

## Chapter 2

# Mining Time-Series Data

Data is being generated in an ever increasing rate by all kinds of human endeavors. A sizable fraction of this data appears in the form of time-series or can be converted to this form. For example, a random sampling of 4000 graphics from 15 of the world's newspapers published from 1974 to 1989 found that 75 % of these graphics were time-series (Ratanamahatana et al. 2010).

This makes time series analysis and mining an important research area that is expected only to become more so over time. Time series analysis is a huge field and it is not possible to exhaustively cover it in an introductory chapter or even in a complete book. This means that in this chapter we had to be selective. The guiding principle for the selections made in this chapter is utility for the ideas presented in subsequent chapters. At the very least, this chapter introduces the notation used throughout the book and forms the background against which the ideas to be presented are painted.

This chapter was written with the social robotics researcher in mind and most of it would be elementary knowledge for practitioners of time-series analysis, or related fields. Nevertheless, a quick pass through the chapter is advised in order for the reader to familiarize herself with the notation and definitions that will be used extensively in the following chapters.

### 2.1 Basic Definitions

A time-series is simply an ordered list with an implied independent variable (usually time) and one or more dependent variables from a predefined domain (usually  $\mathbb{R}$ ).

**Definition 2.1** Time Series  $X_t$  is an ordered list of items  $(x_0, x_1, \dots, x_t, \dots, x_T)$  each of them is called a *point*, where  $t$  is the independent variable belonging to a domain  $D_T$  and is assumed to be monotonically increasing and  $T$  is a scalar specifying the length of the time-series. All items  $x_t$  belong to a predefined domain  $D_X$ .

In most of the time-series described in this book, the domain of the independent variable is the set of integers (i.e.  $D_T = \mathbb{I}^+$ ). If the domain of the dependent variable is the set of real numbers (i.e.  $D_X = \mathbb{R}$ ) then the time-series is called a *real valued time-series*. If the domain of the dependent variable is multidimensional (e.g.  $D_X = \mathbb{R}^N$ ) then the time-series is called a *multidimensional time-series* and the dimensionality of the time series is  $N$  where  $N$  is the dimensionality of each point. When the independent variable is not important or is known we will ignore it and will use  $X$  instead of  $X_t$ . In this book we use the notations  $x_i$  and  $x(i)$  interchangeably to mean the point  $i$  of the time-series  $X$ .

In many cases we need to consider a contiguous list of items belonging to a time-series. This is made more precise in the following definition.

**Definition 2.2** A subsequence  $x_{i:j}$  is a time-series that consists of the tuple  $(x_i, x_{i+1}, \dots, x_j)$  belonging the time-series  $X$ . Another notation is  $x_{i:n}$  which is the time-series that consists of the ordered points  $x_i, x_{i+1}, \dots, x_{i+n-1}$ .

We use  $x(\dots)$  to mean the same thing as  $x_{\dots}$ . In some cases, the length of the subsequence will be known from the context and in these cases we will drop it from the subsequence name (e.g.  $x_i$  will imply  $x_{i:n}$ ). Notice that in this case  $x_i$  is a point while  $x_{i:n}$  is a subsequence starting at point  $i$  with the implicit length  $n$ .

For simplicity and throughout this book we will speak of the independent variable as time ( $t$ ) even though our discussion can be generalized without any modification to any other variable given that it is monotonically increasing at a constant rate. We call this rate of increase/decrease of the independent variable  $\tau$  and in most cases it is assumed to be unity.

## 2.2 Models of Time-Series Generating Processes

Time-series are generated continuously from nearly every kind of information processing or dynamical system. It is not possible to find an exhaustive set of generating processes that can be combined to lead to all possible time-series but there are some time-series models that received more attention from researchers due to their simplicity and/or ability to represent a wide range of phenomena and this section introduces some of them.

### 2.2.1 Linear Additive Time-Series Model

The Linear Additive Time-series Model (LAT) decomposes the time-series into a set of four components that are combined linearly (additively):

$$X = T_0 + C + S + R. \quad (2.1)$$

$T_0$  is called the *trend* and is the long-term non-periodic variation in the time-series. In many cases it is a monotone function (i.e. its first difference does not change sign). For example the identity time-series  $x_t = t$  has only a trend component.

$C$  represents a cyclic component with some period  $T_c \gg \tau$  (remember that  $\tau$  is the rate of increase of the independent variable or the inverse of the sampling rate). A good example of cyclic components is the famous business cycle of economics or fluctuations of electricity consumption based on the time of the year.

$S$  represents another cyclic component but with a much smaller period which is only assumed to be greater than  $\tau$  (i.e.  $T_s > \tau$ ). This seasonal component is useful in modeling short lived behavior in some applications. For example, it can model daily fluctuations in electricity consumption.

$R$  models all other random fluctuations added to the time-series.

The differentiation between  $C$  and  $S$  is ad-hoc and there is no reason that restricts the number of cyclic components to just two. A more general model that we call xLAT (Extended LAT) assumes  $N_c$  cyclic components and can be written as:

$$X = T_0 + R + \sum_{n=1}^{N_c} C_n. \quad (2.2)$$

Notice that the Fourier transform (See Sect. 2.3.3) is a special case of this model assuming the trend to be constant and random fluctuations to be zero while setting  $N_c = \infty$  and each cyclic component to a sinusoidal function.

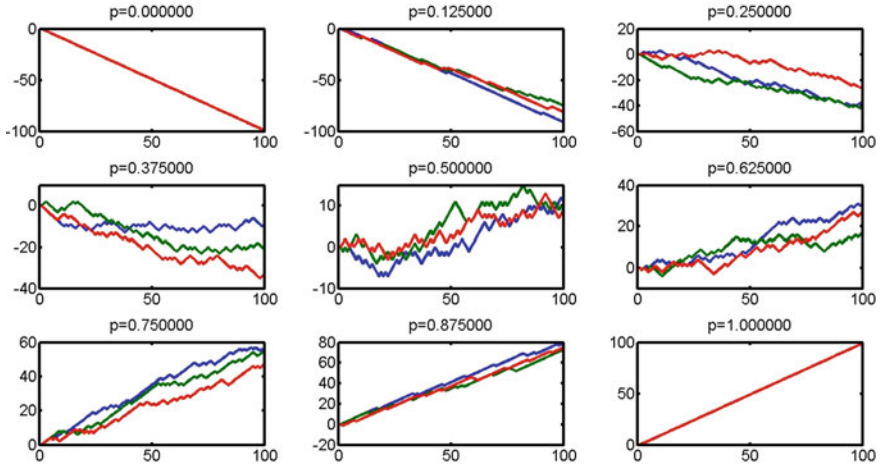
This decomposition of time-series data to different *kinds* of components can be useful in getting a sense of the underlying dynamics. For example, a lot of the controversy about global warming boils down to whether the perceived increase in temperatures is a part of  $T_0$ ,  $C$  or  $R$ .

The linear additive model has several applications specially in economics where the four types of components represent specific socio-economic factors affecting different economic metrics.

### 2.2.2 Random Walk

A random-walk is a time-series that is generated by making small random variations of the current time-series value at every step. One of the simplest random walks involves moving around the integer numbers (usually starting with zero) with an increment of either 1 or  $-1$  based on a fair coin flip. A slightly generalized version of this random walk can be formalized as follows:

$$x_0 = 0, \quad (2.3)$$



**Fig. 2.1** Examples of random walks with variable probability of increase  $p$

$$x_t = \begin{cases} x_{t-1} + \delta & 0 \leq p_t < p \\ x_{t-1} - \delta & \text{otherwise} \end{cases}, \quad (2.4)$$

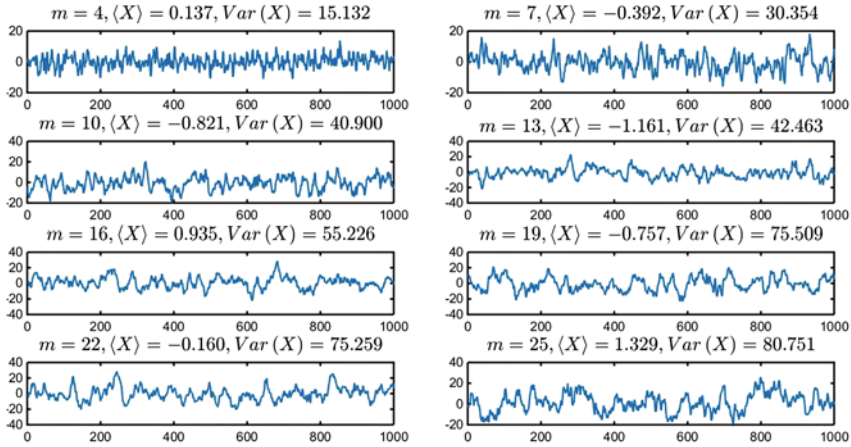
where  $\delta \in \mathbb{R}^+$  is some positive real number,  $p_t$  is a random number between zero and 1 and  $p$  is the probability of increasing.

The  $MC^2$  toolbox accompanying this book contains a data generation function called `generateRandomWalk()` that generates 1-D random walks. Figure 2.1 shows examples of random walks for different increase probabilities ( $p$ ). It is easy to show that the expected value of the time-series average and trend are zero when  $p = 0.5$  but it gets a positive/negative trend when  $p$  is larger/smaller than this critical value. The toolbox function allows the user to control the step size or even make it randomly chosen from a uniform probability distribution.

