MATH4800H Final Report

Time Series Interpolation Algorithms

Melissa Van Bussel

Winter 2019

Contents

Introduction **4**

Background Reading **4**

Time Series . . . 4
Spacing of Observations . . . 4
Evenly Spaced Observations . . . 4
Approximately Evenly Spaced Observations . . . 4
Missing Values . . . 5
Assumptions . . . 5
Sampling Frequency . . . 6
Autocovariance and Autocorrelation . . . 6
Formal Definitions . . . 8
Trends . . . 9
Linear Trends . . . 9
Exponential Trends . . . 10
Periodic Trends . . . 10
Variance . . . 11
Removing trends . . . 12
Logarithmic Transformations . . . 12
Differencing . . . 13
Some Stochastic Processes . . . 14
White Noise Processes . . . 14
Random Walks . . . 16
Stationarity . . . 19
Stationary Processes . . . 19
Weakly Stationary . . . 19
Strictly Stationary . . . 20
Non-Stationary Processes . . . 20
Why does it matter? . . . 20
When is a process stationary? . . . 20
The Autoregressive Model (AR) . . . 20
Modeling and Forecasting AR Processes . . . 24
Moving Average (MA) Processes . . . 27
ARMA Models . . . 29
Why does stationarity matter? . . . 30

Interpolation Algorithms **30**

Nearest Neighbor Interpolation . . . 32
Linear Interpolation . . . 34
The na.approx Family of Functions in R . . . 36
The na.interp Function in R . . . 40
The imputeTS package in R . . . 40
The na.interpolation Function . . . 40
The na.kalman Function . . . 41

The na.locf Function . . . 43
The na.ma Function . . . 44
The na.mean Function . . . 47
The na.random Function . . . 48
Other functions in the imputeTS package . . . 49
Hybrid Wiener Interpolator . . . 50

Evaluating Performance of Algorithms **51**
Coefficient of Correlation . . . 52
Squared Coefficent of Correlation . . . 52
Absolute Differences . . . 52
Mean Bias Error . . . 53
Mean Error . . . 53
Mean Absolute Error . . . 53
Mean Relative Error . . . 53
Mean Absolute Relative Error . . . 53
Mean Absolute Percentage Error . . . 54
Sum of Squared Errors . . . 54
Mean Square Error . . . 54
Root Mean Squares . . . 54
Mean Squares Error . . . 55
Reduction of Error . . . 55
Root Mean Square Error . . . 55
Normalized Root Mean Square Deviation . . . 55
Root Mean Square Standardized Error . . . 55
Discussion of Performance Criteria . . . 56
Problems with Performance Criteria . . . 56

Testing and Benchmarking a Variety of Algorithms **56**
Datasets Used . . . 57
The Air Quality Dataset . . . 57
The Sunspots Dataset . . . 58
The Solar Flux Dataset . . . 59
Results . . . 61
Discussion . . . 66
Improvements . . . 69
Increase Data . . . 69
Increase gap selection methods . . . 69
Repetition . . . 69
Increase and refine algorithms and criteria . . . 69
Future Work . . . 70

Conclusion **70**

References **70**

Introduction

The work presented in this document is for the course MATH4800H - Mathematics Honours Project, during the Winter 2019 Semester. The main goals of this project are:

- Research a variety of time series interpolation algorithms
- Test the algorithms on real-world data
- Evaluate the performance of the algorithms, using a number of performance criteria

Background Reading

This section will provide a "crash course" in time series. This background is necessary in order to understand the context as well as the significance of the goals of this Honours Project. The definitions are taken from [4] and [5].

Time Series

A time series is a sequence of data arranged in chronological order. This type of data is seen everywhere, but is an especially common format for financial or economical data.

For this project, a time series will be denoted $\{X_t\}$, with one observation taken at each time t.

Spacing of Observations

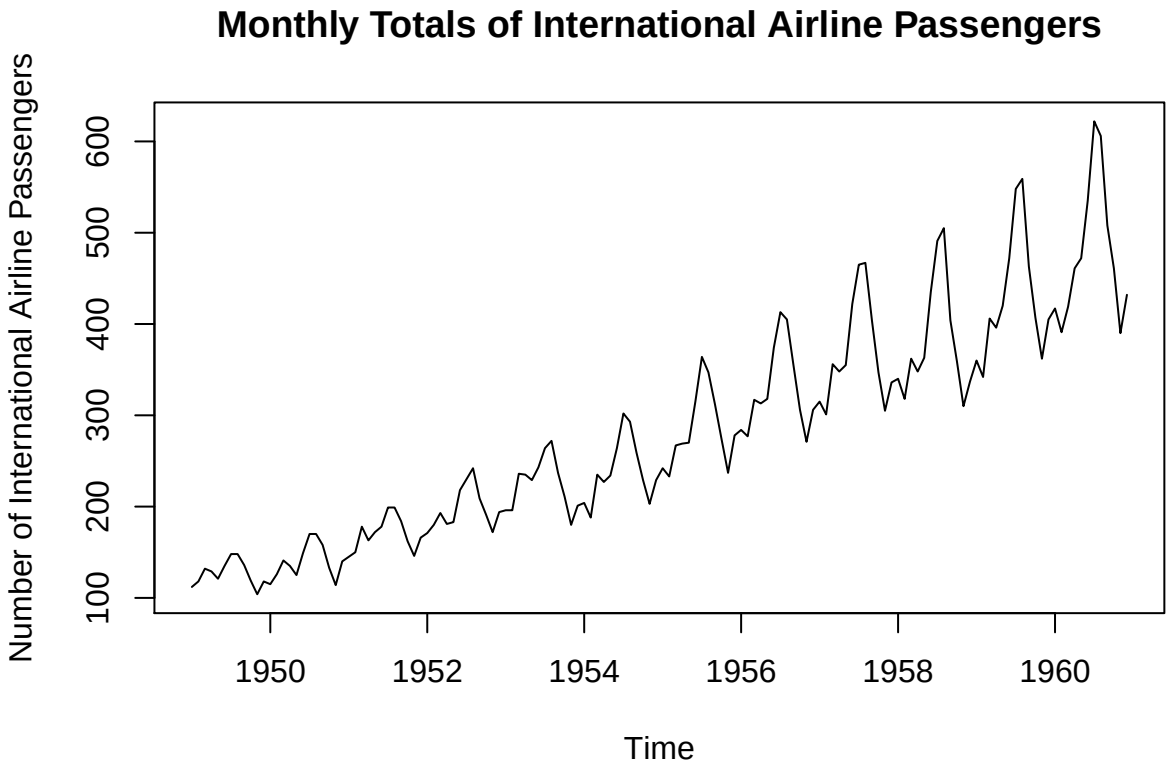
Evenly Spaced Observations

Sometimes, time series data is evenly spaced. This means that each observation is taken at regular time intervals. For example, consider measurements of temperature taken once each hour, on the hour.

Approximately Evenly Spaced Observations

It is also possible to have time series wherein observations are spaced approximately evenly. For example, the dataset `AirPassengers` in `R` records the total number of international airline passengers. There is one observation per month from January 1949 to December 1960. These observations are only "approximately spaced" because the number of days in a month is not the same for each month.

```
data("AirPassengers")
plot(AirPassengers,
     main = "Monthly Totals of International Airline Passengers",
     ylab = "Number of International Airline Passengers",
     type = "l")
```



In reality, most time series have approximately evenly spaced intervals. For example, most years have 365 days, but leap years have 366 to account for the fact that the Earth completes a single rotation around the Sun once every 365.24 years, approximately. Since this is such a small difference, it is reasonable to proceed as if the observations are equally spaced.

Missing Values

Another reality when working with temporal data is that there can often be missing values. This can occur for a variety of reasons. For example, financial datasets might only have observations on business days, so there may be missing values on weekends or holidays. In `R`, missing values are represented with `NA`.

In scientific datasets, equipment failure can lead to missing observations (or incorrect data, which needs to be removed from the dataset, leading to missing values). This will be of particular importance to this Honours Project.

Assumptions

When analyzing time series, the assumption is typically made that observations are taken at evenly spaced time intervals. This assumption is required for majority of methods currently available for modelling time series data.

Sampling Frequency

The *sampling frequency* refers to the number of observations per cycle / period. For example, a time series which has one observation per month would have a sampling frequency of 12, because one period is 12 observations. In `R`, the `frequency()` function can be used to determine the sampling frequency of a time series (`ts`) object. For example, consider again the `Airline Passengers` dataset.

```
frequency(AirPassengers)
```

```
## [1] 12
```

Autocovariance and Autocorrelation

Recall that the *covariance* between two sets of data is a measure of how strongly associated the two sets are. For example,

```
data(iris)
cov(iris$Petal.Length, iris$Petal.Width)
```

```
## [1] 1.295609
```

The problem with covariance is that it is difficult to interpret, because it is dependent on the scale of the two variables. In the above example, it is clear that there is a positive association between the variables `Petal.Length` and `Petal.Width`, but it is unclear how strong that positive association is.

Instead, it is useful to scale the covariance of the two variables by their standard deviations. This is equivalent to calculating the *correlation* between the two variables. This is a more interpetable value – recall that correlations can range from -1 to +1, where -1 represents a perfectly negative linear relationship, +1 represents a perfectly positive linear relationship, and 0 represents no linear assocation.

```
cov(iris$Petal.Length, iris$Petal.Width) / (sd(iris$Petal.Length) * sd(iris$Petal.Width))
```

```
## [1] 0.9628654
```

```
cor(iris$Petal.Length, iris$Petal.Width)
```

```
## [1] 0.9628654
```

Now, it is much easier to tell that there is a near-perfect positive linear relationship between the two variables.

This leads to the concept of autocovariance and autocorrelation. Similar to how covariance and correlation measure how strongly associated two variables are, autocovariance and autocorrelation measure how closely observations in a time series are associated with each other.

The *lag-1* autocorrelation finds the correlation between observations which are one time step apart. To compute the lag-1 autocorrelation of a time series, just use the `cor` function on the original series, along with the series that has been shifted backwards (lagged) by 1. This will mean losing one data point.

```
data(sunspots)
# must remove the last observation in order for
# the vectors to both be the same size
```

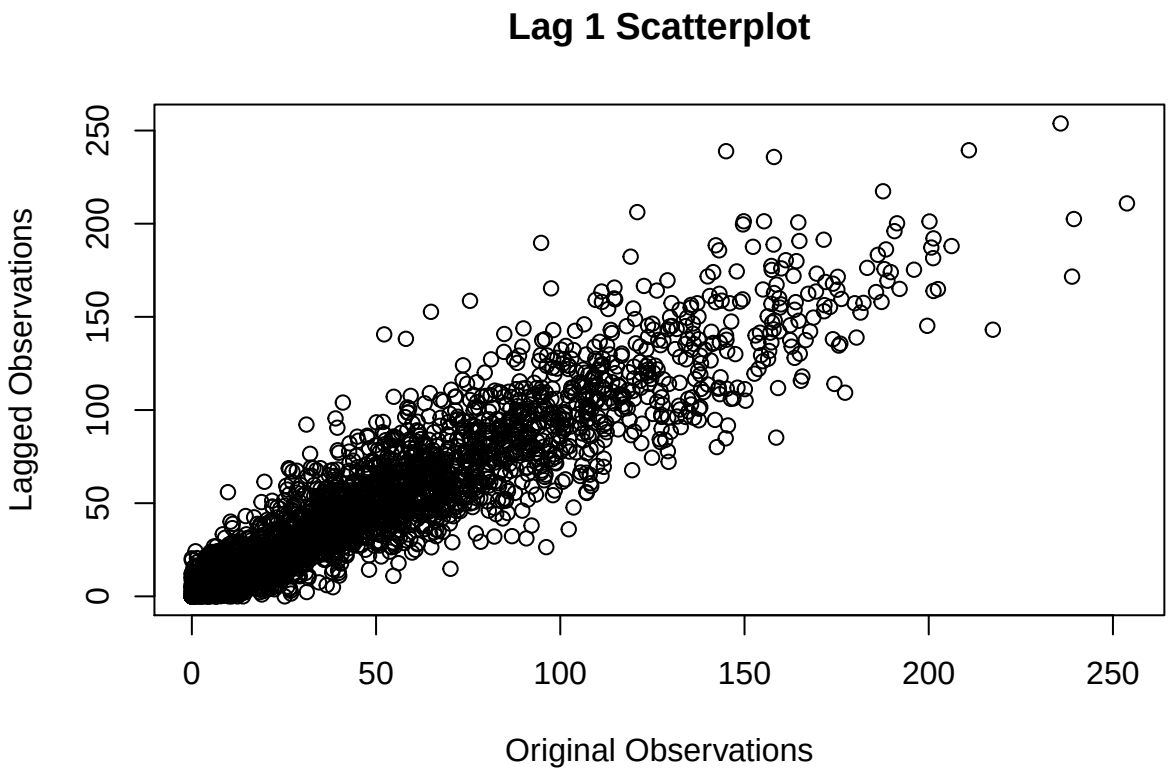
```
original <- sunspots[-length(sunspots)]
lagged_by_1 <- sunspots[-1]
cor(original, lagged_by_1)
```

```
## [1] 0.9217177
```

There is high correlation between successive observations. This means that prediction, or *forecasting*, future values of this series will be quite doable, since successive observations are very similar.

Below is a plot showing each observation in the `sunspots` time series, plotted against the previous observation.

```
plot(original,
     lagged_by_1,
     main = "Lag 1 Scatterplot",
     ylab = "Lagged Observations",
     xlab = "Original Observations")
```



A clear linear relationship can be seen between observations which are lagged by 1.

This process can be repeated. For example the lag-2 autocorrelation can be found:

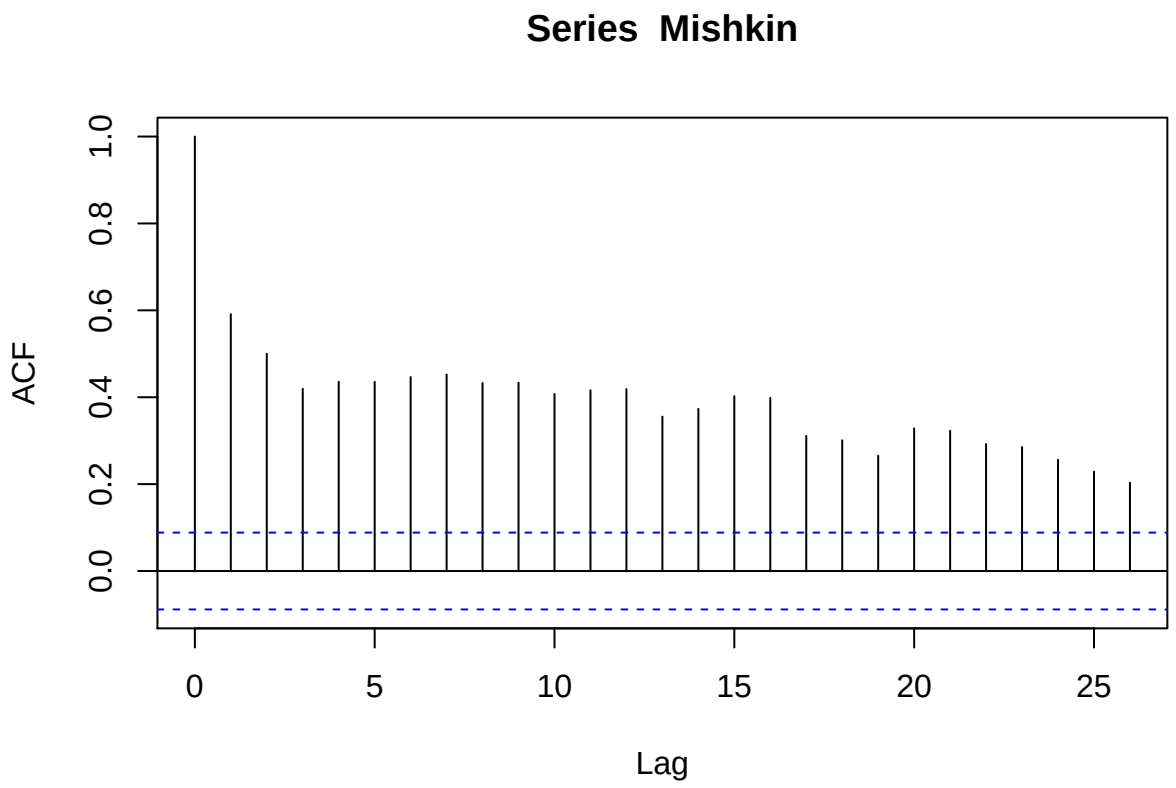
```
original <- sunspots[-(2819:2820)]
lagged_by_2 <- sunspots[-(1:2)]
cor(original, lagged_by_2)
```

```
## [1] 0.8905405
```

There is still a high positive correlation between observations and lag 2 observations.

The *autocorrelation function* (commonly known as the ACF) is a function of the time lag. In `R`, the `acf` function can be used.

```
data(Mishkin, package = "Ecdat")
Mishkin <- as.vector(Mishkin[, 1])
acf(Mishkin)
```



Sort of unsurprisingly, the autocorrelation is highest at smaller lags, and eventually goes down to zero at higher lags, indicating that observations in the above series are highly related to the most recent observations. This is quite common.

Formal Definitions

We define the autocovariance function as

$$\gamma(h) = \text{Cov}[X_t, X_{t+h}]$$

It is important to note that if weak stationarity is assumed, then $\gamma(h) = \gamma(-h)$. It can also be observed that $\gamma(0) = \sigma^2$.

We define the autocorrelation function as

$$\begin{aligned}\rho(h) &= \frac{\gamma(h)}{\sigma^2} \\ &= \frac{\gamma(h)}{\gamma(0)}\end{aligned}$$

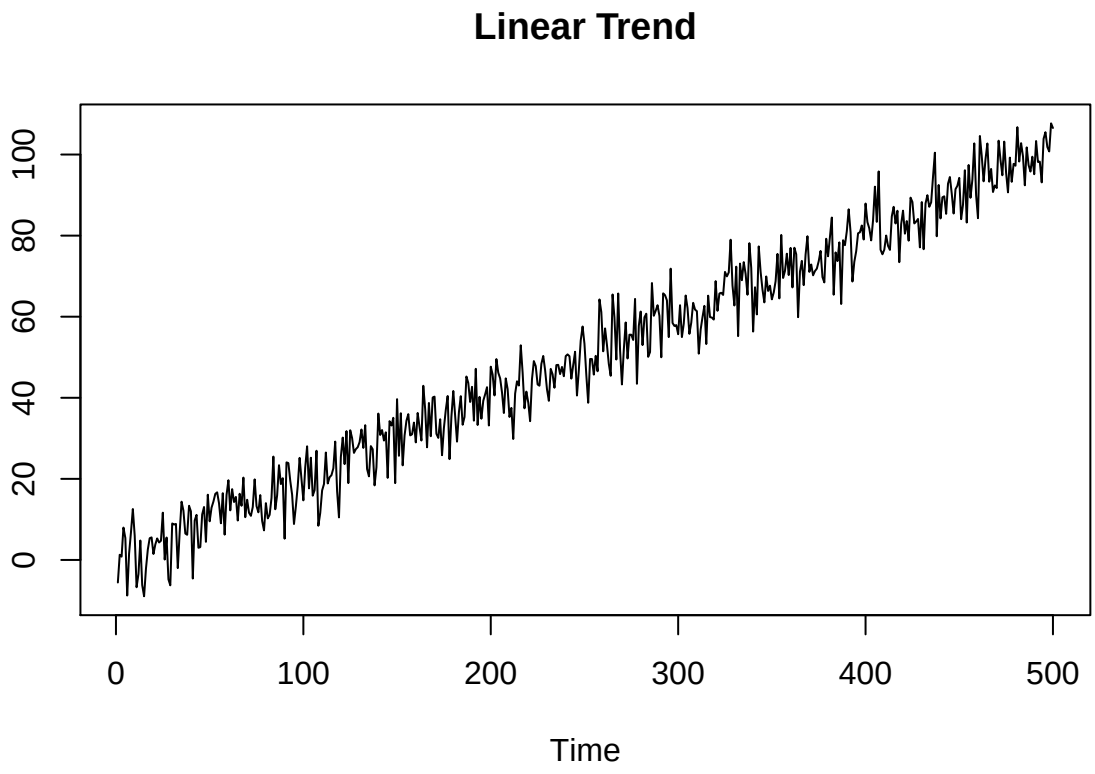
Trends

Many time series exhibit trends over time. These can be classified into a number of categories.

Linear Trends

Data which either increases or decreases over time in a linear fashion is said to have a *linear trend*.

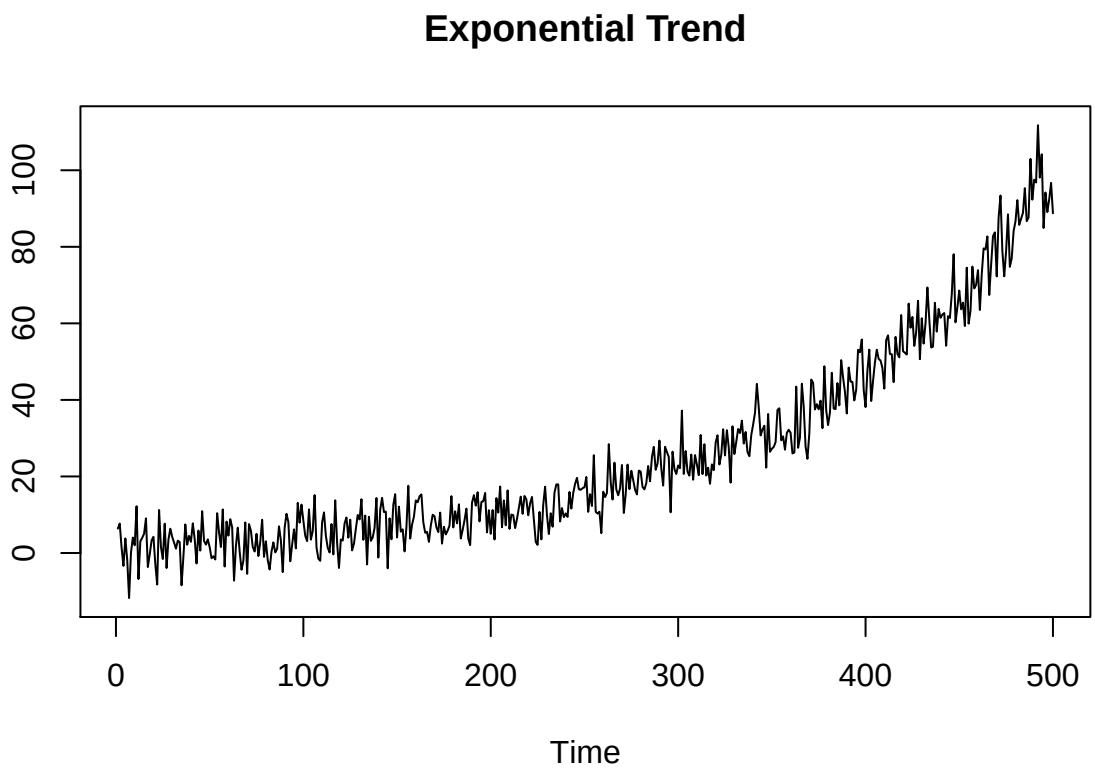
```
noise <- rnorm(500, 0, 5)
linear <- seq(1, 100, length.out = 500) + noise
plot(linear, type = "l",
     main = "Linear Trend", ylab = "", xlab = "Time")
```



Exponential Trends

Data which grows or shrinks in an exponential fashion is said to have an *exponential trend*.

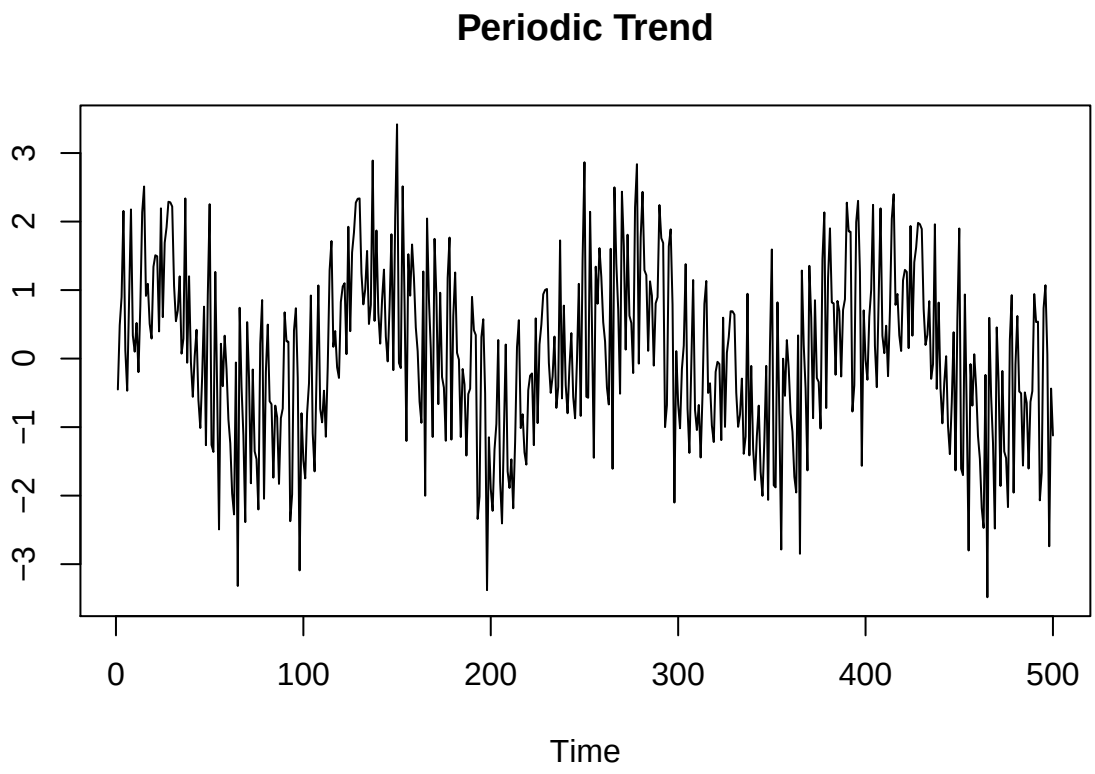
```
noise <- rnorm(500, 0, 5)
time <- seq(1, 5, length.out = 500)
exponential <- 2.5 ^ time + noise
plot(exponential, type = "l",
     main = "Exponential Trend",
     xlab = "Time",
     ylab = "")
```



Periodic Trends

Trends can also be periodic in nature. Often, this is referred to as *cyclical* data.

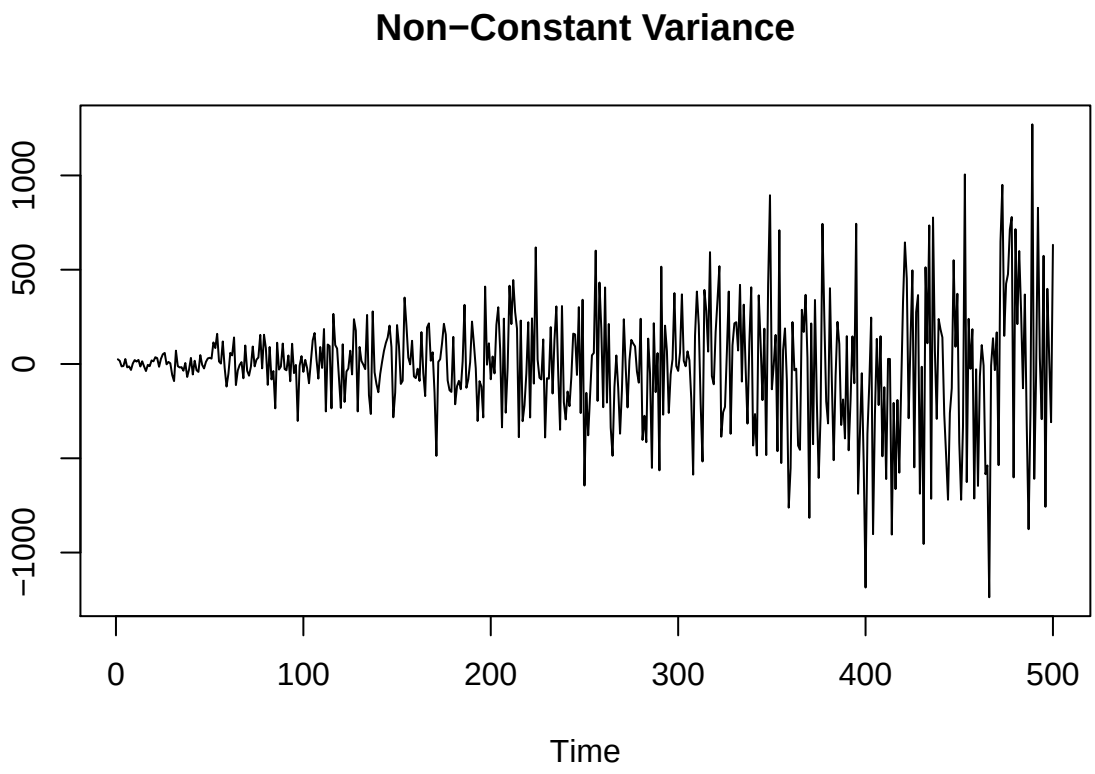
```
noise <- rnorm(100, 0, 1)
periodic <- sin(seq(1, 25, length.out = 500)) + noise
plot(periodic, type = "l",
     main = "Periodic Trend",
     xlab = "Time",
     ylab = "")
```



Variance

In addition to the mean of a time series varying with time (as described above), the variance of a time series can also follow trends. In the following example, the variance grows as a function of time.

```
noise <- rnorm(500, 0, 5)
non_const_var <- seq(1, 100, length.out = 500) * noise + noise * 2
plot(non_const_var, type = "l",
     main = "Non-Constant Variance",
     xlab = "Time",
     ylab = "")
```



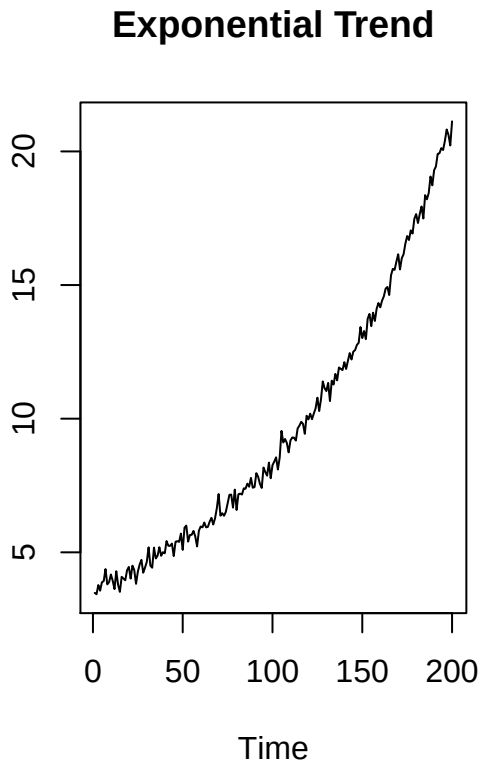
Removing trends

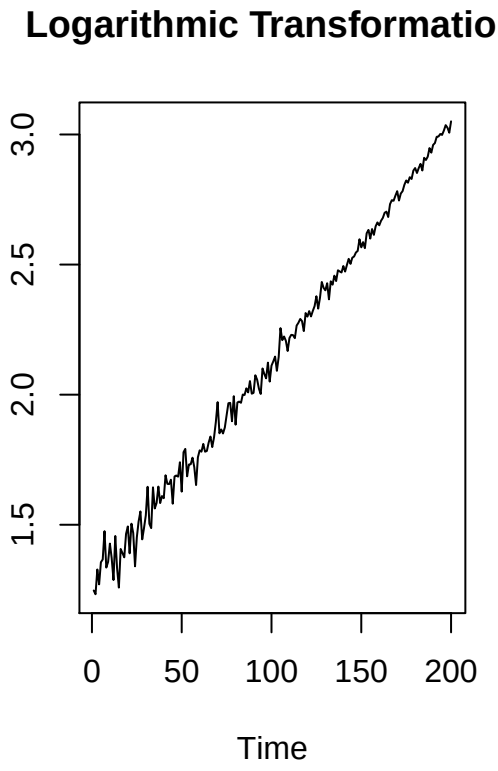
Transformations can be applied in order to remove the trends in time series.

Logarithmic Transformations

Performing a logarithmic transformation on exponential data will remove the exponential trend and will create a linear trend. Obviously, this transformation will only work on data which is positive.

```
noise <- rnorm(200, 0, 0.25)
time <- seq(1, 3, length.out = 200)
exponential <- exp(time) + noise + 1
par(mfrow = c(1, 2))
plot(exponential, type = "l",
     main = "Exponential Trend",
     xlab = "Time",
     ylab = "")
plot(log(exponential), type = "l",
     main = "Logarithmic Transformation",
     ylab = "",
     xlab = "Time")
```

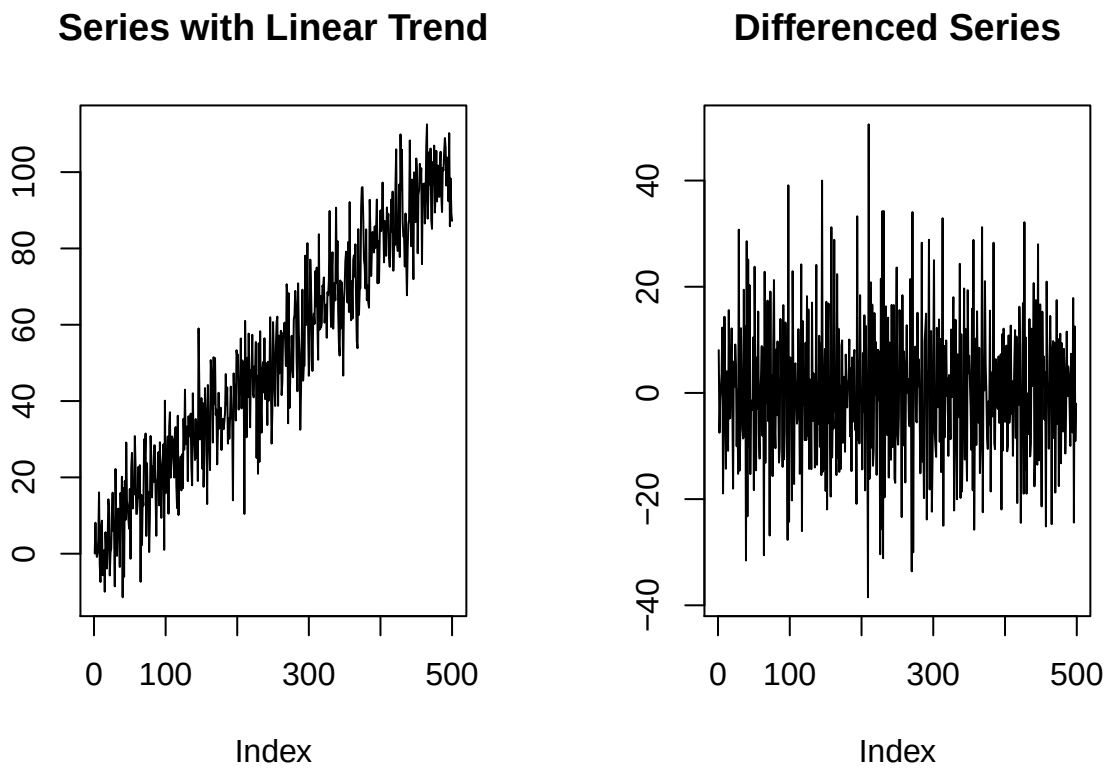




Differencing

The `diff` function can be used to remove a linear trend. *Differencing* refers to computing the difference between two consecutive observations. A differenced series will always have one less observation than the original series.

```
noise <- rnorm(500, 0, 5)
linear <- seq(1, 100, length.out = 500) + noise * 2
differenced <- diff(linear)
par(mfrow = c(1, 2))
plot(linear, type = "l",
     main = "Series with Linear Trend",
     ylab = "")
plot(differenced,
     main = "Differenced Series",
     type = "l",
     ylab = "")
```



Some Stochastic Processes

White Noise Processes

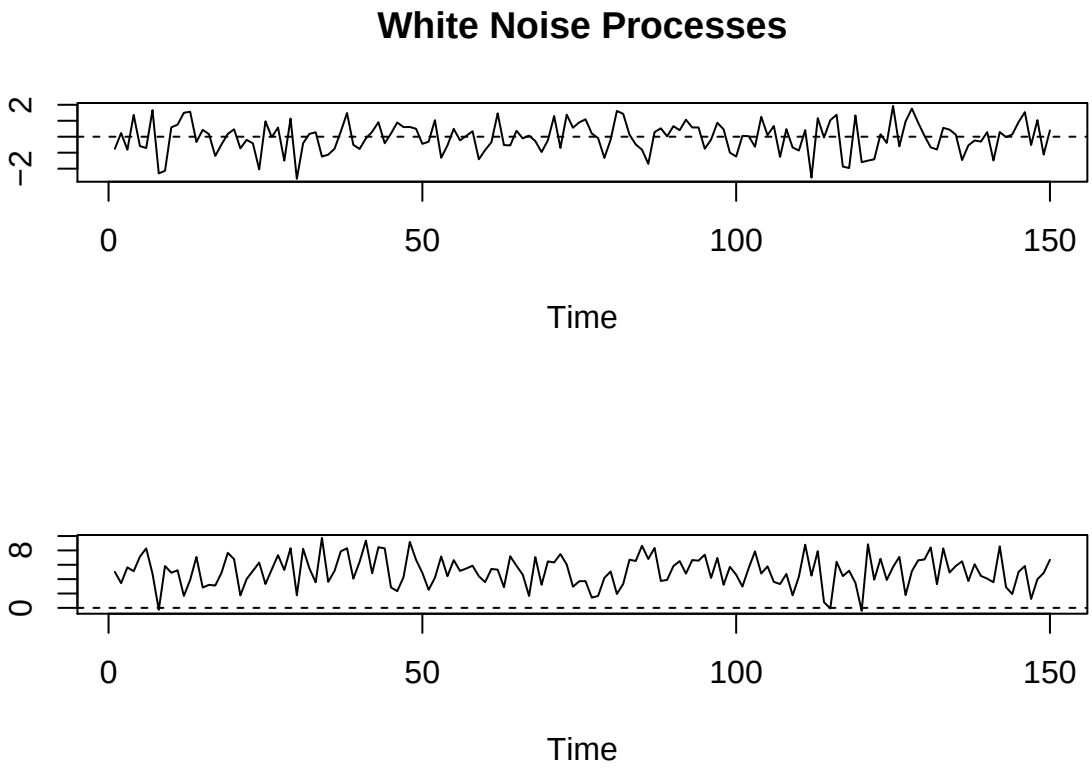
A process is said to be a *weak white noise process* if it has constant mean and variance (i.e., unlike the examples presented immediately above). More formally, let $\{X_t\}$ be a time series. Then,
$\{X_t\} \sim$ weak WN(μ, σ^2) if and only if

1. $\mathbf{E}[X_t] = \mu \quad \forall t$
2. $\mathbf{Var}[X_t] = \sigma^2 \quad \forall t$
3. $\mathbf{Cov}[X_t, X_s] = 0 \quad \forall t \neq s$

The following example will demonstrate two white noise processes – one with WN$(0, 1)$ and the second with WN$(4, \sqrt{2})$.

```
library(forecast)
wn1 <- arima.sim(model = list(order = c(0, 0, 0)),
                 n = 150)
wn2 <- arima.sim(model = list(order = c(0, 0, 0)),
                 n = 150,
```

```
                mean = 4,
                sd = 2)
par(mfrow = c(2, 1))
plot(wn1, main = "White Noise Processes",
     ylab = "")
abline(h = 0, lty = 2)
plot(wn2 + 1, ylab = "")
abline(h = 0, lty = 2)
```



If given a time series, one can use the `arima` function in `R` to estimate the parameters (μ, σ^2) of the white noise process.

```
arima(wn2, order = c(0, 0, 0))
```

```
##
## Call:
## arima(x = wn2, order = c(0, 0, 0))
##
## Coefficients:
##       intercept
##          4.0971
## s.e.     0.1728
##
```

```
## sigma^2 estimated as 4.478:  log likelihood = -325.29,  aic = 654.57
```

The true values of μ and σ^2 were both 4, so the `arima` function estimated the parameters of the process quite effectively.

Random Walks

Random Walks (often abbreviated to RW) are stochastic processes which are much different from White Noise processes. Random Walk processes do **not** have a constant mean or variance, and values which are close to each other are highly correlated. The increments of a random walk process follow a white noise process. To find the i^{th} observation in a RW process, take the $(i-1)^{th}$ observation and add noise. Formally,

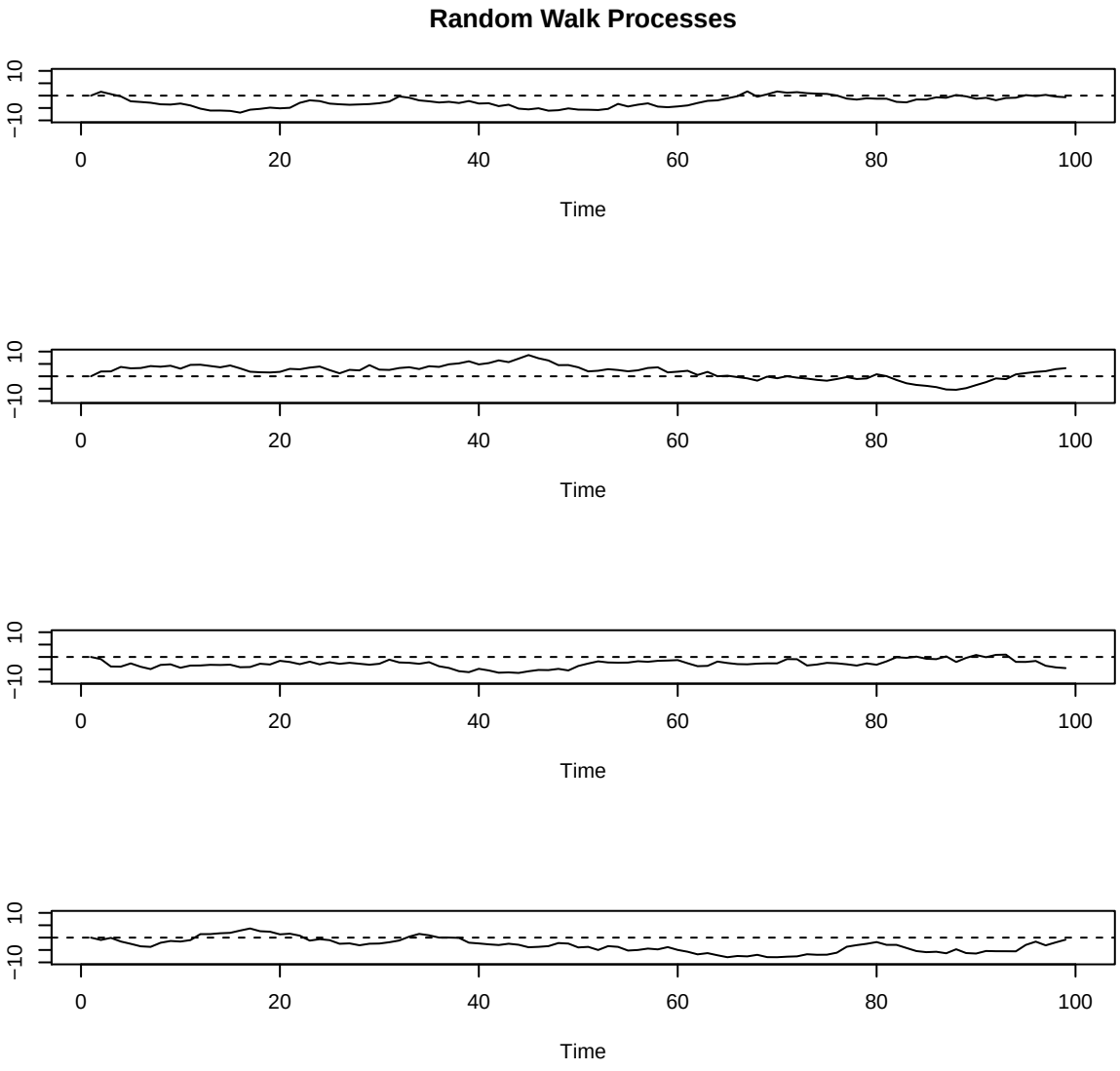
$$X_t = X_{t-1} + \epsilon_t, \quad \epsilon_t \sim \text{WN}(0, \sigma^2)$$

Thus, the only parameter of a RW process is the variance of its associated WN process, σ^2.

To simulate a Random Walk, a starting position must be used. This is typically 0, but doesn't have to be. The following examples demonstrate that even Random Walks which originate from the same starting position and have the same white noise process can turn out very different due to the stochastic nature of the process.

```r
set.seed(12)
wn <- list()
y <- list()
for (i in 1:4) {
  wn[[i]] <- arima.sim(model = list(order = c(0, 0, 0)),
                n = 99)
  y[[i]] <- vector(length = 100)
  y[[i]][1] <- 0
  for (j in 2:100) {
    y[[i]][j] <- y[[i]][j - 1] + wn[[i]][j]
  }
}
par(mfrow = c(4, 1))
plot(y[[1]], type = "l",
     main = "Random Walk Processes",
     ylab = "",
     xlab = "Time",
     ylim = c(-10, 10))
abline(h = 0, lty = 2)
for (i in 2:4) {
  plot(y[[i]], type = "l",
       ylab = "",
       xlab = "Time",
       ylim = c(-10, 10))
  abline(h = 0, lty = 2)
```

```
}
```



Note that the `cumsum` function in `R` can convert a vector containing white noise realizations into a vector containing random walk realizations (because the `cumsum` function returns a vector whose elements are the *cumulative sum* of the elements in the vector passed to it).