### 2.2.3 Moving Average Processes

A moving average process generates time-series points by a linear combination of past values of another time-series (usually white noise) and is parameterized by the number of past points involved.  $MA(m)$  is a moving average process of order  $m$  iff it generates data according to:

$$x_t^{ma(m)} = \sum_{i=0}^m a_i \theta_{t-i}, \quad (2.5)$$



**Fig. 2.2** Examples of time-series generated from a  $MA(m)$  process for different values of  $m$ . In all cases  $a_0 = 1$  and  $a_i = 2$  for  $1 \leq i \leq m$

where  $\Theta$  is a random time-series representing white noise. Moreover,  $a_i \in \mathbb{R}$  for  $0 \leq i \leq m$  and usually  $a_0$  is set to unity. White noise can be generated using `randn()` in MATLAB/Octave or better `wgn()`. Assuming that the mean of the white noise signal is  $\mu_\theta$  and its variance is  $\sigma_\theta^2$ , it can be shown that:

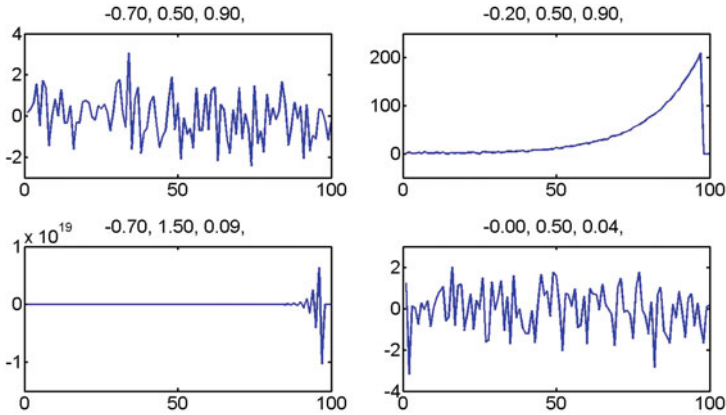
$$\langle X \rangle = \mu_\theta \sum_{i=0}^m a_i, \quad (2.6)$$

$$Var(X) = \sigma_\theta^2 \sum_{i=0}^m a_i^2, \quad (2.7)$$

where as usual  $\langle X \rangle$  and  $Var(X)$  are the expectation and variance of the time-series  $X$ .

The  $MC^2$  toolbox has a function `generateMA()` that can generate time-series from a moving average process  $MA(m)$ . Figure 2.2 shows examples of time-series generated from a  $MA(m)$  process for different values of  $m$ . In all cases  $a_0 = 1$  and  $a_i = 2$  for  $1 \leq i \leq m$ . It is clear that with increased order, the variance of the time-series increases. Notice that the mean of the white noise used was zero which accounts for the observation that the mean of the final time-series is also around zero and did not change with increased model order.

An implementation detail related to moving average processes is what to do with the first  $m$  points in the output time-series for which no enough data is available to apply Eq. 2.5. In our implementation we assume that  $\theta_{-1:-m}$  is generated from the same process from which  $\Theta$  is generated and use these values implicitly to set  $x_{0:m}$  (Fig. 2.3).



**Fig. 2.3** Examples of time-series generated from an  $AR(3)$  process for different values of the parameters  $a_i$ . Each example shows the values of  $a_{1:3}$  used to generate it. For simplicity we assume that the white noise had zero variance (e.g. no white noise)

### 2.2.4 Auto-Regressive Processes

An auto regressive process  $AR(m)$  generates each new data point as a linear weighted combination of the past  $m$  points in the time-series plus an additive white noise value.

$$x_t = \sum_{i=1}^m a_i x_{t-i} + \delta_t. \quad (2.8)$$

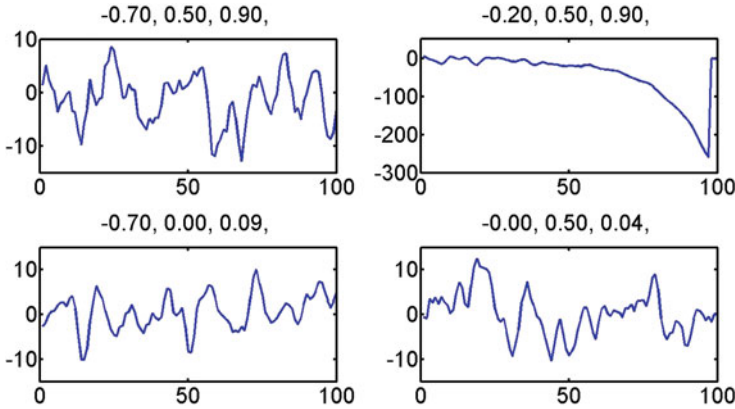
The function *generateAR()* can be used to generate data from an auto-regressive process in  $MC^2$  (Fig. 2.3).

### 2.2.5 ARMA and ARIMA Processes

An  $ARMA(m, n)$  process is a linear summation of an  $AR(m)$  process and a  $MA(n)$  process which can be written in the form:

$$x_t^{arma} = \sum_{i=1}^m a_i x_{t-i}^{arma} + \sum_{i=0}^n b_i \theta_{t-i}. \quad (2.9)$$

An  $ARIMA(m, n, d)$  process is a process when differencing its output  $d$  times will be an  $ARMA(m, n)$  process.  $ARIMA$  processes find many applications in economic theory but will not be used much in the second part of this book to model robot or human behavior.



**Fig. 2.4** Examples of time-series generated from an  $ARMA(3, 5)$  process for different values of the parameters  $a_i$ . Each example shows the values of  $a_{1:3}$  used to generate it. In all cases,  $B = [1, 2, 3, 2, 1]^T$

In  $MC^2$ , you can generate data from an  $ARMA$  process using `generateARMA()` and from an  $ARIMA$  process using `generateARIMA()` (Fig. 2.4).

### 2.2.6 State-Space Generation

Another widely used generation model of time-series is the state-space model.

$$\begin{aligned} s_t &= g(s_{t-1}) + r_1(\varepsilon_t), \\ x_t &= f(s_t) + r_2(\lambda_t). \end{aligned} \quad (2.10)$$

In general all time-series involved in this definition are multidimensional in order to code the full state of the system at every time-step.

Of special interest are affine state-space models that have the specific form:

$$\begin{aligned} s_t &= As_{t-1} + B\varepsilon_t, \\ x_t &= Cs_t + \lambda_t, \end{aligned} \quad (2.11)$$

where  $A$ ,  $B$ ,  $C$  are matrices and the output time-series of the model is  $X$ . Assuming that  $S$  has the dimensionality  $N_s$ ,  $X$  has the dimensionality  $N_x$ , and  $\varepsilon_t \in \mathbb{R}_{\varepsilon}^{N_{\varepsilon}}$  then  $A$  is a  $N_s \times N_s$  matrix,  $B$  is a  $N_s \times N_{\varepsilon}$  matrix, and  $C$  is a  $N_x \times N_s$  matrix. The noise components ( $\varepsilon_t$  and  $\lambda_t$ ) are sampled from two Gaussian distributions with zero mean and known covariance matrices.

This generation model can be used to simulate random walks, MA processes, AR processes, and ARMA models. Consider for example the ARMA model of Eq. 2.9. An equivalent affine state-space model (See Eq. 2.11) will have the following form:

$$s_t = (x_t \ x_{t-1} \ \dots \ x_{t-m+2} \ x_{t-m+1} \ \varepsilon_t \ \varepsilon_{t-1} \ \dots \ \varepsilon_{t-n+2} \ \varepsilon_{t-n+1})^T, \quad (2.12)$$

$$A = \begin{pmatrix} a^T & b^T \\ I^{m \times m} & 0^{m \times m} \\ 0^{n \times n} & I^{n \times n} \end{pmatrix}, \quad (2.13)$$

$$B = (1 \ 0^{1 \times m} \ 1 \ 0^{1 \times n})^T, \quad (2.14)$$

$$C = (1 \ 0^{n+m \times 1}), \quad (2.15)$$

$$\lambda_t = 0. \quad (2.16)$$

The reader can confirm that these definitions when plugged into Eq. 2.11 will recover Eq. 2.9. As random walks, MA and RA processes are all special cases of ARMA processes; we have just shown that affine state-space models can be used as a general generation process of all of these. This model is implemented in the function `generateAffineStateSpace()`.

## 2.2.7 Markov Chains

All of the previous methods for time-series generation are deterministic, in the sense that given the parameters of the time-series all points are known exactly up to the added noise. The Markov chain (MC) generation process, on the other hand, generates time-series points from a probabilistic distribution. The defining assumption of this model is that the time-series point depends only on the previous value of the time-series.

A MC model is defined by its initial distribution  $p(x_0)$  and its transition conditional distribution  $p(x_t|x_{t-1})$ . Both can take any form.

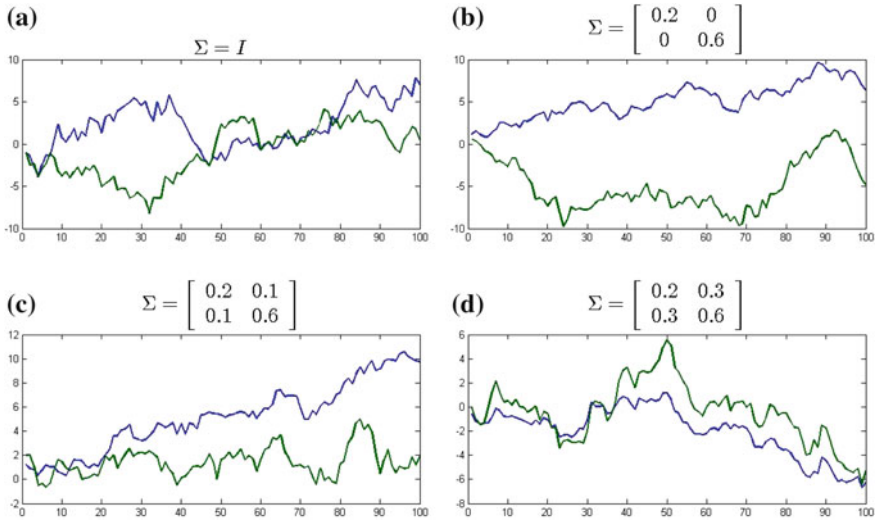
One of the simplest continuous MCs is the Gaussian MC Model (GMC) which is defined as:

$$\begin{aligned} x_0^{gmc} &\sim p(x_0^{gmc}) \equiv \mathcal{N}(\mu_0, \Sigma_0), \\ x_t^{gmc} &\sim p(x_t^{gmc}|x_{t-1}^{gmc}) \equiv \mathcal{N}(x_{t-1}, \Sigma), \ 0 < t \leq T. \end{aligned} \quad (2.17)$$

It has three parameters:

- $\mu_0, \Sigma_0$ : The mean and covariance matrix for generating the first point of the series
- $\Sigma$ : The covariance matrix used to generate  $x_{t+1}$  given  $x_t$  as its mean.

The  $MC^2$  toolbox has a function `generateMarkovChain()` which generates time-series of arbitrary length and dimensionality from a GMC. The implemented version has an option to add an MA process. Notice that this will not be added after the complete time-series is generated but at each time-step. The effect of having a non-zero MA(m) process here depends on the exact values of its  $m$  parameters. For example



**Fig. 2.5** Examples of 2 dimensional time-series (represented by the *two colors*) generated from a Gaussian Markov Chain with  $\mu = \mu_0 = 0$  and unit  $\Sigma_0$  for four different  $\Sigma$  cases (Color in online)

the MA(m) process with  $a = (1, 2, 3, 4, 5, 4, 3, 2, 1)^T$  will tend to smooth out the time-series. Figure 2.5 shows four time-series generated from this function.

It is interesting to see how can we control various aspects of the time-series by changing its three parameters (i.e.  $\mu_0$ ,  $\Sigma_0$ , and  $\Sigma$ ). Given that  $\mu_0$  and  $\Sigma_0$  affect only the first point of the time-series, we will focus on the effect of  $\Sigma$ . Figure 2.5a shows an example 2D time-series when  $\Sigma = I$  which means that the two dimensions of the time-series are independent (because the off-diagonal elements are zeros) and they both have the same overall variance. Figure 2.5b shows what happens when we change the relative values of the diagonal elements. Here the second dimension of the time-series (green) is still uncorrelated with the first dimension but has much higher variability because of increased variance. Figure 2.5c, d shows the effect of increasing off-diagonal elements creating correlations between different dimensions of the time-series.

### 2.2.8 Hidden Markov Models

A slightly more complex probabilistic generation mechanism is the Hidden Markov Model (HMM). For our purposes, a HMM has both unobserved states and observed time-series. An appropriate HMM definition for our purposes is given by:

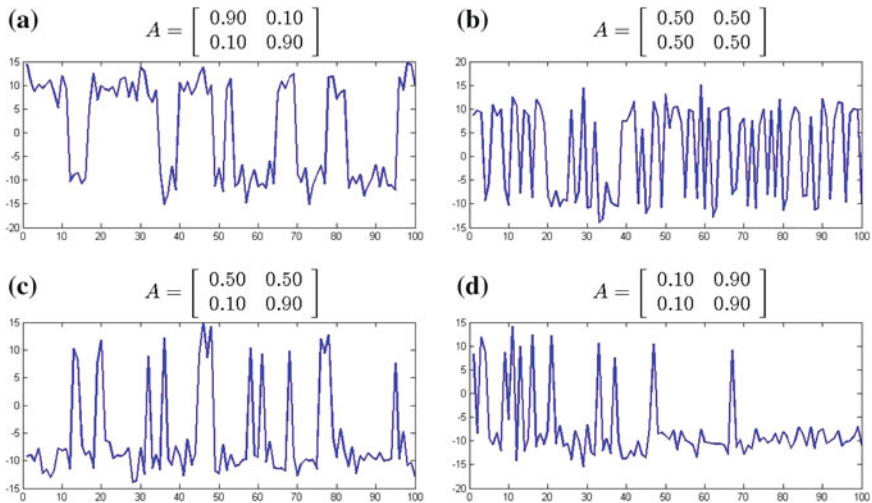
$$\begin{aligned} s_0 &\sim p(s_0), \\ s_t &\sim p(s_t | s_{t-1}), \quad 0 < t \leq T, \\ x_t^{hmm} &\sim p(x_t^{hmm} | s_t). \end{aligned} \quad (2.18)$$

For the rest of this book we will assume—if not otherwise stated—that the conditional probability  $p(x_t^{hmm}|s_t)$  is the same for all values of  $t$  but in general this needs not be the case and the probability distribution used may depend on time. Furthermore, we assume that the HMM has discrete  $N_s$  internal states, that the dimensionality of the output time-series is  $N_x$  and that the observation distribution is a Gaussian. This leads to the following definition of Gaussian HMM (GHMM):

$$\begin{aligned} s_0 &\sim p(s_0) \equiv \pi, \\ s_t &\sim p(s_t|s_{t-1}) \equiv A_{s_{t-1}}^T, 0 < t \leq T, \\ x_t^{ghmm} &\sim p(x_t^{ghmm}|s_t) \equiv \mathcal{N}(\mu_{s_t}, \Sigma_{s_t}). \end{aligned} \quad (2.19)$$

The literature on HMM and related probabilistic models is vast and the interested reader is advised to consult any of the excellent textbooks discussing these versatile structures. For our purposes though, GHMMs will suffice for all our needs in this book. The toolbox implements GHMM generation through the function `generateHMM()` which also has the option of adding an  $MA(m)$  process to the output.

Figure 2.6 shows four examples of the output of a Gaussian HMM with fixed means. The variation of the transition matrix affects the output in an expected way. Figure 2.6a shows a case where each state has high probability of being repeated (0.9) relative to the probability of going to the other state (0.1). This leads to a time-series that looks like a square wave with variable duty cycle. Figure 2.6b shows a middle case where the probability of staying at the same state is equal to the



**Fig. 2.6** Examples of time-series generated from a Gaussian Hidden Markov model with two states and single dimensional output. The means of the observational Gaussians for the two states were set to 10 and -10

probability of changing to the other state. This leads to a time series that jumps between the two states rapidly. Figure 2.6c shows a case with one state (the first) having equal probability of repeating or toggling while the other state (the second) has high probability of repeating (0.9). This leads to a time series that spends more time in the second state. A more extreme example is given in Fig. 2.6d where no matter what is the current state, the second state has higher probability of occurring in the next output compared with the first state (9 times higher). This leads to a time-series that spends even more of its time in the second state.

### 2.2.9 Gaussian Mixture Models

A widely used probabilistic generation model for time-series uses a mixture of probability distributions rather than switching between them as in HMM. In this book we only utilized Gaussian Mixture Models (GMM) so we will focus our attention on them.

Assume that we have a joint distribution of two variables  $x$  and  $y$  (i.e.  $p(x, y)$ ). We can then condition on one of them using the definition of conditioning:

$$p(x|y) = \frac{p(x, y)}{\sum_x p(x, y)}. \quad (2.20)$$

This means that we can generate points of a time-series  $X$  by conditioning on the independent variable (time) at every time-step. The math becomes much easier when we assume that the joint distribution is a Mixture of Gaussians in the form:

$$p(x, t) = \sum_{k=1}^K p(k) p_k(x, t), \quad (2.21)$$

$$p_k(x, t) \equiv \mathcal{N}(\mu_k, \Sigma_k).$$

Given this joint distribution, it is possible to generate time-series points by conditioning on time. This is the basic idea of Gaussian Mixture Regression (GMR) which is being widely applied in both statistics and learning from demonstration communities for providing a middle ground between high-bias parametric approaches and high-variance non-parametric approaches to modeling.

The use of a GMM to model the joint distribution ensures that the conditional distribution is also a GMM. In such cases, Eq. 2.20 simplifies to the following:

$$p(x|t) = \sum_{k=1}^K \pi_k^c p_k^c(x|t), \quad (2.22)$$

$$p_k^c(x|t) \equiv \mathcal{N}((x, t)^T; \mu_k^c, \Sigma_k^c).$$

Now assume that we make the following definitions:

$$\mu_k \equiv (\mu_k^x, \mu_k^t)^T, \quad (2.23)$$

$$\Sigma_k \equiv \begin{pmatrix} \Sigma_k^x & \Sigma_k^{xt} \\ (\Sigma_k^{xt})^T & \Sigma_k^t \end{pmatrix}, \quad (2.24)$$

where  $\mu_k^t \in \mathbb{R}^K$  and  $\Sigma_k^x \in \mathbb{R}^{N \times N}$  while  $\mu_k^x \in \mathbb{R}^{1 \times N}$ .

Given these definitions, the new parameters of the conditional GMM are given by the following set of equations:

$$\pi_k^c = \frac{p(k) \mathcal{N}(t; \mu_k^t, \Sigma_k^t)}{\sum \pi_k^c}, \quad (2.25)$$

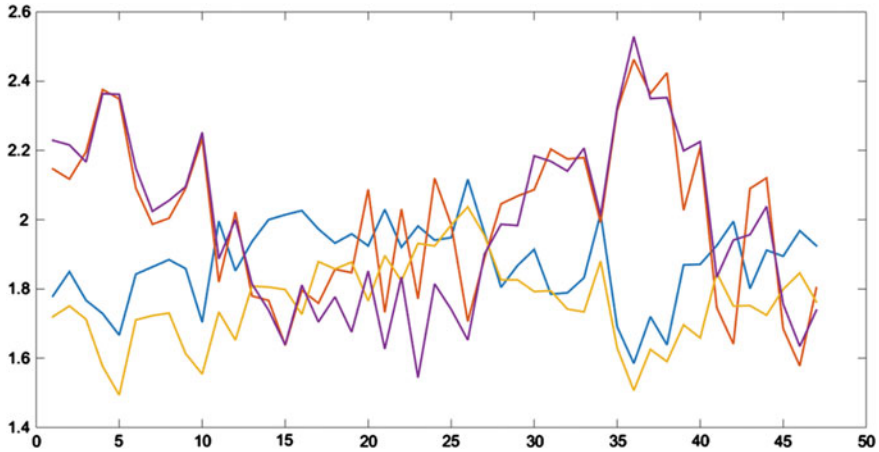
$$\mu_k^c = \mu_k^x + \Sigma_k^{xt} (\Sigma_k^t)^{-1} (t - \mu_k^t), \quad (2.26)$$

$$\Sigma_k^c = \Sigma_k^x - \Sigma_k^{xt} (\Sigma_k^t)^{-1} (\Sigma_k^{xt})^T. \quad (2.27)$$

Furthermore, using properties of Gaussians, Eq. 2.22 can be further simplified by approximating the GMM with a single Gaussian:

$$p(x|t) \cong \mathcal{N}(x; \bar{\mu}_x, \bar{\Sigma}_x), \quad (2.28)$$

where  $\bar{\mu}_x = \sum_{k=1}^K \pi_k^c \mu_k^c$  and  $\bar{\Sigma}_x = \sum_{k=1}^K (\pi_k^c)^2 \Sigma_k^c$ .



**Fig. 2.7** Examples of a four dimensional time-series generated from a Gaussian Mixture Regression model (the *four colors* represent the four dimensions). The mean and covariance matrices were taken from the data provided in Calinon et al. (2006) for the first demo of Learning from Demonstration using GMM/GMR (Color in online)

A time-series can then be generated from this GMM by sampling from  $p(x|t)$  for every value of  $t$  from 1 to the desired  $T$ . This procedure is implemented in the toolbox using the function `generateGMR()`. Figure 2.7 shows an example time-series generated using this function.