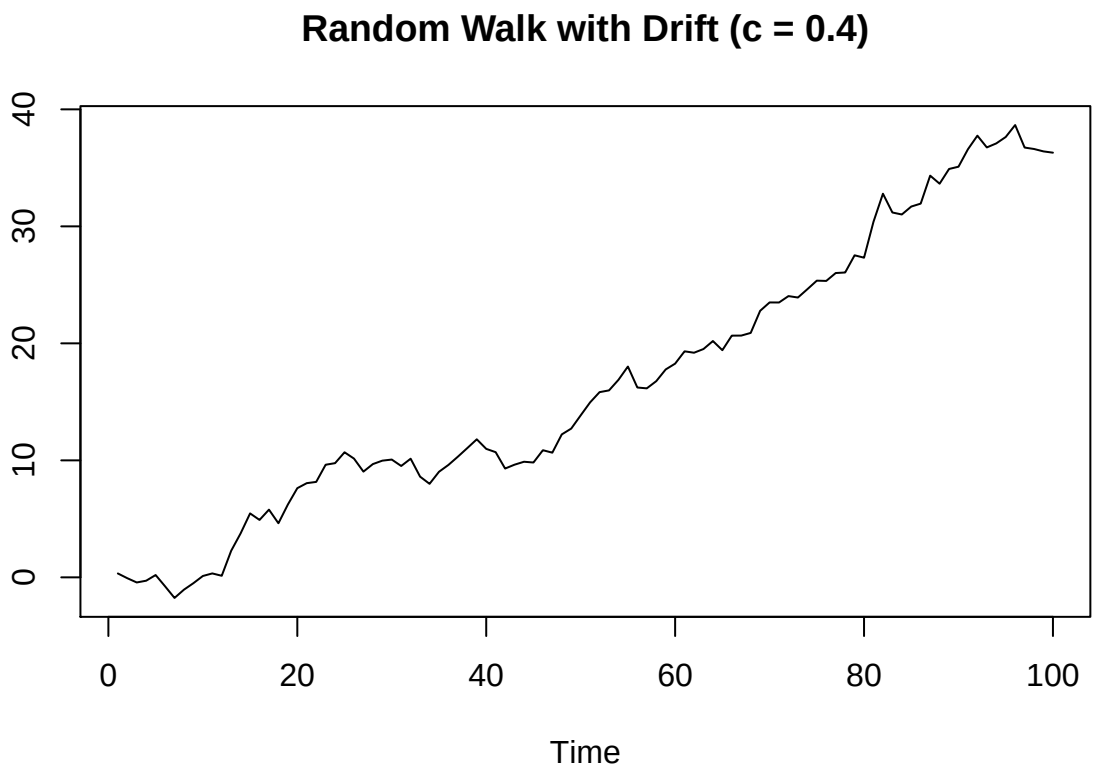
Random walks with drifts are Random Walk processes with a constant term added in. Formally,

$$X_t = X_{t-1} + c + \epsilon_t, \quad \epsilon_t \sim \text{WN}(0, \sigma^2), \quad c \in \mathbb{R}$$

Thus, a Random Walk with drift has two paramters: c and σ^2.

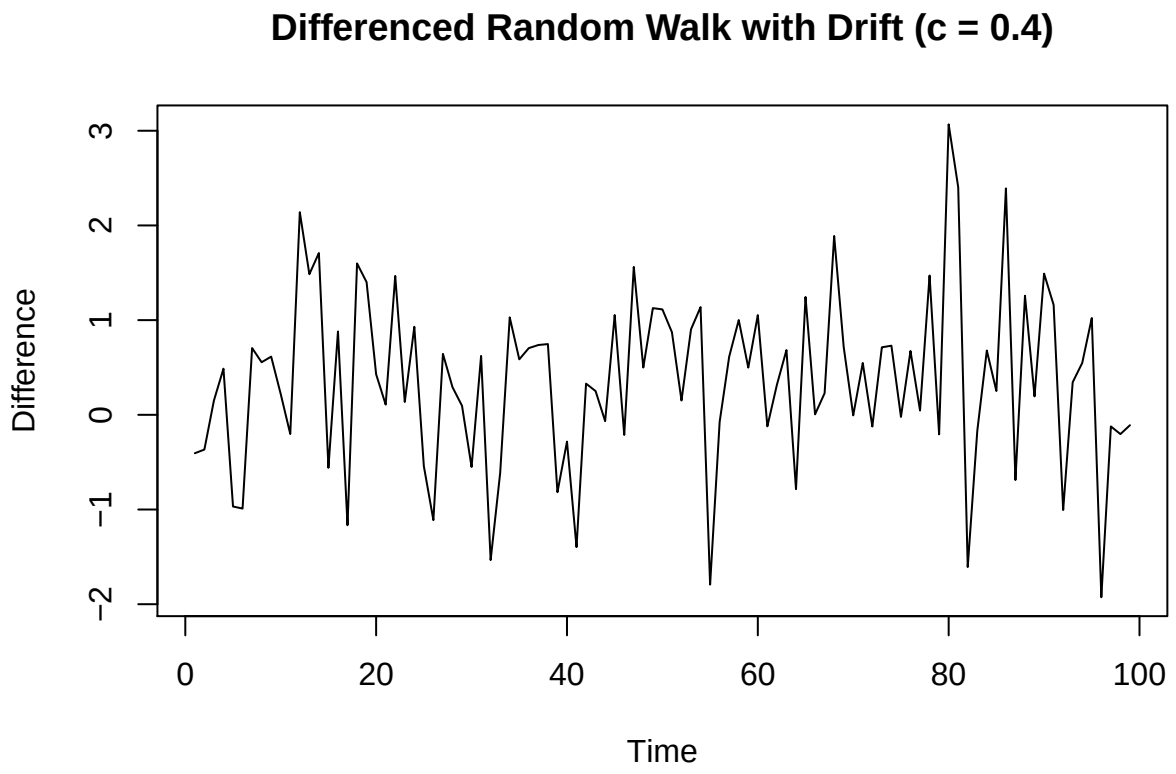
```
wn <- arima.sim(model = list(order = c(0, 0, 0)),
                n = 100,
                mean = 0.4)
```

```r
rw_drift <- cumsum(wn)
plot(rw_drift, type = "l",
     main = "Random Walk with Drift (c = 0.4)",
     xlab = "Time",
     ylab = "")
```



Random Walk processes can be arranged as a differenced series. The difference between any two consecutive terms is ϵ_t (or $\epsilon_t + c$ for a RW with drift). Thus, $\texttt{diff}(X_t) \sim \text{WN}(c, \sigma^2)$. For an RW without a drift, $c = 0$.

```r
plot(diff(rw_drift), type = "l",
     main = "Differenced Random Walk with Drift (c = 0.4)",
     xlab = "Time",
     ylab = "Difference")
```



Stationarity

Stationary Processes

Roughly speaking, a stationary process is one which isn't a function of time. This will be defined more rigorously below, but for now, it is useful to know that White Noise processes are stationary, while Random Walk processes are not.

Weakly Stationary

A process which is *weakly stationary* is one for which the mean, variance, and covariance are constant over time (and finite). Formally, $X_1, X_2, ...$ is weakly stationary iff:

1. $\mathbf{E}[X_t] = \mu \quad \forall t$
2. $\mathbf{Var}[X_t] = \sigma^2 \quad \forall t, \sigma^2 < \infty$
3. $\mathbf{Cov}[X_t, X_s] = \gamma(|t - s|) \quad \forall t, s$ for some $\gamma(h)$

where $\gamma(h)$ is the autocovariance function.

In other words, the third condition says that the covariance between two observations depends only on the *lag* (the distance $|t - s|$) between observations, and not on the indices s and t themselves. The covariance between X at time t and time s depends only on how close t and s are. For example, the covariance of X between times 1 and 4 is the same as the covariance between the times 5 and 8, since they both have the same lag.

Note that some literature refers to weakly stationary as *covariance stationary*.

Strictly Stationary

Strict stationarity is a much stronger assumption, that occurs much less often in real data. A process is said to be *stricly stationary* if all aspects of its behavior are unaffected by shifts in time. This means that if you were to select a set of n consecutive observations from the process, this set would have the same distribution as any other set of n consecutive observations from the same process. Formally,

$$(X_1, X_2, ..., X_n) \sim (X_{1+m}, X_{2+m}, ..., X_{n+m})$$

Non-Stationary Processes

Any process which is not stationary is referred to as *non-stationary*. Random Walks are one such example, as are the trend examples presented earlier (linear, exponential, periodic, etc.).

Why does it matter?

Stationary processes are much easier to model, as they have less parameters. The estimators for these processes are quite straightforward as well. For example, the estimator of the mean, μ, is simply the sample average: that is, $\hat{\mu} = \bar{y}$.

When is a process stationary?

In reality, it can be hard to find processes which are stationary. For example, it is quite common for financial/economical data to be non-stationary. With this being said, however, the changes in the series may exhibit stationarity. Transformations as discussed earlier can also be applied to time series before analysis and modeling in order to obtain stationary processes.

The Autoregressive Model (AR)

Autogressive (AR) processes are an entire category of processes. Begin by considering the simplest case, AR(1). An AR(1) process regresses the current observation on the previous observation. It is useful to use the mean-centred version of this model. That is,

$$X_t - \mu = \phi(X_{t-1} - \mu) + \epsilon_t, \quad \epsilon_t \sim \text{WN}(\mu, \sigma_\epsilon^2)$$

The three parameters are: μ, the mean of the $\{X_t\}$ process, ϕ, the slope coefficent, and σ_ϵ^2, the variance of the white noise.

It can be observed that if $\phi = 0$, then X_t is simply a white noise process with parameters μ and σ_ϵ^2.

Some useful properties arise in the case when $|\phi| < 1$:

$$\mathbf{E}[X_t] = \mu$$

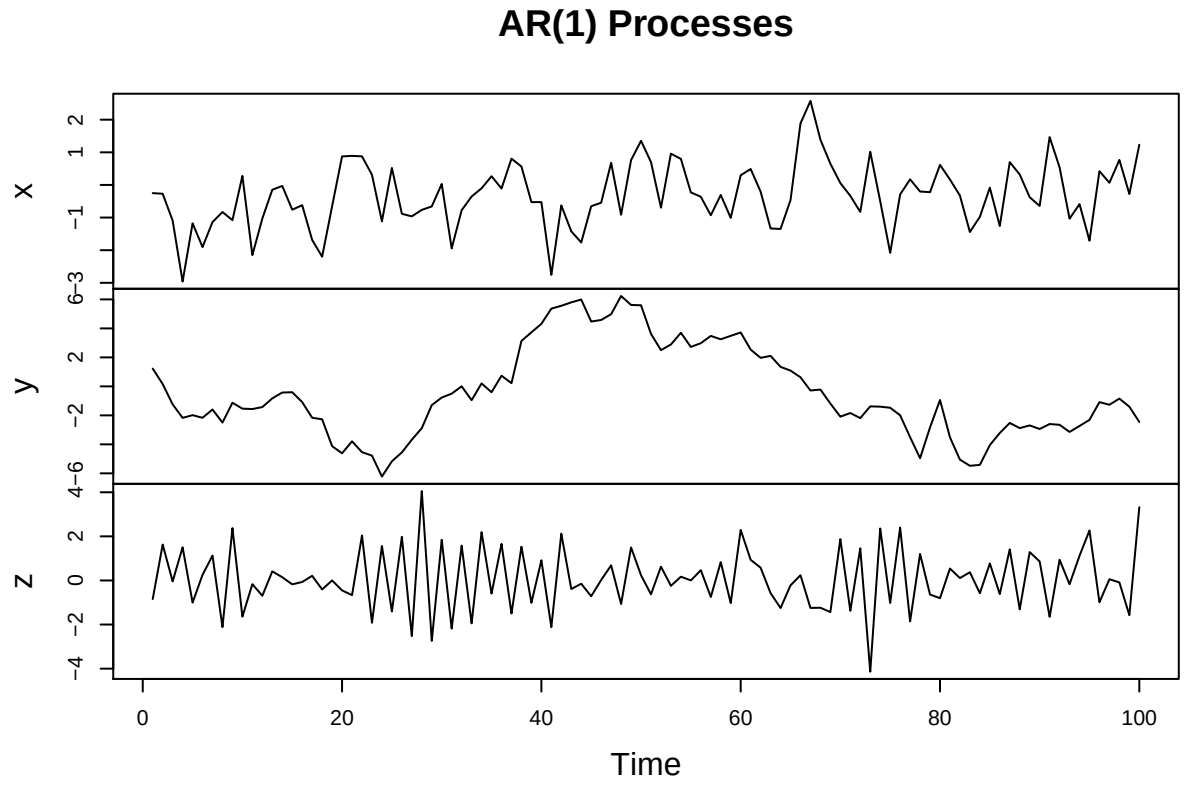
$$\mathbf{Var}[X_t] = \sigma_X^2 = \frac{\sigma_\epsilon^2}{1 - \phi^2}$$

It can also be seen that if $\mu = 0$ and $\phi = 1$, then the process is a random walk (non-stationary).

The higher the value of the slope coefficient, the higher the autocorrelation will be. If the slope coefficient is negative, then the time series is periodic.

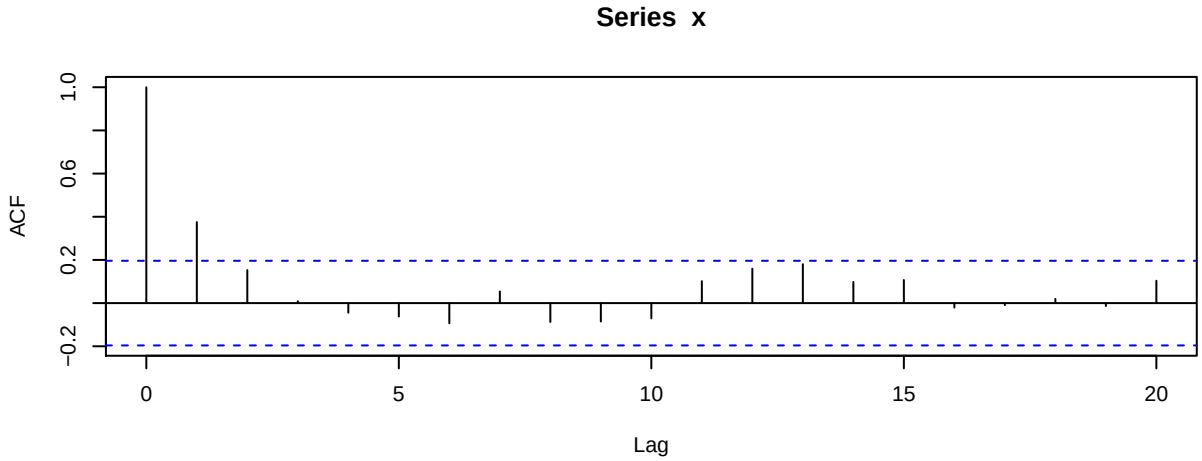
Consider the following examples of AR(1) processes, with slope coefficients 0.5, 0.9, and -0.75, respectively.

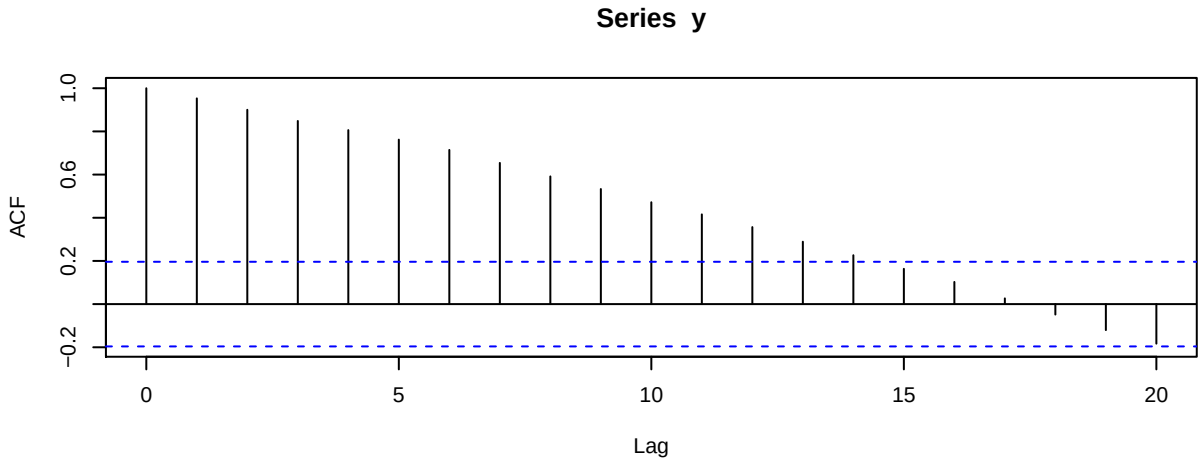
```
# phi = 0.5
x <- arima.sim(model = list(ar = 0.5), n = 100)
# phi = 0.9
y <- arima.sim(model = list(ar = 0.9), n = 100)
# phi = -0.75
z <- arima.sim(model = list(ar = -0.75), n = 100)

# Plot your simulated data
plot(cbind(x, y, z),
     main = "AR(1) Processes")
```

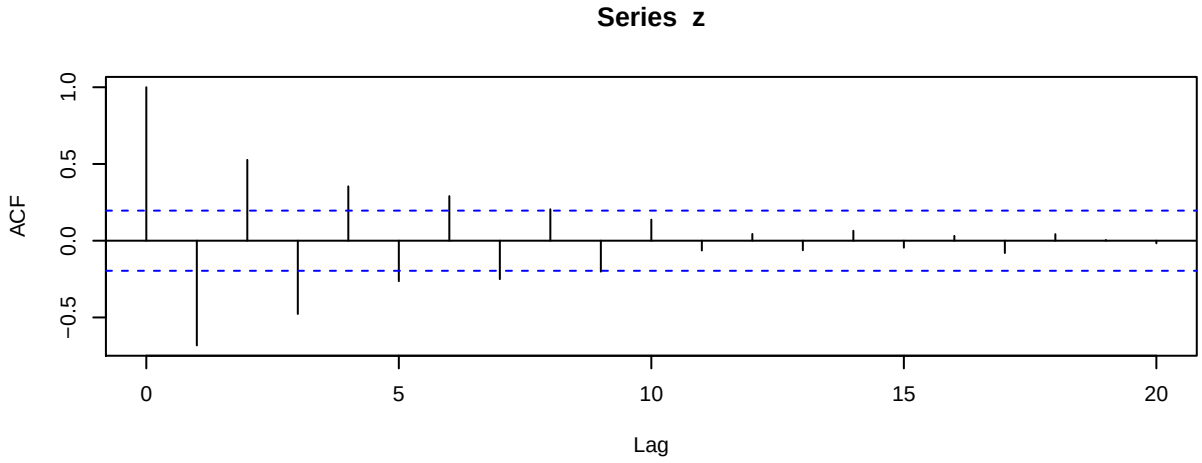


Looking at the autocorrelation functions of the above processes will provide insight, particularly about the periodic nature of the series **z**.

```
par(mfrow = c(3, 1))
acf(x)
acf(y)
acf(z)
```


Series x
ACF
1.0
0.6
0.2
−0.2
0
5
10
15
20
Lag

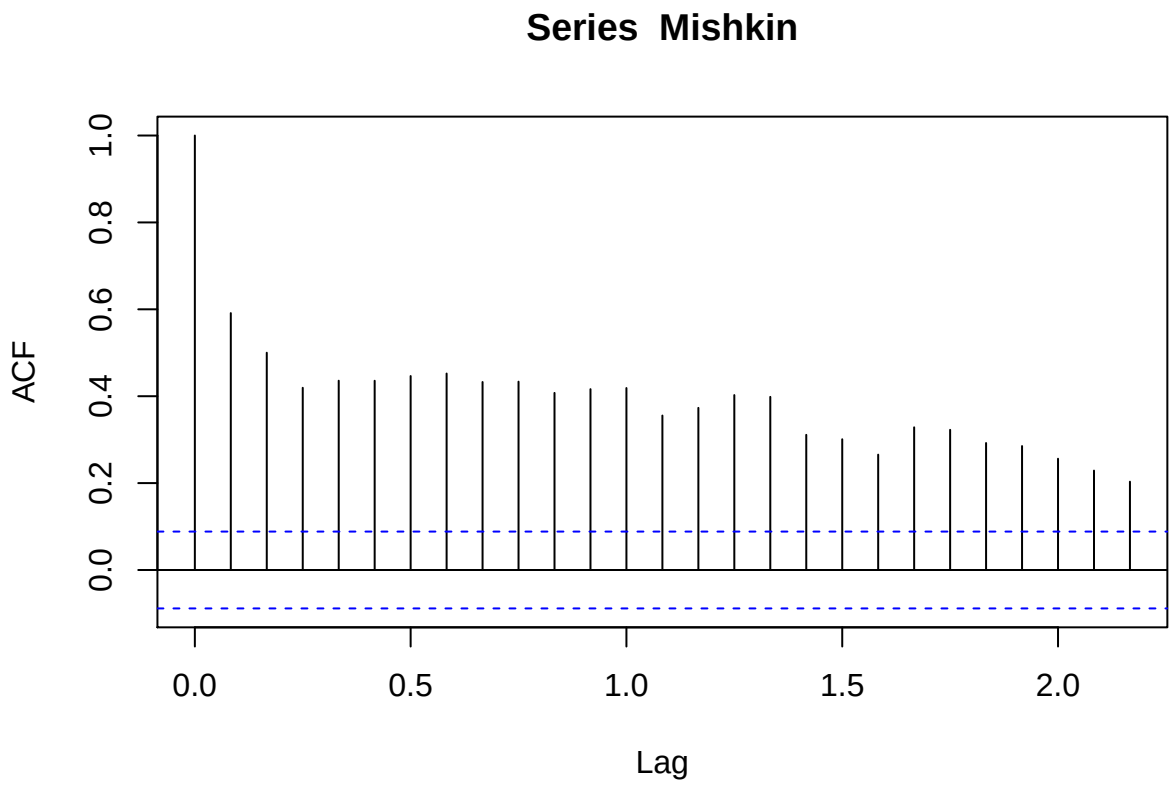

Series y
ACF
1.0
0.6
0.2
−0.2
0
5
10
15
20
Lag


Series z
ACF
1.0
0.5
0.0
−0.5
0
5
10
15
20
Lag

Modeling and Forecasting AR Processes

Consider the `Mishkin` dataset from before, from the package `Ecdat`. We are interested in the first column of this dataset, which is a series that contains the US inflation rate by month.

```
data(Mishkin, package = "Ecdat")
Mishkin <- Mishkin[, 1]
acf(Mishkin)
```



Analyzing the ACF of this data shows a large peak at lag 1, suggesting an AR(1) model may be appropriate.

An AR(1) model is now fitted to the above data:

```
AR1_Mishkin <- arima(Mishkin, order = c(1, 0, 0))
AR1_Mishkin
```

```
##
## Call:
## arima(x = Mishkin, order = c(1, 0, 0))
##
## Coefficients:
##          ar1  intercept
##       0.5960     3.9745
## s.e.  0.0364     0.3471
##
```

```
## sigma^2 estimated as 9.713:  log likelihood = -1255.05,  aic = 2516.09
```

The slope was estimated as $\hat{\phi} = 0.596$ while the mean was estimated as $\hat{\mu} = 3.974$, and the variance of the white noise was estimated to be $\hat{\sigma_\epsilon^2}$ =9.713.