GMR was originally designed as a regression algorithm that uses a GMM learned from example data points to smoothly find the value of learned function at unseen points. Using it for data generation as we did in this example is not a standard procedure and is not even recommended. The main disadvantage for using GMR for generation is that the final step generates the data at every time-step independent from the data at nearby time-steps. This results in rough functions as seen in Fig. 2.7. This problem can be alleviated if the correlation between nearby points is taken into account which is what can be achieved by Gaussian Processes.

### 2.2.10 Gaussian Processes

HMM and GMM/GMR provide two alternative statistical methods for modeling time-series generation. The main problem with HMM is the need to select an appropriate number of states in advance. This problem can be somehow alleviated by using model selection techniques (e.g. Bayesian Information Criteria) if we have time-series data to use as example for the system to be modeled. This requires though multiple trainings with different values for the number of states.

GMM/GMR modeling discussed in the previous section had the problem of independence between successive time steps even though it could capture the covariance between different dimensions at the same time-step.

Gaussian Processes can overcome both of these problems by directly modeling the correlation between different dimensions of the time-series at all time-steps. It requires no selection of a number-of-states and can generate smooth time-series output.

A Gaussian Process ( $\mathcal{GP}$ ) is defined as a collection of random variables (possibly infinite), any finite number of which have a joint Gaussian distribution (Rasmussen and Williams 2006).

A  $\mathcal{GP}$  is completely specified by its mean function  $m(x)$  and covariance functions  $k(x, x')$  both are direct extensions of the mean and covariance matrix of a standard multivariate Gaussian distribution. Notice that for this section we use  $x$  as the independent variable instead of  $t$  as the  $\mathcal{GP}$  formalism is general and can handle any kind of real valued independent variable not only time.

The standard formalism for Gaussian Processes is:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')), \quad (2.29)$$

where the mean and covariance matrices are defined as:

$$\begin{aligned} m(x) &= \mathbb{E}(f(x)), \\ k(x, x') &= \mathbb{E}((f(x) - m(x))(f(x') - m(x'))). \end{aligned} \quad (2.30)$$

Given a Gaussian Process, it is easy to generate time-series by sampling at different values of  $x$ . For example, consider the case where  $m(x) = 0$  and the covariance function is defined as:

$$k(x, x') = k_{se}(x, x') = \lambda e^{-(x-x')^T(x-x')/l}. \quad (2.31)$$

This covariance function is called the squared exponential and results in smooth time-series. We will use  $\lambda = 1$  and  $l = 0.5$  for our example. To generate a time-series spanning the time from 0 to 20, we select a sampling rate (100 in our case) and then calculate the mean of the function to be sampled which will equal to  $\mu = m(0 : 0.01 : 20)$  then calculate the covariance matrix at the values of the input  $x = 0 : 0.01 : 20$  by applying Eq. 2.31 for each pair of values in this set. The resulting covariance matrix  $\Sigma$  is then used with the mean  $\mu$  to generate multivariate Gaussian values. This can easily be achieved by finding the Eigen Value Decomposition of  $\Sigma$  and sampling from independent Gaussians on this basis then projecting back using  $V\lambda^{-1/2}$  where  $V$  is the set of Eigen vectors of  $\Sigma$  and  $\lambda$  is the set of Eigen values. This is implemented in *grand()* in the Toolbox. Figure 2.8 shows five example time-series sampled using this method.

It is trivial to add noise to this system by adding it directly to the mean vector  $\mu$  during sampling.

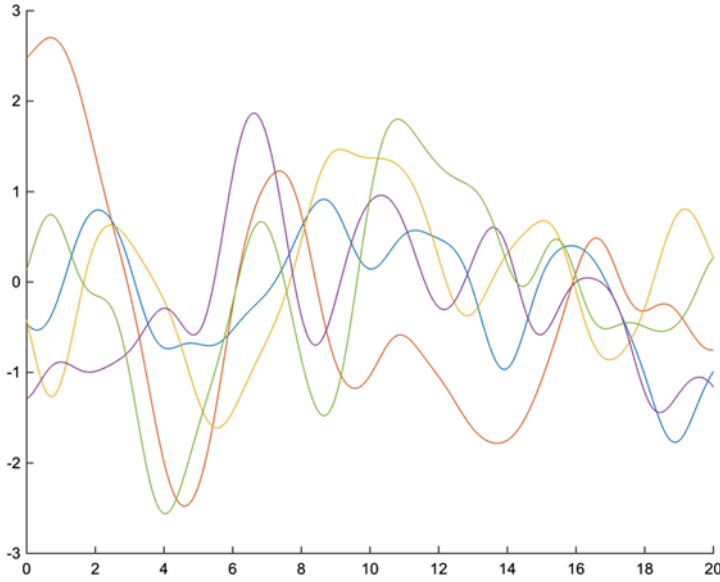
Figure 2.9a shows a set of functions sampled from a 1D GP with the squared exponential covariance function (after adding the Gaussian noise). An important advantage of GPs is that it can update its internal generation model incrementally and with few data points. For example, Fig. 2.9b–d shows the mean and variance of the same GP used in Fig. 2.9a after fixing three points in succession (shown as red circles). It is clear that, even with few data points, the GP can easily learn a model that can be used to interpolate and extrapolate the function for all time. To achieve this incremental learning, GPs exploit the properties of Gaussians specially the decomposability of the covariance matrix. Using our previous notation:

$$\text{cov}(y_i, y_j) = \lambda e^{\|x_i - x_j\|^2/l} + \sigma_n \delta_{ij}, \quad (2.32)$$

where  $\delta_{ij}$  is the Kronecker delta function. In matrix notation this leads to:

$$\text{cov}(\mathbf{y}) = K_{XX} + \sigma_n^2 I, \quad (2.33)$$

where  $\mathbf{y}$  is a column vector of the outputs observed and  $K_{XX}$  is the covariance matrix calculated by applying Eq. 2.31 to all pairs of inputs. Notice that the noise model is decoupled and iid which is expected for a Gaussian noise generation process.



**Fig. 2.8** Five examples of time-series sampled from a GP with zero mean and squared exponential covariance function with unity parameters

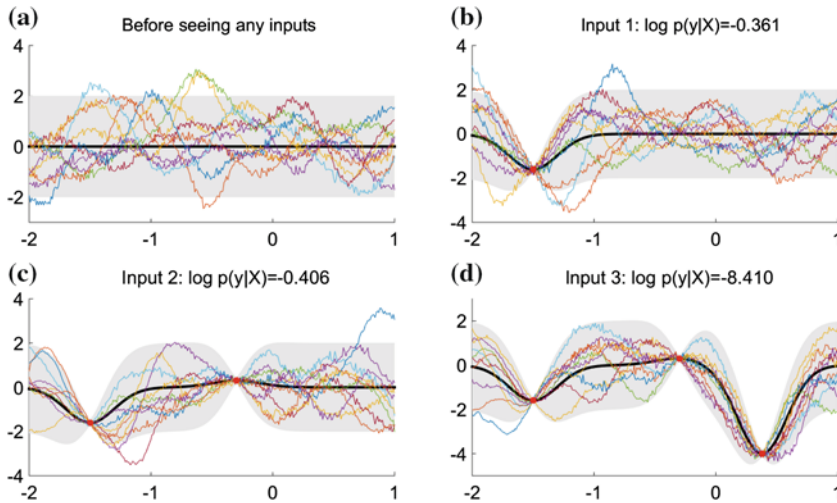
Now given a set of input output pairs  $(X, y)$  and a test point  $x_t$ , we can calculate the distribution of the output at this test point using the rules for conditioning Gaussian distributions as follows:

$$p(f_t | X, y, x_t) = \mathcal{N} \left( K_{x_t X} (K_{XX} + \sigma_n^2 I)^{-1} y, K_{x_t x_t} - K_{x_t X} (K_{XX} + \sigma_n^2 I)^{-1} K_{X X}^T \right). \quad (2.34)$$

Equation 2.34 was used to generate the estimates shown in Fig. 2.9. Most applications of GPs use the squared exponential function which is infinitely differentiable. This may lead to over-smoothing of the output and makes it harder for the system to represent abrupt changes in system dynamics due for example to a sudden change in the load for a manipulator. One example covariance function that can handle sudden changes better than the squared exponential is the Matern covariance function which has the form:

$$k_{\text{Matern}}(x, x') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left( \frac{\sqrt{2\nu} |x - x'|}{l} \right)^\nu K_\nu \left( \frac{\sqrt{2\nu} |x - x'|}{l} \right). \quad (2.35)$$

There are many possible covariance functions other than the squared exponential and Matern and they can be combined to lead to a large variety of generation schemes (for several examples, please refer to Rasmussen and Williams 2006).



**Fig. 2.9** Sample functions from a SE Gaussian Process showing the adaptation to input data and the limitation on the variance even outside the seen samples. The **bold black** signal represents the mean and other signals represent samples from the process. The *gray color* shows two standard deviations at each point (Color in online)

The GP generation process is implemented in the toolbox using a set of functions. The function *grand()* can be used to generate a time-series from a Gaussian Process given the mean and covariance matrices corresponding to the time-steps. To generate these matrices, you can use either *generateGP()* to generate a GP from a set of sample points or the functions *createGP()* to create a GP with no restrictions similar to the one used to generate the time-series shown in Fig. 2.8. The function *add2GP()* can be used to add points of known values to the GP and *generateFromGP()* can be used—at any time—to create the mean and covariances matrices needed for *grand()*.

The implementation of GP in the *MC<sup>2</sup>* toolbox accompanying this book is minimal with only one covariance function (the squared exponential) and two mean functions (linear and sinusoidal) and with no optimization tricks in the implementations to facilitate understanding. These can easily be extended. There are several GP toolboxes for Matlab that implement more covariance functions already implemented and with more advanced GP algorithms for the interested reader. The information given here will, nevertheless, be enough for all our purposes in this book.

## 2.3 Representation and Transformations

In many cases, it is desirable to represent the time-series differently from the direct representation presented in Definition 2.1. The most effective representation scheme depends largely on the application. This section introduces some of the most useful

representations for time-series. We will focus first on single-dimensional time-series. Multidimensional extensions will be discussed later.