The model produces the following estimates for the series:

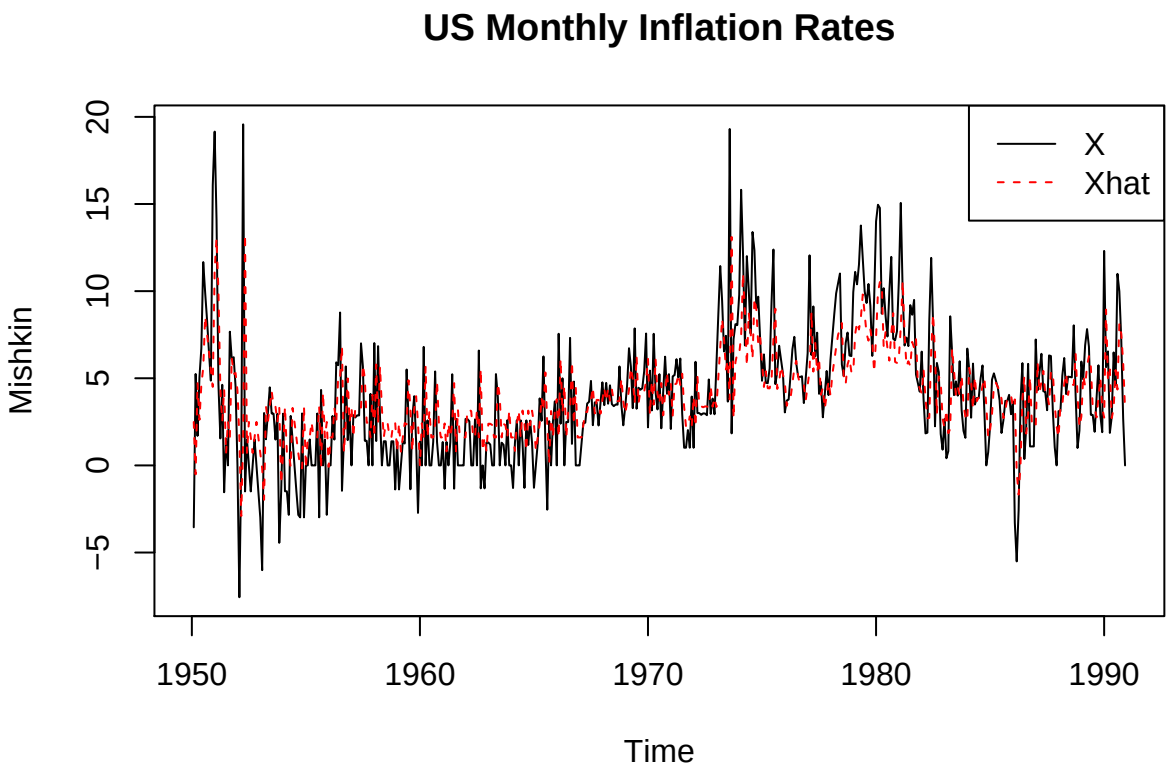
$$\hat{X}_t = \hat{\mu} + \hat{\phi}(X_{t-1} - \mu)$$

and the residuals will be estimated as

$$\hat{\epsilon_t} = X_t - \hat{X}_t$$

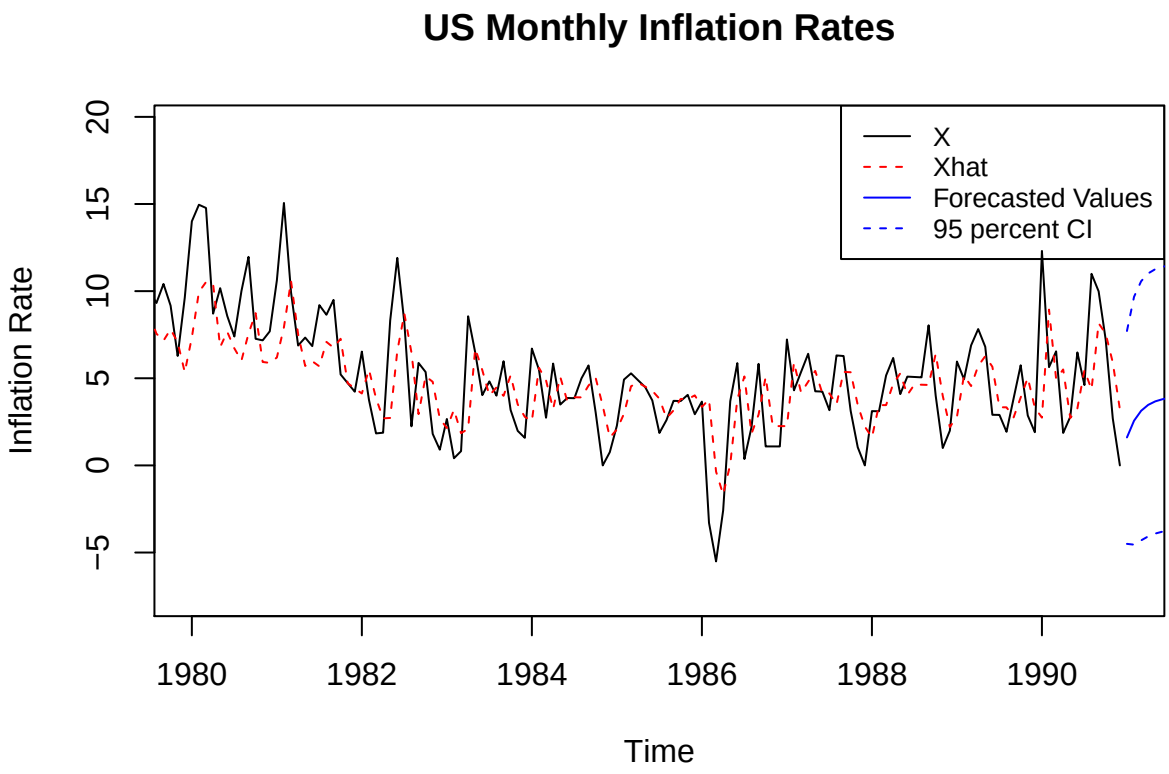
Thus the original series along with the model can be plotted.

```
fitted_vals <- Mishkin - residuals(AR1_Mishkin)
plot(Mishkin, main = "US Monthly Inflation Rates",
     xlim = c(1950, 1991))
lines(fitted_vals, type = "l",
      ylab = "Inflation Rate",
      col = "red",
      lty = 2)
legend("topright",
       legend = c("X", "Xhat"),
       col = c("black", "red"),
       lty = c(1, 2))
```



This can be extended by *forecasting*, or predicting, the next few values in the series. A 95% confidence interval prediction is added to the previous plot (zooming in at the end of the series in order to demonstrate the forecasting):

```
AR_forecast <- predict(AR1_Mishkin, n.ahead = 12)$pred
AR_forecast_se <- predict(AR1_Mishkin, n.ahead = 12)$se
plot(Mishkin, main = "US Monthly Inflation Rates",
     xlim = c(1980, 1991),
     ylab = "Inflation Rate")
lines(fitted_vals, type = "l",
      col = "red",
      lty = 2)
points(AR_forecast, type = "l", col = "blue")
points(AR_forecast - 1.96 * AR_forecast_se,
       type = "l", col = "blue", lty = 2)
points(AR_forecast + 1.96 * AR_forecast_se,
       type = "l", col = "blue", lty = 2)
legend("topright",
       legend = c("X", "Xhat", "Forecasted Values", "95 percent CI"),
       col = c("black", "red", "blue", "blue"),
       lty = c(1, 2, 1, 2),
       cex = 0.8)
```



Note that the AR(1) process is just one of many in the family of AR(p) processes. This extension is as one would expect: An AR(p) process is defined as

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} + \epsilon_t, \quad \epsilon_t \sim \text{WN}(\mu, \sigma_\epsilon^2)$$

The extension in `R` is straightforward: just specify `order = c(p, 0, 0)` within the `arima()` function call.

Moving Average (MA) Processes

Moving Average (MA) processes are another family of stochastic processes. Consider the simplest Moving Average (MA): an MA(1) process. A time series $X_1, X_2, ...$ is said to be a *simple moving average process* (an MA(1) process) iff:

$$X_t = \mu + \epsilon_t + \theta \epsilon_{t-1}$$

where $\epsilon_t \sim \text{WN}(\mu, \sigma_\epsilon^2)$. This is quite similar, conceptually, to the AR(1) process, except that the current observation is regressed on the noise of the previous observation. Again, this process has 3 paramters. This time, the slope coefficent is represented by θ. Once again, if the slope coefficient is zero, then the series X_t is a white noise process. Additionally, once again, larger values of the slope coefficient correspond to higher autocorrelation, while negative values of θ correpond to series with periodic behavior.

The following are properties of an MA(1) process:

$$\rho(1) = \frac{\theta}{1+\theta^2}$$
$$\rho(h) = 0 \quad \forall h > 1$$

The MA(1) process can only predict a 1-step forecast.

The same dataset as before is used to demonstrate modeling and prediction using an MA(1) process. Similar to before, the values are modeled as:

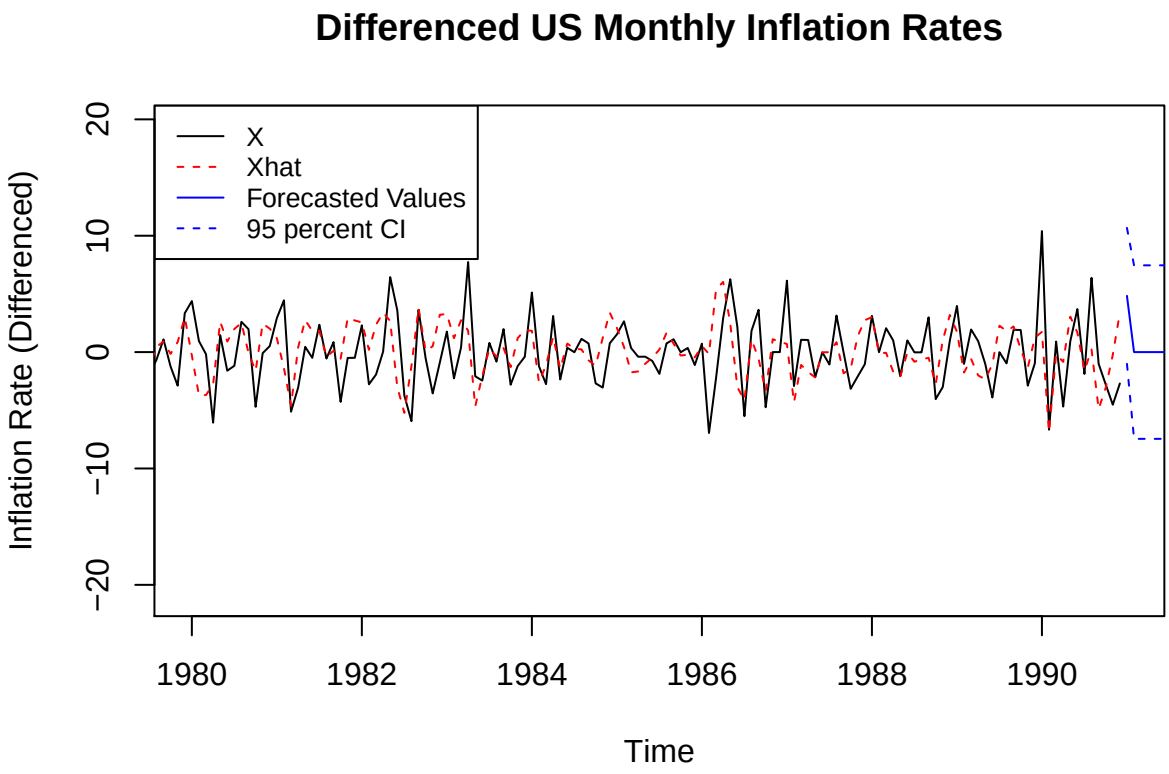
$$\hat{X}_t = \hat{\mu} + \hat{\theta}\hat{\epsilon_{t-1}}$$

with residuals

$$\hat{\epsilon_t} = X_t - \hat{X}_t$$

This time, instead of modeling the original series, the differenced series is used.

```
MA1_Mishkin_diff <- arima(diff(Mishkin), order = c(0, 0, 1))
fitted_vals <- diff(Mishkin) - residuals(MA1_Mishkin_diff)
MA_forecast <- predict(MA1_Mishkin_diff, n.ahead = 12)$pred
MA_forecast_se <- predict(MA1_Mishkin_diff, n.ahead = 12)$se
plot(diff(Mishkin), main = "Differenced US Monthly Inflation Rates",
     xlim = c(1980, 1991),
     ylab = "Inflation Rate (Differenced)")
lines(fitted_vals, type = "l",
      col = "red",
      lty = 2)
points(MA_forecast, type = "l", col = "blue")
points(MA_forecast - 1.96 * MA_forecast_se,
       type = "l", col = "blue", lty = 2)
points(MA_forecast + 1.96 * MA_forecast_se,
       type = "l", col = "blue", lty = 2)
legend("topleft",
       legend = c("X", "Xhat", "Forecasted Values", "95 percent CI"),
       col = c("black", "red", "blue", "blue"),
       lty = c(1, 2, 1, 2),
       cex = 0.8)
```



Note that the prediction line goes flat after the first prediction point. This is because the MA(1) model can only predict 1 step forward.

MA(1) is just one simple example of the MA(q) family. A process is MA(q) if:

$$\begin{aligned} X_t &= \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \\ &= \mu + (1 + \theta_1 B + \cdots + \theta_q B^q)\epsilon_t \end{aligned}$$

where the error terms are once again white noise.

ARMA Models

Autoregressive Moving Average (ARMA) models describe stochastic processes in terms of two polynomials: one for the autoregression (AR) and one for the moving average (MA).

A process $\{X_t\}$ is an ARMA(p, q) process iff:

1. It is stationary.
2. It satisfies the following equation:

$$X_t - \phi_1 X_{t-1} - \cdots - \phi_p X_{t-p} = \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}$$

where $\epsilon_t \sim \text{WN}(0, \sigma^2)$.

This equation is often abbreviated using the backshift operator:

$$\phi(B)X_t = \theta(B)\epsilon_t$$

where

$$\phi(z) = 1 - \phi_1 z - \dots \phi_p z^p$$

and

$$\theta(z) = 1 + \theta_1 z + \cdots + \theta_q z^q$$

In `R`, modelling time series using ARMA models is as simple as using the `auto.arima` function.

Why does stationarity matter?

The Wold Decomposition states that any stationary time series can be written as a linear combination of white noise, which is exactly what an ARMA process is. Thus, ARMA models can model stationary data effectively.

Non-Stationary processes can be transformed into stationary processes. Typically, this is done through a combination of logarithmic transformations and differencing. Once the process is stationary, ARMA models can be used, since ARMA processes are stationary.

Interpolation Algorithms

The models discussed in the previous section work very well, and are easy to use. Unfortunately, these methods are only usable when the data are "nice" – that is, the time series must have a single observation at regularly spaced intervals. In reality, it is possible to obtain data which have missing points. This is problematic, because it breaks the assumption that the observations are evenly spaced.

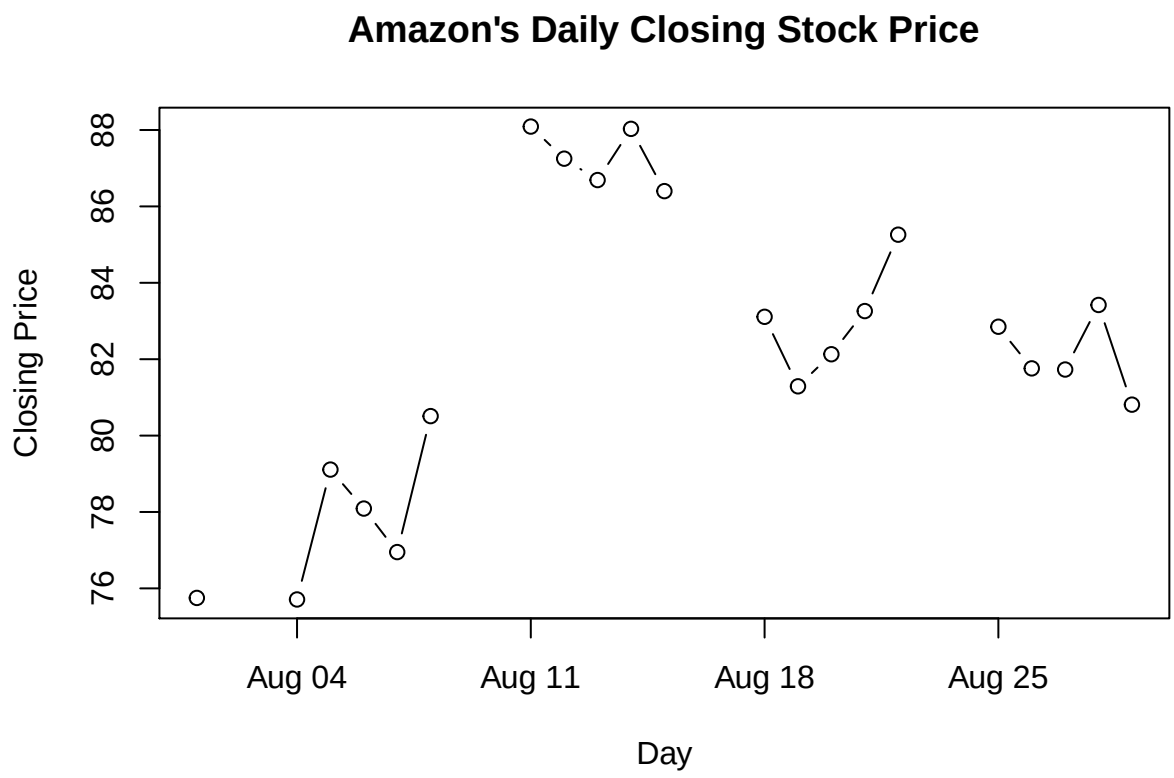
Missing observations in data can occur for a number of reasons. For example, consider closing stock prices:

```
library(quantmod)
getSymbols("AMZN", from = "2008-08-01", to = "2008-09-01")
```

```
## [1] "AMZN"
```

```
dates <- seq(as.Date("2008-08-01"),
             as.Date("2008-08-29"),
             by = "days")
AMZNdf <- data.frame(dates = dates,
                     close = rep(NA, length(dates)))
AMZN <- as.data.frame(AMZN)
```

```
for (i in 1:length(AMZNdf$dates)) {
  index <- which(rownames(AMZN) == AMZNdf$dates[i])
  if (length(index) != 0) {
    AMZNdf$close[i] <- AMZN$AMZN.Close[index]
  }
}
plot(x = AMZNdf$dates, y = AMZNdf$close,
     main = "Amazon's Daily Closing Stock Price",
     xlab = "Day",
     ylab = "Closing Price",
     type = "b")
```



In this case, information is only available for business days, so there will be missing observations on weekends and holidays.

This is just one possible reason why a time series may have missing values. As mentioned in the background section, another common reason that this occurs is because of equipment failure. It is also possible to have time series in which some observations need to be removed due to transcription error or unreliable results.

When there are missing values in a time series, these points must be *interpolated* or *imputed*. This is a necessary step in order to model the series using the models discussed previously.

In this section, some interpolation algorithms will be presented. The information in the following section comes mainly from `R` documentation and [3], with other sources noted as used.

For each of the following algorithms, a toy example will be used in order to demonstrate how the algorithms would work on a small scale. In a later section, all of these algorithms will be tested on real-world, large scale datasets. For now, though, consider the following example time series with missing points (denoted by NA):

$$\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$$

It is important to note that, while some of these algorithms are very simplistic, even the simplest of algorithms (Nearest-Neighbor) are present in the literature as recently as 2014[3]. This demonstrates that this is still a very active field of research, and that there is much work to be done in this area.

Nearest Neighbor Interpolation

The simplest time series interpolation method is known as Nearest-Neighbor Interpolation (this algorithm should not be confused with k-nearest-neighbors methods). In this method, the value of the nearest neighboring non-empty data point is assigned to the observation with the missing value. Formally, if X_i is a missing value, then:

$$X_i = \begin{cases} x_A & \text{if } i < \frac{a+b}{2} \\ x_B & \text{else} \end{cases}$$

where a is the index of x_A, b is the index of x_B, and x_A and x_B are the left and right neighbors respectively.

For example, if $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$, the Nearest-Neighbor Interpolation algorithm would complete the time series as following: $\{X_t\} = \{4, 7, 9, 9, 6, 6, 3, 5, 12, 12, 15\}$

Here is the implementation in R for the Nearest-Neighbor Interpolation algorithm.

```
nearestNeighbor <- function(x) {
  stopifnot(is.ts(x))

  findNearestNeighbors <- function(x, i) {
    leftValid <- FALSE
    rightValid <- FALSE
    numItLeft <- 1
    numItRight <- 1
    while (!leftValid) {
      leftNeighbor <- x[i - numItLeft]
      if (!is.na(leftNeighbor)) {
        leftValid <- TRUE
        leftNeighbor <- i - numItLeft
      }
      numItLeft <- numItLeft + 1
    }
    while (!rightValid) {
      rightNeighbor <- x[i + numItRight]
```

```
      if (!is.na(rightNeighbor)) {
        rightValid <- TRUE
        rightNeighbor <- i + numItRight
      }
      numItRight <- numItRight + 1
    }
    return(c(leftNeighbor, rightNeighbor))
  }

  for (i in 1:length(x)) {
    if (is.na(x[i])) {
      nearestNeighborsIndices <- findNearestNeighbors(x, i)
      a <- nearestNeighborsIndices[1]
      b <- nearestNeighborsIndices[2]
      if (i < ((a + b) / 2)) {
        x[i] <- x[a]
      } else {
        x[i] <- x[b]
      }
    }
  }
  return(x)
}
```

To ensure that the function works as proposed, consider the example presented earlier:

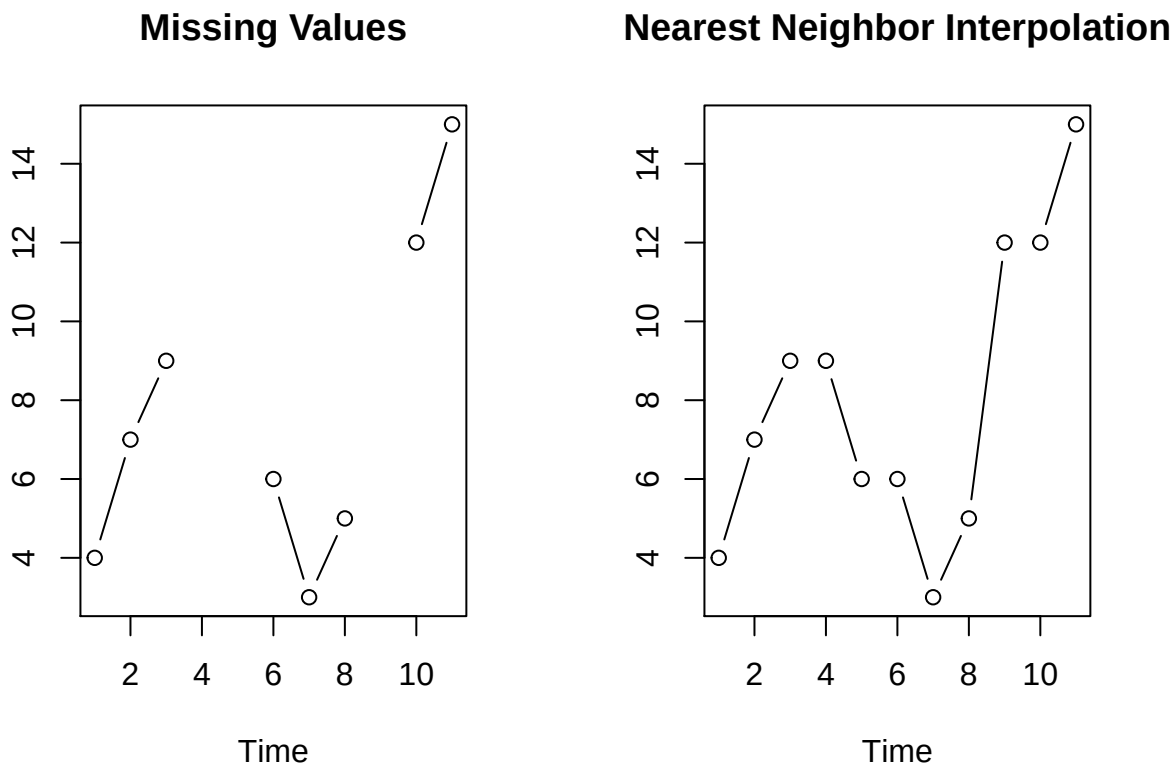
```
x <- as.ts(c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15))
nearestNeighbor(x)
```

```
## Time Series:
## Start = 1
## End = 11
## Frequency = 1
##  [1]  4  7  9  9  6  6  3  5 12 12 15
```

The function produced the desired result.