### 2.3.1 Piecewise Aggregate Approximation

Piecewise Aggregate Approximation (PAA) is one of the simplest time-series compression and representation approaches. The idea of PAA is very simple. Rather than keeping the whole time-series, it is divided into equal length segments and we keep only the mean of each segment. These means are then concatenated to form another time-series that *represents* the original time-series albeit being shorter. This can be expressed as follows assuming that the segment length is  $m$ :

$$x_i^{paa} = \frac{1}{m_i} \sum_{j=im}^{\min(T-1, (i+1)m-1)} x_j, \quad (2.36)$$

for  $0 \leq i \leq \lceil T/m \rceil$  where  $m_i = m$  for all  $i$  but the last and equals  $T - m \times \lfloor T/m \rfloor$  for the last point of  $X^{paa}$ .

Assuming that the segment length is  $m$  then the output of PAA is  $m$  times shorter than the original time-series. The PAA representation requires a single parameter from the user, which is the length of the segment  $m$ . In the extreme case where  $m = 1$ , PAA simply returns the input time-series without any change. In the other extreme case when  $m = T$  where  $T$  is the length of the input time-series, PAA returns a single number representing the mean of the time-series (i.e. the DC or zero-frequency component).

Despite its simplicity, PAA can be very useful in many applications. For example, it can be shown that the Euclidean distance between any two time-series is larger than or equal the Euclidean distance between their PAA representations:

$$\sum (x_i - y_i)^2 \geq \sum (x_i^{paa} - y_i^{paa})^2. \quad (2.37)$$

This property (called lower bounding) allows linear time generation of indexing structures that can be used to search through large numbers of time-series efficiently. Another important feature of the PAA transform or representation is that all calculations are local which means that it can be applied to the data in real time.

It is also trivial to extend PAA to the multidimensional case. Simply apply PAA to each dimension of the input time-series.

A simple extension of PAA that we do not utilize much in this book is called Adaptive Piecewise Constant Approximation (APCA) in which the length of each segment is not fixed but adapted to the time-series.

**Table 2.1** The location of break points for the first 5 cases of alphabet sizes ( $n_a$ )

$n_a$	1	2	3	4	5
$\beta_a$					
$\beta_1$	−0.43	−0.67	−0.84	−0.97	−1.07
$\beta_2$	0.43	0	−0.25	−0.43	−0.57
$\beta_3$	−	0.67	0.25	−0	−0.18
$\beta_4$	−	−	0.84	0.43	0.18
$\beta_5$	−	−	−	0.84	0.57
$\beta_6$	−	−	−	−	1.07

A more complete table (up to  $n_a = 10$ ) can be found in (Lin et al. 2003)

### 2.3.2 Symbolic Aggregate Approximation

Symbolic Aggregate approXimation (SAX) is a transformation algorithm that converts a single-dimensional time-series to a string. The main advantage of SAX is that it is designed to achieve equiprobable symbol production. Because it is based on PAA, it also provides a lower bound on the Euclidean distance. SAX was proposed by Lin et al. (2003) and since then became one of the most widely used transformation for data mining of time-series data.

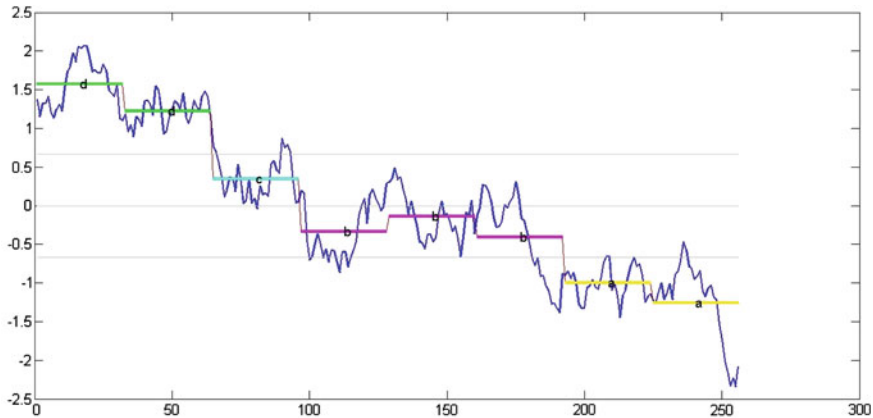
SAX is built upon the finding that z-score normalized time-series have Gaussian distributions (Larsen and Marx 1986). Based on that, the normal distribution is divided into bands where the probability of falling within all bands is the same. The number of bands is selected to equal to the number of symbols in the alphabet to be used for discretization ( $n_a$ ). The limits of these bands are called *break points*  $\beta_a$  where  $a \in \{1, 2, \dots, n_a - 1\}$ . These break points can be calculated easily from tables of cumulative normal function. Table 2.1 shows the break points for the first five alphabet sizes.

The locations of the break points can be calculated and stored for any finite alphabet size.

Given these break point locations and the alphabet size ( $n_a$ ), we can transform a single-dimensional time-series into a string by first z-score normalizing it (i.e. subtracting the mean from each point and dividing the standard deviation). The normalized time-series is then transformed to a shorter time-series using PAA (See Sect. 2.3.1). The break points corresponding to the chosen alphabet size are then used to map values of this shorter time-series into symbols by issuing the symbol corresponding to the band within which each PAA value lies.

The implementation of this transform in  $MC^2$  is encapsulated by the function *time-series2symbol()* which is provided by Lin et al. (2003). Figure 2.10 shows the processing steps of this function leading to the final string representation of the time-series.

The aforementioned SAX transform assumes that the input is a single-dimensional time-series. Mohammad and Nishida (2014) proposed three different extensions of this algorithm to multi-dimensional time-series. There are several methods to extend



**Fig. 2.10** Example time-series and its transformation steps using SAX. The original time-series is shown as well as its PAA transformation. The break points corresponding to a 4-letters alphabet are also shown (light gray). Finally the final symbols from the set  $\{ddcbbbaa\}$  is shown

SAX to handle multidimensional time-series. The first approach is not to modify SAX in any way and start by reducing the dimensionality of the time-series to one then apply the traditional SAX transformation. This approach is called SAX-PCA because we use Principal Component Analysis (PCA) for dimensionality reduction.

We start by z-score normalizing each dimension then applying Singular Value Decomposition (SVD) to represent the time-series  $USV^T = X$ . The output single dimensional time-series  $x$  is then obtained by projecting the time-series on the first singular vector:  $x = U_1X$ , where  $U_1$  is the singular vector of  $U$  corresponding to the largest singular value in  $S$ . The time-series  $x$  is then passed as an input to the traditional SAX algorithm.

Another, even simpler, approach is to apply SAX to every dimension of the data separately and then combine the resulting string by assigning every possible combination of symbols in the resulting  $D$  strings a unique identifier. This leads to a string of length  $N$  but from an extended alphabet of length  $M^D$ . In most cases, most of the symbols of this extended alphabet will not appear in the final string. To keep the requirement that the final string is from an  $M$ -symbols alphabet, we can simply cluster the resulting characters into  $M$  clusters using K-means and replace each character with the centroid of its cluster. This approach is called SAX-REPEAT.

A third approach (called SAX-z-score hereafter) is to modify the normalization step in SAX to use an extended multidimensional version of the z-score normalizer. This can be done by calculating an intermediate time-series  $\hat{x}$  that has a zero-mean unit-variance distribution. The PAA step can then be applied to the  $L_2$  norm of the data. Assuming that the covariance matrix of  $X$  is  $C$  and that the mean of its columns is  $\mu$ , the intermediate time-series is calculated as:

$$\hat{x}(t) = C^{-1/2}(X - \mu). \quad (2.38)$$

The PAA step is then modified to be:

$$\bar{x}(n) = \frac{N}{T} \sum_{t=\frac{T}{N}(n-1)+1}^{\frac{T}{N}} \|\hat{x}(t)\|_2. \quad (2.39)$$

Multidimensional SAX (MSAX) provides a fast way to encode the time-series information in a symbolic way which opens the way for utilizing existing text-retrieval and manipulation techniques.

### 2.3.3 Discrete Fourier Transform

PAA represents the time-series with a shorter one that has the same independent variable (usually time). A completely different approach is taken by the Discrete Fourier Transform (DFT) which represents the time-series using a different yet related independent variable. If the independent variable of the original time series was time, then the transformed version will be represented using frequency as the independent variable.

DFT works with complex time-series. Given a time-series  $X$ , we can calculate its DFT transform using:

$$x_i^{dft} = \sum_{k=0}^{T-1} x_k e^{-j2\pi ik/T}, \quad (2.40)$$

where  $j^2 = -1$  and the time-series length is  $T$  as usual.

It is clear that the transform is not local in the sense that calculating the value of  $x_i^{dft}$  depends on ALL the values of  $X$ . This means that—in contrast to PAA for example—the transform cannot be implemented incrementally. Even though that naive implementation of DFT requires  $O(T^2)$  operations, implementations based on the Fast Fourier Transform (FFT) can achieve time complexity of  $O(T \log T)$ .

DFT is used extensively in the digital signal processing community and has several applications. For example, keeping only 10% of the output can preserve 90% of signal energy of random walks which makes DFT a good candidate for compression applications.

As with the case in PAA, DFT provides a lower bound on Euclidean distance:

$$\sum (x_i - y_i)^2 \geq \sum (|x_i^{dft}| - |y_i^{dft}|)^2. \quad (2.41)$$

This can be the basis of an indexing application and it usually achieves speedups between 3–100 compared with using the original time-series. Extending DFT to multiple dimensions can be achieved by applying the same transform to each dimension of the input.