As this method is the simplest, it does not perform well. This will be demonstrated in later sections. This can also be represented visually:

```
par(mfrow = c(1, 2))
x <- as.ts(c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15))
X <- nearestNeighbor(x)
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Nearest Neighbor Interpolation", type = "b", ylab = "")
```



The nearest neighbor interpolation algorithm produces very jumpy, unrealistic results.

While this algorithm may seem very simple, all of these algorithms are relatively "new", as these sorts of algorithms were not viable on a large scale before the invention of computers.

Linear Interpolation

Linear interpolation searches for a straight line which passes through the end points x_A and x_B (note that these points are the same as the ones defined in the nearest neighbor interpolation algorithm).

$$X_i = \frac{x_A - x_B}{a - b}(i - b) + x_B$$

This can be implemented in R:

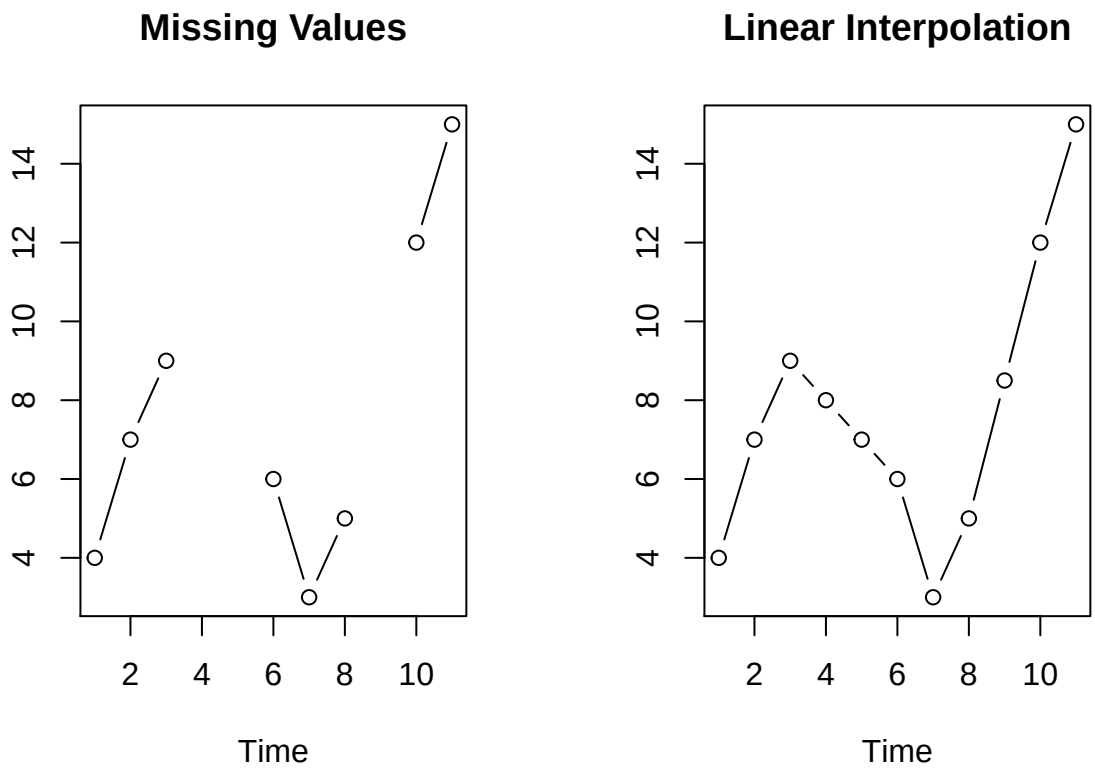
```
linearInterpolator <- function(x) {
  stopifnot(is.ts(x))

  findNearestNeighbors <- function(x, i) {
    leftValid <- FALSE
    rightValid <- FALSE
    numItLeft <- 1
    numItRight <- 1
```

```
    while (!leftValid) {
      leftNeighbor <- x[i - numItLeft]
      if (!is.na(leftNeighbor)) {
        leftValid <- TRUE
        leftNeighbor <- i - numItLeft
      }
      numItLeft <- numItLeft + 1
    }
    while (!rightValid) {
      rightNeighbor <- x[i + numItRight]
      if (!is.na(rightNeighbor)) {
        rightValid <- TRUE
        rightNeighbor <- i + numItRight
      }
      numItRight <- numItRight + 1
    }
    return(c(leftNeighbor, rightNeighbor))
  }

  for (i in 1:length(x)) {
    if (is.na(x[i])) {
      nearestNeighborsIndices <- findNearestNeighbors(x, i)
      a <- nearestNeighborsIndices[1]
      b <- nearestNeighborsIndices[2]
      x[i] <- (x[a] - x[b]) / (a - b) * (i - b) + x[b]
    }
  }
  return(x)
}
```

We can use the sample data from earlier to ensure that this function works properly:

If $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$, the Linear Interpolation Algorithm would complete the time series as following: $\{X_t\} = \{4, 7, 9, 8, 7, 6, 3, 5, 8.5, 12, 15\}$

```
par(mfrow = c(1, 2))
x <- as.ts(c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15))
X <- linearInterpolator(x)
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Linear Interpolation", type = "b", ylab = "")
```



So, the function works as desired.

Even with this toy example, it is easy to see that this algorithm would perform better than the Nearest-Neighbor algorithm on real, large-scale data.

The na.approx Family of Functions in R

In `R`, there are a number of time series interpolation algorithms which have been implemented already. For example, the `zoo` package contains a family of functions, `na.approx`, for this purpose. The simplest of these is the `na.approx` function itself, which uses linear interpolation to fill missing values. This is indeed the exact same linear interpolation algorithm which was discussed above, except that the `na.approx` function also takes additional arguments (for example, the user can specify the `maxgap` argument, which represents the maximum number of consecutive NAs to fill. Any gaps which are larger than the value of `maxgap` will be left unchanged).

To demonstrate that they are indeed the same, consider our example from before:

```
library(zoo)
# restating for convenience
x <- as.ts(x <- as.ts(c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)))
linearInterpolator(x)
```

```
## Time Series:
```

```
## Start = 1
## End = 11
## Frequency = 1
##  [1]  4.0  7.0  9.0  8.0  7.0  6.0  3.0  5.0  8.5 12.0 15.0
```

```
na.approx(x)
```

```
## Time Series:
## Start = 1
## End = 11
## Frequency = 1
##  [1]  4.0  7.0  9.0  8.0  7.0  6.0  3.0  5.0  8.5 12.0 15.0
```

Thus, the `na.approx` method is the exact same as the one presented in the subsection above.

It might seem like this is harmless, but a rather shocking fact is that a Google search for "Time Series Interpolation in R" yields a top result as a Stack Overflow question in which the person is told to use the `na.approx` function to interpolate the missing points in their series. It is alarming that such a simplistic method is being used on such a large scale.

Another member of the `na.approx` family of functions from the `zoo` package in R is the `na.spline` function. This function uses cubic spline interpolation[7].

Cubic splines, in general, are interpolating polynomials. Given a set of $n+1$ data points at regularly spaced intervals, a cubic spline is denoted $S(x)$ and is a continuous cubic function connecting the endpoints. More formally,

Given $n+1$ points, (x_i, y_i), let $a = x_0 < x_1 < \cdots < x_n = b$. Then the cubic spline $S(x)$ satisfies:

1. $S(x) \in C^2[a, b]$
2. On each subinterval $([x_{i-1}, x_i]),$ the cubic spline is a polynomial of degree 3
3. $S(x_i) = y_i \quad \forall i = 1, .., n$

Then we assume that the cubic spline is a piecewise defined function,

$$S(x) = \begin{cases} C_1(x), & x_0 \leq x \leq x_1 \\ \cdots & \\ C_i(x), & x_{i-1} < x \leq x_i \\ \cdots & \\ C_n(x), & x_{n-1} < x \leq x_n \end{cases}$$

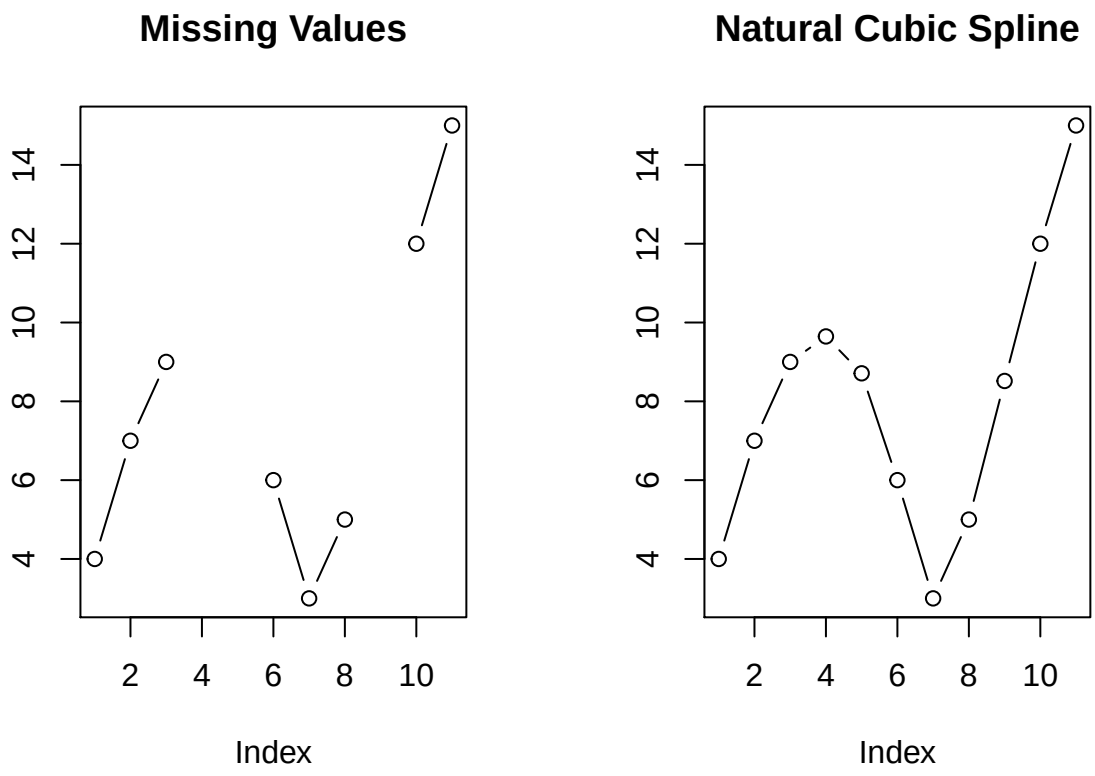
such that each $C_i = a_i + b_i x + c_i x^2 + d_i x^3$ where the coefficient in front of the cubic term cannot be zero, and $i = 1, ..., n$.

Obviously, we need a way in which to determine the coefficients a, b, c, d for each subinterval of the spline. The way in which this is done depends on the kind of cubic spline that is being used. For example, one kind of cubic spline is a *natural cubic spline*, where

$$C_i(x_{i-1}) = y_{i-1} \text{ and } C_i(x_i) = y_i,\ i = 1, ..., n$$
$$C_i^{'}(x_i) = C_{i+1}^{'}(x_i),\ i = 1, ..., n-1$$
$$C_i^{''}(x_i) = C_{i+1}^{''}(x_i),\ i = 1, ..., n-1$$
$$C_1^{''}(x_0) = C_n^{''}(x_n) = 0$$

To use this method of interpolation on our series, set the `method` argument to `natural` when using the `na.spline` function from the `zoo` package:

```
library(zoo)
par(mfrow = c(1, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X <- na.spline(x)
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Natural Cubic Spline", type = "b", ylab = "")
```



Already, a huge improvement is seen, compared to the nearest neighbor or linear interpolation methods – the data appear to flow much more smoothly, more like they would in real life.

Other `method` arguments that the `na.spline` function can take are:

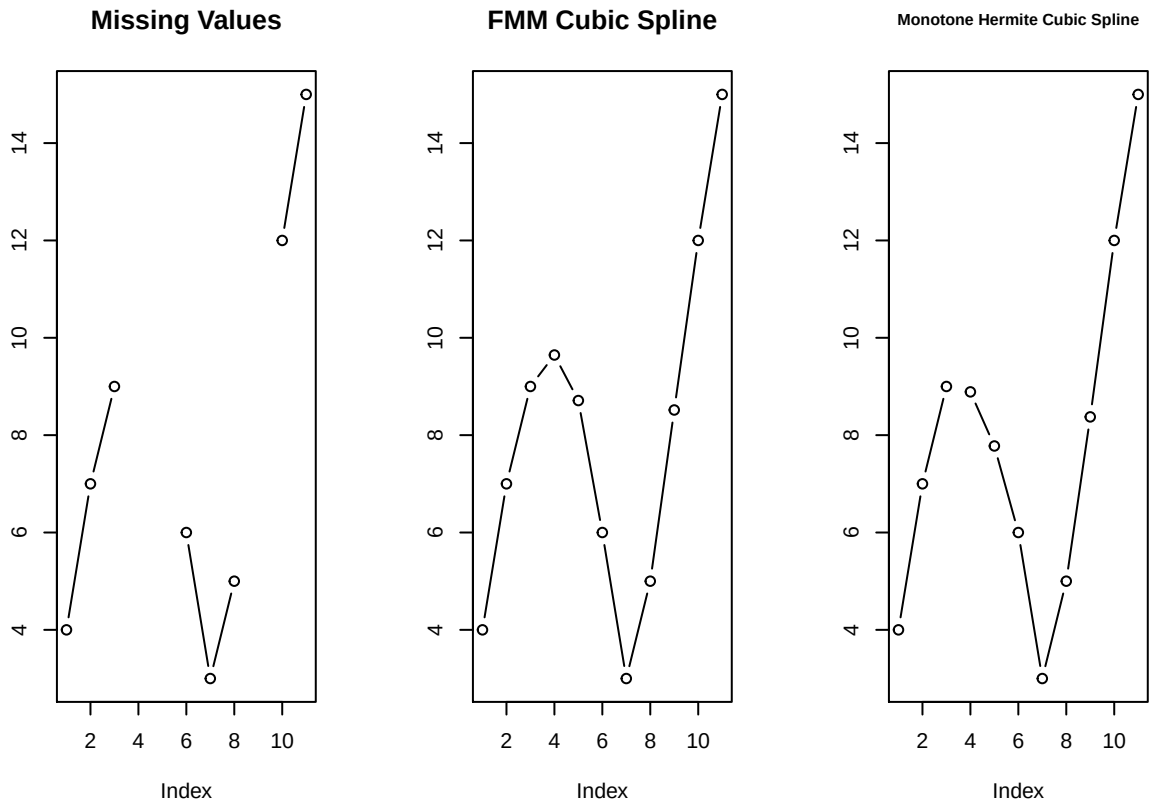
- `"fmm"` (an exact cubic spline is fitted through the four points at each end of the data, and this is used to determine end conditions; method proposed by Forsythe, Malcolm, and Moler in 1977)

- `"periodic"` (if the exact function $f(x)$ is a periodic function with period $x_n - x_0$, then its cubic spline should be a periodic function of the same period. This is not often the case when we are working with real-world time series data, since we do not know the *exact* period of our data and it may be unwise to impose these kinds of assumptions on our data)
- `"monoH.FC"` (computes a monotone Hermite spline, as described by Fritsch and Carlson)
- `"hyman"` (computes a monotone cubic spline using Hyman filtering of a `method = "fmm"` input. Note that this method can only be used on inputs which are monotone increasing or monotone decreasing, which is very unlikely in time series data, so we will not explore this method)

Note that these cubic spline interpolation methods are also presented in [3].

The method presented by Forsythe, Malcolm, and Moler, as well as the method presented by Fritsch and Carlson can be used on our sample series from earlier.

```
library(zoo)
par(mfrow = c(1, 3))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X1 <- na.spline(x, method = "fmm")
X2 <- na.spline(x, method = "monoH.FC")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X1, main = "FMM Cubic Spline", type = "b", ylab = "")
plot(X2, main = "Monotone Hermite Cubic Spline", type = "b", ylab = "", cex.main = 0.8)
```



Just from first glance, it can be seen that the Hermite spline allowed for smoother transitions between

imputed values and non-missing values. This will be explored further in later sections.

It is important to note that cubic splines are a method regularly taught in numerical analysis courses – in fact, at Trent, natural cubic splines are touched briefly in Numerical Methods.

The na.interp Function in R

The `forecast` package is a package which includes many useful functions for the analysis of time series data. This package includes a function called `na.interp` for the interpolation of NA (missing) values in time series objects. Surprisingly, it simply uses linear interpolation to do so. Indeed, using our example from before yields the exact same results as the linear interpolator:

```
library(forecast)
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
na.interp(x)
```

```
## Time Series:
## Start = 1
## End = 11
## Frequency = 1
##  [1]  4.0  7.0  9.0  8.0  7.0  6.0  3.0  5.0  8.5 12.0 15.0
```

The imputeTS package in R

A package called `imputeTS` was developed in R, which, according to its developers, "specializes on univariate time series imputation". As will be shown later, this is quite an overstatement, as the methods used in the `imputeTS` series are once again, quite simplistic.

The na.interpolation Function

This package includes a function called `na.interpolation` for the interpolation of missing values in time series data. This function takes an argument called `option`, which has the following possible values:

- `"linear"` (using this value will just compute linear interpolation as described earlier, by calling the `na.approx` function)
- `"spline"` (using this value will just compute cubic spline interpolation as described previously, by calling the `na.spline` function)
- `"stine"` (using this value will compute Stineman interpolation, by calling the `stinterp` function from the `stinepack` package in R)

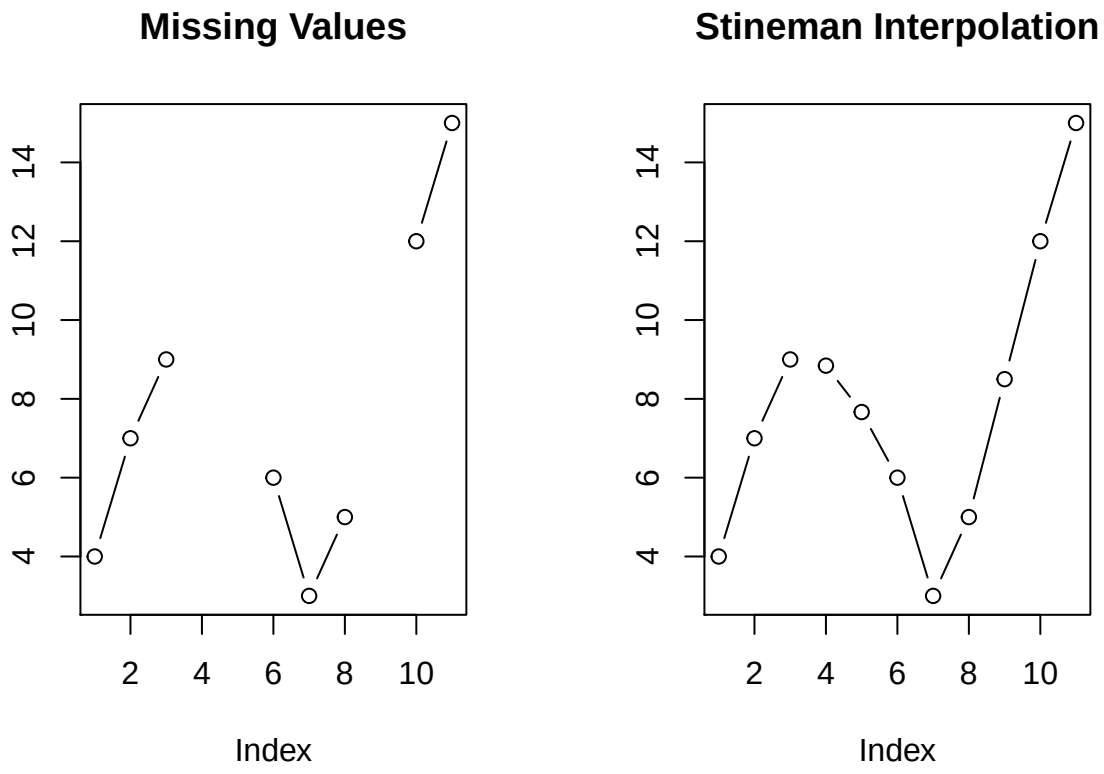
It is interesting to see that, once again, linear interpolation is such a popular method. In fact, the default value of the `option` argument for the `na.interpolation` function is `"linear"`, which means that the unsuspecting user will have their time series interpolated using this extremely simplistic method.

The only new method presented in the function above is Stineman interpolation, which was an algorithm developed by (you guessed it) Stineman in 1980. Locating the original paper or any information on the original paper has proven challenging, but examining the source code directly for the `stinterp` function

reveals that the Stineman interpolation algorithm is again nothing more than a simple polynomial interpolator.

Using this method on our previous example series yields:

```
library(imputeTS)
par(mfrow = c(1, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X <- na.interpolation(x, option = "stine")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Stineman Interpolation", type = "b", ylab = "")
```



The na.kalman Function

The `na.kalman` function in the `imputeTS` package uses Kalman Smoothing to impute (interpolate) missing values. In order to do this, the function requires a model for which the Kalman Smoothing is performed. Thus, the `na.kalman` function takes one of the following two options for the `model` argument:

- `"auto.arima"`(an ARIMA model is used, by calling the `auto.arima` function, seen earlier)
- `"StructTS"` (a structural model, which is fitted by using the method of maximum likelihood; the `StructTS` function is called)

The Kalman filter is also sometimes known as the Linear Quadratic Estimation (LQE) algorithm. It is a two-step algorithm: Step 1 is a prediction step, wherein the current state variables are estimated, along with

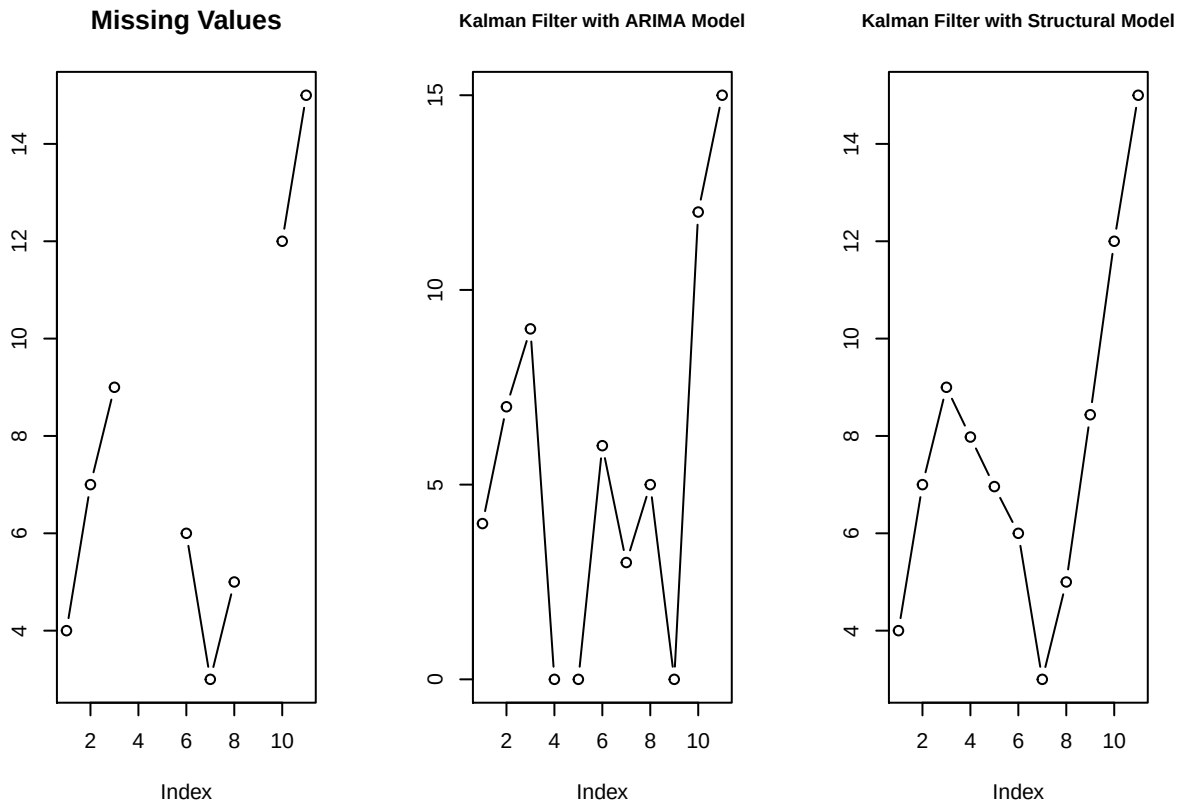
their uncertainties. Step 2 updates these estimates, by using a weighted average. The outcome of the next measurement is observed, and the estimates with higher certainty are given higher weights. This is a recursive process.

An interesting history piece is that one of the very first applications of the Kalman filter was in the Apollo navigation computer. It was applied to the nonlinear problem of trajectory estimation.

Kalman filters are used widely in other scientific fields. For example, they can be used for robotic motion planning and control, as they can be used to model the way that the central nervous system controls our movement[6].

Applying the `na.kalman` function to our example series from before:

```
par(mfrow = c(1, 3))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X1 <- na.kalman(x, model = "auto.arima")
X2 <- na.kalman(x, model = "StructTS")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X1, main = "Kalman Filter with ARIMA Model", type = "b", ylab = "",
     cex.main = 0.9)
plot(X2, main = "Kalman Filter with Structural Model", type = "b", ylab = "",
     cex.main = 0.9)
```



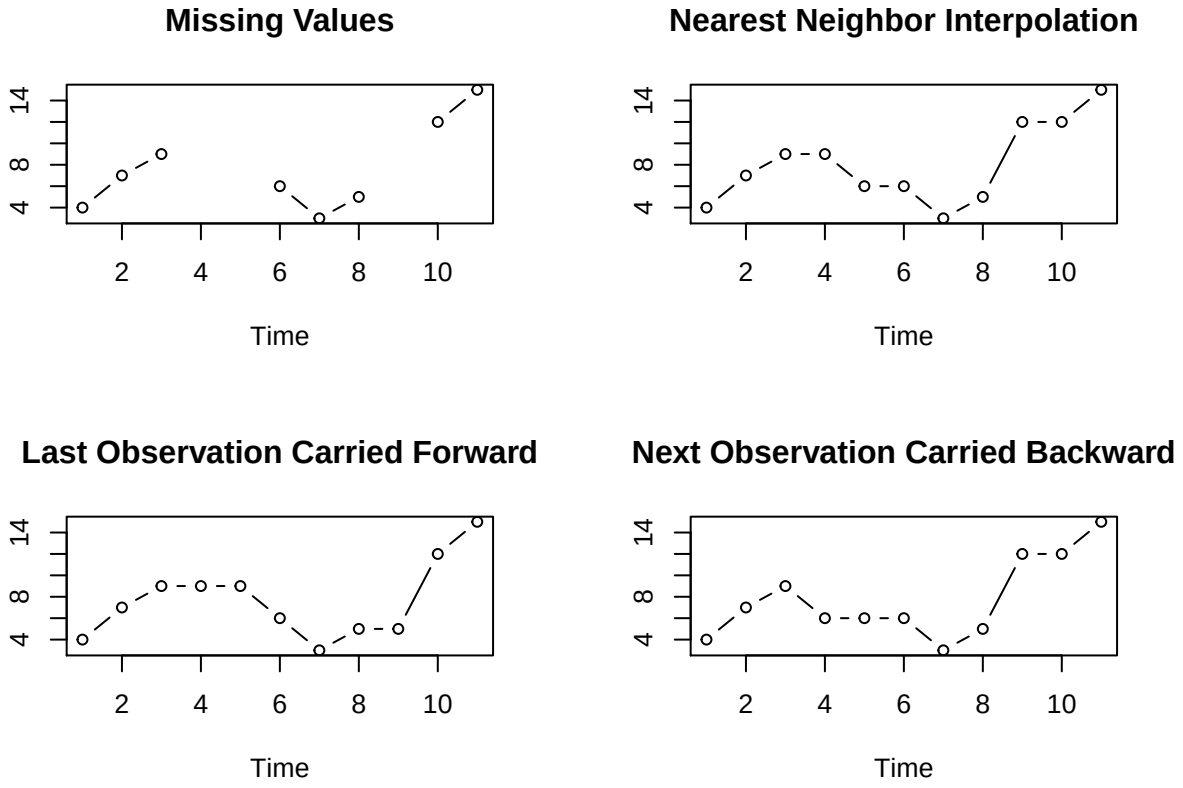
The na.locf Function

The `na.locf` function in the `imputeTS` package stands for Last Observation Carried Forward. It will interpolate the missing values using either Last Observation Carried Forward (using `option = "locf"`) or Next Observation Carried Backward (using `option = "nocb"`). This is exactly what it sounds like – Nearest Neighbor interpolation, but worse. Using this function is actually worse than using the Nearest Neighbor interpolation algorithm, as the `na.locf` function forces the user to choose either a left or right neighbor, rather than choosing the **nearest** neighbor.

The reader may (and should) be shocked to learn of the existence of this function, given that it provides horrid estimates. It should also be noted that a similar function is also included in the `zoo` package. Thus, in not just one, but **two** packages which are heavily used for the analysis of time series, an extremely simple algorithm is implemented and is regularly used.

These functions can be compared on our example series from earlier:

```
par(mfrow = c(2, 2))
x <- as.ts(c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15))
X1 <- nearestNeighbor(x)
X2 <- imputeTS::na.locf(x, option = "locf")
X3 <- imputeTS::na.locf(x, option = "nocb")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X1, main = "Nearest Neighbor Interpolation", type = "b", ylab = "")
plot(X2, main = "Last Observation Carried Forward", type = "b", ylab = "")
plot(X3, main = "Next Observation Carried Backward", type = "b", ylab = "")
```



There should be no reason why anyone would ever use LOCF or NOCB. The Nearest Neighbor algorithm is superior, and even it shouldn't be used.

The na.ma Function

The `na.ma` function from the `imputeTS` package uses a weighted moving average to replace missing values in a time series. Essentially, the average of the values surrounding the missing value is used.

One of the arguments that this function takes is the width of the moving average window, set by the argument `k`. For example, setting `k = 2` means that 2 observations on the left of the missing value and 2 observations on the right of the missing value will be taken into account. An important note is that if there is a gap in the series which is larger than the user-defined width of the moving average window, then the function will automatically increase the window until there are at least 2 non-missing values present in the window.

There are 3 different types of weighting that can be used, set by the `weighting` argument of the function:

- `"simple"` (a Simple Moving Average, or SMA)
- `"linear"` (a Linear Weighted Moving Average, or LWMA)
- `"exponential"` (an Exponential Weighted Moving Average, or EWMA)

It is useful to explain these various weightings in more detail. Let i be the indices of the values in the moving average window of width k surrounding the missing value in the series. Let j represent the index of the missing value in the moving average window of width k. Then the Simple Moving Average is given by:

$$X_j = \frac{1}{2k} \sum_{i=j-k}^{j+k} x_i \qquad i \neq j$$

Consider the example from before: $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$.

Let $k = 2$. If we want to interpolate the third missing value, then we have $j = 9$, so $i = 7, 8, 10, 11$. Thus, the missing value would be interpolated as $\frac{1}{4}(3 + 5 + 12 + 15) = 8.75$. This can be demonstrated by using the function:

```
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
na.ma(x, k = 2, weighting = "simple")[9]
```

```
## [1] 8.75
```

The Linear Weighted Moving Average is similar, but points which are closer to the missing value are weighted more heavily than points that are further away. Let W_i denote the weight of the i^{th} data point. Then,

$$W_i = \frac{1}{|i - j| + 1} \qquad i \neq j,\ i = j - k, ..., j + k$$

The missing value is then interpolated as:

$$X_j = \sum_{i=j-k}^{j+k} \left(\frac{x_i \cdot W_i}{\sum_{i=j-k}^{j+k} W_i} \right) \qquad i \neq j$$

Consider again the example used earlier, where we are interested in interpolating the third missing value from the series, using a window of $k = 2$ (recall: $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$). Then we have $j = 9$, $i = 7, 8, 10, 11$, and we have weights $\frac{1}{3}, \frac{1}{2}, \frac{1}{2}, \frac{1}{3}$. The missing value would be interpolated as:

$$\begin{aligned}
X_9 &= \sum_{i=4-2}^{4+2} \left(\frac{x_i \cdot W_i}{\sum_{i=4-2}^{4+2} W_i} \right) \qquad i \neq j \\
X_9 &= \frac{3 \cdot \frac{1}{3} + 5 \cdot \frac{1}{2} + 12 \cdot \frac{1}{2} + 15 \cdot \frac{1}{3}}{(\frac{1}{3} + \frac{1}{2} + \frac{1}{2} + \frac{1}{3})} \\
X_9 &= 8.7
\end{aligned}$$

This can be verified by using the `na.ma` function:

```
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
na.ma(x, k = 2, weighting = "linear")[9]
```

```
## [1] 8.7
```

Finally, the Exponential Weighted Moving Average is calculated in the same fashion as the Linear Weighted Moving Average, except that the weights are instead given by:

$$W_i = \frac{1}{2^{|i-j|}} \qquad i \neq j,\ i = j - k, ..., j + k$$

Thus, with the same example we used for the other two moving averages (recall: $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$), we have $j = 9$, $i = 7, 8, 10, 11$.

The weights are $\frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{4}$. The missing value would be interpolated as:

$$X_9 = \sum_{i=4-2}^{4+2} \left(\frac{x_i \cdot W_i}{\sum_{i=4-2}^{4+2} W_i} \right) \qquad i \neq j$$

$$X_9 = \frac{3 \cdot \frac{1}{4} + 5 \cdot \frac{1}{2} + 12 \cdot \frac{1}{2} + 15 \cdot \frac{1}{4}}{(\frac{1}{4} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4})}$$

$$X_9 = 8.\bar{6}$$

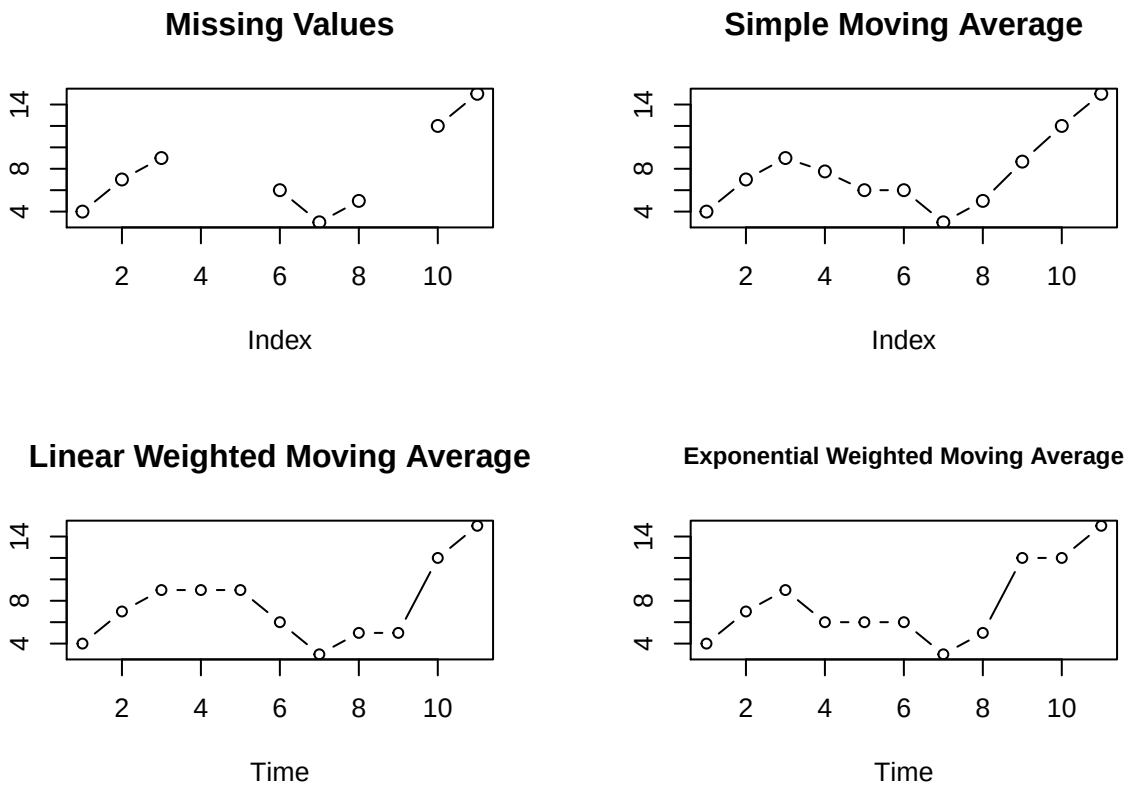
This can be checked by using the function:

```
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
na.ma(x, k = 2, weighting = "exponential")[9]
```

```
## [1] 8.666667
```

Now that the three types of moving averages in the `na.ma` function have been explained, it will be helpful to visualize all three at once.

```
library(imputeTS)
par(mfrow = c(2, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X1 <- na.ma(x, k = 2, weighting = "simple")
X1 <- na.ma(x, k = 2, weighting = "linear")
X1 <- na.ma(x, k = 2, weighting = "exponential")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X1, main = "Simple Moving Average", type = "b", ylab = "")
plot(X2, main = "Linear Weighted Moving Average", type = "b", ylab = "")
plot(X3, main = "Exponential Weighted Moving Average", type = "b", ylab = "",
     cex.main = 0.9)
```



The example that was used above was so simplistic that it resulted in very similar interpolated values for all three moving averages. This is only because of the simplistic example, and this will not always occur.

The na.mean Function

When the reader first hears the title of this function, they are likely to assume that this method will be similar to that of the moving average. The reader would be very wrong in this assumption, and in fact, the `na.mean` function provides far less sophisticated estimates than a moving average process. The `na.mean` function has an argument `option` which takes one of the following three values:

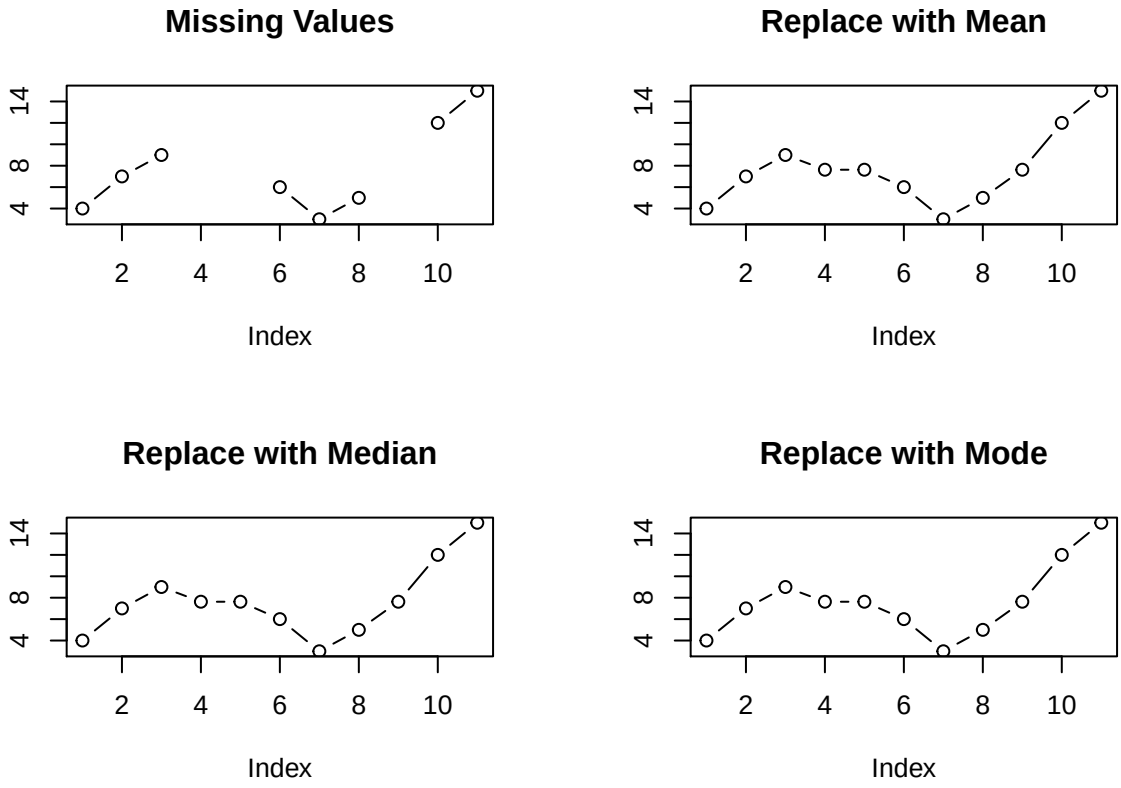
- `"mean"` - take the mean of the entire time series and replace all missing values with this value
- `"median"` - take the median of the entire time series and replace all missing values with this value
- `"mode"` - take the mode of the entire time series and replace all missing values with this value

This method is *terrible* – it assumes that all missing values should be replaced with the same value, and completely disregards any kinds of patterns / trends that might be present in the data. This algorithm is, in most cases, even worse than the LOCF or NOCB.

Nonetheless, recall the example time series from before: $\{X_t\} = \{4, 7, 9, \text{NA}, \text{NA}, 6, 3, 5, \text{NA}, 12, 15\}$. Using the `na.mean` function on the above series for all three options gives:

```
par(mfrow = c(2, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X1 <- na.mean(x, option = "mean")
```

```
X2 <- na.mean(x, option = "median")
X3 <- na.mean(x, option = "mode")
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X1, main = "Replace with Mean", type = "b", ylab = "")
plot(X1, main = "Replace with Median", type = "b", ylab = "")
plot(X1, main = "Replace with Mode", type = "b", ylab = "")
```

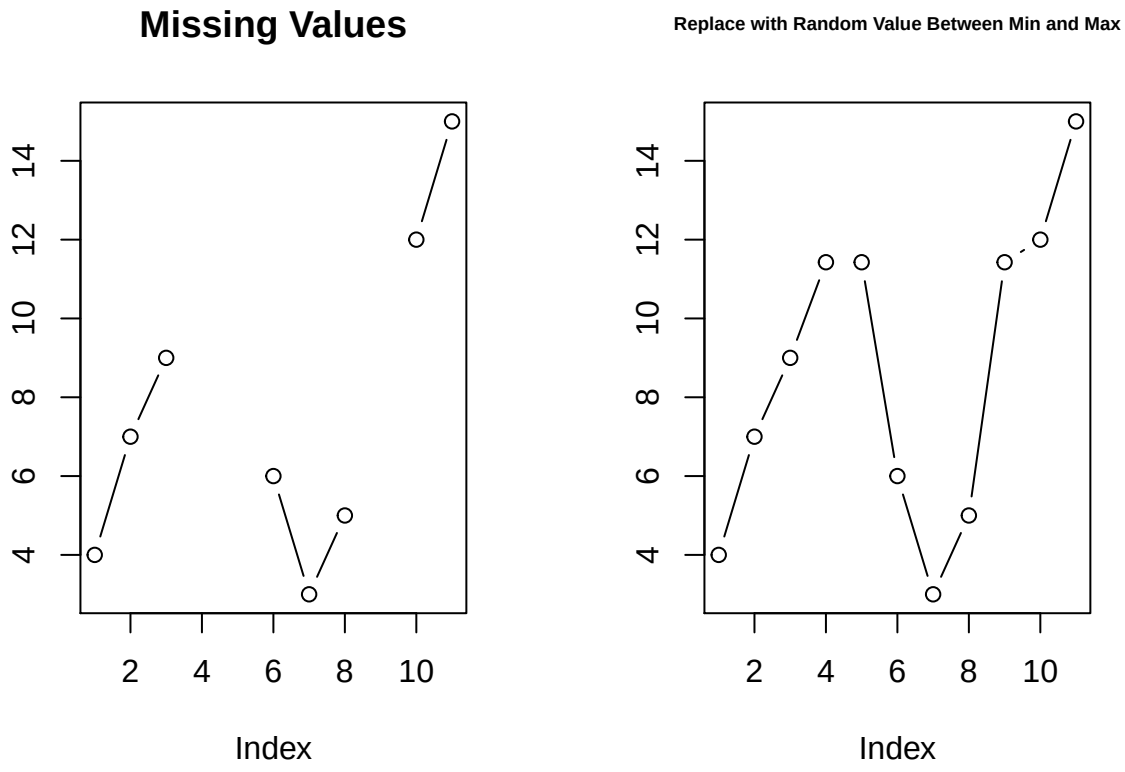


The `option = "mode"` performs particularly poorly, as the algorithm falls apart if all values in the series are observed the same number of times, as they were in the example. In this case, the function assigns the minimum value of the series as the "mode", even though all values in the series were observed the same number of times as the minimum value of the series, and this choice is extremely arbitrary. This function should not be used, ever, as an intelligent Kindergartener would probably be able to guess the missing value with the same or higher degree of precision than this function could.

The na.random Function

Just when it seemed that the methods presented above could not get any crazier, the `na.random` function from the `imputeTS` package rears its ugly head. This function replaces missing values with a randomly generated value between user-defined bounds. If the user does not specify bounds, the function will, by default, generate random points between the minimum and maximum values observed in the given time series. For example:

```
par(mfrow = c(1, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X <- na.random(x)
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Replace with Random Value Between Min and Max", type = "b", ylab = "",
     cex.main = 0.6)
```



This is problematic for so many reasons. To begin with, unless the user sets a seed, they will get different results every single time they use the function. If the user forgets to set a seed, they may have to redo some or all of their analysis. Furthermore, picking a random value completely ignores any trends that may be occuring in the data, especially since the indices of the random values generated are coming from a uniform distribution, disregarding any kind of structure that may be present and useful for imputing the missing values. Additionally, this function does not take into account the fact that certain values in a time series might be unreachable or might be extremely unlikely to occur. For example, it is possible that observations must be restricted to integers (or that it is highly unlikely to obtain a non-integer observation). The function also replaces all missing values with the same random number, so there might be weird jumps in the interpolated time series that actually make the series harder to analyze overall.