### 2.3.4 Discrete Wavelet Transform

PAA was clearly local while DFT was a clearly global transform of the time-series. Some transforms (wavelet transforms) give a hierarchical view of the time series allowing it to be examined at multiple resolutions. There are several wavelet transforms but all share the common feature of having a function called the mother wavelet  $\psi$  that is translated and scaled to get other similar functions on a different scale and/or position called child wavelets according to the following equation:

$$\psi(t)_{\tau,s} = \frac{1}{\sqrt{s}} \psi\left(\frac{t-\tau}{s}\right). \quad (2.42)$$

The parameter  $s$  controls the scale while the parameter  $\tau$  controls the positioning of the wavelet. The simplest yet most widely used wavelet transform in time-series mining is the *Haar* wavelet. The mother wavelet of the Haar transform is defined as:

$$\psi^{Haar}(t) = \begin{cases} 1 & 0 \leq t < 0.5 \\ -1 & 0.5 \leq t < 1 \\ 0 & \text{otherwise} \end{cases}. \quad (2.43)$$

Applying this wavelet to any time-series is very efficient and can be executed in linear time using only summation and difference operators between pairs of points. Given a time-series  $X$  of length  $T$  which is assumed to be a power of 2, we start by finding the averages and differences of all consecutive pairs of points according to:

$$x_t^0 = x_{2t+1} - x_{2t}, \quad (2.44)$$

for  $0 \leq t \leq T/2$ . Another temporary time-series is calculated using the sum operator applied to the same pairs

$$\bar{x}_t^0 = x_{2t+1} + x_{2t}, \quad (2.45)$$

for  $0 \leq t \leq T/2$ . The same two operations are applied again to  $\bar{X}^0$  leading to:

$$x_t^1 = \bar{x}_{2t+1}^0 - \bar{x}_{2t}^0, \quad (2.46)$$

$$\bar{x}_t^1 = \bar{x}_{2t+1}^0 + \bar{x}_{2t}^0. \quad (2.47)$$

Now  $t$  will range from 0 to  $T/4$ .

In general:

$$x_t^i = \bar{x}_{2t+1}^{i-1} - \bar{x}_{2t}^{i-1}, \quad (2.48)$$

$$\bar{x}_t^i = \bar{x}_{2t+1}^{i-1} + \bar{x}_{2t}^{i-1}, \quad (2.49)$$

until we get a time-series of a single point for both  $X^i$  and  $\bar{X}^i$  which will happen when  $i = \log_2 T$ . The time-series named  $X^i$  are then concatenated to give the Haar transform of the original time-series:

$$X^{Haar} = (X^0, X^1, \dots, X^{\log_2 T}). \quad (2.50)$$

Notice that each successive  $X^i$  gives a view of the original time-series that is progressively coarser.

Again, multidimensional time-series can be treated by simply applying the aforementioned transformation on every dimension independently.

The main advantages of Haar transform are simplicity of calculation and the ability to *inspect* the time-series at different resolutions which may reveal for example hierarchical repeated pattern (an important application for social robotics).

### 2.3.5 Singular Spectrum Analysis

Singular spectrum analysis (SSA) is a relatively new approach for analyzing and representing time-series data. This approach has several advantages. Firstly, it can be applied to any time-series either of a single or multiple dimensions and it can be applied in a single scan of the time series to generate a representation that requires a predefined multiple of the original time-series length. Secondly, the approach is flexible enough to allow both unsupervised and semi-supervised analysis of the data. Moreover, unlike Fourier analysis for example, it is data driven even in deciding the basis functions used in the decomposition of the input time-series. Compare that to the Haar transform which as we have just seen uses a predefined basis function that is applied repeatedly to the time-series.

Singular spectrum analysis has seen increased interest in the current century and has found applications in several areas including forecasting, noise rejection, causality analysis and change point discovery among many other approaches (Elsner and Tsonis 2013). This section will introduce the technique showing how to use it for time-series representation. Most of the change point discovery techniques used in this book (See Chap. 3) are based on this representation.

The main idea behind SSA is to represent the signal through a set of data-driven basis functions that are generated by subdimensional representation of the Hankel matrix representing the original time-series. This subdimensional representation, if carefully selected, can remove unwanted signal components (e.g. noise) and can be used to discover the basic building blocks of the time-series.

Given a time-series  $X$ , SSA analysis generates a set of times-series  $X_i^{ssa}$  where each of these components can be identified as a trend, a periodic, a quasi-periodic or a noise component while allowing reconstruction of  $X$  as:

$$X = \sum X_i^{ssa}.$$

This means that SSA analysis is done in three stages: decomposition, selection, and reconstruction. At the first stage, the time-series is decomposed into its set of basis time-series. At the selection stage, we select only a subset of these basis time-series to use for reconstruction in the hope that they will have lower noise levels or reveal some underlying property of the original time-series (e.g. its trend). Finally, the selected basis time-series are used to reconstruct the original time-series. If not all basis time-series are selected in the second stage, the final reconstruction will not generate the exact input time-series but a modified version of it that highlights the purpose of the analysis.

The decomposition stage consists of two steps: Hankel matrix embedding and SVD decomposition. Given a time-series  $X$  of length  $T$ , the Hankel matrix embedding is the matrix  $H$  of size  $L \times K$  where  $L$  is called the lag parameter and must be between 2 and  $T$  and  $K = T - L + 1$ . Row  $k$  of  $H$  consists of the subsequence  $x_{i,L}$  in our terminology. This means that  $H$  is a Hankel matrix (i.e. it has the property that  $H_{i,j} = H_{i',j'}$  for  $i + j = i' + j'$ ) and  $H \in \mathbb{R}^{K \times L}$ .

Consider the following time-series:

$$X = [0.05 \ 0.49 \ 0.36 \ 0.90 \ 0.98 \ 0.87 \ 0.91 \ 0.84 \ 0.95 \ 0.59 \ -0.13 \ -0.01],$$

and a lag parameter ( $L$ ) equal to 3, the corresponding Hankel matrix is given by:

$$H = \begin{bmatrix} 0.05 & 0.49 & 0.36 \\ 0.49 & 0.36 & 0.90 \\ 0.36 & 0.90 & 0.98 \\ 0.90 & 0.98 & 0.87 \\ 0.98 & 0.87 & 0.91 \\ 0.87 & 0.91 & 0.84 \\ 0.91 & 0.84 & 0.95 \\ 0.84 & 0.95 & 0.59 \\ 0.95 & 0.59 & -0.13 \\ 0.59 & -0.13 & -0.01 \end{bmatrix}.$$

This Hankel matrix can be created in MATLAB using the `hankel()` function. Given the Hankel matrix  $H$ , we can create the set of basis functions  $X_i^{ssa}$  in two steps: SVD decomposition and Hankelization. SVD decomposition consists of simply applying Singular Value Decomposition to the Hankel matrix (e.g. using `svd()` in Matlab). Mathematically this corresponds to finding  $U \in \mathbb{R}^{K \times K}$ ,  $S \in \mathbb{R}^{K \times L}$ ,  $V \in \mathbb{R}^{L \times L}$  where:

$$H = USV^T. \quad (2.51)$$

In our previous example, we get:

$$U = \begin{bmatrix} -0.13 & -0.22 & 0.35 & -0.37 & -0.28 & -0.33 & -0.24 & -0.45 & -0.48 & 0.08 \\ -0.26 & -0.26 & -0.54 & -0.09 & -0.32 & -0.15 & -0.32 & 0.10 & 0.13 & -0.56 \\ -0.33 & -0.43 & 0.20 & -0.09 & -0.04 & -0.08 & -0.11 & 0.06 & 0.66 & 0.44 \\ -0.41 & 0.00 & 0.08 & 0.86 & -0.10 & -0.12 & -0.09 & -0.16 & -0.15 & 0.04 \\ -0.41 & 0.05 & -0.17 & -0.15 & 0.84 & -0.15 & -0.15 & -0.14 & -0.10 & -0.05 \\ -0.39 & 0.01 & 0.01 & -0.13 & -0.11 & 0.88 & -0.10 & -0.15 & -0.13 & 0.02 \\ -0.40 & -0.03 & -0.18 & -0.15 & -0.15 & -0.14 & 0.85 & -0.12 & -0.04 & -0.01 \\ -0.35 & 0.15 & 0.29 & -0.12 & -0.07 & -0.10 & -0.05 & 0.81 & -0.29 & 0.02 \\ -0.21 & 0.71 & 0.31 & -0.12 & -0.14 & -0.12 & -0.08 & -0.23 & 0.42 & -0.28 \\ -0.06 & 0.42 & -0.55 & -0.09 & -0.22 & -0.12 & -0.21 & -0.01 & -0.08 & 0.63 \end{bmatrix},$$

$$S = \begin{bmatrix} 3.92 & 0.00 & 0.00 \\ 0.00 & 1.06 & 0.00 \\ 0.00 & 0.00 & 0.59 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 \end{bmatrix},$$

$$V = \begin{bmatrix} -0.57 & 0.73 & -0.37 \\ -0.60 & -0.06 & 0.80 \\ -0.56 & -0.68 & -0.47 \end{bmatrix}.$$

It can easily be shown that Eq. 2.11 holds for these matrices. The following properties hold for all SVD decompositions assuming that  $A_i$  is column  $i$  of matrix  $A$ ,  $A_{i,j}$  is element  $j$  of vector  $A_i$ , and  $\langle V, W \rangle$  is the dot product of vectors  $V$  and  $W$ :

$$\langle U_i, U_j \rangle = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}, \quad (2.52)$$

$$\langle V_i, V_j \rangle = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}. \quad (2.53)$$

This means that the sets of vectors  $U_i$  and  $V_i$  each form an orthonormal vector set.

$$S_{i,j} = \begin{cases} \sqrt{\lambda_i} & i = j \wedge i, j \leq \min(K, L) \\ 0 & \text{otherwise} \end{cases}, \quad (2.54)$$

where  $\lambda_i$  is the  $i$ th Eigen vector of the matrix  $HH^T$ . Notice that at most  $\min(K, L)$  elements are nonzero.  $S_{i,i}$  are called the singular values of  $H$ . We will always assume that  $S_{i,i} \geq S_{j,j}$  for  $i \geq j$  and will use  $S_{i,i}$  and  $s_i$  interchangeably. The number of nonzero

singular values are denoted  $d$  hereafter. In our running example,  $s_1, s_2$  and  $s_3$  are all nonzero which means that  $d = 3$ . The value for  $d$  will always be less than or equal  $\min(K, L)$  and represents the rank of the original matrix  $H$ .

The matrix  $H$  can then be expressed as a summation ( $H = \sum_{i=1}^d H^i$ ) where:

$$H^i = s_i U_i V_i^T. \quad (2.55)$$

The tuple  $(s_i, U_i, V_i)$  is called the  $i$ th Eigen triple in SSA literature (Hassani 2007). The vectors  $U_i$  are called *factor empirical orthogonal functions* or EOFs, and the vectors  $V_i$  are called *principal components*.