Other functions in the imputeTS package

The `imputeTS` package also includes another two functions which are practically useless – one which simply removes missing values, and one which replaces missing values with a user-defined value. These functions

really serve no purpose, because anyone who is looking to interpolate missing values in their time series data is obviously not interested in throwing points away (and destroying any seasonality their data might have in the process) or at making their own guesses (otherwise they would not be using the `imputeTS` package at all).

All in all, the `imputeTS` package lacks functionality, and actually provides a disservice to the statistical community, by falsely claiming that it "specializes" in time series imputation.

Hybrid Wiener Interpolator

The final interpolation algorithm that will be explored in this Honours Project is the Hybrid Wiener Interpolator, available in the `tsinterp` package[1] through the `interpolate` function.

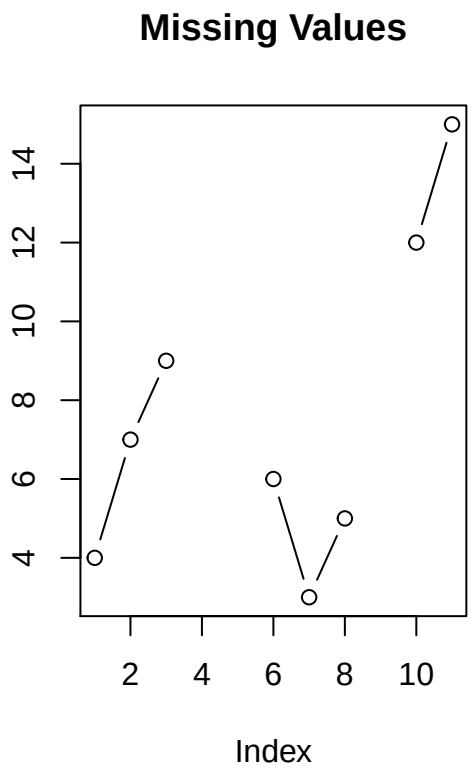
The specifics of this interpolator are far beyond the scope of this course or the author's knowledge base, but a brief overview of the algorithm will be provided. In essence, the algorithm behind this interpolator can be thought of as an "EM algorithm". The first step is an *expectation* step, where the autocovariance function of the process is estimated. Step two of the algorithm is a *maximization* step, where the missing values are estimated, using the resulting ACVF estimate from the first step[2].

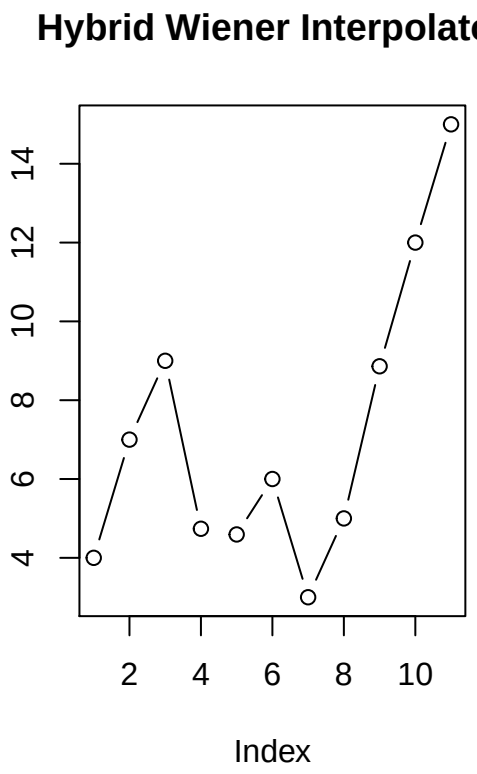
The `interpolate` function of the `tsinterp` package requires both the time series itself, as well as the indices of the missing values in the series. This is noticeably different from other functions presented in this project, which locate the gaps for the user. The gaps can be found very easily by using `which(is.na(x))`.

```
library(tsinterp)
par(mfrow = c(1, 2))
x <- c(4, 7, 9, NA, NA, 6, 3, 5, NA, 12, 15)
X <- interpolate(x, which(is.na(x)))[[1]]
```

```
## Iteration 0:  N/A  (.)
## Iteration 1: 0.732008
## Iteration 2: 0.280483
## Iteration 3: 0.107713
## Iteration 4: 0.041326
## Iteration 5: 0.015844
## Iteration 6: 0.006073
## Iteration 7: 0.002327
## Iteration 8: 0.000892
```

```
plot(x, main = "Missing Values", type = "b", ylab = "")
plot(X, main = "Hybrid Wiener Interpolator", type = "b", ylab = "")
```





Evaluating Performance of Algorithms

Once an interpolation algorithm has been used on a time series with gaps, the performance of the algorithm must be evaluated.

One of the existing problems in this field of research is that the criteria used for evaluating the performance of interpolation algorithms has been extremely inconsistent[3]. This may be due to personal opinions of authors, or perhaps it is because authors choose to use whichever criteria will best support their research – "cherry picking". Another problem that has arisen is that authors in the literature seem to be using different names for the same criteria, perhaps out of personal preference or perhaps in an attempt to sound overly sophisticated. To combat these inconsistencies, it will be useful to have a function which will allow a user to input the original, unaltered and complete time series (let this be known as x), and the interpolated series after gaps have been imposed and filled (let the interpolated series be known as X). This function should then output a number of performance criteria.

In the following sections, performance criteria from Table 1 of *Interpolation in Time Series: An Introductive Overview of Existing Methods, Their Performance Criteria and Uncertainty Assessment* will be presented. They will be implented in R.

For the following methods, let $\{x_i\}_{i=1}^n$ be the true, original series, and let $\{X_i\}_{i=1}^n$ be the series *after* gaps were imposed and an interpolation algorithm was used to estimate the missing values.

Coefficient of Correlation

The Coefficient of Correlation comes with many different names. For example, sometimes it is referred to as correlation, Pearson's Correlation Coefficient, Pearson's Coefficient of Correlation, Pearson's r, point-biserial correlation, or just simply r. Correlation is a measure of how closely related two sets of numbers are, and its value can range from -1 to +1. -1 represents a perfectly negative correlation, and +1 represents a perfectly positive correlation. A correlation of 0 implies no correlation between the two sets of numbers.

In Table 1, the formula presented is

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(x_i - \bar{x})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \sum_{i=1}^{n}(x_i - \bar{x})^2}}$$

However, upon using this formula on real data, it was realized that this formula is actually incorrect. The formula *should* have been presented in Table 1 as:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(x_i - \bar{x})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}}$$

which is significantly different from the formula presented in [3]. The correct formula forces values to be between -1 and +1.

Squared Coefficent of Correlation

The squared coefficient of correlation is exactly what it sounds like – the square of the coefficent of correlation, which was presented above. For this reason, it is often referred to as r^2.

Since r can range from -1 to +1, r^2 is a positive number, which can range from 0 to 1. Conceptually, the value of r^2 represents how much of the variation of one set of numbers can be described by the other set of numbers. Higher values of r^2 represents a stronger relationship between the two sets of numbers.

Since Table 1 had the incorrect formula for r, it also had the incorrect formula for r^2. The correct formula is:

$$r^2 = \left(\frac{\sum_{i=1}^{n}(X_i - \bar{X})(x_i - \bar{x})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}} \right)^2$$

Absolute Differences

The absolute differences criterion simply sums up the differences between the true values and the interpolated values, and takes the absolute value of these differences at each step. Since there is no scale factor, this criterion can often be difficult to interpet. This criterion is defined as:

$$\sum_{i=1}^{n} |X_i - x_i|$$

Mean Bias Error

The Mean Bias Error (MBE) is sometimes referred to as simply the bias. It is the sum of the differences between the interpolated values and the true values, scaled by n:

$$\mathbf{MBE} = \frac{\sum_{i=1}^{n} X_i - x_i}{n}$$

Mean Error

The Mean Error (ME) is very similar to the MBE, except that it is the sum of the differences between the true values and the interpolated values, scaled by n. Thus, it has the same magnitude but opposite sign as the MBE.

$$\mathbf{ME} = \frac{\sum_{i=1}^{n} x_i - X_i}{n}$$

Mean Absolute Error

The Mean Absolute Error (MAE) is very similar to both the MBE and the ME. It just takes the absolute value of the differences. Thus, it can be more useful than calculating the other two (though the other two may be useful if for some reason, the sign is imporant). The MAE is defined as:

$$\mathbf{MAE} = \frac{\sum_{i=1}^{n} |X_i - x_i|}{n}$$

Mean Relative Error

The Mean Relative Error (MRE) takes the relative differences between the true values and the interpolated values.

The formula presented in Table 1 of [3] is not represented in a mathematically correct way. The correct formula is:

$$\mathbf{MRE} = \sum_{i=1}^{n} \frac{x_i - X_i}{x_i}$$

Note that this cannot be used if $x_i = 0$ for some i (In my code, I return NA if this is the case).

Mean Absolute Relative Error

The Mean Absolute Relative Error (MARE) is essentially a combination of ideas discussed above. It takes the same approach as the MRE, except that the absolute value is taken at each step, and the entire measurement is scaled by n:

$$\textbf{MARE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{x_i - X_i}{x_i}\right|$$

Again, this cannot be used if any values in the original time series were 0.

Mean Absolute Percentage Error

The Mean Absolute Percentage Error (MAPE) will be familiar to anyone who has taken physics courses at Trent, as this is the criterion that students are typically required to report when completing lab experiments.

It is essentially the MARE converted into a percentage, which makes interpretation much easier. The MAPE is defined as:

$$\textbf{MAPE} = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{x_i - X_i}{x_i}\right|$$

Again, this is only usable if $x_i \neq 0 \quad \forall i$.

Sum of Squared Errors

The Sum of Squared Errors (SSE) is also sometimes referred to as "quadratic differences". It is the sum of the squared differences (which can be thought of as "errors"). The SSE is defined as:

$$\textbf{SSE} = \sum_{i=1}^{n}(X_i - x_i)^2$$

Mean Square Error

The Mean Square Error (MSE) is the SSE, but scaled by n. It is also sometimes referred to in papers as the "standard errors". Mathematically, it is:

$$\textbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(X_i - x_i)^2$$

Root Mean Squares

The Root Mean Squares (RMS) is also referred to as the Root Mean Square Errors of Prediction (RMSEP), and is *almost* the square root of the MSE, except that the squared errors are scaled by the true observations in the time series:

$$\textbf{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{X_i - x_i}{x_i}\right)^2}$$

Again, this is only usable if $x_i \neq 0 \quad \forall i$.

Mean Squares Error

The Mean Squares Error (denoted NMSE) is not really what it sounds like. According to [3], it appears to be rarely present in the literature. Its definition is:

$$\mathbf{NMSE} = \frac{\sum_{i=1}^{n}(x_i - X_i)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Reduction of Error

The Reduction of Error (RE) is used interchangeably with the name Nash-Sutcliffe coefficient (NS). It is 1 - NMSE:

$$\mathbf{RE} = 1 - \frac{\sum_{i=1}^{n}(x_i - X_i)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Root Mean Square Error

The Root Mean Squares Deviation (RMSD) is the square root of the Mean Square Error (MSE). For this reason it is also sometimes referred to as the Root Mean Squares Error, or RMSE:

$$\mathbf{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - x_i)^2}$$

Normalized Root Mean Square Deviation

Recall that the previous criterion was sometimes referred to as RMSD. The Normalized Root Mean Square Deviation (NRMSD) is a normalization of the previous criterion (RMSD or RMSE). The RMSD is scaled by the range of the true values in the time series, and then multiplied by 100 for easier interpretation. Mathematically:

$$\mathbf{NRMSD} = 100 \times \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - x_i)^2}}{\max(x_i) - \min(x_i)}$$

Root Mean Square Standardized Error

The Root Mean Square Standardized Error (RMSS) is similar to the Root Mean Square (RMS), but the denominator has the standard deviation of the true values of the time series, rather than just the values themselves.

$$\mathbf{RMSS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{X_i - x_i}{\sigma_x}\right)^2}$$

(This cannot be computed if the standard deviation is zero, but this will almost never happen in practice).

Discussion of Performance Criteria

While there are a few other formulae presented in Table 1 of *Interpolation in Time Series: An Introductive Overview of Existing Methods, Their Performance Criteria and Uncertainty Assessment*, the ones presented above are the ones which can be applied to any pair of observed and interpolated time series. Other methods presented in the table require that certain algorithms be used, so they cannot be used to compare performance of different algorithms applied to the same test series. For this reason, the other methods presented in Table 1 of the above paper will not be included in the function.

Problems with Performance Criteria

Many of the criteria described previously present a number of problems. For example, none of the above criteria take into account the distances between gaps. Obviously, a time series which has 50 missing points in a row will be more difficult to interpolate than a time series which has 50 missing points evenly dispersed throughout the series.

Another problem is that many of the performance criteria are difficult to interpret, because they are not scaled or normalized in any way that will allow for meaningful comparisons.

Furthermore, taking only the size of the time series (n) into account may lead to problems. Perhaps criteria should also take into account the number of missing points as well, or the percentage of missing points. For example, 10 missing points in a series with 100 observations is a lot different than 10 missing points in a series with 10000 observations. Thus, when scaling only by n or something which depends on n directly, it is possible that certain algorithms will appear to have performed "better" by having falsely inflated criteria that do not take the number of missing points into account.

Interesting future work might be to create a new criterion for evaluating the performance of time series interpolation algorithms, taking into account several factors: the original time series, the interpolated time series, the length of the time series, the number of missing points in the time series, and the distances between the missing points in the time series.

Testing and Benchmarking a Variety of Algorithms

Now that a number of algorithms and performance criteria have been presented, they will be tested on a variety of real-world time series and their performance will be evaluated. The code for this analysis is far too long to include in this report (and is instead included in the accompanying folder), but the following general approach was taken for each obtained time series:

- Use the `sample` function to randomly remove $\alpha\%$ of points from the original, unaltered time series; call this series `gapped_series`
- Use a loop to pass `gapped_series` to all 18 of the interpolation algorithms described above
- Use a nested loop to calculate the performance of each of the 18 interpolation algorithms, using the 17 performance criteria described above
- Repeat the above steps for each of the obtained time series

- Repeat the above steps for various levels of α (such as 5%, 10%, 15%, etc.)

Datasets Used

In the following sections, the datasets chosen for this experiment will be described.

All of the series chosen will be non-stationary, for the reasons discussed earlier in this paper.

In order to use the 17 performance criteria that were described in this paper, the original, unaltered time series (with no missing values) is required. For this reason, only time series which do not have missing values were included in this analysis.

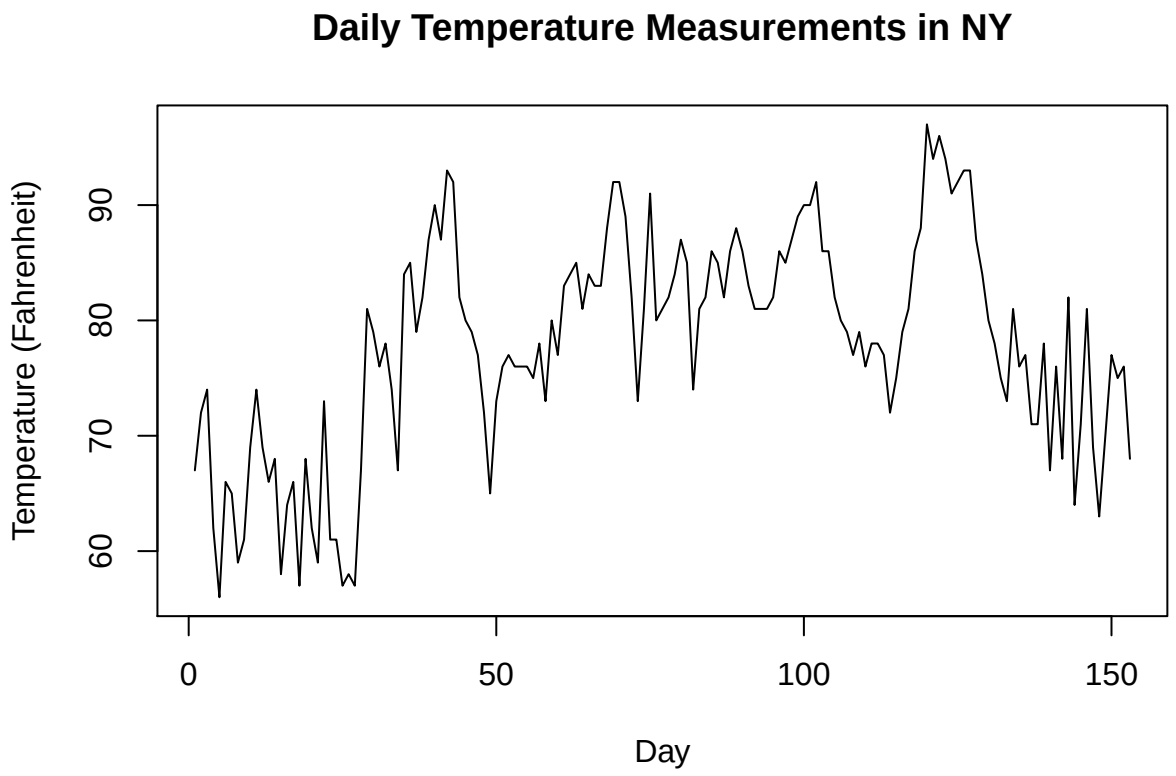
It was discovered early on in the running of the code that the loop is quite computationally expensive. This is due to the fact that there are 18 algorithms, 17 performance criteria, and 5 levels of α, for each dataset. For this reason, only a few datasets were used, but this experiment can easily be improved by adding computational power. Future steps will be discussed in the discussion section.

Datasets which had varying length (number of observations) and varying sampling frequency (e.g., daily, monthly observations) were used.

The Air Quality Dataset

The first dataset used is quite simplistic: the `airquality` dataset, available in base `R`. This dataset contains a number of variables, but the variable chosen for inclusion in this analysis was the `Temp` variable, which contains daily measurements of temperature (in Fahrenheit) in New York, from May to September of 1973. This dataset consists of 153 observations in total.

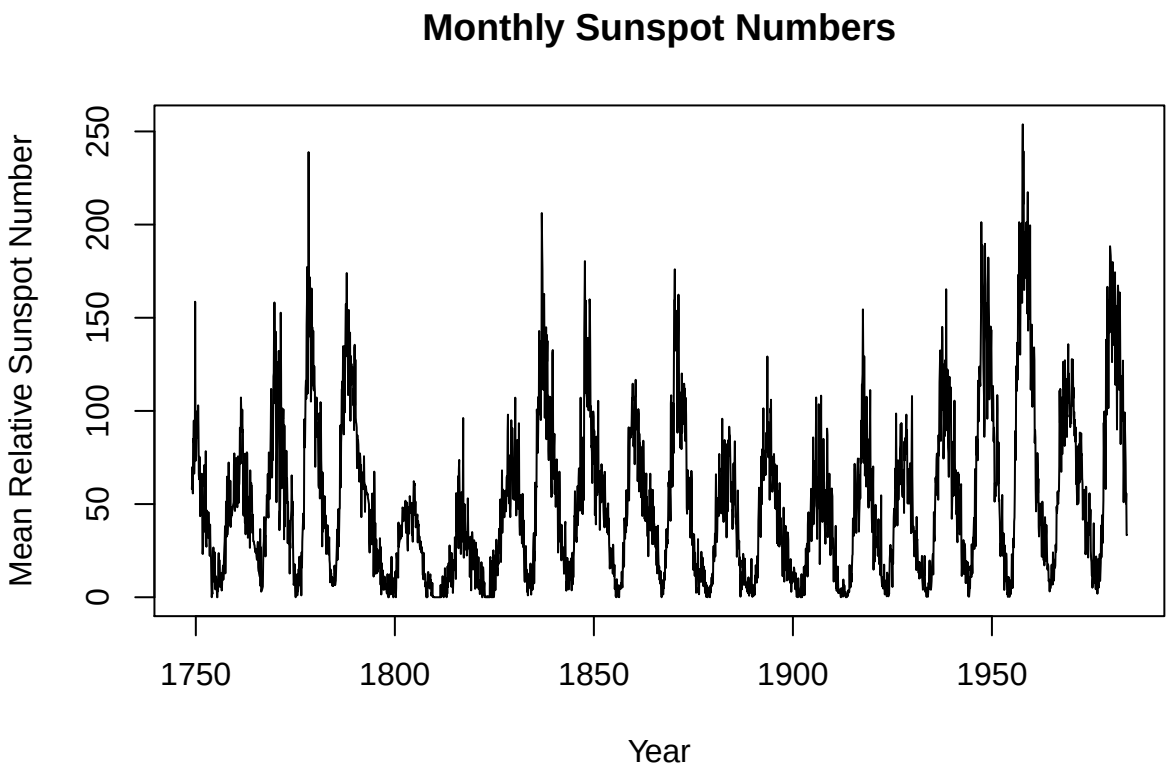
```
plot(airquality$Temp, type = "l",
     xlab = "Day",
     ylab = "Temperature (Fahrenheit)",
     main = "Daily Temperature Measurements in NY")
```



The Sunspots Dataset

The second dataset chosen for this project is the `sunspots` dataset, which is also available in base `R`. This data was chosen because it is a classic example that is regularly used in time series analysis. It consists of monthly mean relative sunspot numbers from 1749 to 1983, and thus consists of 2820 observations in total.

```
plot(sunspots, type = "l",
     xlab = "Year",
     ylab = "Mean Relative Sunspot Number",
     main = "Monthly Sunspot Numbers")
```



The Solar Flux Dataset

The third and most complex dataset that was used for this project is one which is readily available in Wesley Burr's `tsinterp` package[1], entitled `flux`. This dataset contains daily measurements of solar flux. *Solar flux* is a measure of how much light energy is being radiated in a given area, and in this dataset it is measured in solar flux units (SFU)[8].

This dataset contains two separate time series of daily noon 10.7cm solar flux measurements. The first is from the United States Air Force (USAF) monitoring station in Sagamore Hill, Massachussetts. The second is from the Dominion Radio and Astronomical Observatory (DRAO) in Penticton, British Columbia. The only time period of interest is that for which measurements are available in both of the aforementioned series. For this reason, both series include measurements from May 1st, 1966, until January 31st, 2008. The Penticton series is continuous, with no missing data points, while the Sagamore Hill series has a number of missing points.