For our running example, we can calculate  $H^1, H^2$ , and  $H^3$  from Eq. 2.55 to be:

$$H^1 = \begin{bmatrix} 0.30 & 0.32 & 0.30 \\ 0.57 & 0.60 & 0.56 \\ 0.74 & 0.77 & 0.73 \\ 0.91 & 0.95 & 0.89 \\ 0.91 & 0.95 & 0.90 \\ 0.87 & 0.90 & 0.85 \\ 0.89 & 0.93 & 0.87 \\ 0.79 & 0.82 & 0.77 \\ 0.47 & 0.49 & 0.46 \\ 0.14 & 0.15 & 0.14 \end{bmatrix},$$

$$H^2 = \begin{bmatrix} -0.17 & 0.01 & 0.16 \\ -0.20 & 0.02 & 0.18 \\ -0.33 & 0.03 & 0.31 \\ 0.00 & -0.00 & -0.00 \\ 0.04 & -0.00 & -0.03 \\ 0.01 & -0.00 & -0.00 \\ -0.02 & 0.00 & 0.02 \\ 0.12 & -0.01 & -0.11 \\ 0.55 & -0.04 & -0.51 \\ 0.32 & -0.03 & -0.30 \end{bmatrix},$$

$$H^3 = \begin{bmatrix} -0.08 & 0.16 & -0.10 \\ 0.12 & -0.25 & 0.15 \\ -0.04 & 0.10 & -0.06 \\ -0.02 & 0.04 & -0.02 \\ 0.04 & -0.08 & 0.05 \\ -0.00 & 0.00 & -0.00 \\ 0.04 & -0.09 & 0.05 \\ -0.06 & 0.13 & -0.08 \\ -0.07 & 0.14 & -0.08 \\ 0.12 & -0.26 & 0.15 \end{bmatrix}.$$

Notice for now that all of these matrices are not Hankel matrices. From linear Algebra, it is known that the Frobenius norm of a matrix can be found from its singular values using the following equation:

$$\|H\|^2 = \sum_{i=1}^K \sum_{j=1}^L (H_{i,j})^2 = \sum_{i=1}^d \lambda_i = \sum_{i=1}^d s_i^2. \quad (2.56)$$

For our running example the Frobenius norm of  $H$  calculated any way of the three ways in Eq. 2.56 will equal 16.8045. Moreover, we know that each one of the expansion matrices ( $H^i$ ) is a rank one matrix (See how they are created according to Eq. 2.55). This means that they will have a single nonzero singular value which equals  $s_i$  because both  $U_i$  and  $V_i$  are unit vectors. This leads to:

$$\|H^i\| = s_i. \quad (2.57)$$

From Eqs. 2.56 and 2.57, it is straight forward to see that:

$$\|H\|^2 = \sum_{i=1}^d \|H^i\|^2. \quad (2.58)$$

If we define  $\zeta_i$  as the contribution of expansion matrix  $H^i$  to  $H$ , it is easy to see that

$$\zeta_i = \frac{s_i^2}{\sum_{j=1}^d s_j^2}. \quad (2.59)$$

In our running example,  $\zeta_1 = 0.9129$ ,  $\zeta_2 = 0.0667$ ,  $\zeta_3 = 0.0204$  which means that 91.29% of the total variance in the data is captured in the first expansion matrix. Notice that SVD is optimal in the sense that taking the  $K$  columns of  $U$  and  $V$  corresponding to the top  $K$  singular values, we can reconstruct  $\bar{H}$  with the guarantee that  $\|H - \bar{H}\|$  is minimum compared to any other linear decomposition. This means that for any value  $K$ , and assuming that the singular values are ordered in descending order, it is guaranteed that  $\left\|H - \sum_{k=1}^K H^k\right\|$  is minimal compared to any other linear decomposition of  $H$  to matrices.

Giving the decomposition of  $H$  into the set  $\{H^i\}$  and their weights  $\zeta_i$ , we can then generate approximations of  $H$  by just grouping together some of the expansion matrices using simple summation. This leads to the the following two-steps reconstruction process.

The first step of reconstruction is to group the expansion matrices then summing the matrices in each group. Given the  $d$  nonzero expansion matrices, we generate a set of sets representing the groups:  $\{I_1, I_2, \dots, I_g\}$  where  $1 \leq g \leq d$  is the number of groups and each  $I_i$  is a set  $\{i_1, \dots, i_{n_i}\}$  where  $1 \leq i_j \leq d$  and  $1 \leq j \leq n_i$ . The sets  $I_j$  are disjoint. Each one of these groups should correspond to some meaningful component of the original time-series. This process is called Eigen-triple grouping. The

expansion matrices of all members of a group are added to generate a matrix representing the group using Eq. 2.60. The contributions of all member expansion matrices in a group are simply added to get the total contribution of the group according to Eq. 2.61.

$$H^{I_i} = \sum_{j \in I_i} H^j. \quad (2.60)$$

$$\zeta_{I_i} = \sum_{j \in I_i} \zeta_j. \quad (2.61)$$

In our running example, we can create two groups  $\{I_1 = \{1, 2\}, I_2 = \{3\}\}$  with the following corresponding expansion matrices:

$$H^{I_1} = \begin{bmatrix} 0.13 & 0.33 & 0.46 \\ 0.38 & 0.61 & 0.75 \\ 0.41 & 0.80 & 1.04 \\ 0.91 & 0.95 & 0.89 \\ 0.95 & 0.95 & 0.86 \\ 0.87 & 0.90 & 0.85 \\ 0.87 & 0.93 & 0.89 \\ 0.91 & 0.81 & 0.66 \\ 1.01 & 0.44 & -0.05 \\ 0.47 & 0.12 & -0.16 \end{bmatrix},$$

$$H^{I_2} = \begin{bmatrix} -0.08 & 0.16 & -0.10 \\ 0.12 & -0.25 & 0.15 \\ -0.04 & 0.10 & -0.06 \\ -0.02 & 0.04 & -0.02 \\ 0.04 & -0.08 & 0.05 \\ -0.00 & 0.00 & -0.00 \\ 0.04 & -0.09 & 0.05 \\ -0.06 & 0.13 & -0.08 \\ -0.07 & 0.14 & -0.08 \\ 0.12 & -0.26 & 0.15 \end{bmatrix}.$$

The final step of reconstruction is to convert the matrices corresponding to groups into time-series that comprise the final expansion of the input time-series. To achieve that, the expansion matrix corresponding to each group ( $H^{I_i}$ ) is converted into a Hankel matrix by averaging all members that have row and column numbers summing to the same integer. In our running example this leads to the following two hankelized matrices:

$$\bar{H}^{I_1} = \begin{bmatrix} 0.13 & 0.35 & 0.49 \\ 0.35 & 0.49 & 0.82 \\ 0.49 & 0.82 & 0.98 \\ 0.82 & 0.98 & 0.90 \\ 0.98 & 0.90 & 0.88 \\ 0.90 & 0.88 & 0.89 \\ 0.88 & 0.89 & 0.91 \\ 0.89 & 0.91 & 0.52 \\ 0.91 & 0.52 & 0.04 \\ 0.52 & 0.04 & -0.16 \end{bmatrix},$$

$$\bar{H}^{I_2} = \begin{bmatrix} -0.08 & 0.14 & -0.13 \\ 0.14 & -0.13 & 0.08 \\ -0.13 & 0.08 & 0.01 \\ 0.08 & 0.01 & -0.03 \\ 0.01 & -0.03 & 0.03 \\ -0.03 & 0.03 & -0.05 \\ 0.03 & -0.05 & 0.04 \\ -0.05 & 0.04 & 0.06 \\ 0.04 & 0.06 & -0.17 \\ 0.06 & -0.17 & 0.15 \end{bmatrix}.$$

It is now trivial to read-off the corresponding two expansion time-series from the hankelized matrices which leads to:

$$X^1 = [0.13 \ 0.35 \ 0.49 \ 0.82 \ 0.98 \ 0.90 \ 0.88 \ 0.89 \ 0.91 \ 0.52 \ 0.04 \ -0.16],$$

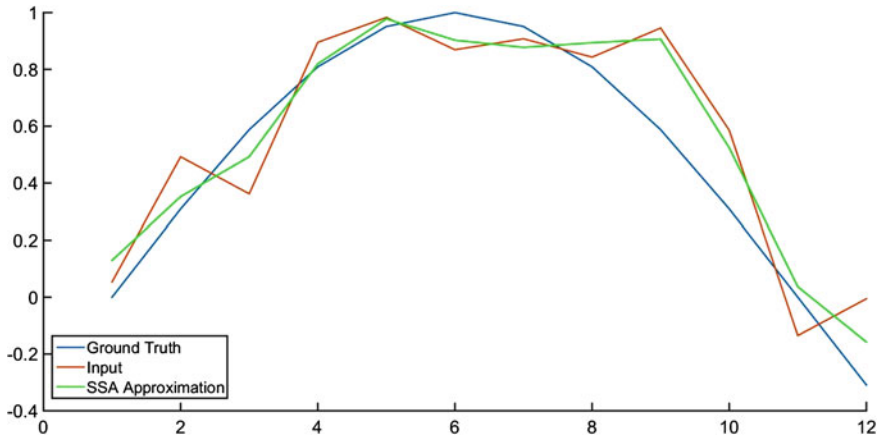
$$X^2 = [-0.08 \ 0.14 \ -0.13 \ 0.08 \ 0.01 \ -0.03 \ 0.03 \ -0.05 \ 0.04 \ 0.06 \ -0.17 \ 0.15].$$

It can easily be confirmed that the summation of these two time-series gives the original time-series. Notice that  $X^2$  is very near to zero and fluctuates between small negative and positive numbers. This corresponds to an estimation of the noise on the time-series. The actual noise in this case was:

$$[0.05 \ 0.18 \ -0.23 \ 0.09 \ 0.03 \ -0.13 \ -0.04 \ 0.03 \ 0.36 \ 0.28 \ -0.13 \ 0.30].$$

We can compare the Euclidean norm of the difference between the original noisy time-series and its ground truth to be 0.6567 while the Euclidean norm between the first expansion time-series ( $X^1$ ) and the ground-truth is only 0.3331 which means that SSA analysis was able to remove around 49 % of the noise in the original time-series. Figure 2.11 shows the ground-truth, original time-series, and the time-series resulting from hankelizing  $H^{I_1}$  which is SSA's best approximation to the ground truth.

Even though the example we used in this section was small with very small  $T$  and  $L$  values, the extension to longer time-series and larger Hankel matrices is trivial.