In the `flux` dataset itself, we have the following variables:

- `Yr` - The year of the obseration; ranges from 1966 to 2008
- `Mnt` - The month of the observation; a number from 1 to 12
- `Day` - The day of the month of the observation; a number from 1 to 31
- `SagOrig` - The daily noon solar flux measurement in Sagamore Hill; measured in solar flux units (SFU)
- `PentOrig` - The daily noon solar flux measurement in Penticton; measured in solar flux units (SFU)

- `S` - An indicator variable, indicating whether or not the observation is missing for the Sagamore Hill series; either `TRUE` or `FALSE`

The dataset has 15251 observations in total. For the Penticton dataset, every day in the series has an observation. For the Sagamore Hill series, there are many missing observations due to instrument failures and repairs. In the dataset, a value of `-99` in the `SagOrig` variable represents a day for which there was no observation. In total, there are 1312 (roughly 8.6%) missing observations in this dataset. To demonstrate the above, consider the following subset of the `flux` dataset from the `tsinterp` package.

```
library(tsinterp)
data("flux")
kable(flux[85:94, ], "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = "striped")
```

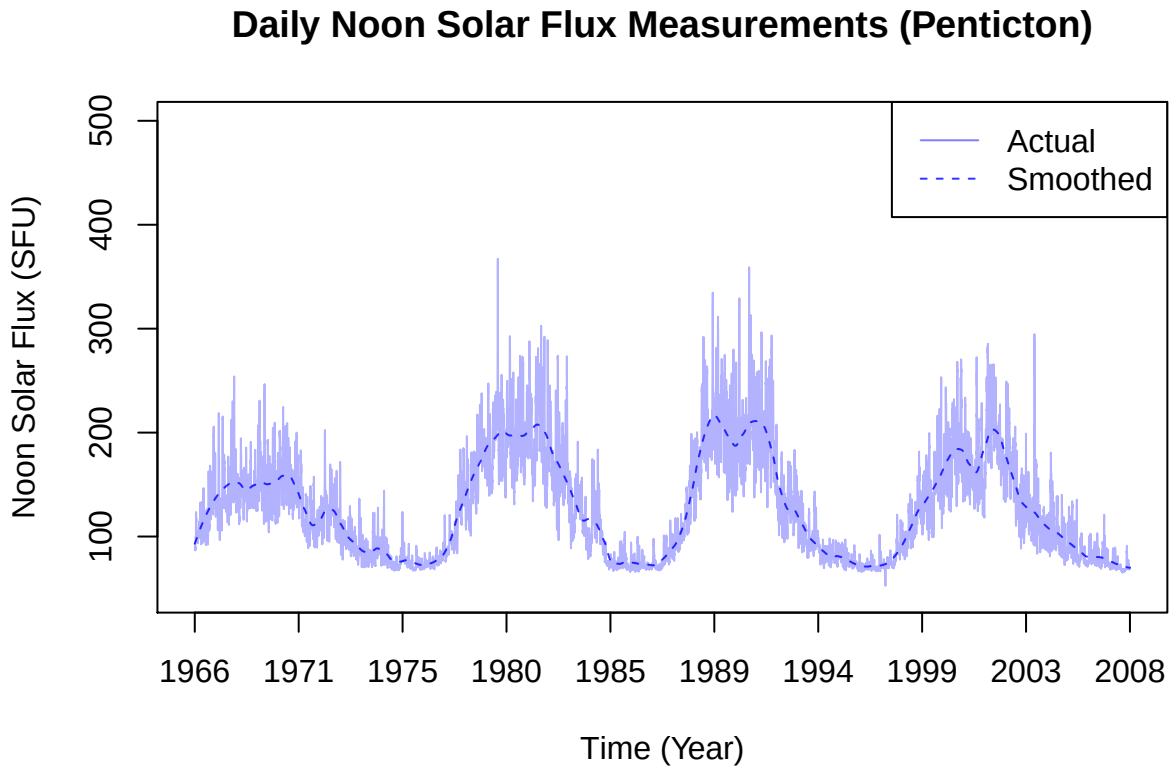
	Yr	Mnt	Day	SagOrig	PentOrig	S
85	1966	7	24	115.0	120.6	TRUE
86	1966	7	25	122.0	126.0	TRUE
87	1966	7	26	126.0	127.6	TRUE
88	1966	7	27	122.7	123.8	TRUE
89	1966	7	28	-99.9	124.2	FALSE
90	1966	7	29	-99.9	132.8	FALSE
91	1966	7	30	123.9	128.0	TRUE
92	1966	7	31	121.1	124.6	TRUE
93	1966	8	1	117.4	125.9	TRUE
94	1966	8	2	117.5	119.5	TRUE

From this peek at the `flux` dataset, it can be seen that there are missing noon solar flux measurements for the Sagamore Hill series on July 28th and 29th, 1966. Since this analysis requires complete time series, the `PentOrig` variable is used.

It is also valuable to visualize the series. The `PentOrig` series is plotted below, with the true series and a superimposed Loess regression smooth curve.

```
library(tsinterp)
data(flux)
df <- data.frame("pent" = flux$PentOrig,
                 "index" = 1:length(flux$PentOrig))
smoothed_pent <- loess(pent ~ index, data = df, span = 0.05)
smoothed_pent <- predict(smoothed_pent)
plot(flux$PentOrig,
     main = "Daily Noon Solar Flux Measurements (Penticton)",
     xlab = "Time (Year)",
     ylab = "Noon Solar Flux (SFU)",
     type = "l",
     xaxt = "n",
     col = rgb(0, 0, 1, 0.3),
     ylim = c(45, 500))
axis(side = 1, at = seq(0, 15251, length.out = 10),
```

```
     labels = round(seq(1966, 2008, length.out = 10)))
lines(smoothed_pent, col = rgb(0, 0, 1, 0.8), lty = 2)
legend("topright",
       legend = c("Actual",
                  "Smoothed"),
       col = c(rgb(0, 0, 1, 0.5),
               rgb(0, 0, 1, 0.8)),
       lty = c(1, 2))
```



Results

The results of my experiment were saved to a number of `RDa` objects, called `performance_alpha.RDa`, where α can be one of 0.05, 0.10, 0.15, 0.20, or 0.25 (corresponding to 5% imposed gaps, 10% imposed gaps, and so on). For example, consider `performance_0.05.RDa`:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("performance_0.05.RDa")
summary(performance_0.05)
```

```
##                                 Length Class  Mode
## Nearest Neighbor                3      -none- list
## Linear Interpolation            3      -none- list
```

```
## Natural Cubic Spline                3      -none- list
## FMM Cubic Spline                    3      -none- list
## Hermite Cubic Spline                3      -none- list
## Stineman Interpolation              3      -none- list
## Kalman - ARIMA                      3      -none- list
## Kalman - StructTS                   3      -none- list
## Last Observation Carried Forward    3      -none- list
## Next Observation Carried Backward   3      -none- list
## Simple Moving Average               3      -none- list
## Linear Weighted Moving Average      3      -none- list
## Exponential Weighted Moving Average 3      -none- list
## Replace with Mean                   3      -none- list
## Replace with Median                 3      -none- list
## Replace with Mode                   3      -none- list
## Replace with Random                 3      -none- list
## Hybrid Wiener Interpolator          3      -none- list
```

The structure of `performance_0.05.RDa` is 18 lists (one for each algorithm).

For each algorithm, there are 3 lists (one for each dataset), each of which contain 17 values (one for each performance criteria):

```
summary(performance_0.05[[1]])
```

```
##                 Length Class  Mode
## airquality Temp 17     -none- list
## sunspots        17     -none- list
## flux Penticton  17     -none- list
```

So, to access the value of r^2 for the `sunspots` dataset with the Exponenital Weighted Moving Average, use the following:

```
performance_0.05[[13]][[2]]$r_squared
```

```
## [1] 0.9930714
```

Through observing the structure of these objects, the reader should now be able to see the reasoning for the computational power required for this experiment: There are 5 gap levels, 18 algorithms, 17 performance criteria, and 3 datasets. Altogether, that is a total of $(5)(18)(17)(3) = 4590$ computations, all of which are fairly computationally expensive.

Given the complexity of these objects, it is useful to create smaller objects, which contain summaries of the information that might be most useful. For example, say one was interested in viewing a table of all of the values for r^2, across all 18 algorithms and all 3 datasets. For 5% gap level:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("performance_matrices_0.05.RDa")
kable(performance_matrices_0.05[[2]], "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = "striped")
```

	airquality Temp	sunspots	flux Penticton
Nearest Neighbor	0.9670885	0.9848461	0.9982521
Linear Interpolation	0.9837846	0.9894742	0.9992776
Natural Cubic Spline	0.9787221	0.9846143	0.9990522
FMM Cubic Spline	0.9788105	0.9846144	0.9990522
Hermite Cubic Spline	0.9823792	0.9878771	0.9992216
Stineman Interpolation	0.9827157	0.9891869	0.9992867
Kalman - ARIMA	0.9780451	0.9900272	0.9992932
Kalman - StructTS	0.9797611	0.9900842	0.9992813
Last Observation Carried Forward	0.9819630	0.9830567	0.9977042
Next Observation Carried Backward	0.9723929	0.9845556	0.9981428
Simple Moving Average	0.9688837	0.9889126	0.9981016
Linear Weighted Moving Average	0.9749554	0.9898442	0.9986076
Exponential Weighted Moving Average	0.9791400	0.9902567	0.9989614
Replace with Mean	0.9214637	0.9012774	0.9023975
Replace with Median	0.9319057	0.8995387	0.8993690
Replace with Mode	0.9385171	0.8013091	0.8058380
Replace with Random	0.5704952	0.8956058	0.5787502
Hybrid Wiener Interpolator	0.9777926	0.9884486	0.9990778

Objects with the name `performance_matrices_alpha.RDa` were created for each value of α (0.05, 0.10, 0.15, 0.20, 0.25). Each one is a list of 17 matrices (one for each performance criteria).

While these matrices are slightly more informative, there is still a significant amount of scanning by eye that one has to do if they wish to interpret the contents of one of the tables like the one presented above. Instead, it would be useful to determine which algorithm performed the "best" for the each dataset and each algorithm. In the case of r^2, the "best" value will be the one which is closest to 1. So, for the above table, with the `sunspots` dataset:

```
index <- which.max(performance_matrices_0.05[[2]][,2])
rownames(performance_matrices_0.05[[2]])[index]
```

```
## [1] "Exponential Weighted Moving Average"
```

So, for the `sunspots` dataset, at 5% gap level, the algorithm which performed the "best" in terms of maximizing the value of r^2 was the Exponential Weighted Moving Average.

Similarly, one might be interested in determining which algorithm performed the "worst" (r^2 closest to 0) for the above set up:

```
index <- which.min(performance_matrices_0.05[[2]][,2])
rownames(performance_matrices_0.05[[2]])[index]
```

```
## [1] "Replace with Mode"
```

Therefore, the algorithm which resulted in the lowest value of r^2 for the `sunspots` dataset at 5% gap level is Replace with Mode.

The next logical step would be to create a table displaying the name of the algorithm that performed "best" for each of the 17 criteria and each of the 3 datasets (17 rows by 3 columns). Note that for the case of error

criteria (such as MSE, MAE, etc), the "best" value will be the one which is smallest (and thus, the "worst" value will be the one which is largest).

Tables of this nature were created and saved to objects named `best_performance_alpha.RDa`. For 5% gap levels:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("best_performance_0.05.RDa")
kable(best_performance_0.05, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "scale_down"))
```

	airquality Temp	sunspots	flux Penticton
pearson_r	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
r_squared	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
abs_differences	Last Observation Carried Forward	Kalman - StructTS	Kalman - ARIMA
MBE	Last Observation Carried Forward	Replace with Random	Replace with Random
ME	Replace with Random	Replace with Mode	Replace with Mode
MAE	Nearest Neighbor	Linear Interpolation	Next Observation Carried Backward
MRE	NA	NA	NA
MARE	NA	NA	NA
MAPE	NA	NA	NA
SSE	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
MSE	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
RMS	NA	NA	NA
NMSE	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
RE	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
RMSE	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
NRMSD	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
RMSS	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA

Similarly, for the worst performance:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("worst_performance_0.05.RDa")
kable(worst_performance_0.05, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "scale_down"))
```

	airquality Temp	sunspots	flux Penticton
pearson_r	Replace with Random	Replace with Mode	Replace with Random
r_squared	Replace with Random	Replace with Mode	Replace with Random
abs_differences	Replace with Random	Replace with Mode	Replace with Random
MBE	Replace with Random	Replace with Mode	Replace with Mode
ME	Last Observation Carried Forward	Replace with Random	Replace with Random
MAE	Replace with Random	Replace with Mode	Replace with Random
MRE	NA	NA	NA
MARE	NA	NA	NA
MAPE	NA	NA	NA
SSE	Replace with Random	Replace with Mode	Replace with Random
MSE	Replace with Random	Replace with Mode	Replace with Random
RMS	NA	NA	NA
NMSE	Replace with Random	Replace with Mode	Replace with Random
RE	Replace with Random	Replace with Mode	Replace with Random
RMSE	Replace with Random	Replace with Mode	Replace with Random
NRMSD	Replace with Random	Replace with Mode	Replace with Random
RMSS	Replace with Random	Replace with Mode	Replace with Random

An important note is that some of the values in the table are missing. This is because the formulae for computing these criteria can occasionally result in divide-by-zero errors. As a result, these values were turned to `NA` intentionally, to avoid `NaN`s.

Finally, the last logical step is to create a table that will display the "best" algorithm for each of the 3 datasets, at each of the 5 gap levels (3 columns, 5 rows). This was determined by picking the algorithm which had the highest frequency in the `best_performance_alpha` tables. For example, in the table presented immediately above, the "best" algorithms would have been Linear Interpolation, Exponential Weighted Moving Average, and Kalman ARIMA for the `airquality Temp`, `sunspots`, and `flux Penticton` series respectively.

This was also done for the worst algorithms for each of the 3 datasets and 5 gap levels.

The best algorithms:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("best.RDa")
kable(best, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = "striped")
```

	airquality Temp	sunspots	flux Penticton
0.05	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
0.10	Linear Interpolation	Exponential Weighted Moving Average	Kalman - ARIMA
0.15	Linear Interpolation	Kalman - ARIMA	Kalman - ARIMA
0.20	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
0.25	Exponential Weighted Moving Average	Kalman - StructTS	Kalman - ARIMA

The worst algorithms:

```
setwd("C:/Users/Melissa/Documents/School/Trent/Fourth Year/MATH4800/Performances")
load("worst.RDa")
kable(worst, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = "striped")
```

	airquality Temp	sunspots	flux Penticton
0.05	Replace with Random	Replace with Mode	Replace with Random
0.10	Replace with Random	Replace with Mode	Replace with Random
0.15	Replace with Random	Replace with Random	Replace with Random
0.20	Replace with Random	Replace with Random	Replace with Mode
0.25	Replace with Random	Replace with Mode	Replace with Random

The tables can be read in the following way (as an example): for the `sunspots` dataset, the best performing algorithm at 25% gap level was the Kalman Filter with a Structural Model. The worst performing algorithm at 25% gap level for the `sunspots` dataset was Replace with Mode.

While these results may seem small, they will be very informative and useful when this experiment is scaled up in the future.

Discussion

There are a number of discussion points on the results of this experiment. This discussion will begin in the most obvious place: the final results.

It can be observed that there was not one "magic" algorithm that performed the best in every single scenario. The algorithm of choice depends largely on the data given, and on the gaps. As the level of gaps increases, more sophisticated models (such as Kalman Filters or Exponential Weighted Moving Averages) were favoured. In all circumstances, replacing missing values with random numbers or the mode were the worst choices, which was highly expected.

The next point of discussion is the comparison of the algorithms. While the final tables created in this project declared just one "winner" (and one "loser") for each dataset and gap level, this was a very close call. The algorithms which performed well actually performed *very well*, and very comparably to one another. For example, consider the value of the correlation coefficient, r, for 25% gap level:

```
load("~/School/Trent/Fourth Year/MATH4800/Performances/performance_matrices_0.25.RDa")
df <- performance_matrices_0.25[[1]][order(performance_matrices_0.25[[1]]$sunspots,
                                    decreasing = TRUE), ][2]
kable(df, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = "striped")
```

	sunspots
Kalman - StructTS	0.9867151
Kalman - ARIMA	0.9864368
Exponential Weighted Moving Average	0.9863733
Linear Weighted Moving Average	0.9860075
Hybrid Wiener Interpolator	0.9855957
Linear Interpolation	0.9855785
Simple Moving Average	0.9849504
Stineman Interpolation	0.9848180
Hermite Cubic Spline	0.9831164
Nearest Neighbor	0.9794587
Next Observation Carried Backward	0.9782892
FMM Cubic Spline	0.9773758
Natural Cubic Spline	0.9773755
Last Observation Carried Forward	0.9740187
Replace with Mean	0.8644244
Replace with Median	0.8571737
Replace with Random	0.7956341
Replace with Mode	0.7380262

In this case, the majority of the algorithms were the same to the first two decimal places, with the first few matching 3 decimal places. It was a "close call" – the "top" algorithm did not "win" by a landslide.

Another important note is that some of the performance criteria are largely ineffective and useless. A good performance criterion should be able to discern a well-performing algorithm from a poorly-performing algorithm, but this was not the case with some of the criteria that were used. For example, consider the best and worst performing algorithms at the 20% gap level:

```
load("~/School/Trent/Fourth Year/MATH4800/Performances/best_performance_0.20.RDa")
load("~/School/Trent/Fourth Year/MATH4800/Performances/worst_performance_0.20.RDa")
kable(best_performance_0.20, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "scale_down"))
```

	airquality Temp	sunspots	flux Penticton
pearson_r	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
r_squared	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
abs_differences	Natural Cubic Spline	Kalman - ARIMA	Kalman - StructTS
MBE	Replace with Random	Replace with Random	Replace with Random
ME	Next Observation Carried Backward	Replace with Mode	Replace with Mode
MAE	Replace with Median	Last Observation Carried Forward	Stineman Interpolation
MRE	NA	NA	NA
MARE	NA	NA	NA
MAPE	NA	NA	NA
SSE	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
MSE	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
RMS	NA	NA	NA
NMSE	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
RE	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
RMSE	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
NRMSD	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA
RMSS	Natural Cubic Spline	Kalman - ARIMA	Kalman - ARIMA

```
kable(worst_performance_0.20, "latex", booktabs = TRUE) %>%
  kable_styling(latex_options = c("striped", "scale_down"))
```

	airquality Temp	sunspots	flux Penticton
pearson_r	Replace with Random	Replace with Random	Replace with Mode
r_squared	Replace with Random	Replace with Random	Replace with Mode
abs_differences	Replace with Mean	Replace with Random	Replace with Mode
MBE	Next Observation Carried Backward	Replace with Mode	Replace with Mode
ME	Replace with Random	Replace with Random	Replace with Random
MAE	Replace with Random	Replace with Random	Replace with Mode
MRE	NA	NA	NA
MARE	NA	NA	NA
MAPE	NA	NA	NA
SSE	Replace with Random	Replace with Random	Replace with Mode
MSE	Replace with Random	Replace with Random	Replace with Mode
RMS	NA	NA	NA
NMSE	Replace with Random	Replace with Random	Replace with Mode
RE	Replace with Random	Replace with Random	Replace with Mode
RMSE	Replace with Random	Replace with Random	Replace with Mode
NRMSD	Replace with Random	Replace with Random	Replace with Mode
RMSS	Replace with Random	Replace with Random	Replace with Mode

The MBE, ME, and MAE show that terrible algorithms (replace with random, replace with mode, replace with median, NOCB, LOCF) are somehow simultaneously best and worst. This indicates that these performance criteria are not reliable for measuring the performance of an interpolation algorithm, and should not be used. This is particularly interesting, since there have been actual scientific papers that were published, using these criteria.

Improvements

While this was interesting for an Honours Project, there are many improvements to be made and steps to take to make the results publishable.

Increase Data

The most obvious improvement is to add more data. More time series should be run through the program, and there should be a large variety of series chosen. For example, there should be datasets that are economic, environmental, biological, and so on. Within each of these categories, there should also be series which are of varying length, periodicity, and sampling frequency.

Increase gap selection methods

For this project, the gaps were simply selected using the `sample` function in `R`. This could be significantly improved, as it is rare in real-world data to find missing values that are truly *random*. Typically, missing values are due to some underlying structure, or equipment failure. For this reason, the number of ways in which the gaps are selected should be increased. For example, it would be interesting to remove points at regular intervals (e.g., every 7th value for observations taken daily). It would also be valuable to simulate equipment failure, by removing large chunks of subsequent observations (e.g., removing 20 consecutive values), and to vary the maximum length of these chunks. Additionally, the maximum and minimum distances between gaps could be varied.

Repetition

This entire experiment was only run once, with one seed value. It would be ideal to run this experiment thousands of times with many different seeds, in order to obtain better results, but this would require significant computational power.

Increase and refine algorithms and criteria

An obvious extension would be to add more algorithms and performance criteria to the experiment. A great starting place would be to add a "calculation time" criterion, which would favour algorithms that are fast.

A revision should also be done to determine which algorithms and criteria are worth including in the experiment. For example, it is a bit silly to waste computational resources on methods such as "replace with random" or "replace with mode", when they will never be the best performers. Instead, by keeping these methods in the analysis, more interesting results are being hidden (i.e., perhaps one of the "better" algorithms is performing terribly, but since it is being compared to "replace with random", its performance appears to be falsely inflated). Similarly, criteria which were ineffective should be not included, because they waste time and distract the analyst from more important results.

Additionally, the way in which the "best" or "worst" algorithm is chosen should be refined. The current method of taking whichever algorithm had the highest frequency of "winning" results for the performance

criteria is likely not the best way to determine this, because some criteria are better than others, and should probably be weighted more heavily.

Future Work

In addition to improving the experiment itself, there are also a number of recommendations for future work on this project.

A nice extension would be to create a function and add it to the `tsinterp` package which will allow a user to input a time series and a gap level, and the function will perform the experiment (randomly impose gaps, then use the interpolation algorithms, and calculate performance criteria) and return to the user a recommendation for which algorithm they should use for the specific dataset they provided. This might not sound helpful, since it requires the user to have a time series with no missing values. While this is a good point, if the user has a series which is mostly complete, they can take a subset of the series which is complete, and use it to gain insight into which algorithm might be best to use for the entire series (which may include real missing values). Alternatively, if the user has a similar time series which is complete and is similar to a time series of interest with missing values, they would be able to use this function. For example, using this function on the `flux$PentOrig` series may provide valuable advice for determining which algorithm to use on the `flux$SagOrig` series, which does include missing values.

Finally, a very ambitious goal would be to publish a paper which not only states the results of performing this experiment on a larger scale, but which is also able to provide recommendations to the Statistical community about which algorithm is "best" for the specific type of data that one may have.

Conclusion

In this Honours Project, 18 different time series interpolation algorithms were researched and tested on 3 real-world datasets. The performance of these algorithms on the data was evaluated, using a variety performance criteria.

Kalman Filters and Weighted Moving Averages were found to perform particularly well for the specific data used in this experiment, though there were a number of algorithms that performed comparably.

References

[1] Wesley S. Burr. *Air Pollution and Health: Time Series Tools and Analysis.* Queen's University, PhD thesis. 2012.

[2] Wesley S. Burr (2012). *tsinterp: A Time Series Interpolation Package for R.* `R` Package.

[3] Mathieu Lepot, Jean-Baptiste Aubin, and Francois H.L.R. Clemens. *Interpolation in Time Series: An Introductive Overview of Existing Methods, Their Performance and Uncertainty Assessment.* Water 2017, 9(10), 796.

[4] Peter J. Brockwell and Richard A. Davis. *Time Series: Theory and Methods.* Springer. 2006.

[5] Data Camp. *Introduction to Time Series Analysis.* Data Camp course.

[6] Wikipedia, the Free Encyclopedia. *Kalman Filter*. Web. Retrieved 20 February 2019, https://en.wikipedia.org/wiki/Kalman_filter.

[7] Wikiversity. *Cubic Spline Interpolation*. Web. Retrieved 20 February 2019, https://en.wikiversity.org/wiki/Cubic_Spline_Interpolation.

[8] Desmond, Joe. *Understanding Solar Flux: Science vs. Science Fiction*. Web. 19 August 2014. Retrieved 30 January 2019, http://www.brightsourceenergy.com/understanding-solar-flux-science-vs-science-fiction-1.