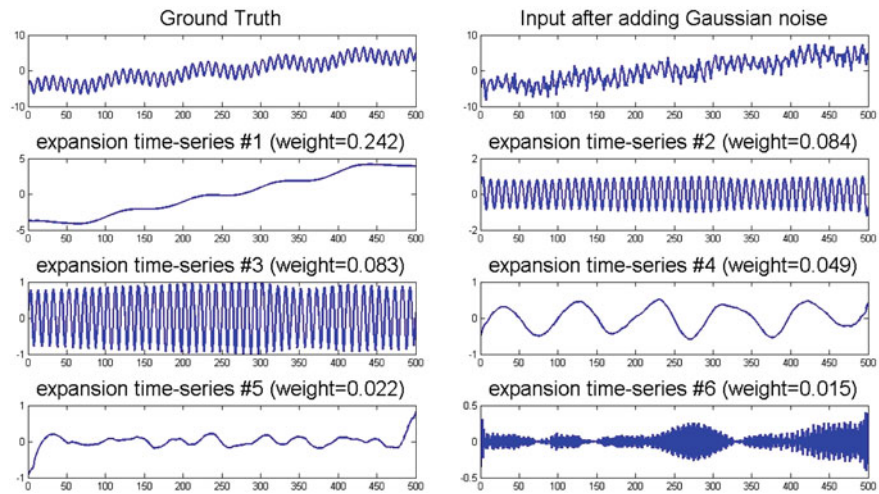
**Fig. 2.11** Example time-series and its corresponding SSA analysis. See text for details

The problem in this case is that the number of non-zero singular values  $d$  increases dramatically with increased  $L$ . In most cases, many of these singular values will be zeros or very near to zeros and they can be ignored as corresponding to noise in the input (similar to  $s_3$  in our running example). The remaining expansion matrices then will need to be grouped intelligently to generate meaningful expansion. There is no optimal grouping criterion that works for all applications and in this book we mostly will use simple grouping strategies like grouping the first expansion matrices with total weights over some threshold together in a single group and ignoring everything else.

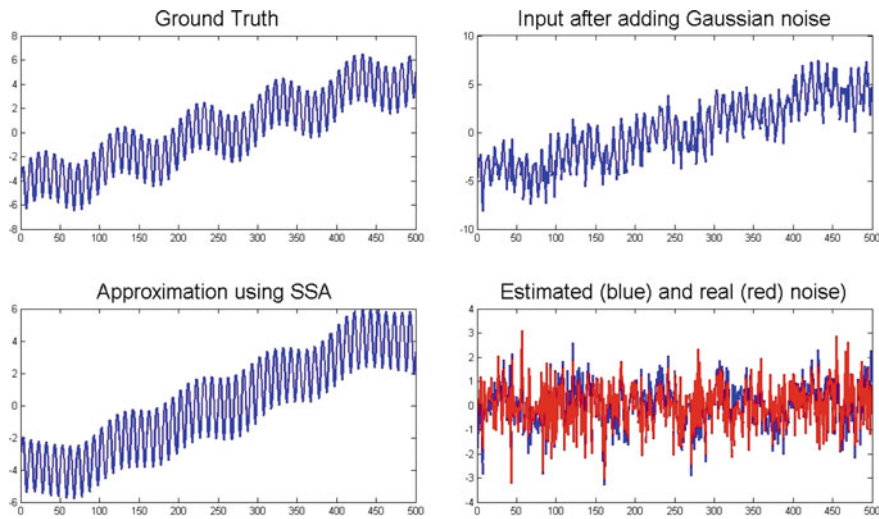
A slightly more complex example is shown in Figs. 2.12 and 2.13. The ground truth time-series consists of a linear trend and two sinusoidal as defined by Eq. 2.62 which is shown in Fig. 2.12a (Top-left).

$$x(t) = \sin(0.02t\pi) + 2\sin(0.2t\pi) + 0.02t \quad (2.62)$$

Gaussian noise with zero mean and unit variance is then added to generate the input signal to the SSA algorithm as shown in Fig. 2.12b (Top-right). The major features of the ground truth signal including the trend and the oscillating components are still preserved. We then applied SSA with a lag parameter of 50 to this time-series. The top six expansion time-series (without grouping) are shown in Fig. 2.12 (second to last rows) along with their weights. The trend is visible in the first expansion time-series. The high frequency sinusoidal is clear in the second and third components and the low frequency sinusoidal (despite being distorted) is visible in the fourth expansion series. notice that the sixth expansion time-series has much lower peak-to-peak (PP) value compared with the five before it and smaller weight signaling that it is a noise component.



**Fig. 2.12** Example time-series and its corresponding SSA analysis. See text for details



**Fig. 2.13** Example time-series and its corresponding SSA approximation using two groups (signal and noise approximations). See text for details

We then used the very simple grouping strategy of combining all expansion matrices with weights summing up to 45 % of the total in an approximation of the input time-series and combine the rest in another time-series estimating the noise. These are the two signals shown in Fig. 2.13’s second row.

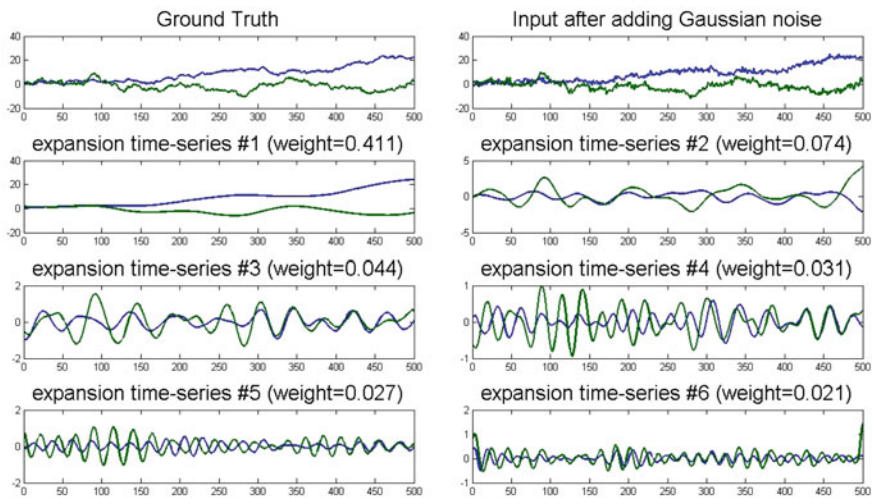
It is clear that the SSA approximation could recover most of the information in the ground-truth while eliminating the additive Gaussian noise. To quantify this intuition, we calculated the Euclidean distance between the ground truth and the

input time-series to be 21.0518. The Euclidean distance between the approximated signal and the ground truth is only 11.2344. This suggests that the SSA analysis could, again, eliminate around half of the distance introduced by the noise.

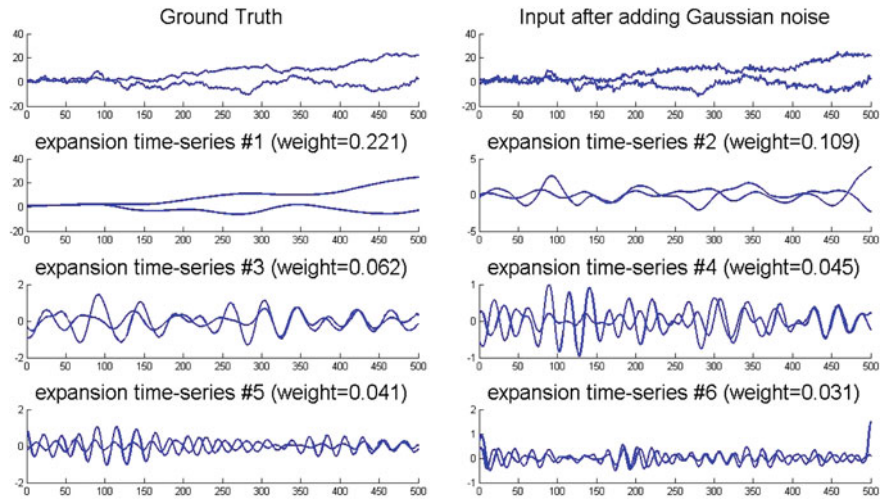
In general it is advisable to use the largest possible lag parameter value less than half the time-series (in order not to have more columns than rows in  $H$ ). This shows one problem with SSA analysis. The complexity of most implementations of SVD is cubic in matrix dimensions when they are similar which is the case in SSA. This means that for long time-series, SSA application may be prohibitively time-consuming. Several approaches have been proposed to deal with this problem but they are outside the scope of this book.

Extending SSA to multidimensional time-series is straight forward. We simply stack the Hankel matrices of all dimensions together to generate a combined Hankel matrix. This matrix is then fed to the rest of the algorithm as it is without any need to modify any parts except the hankelization process which now averages the corresponding parts of each dimension alone. This algorithm is called SSAM hereafter. SSAM is not the same as applying SSA to each dimension and combining the results because  $V$  will be shared between all the dimensions.

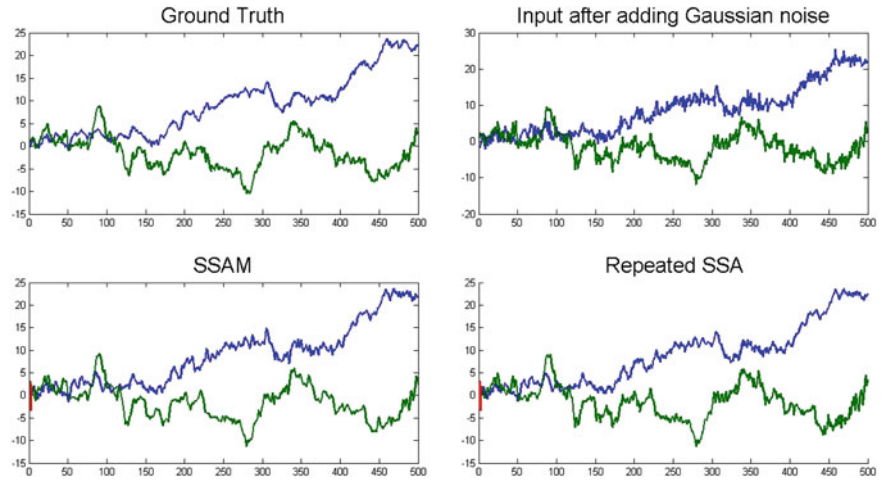
Figure 2.14 shows a time-series generated from a markov chain with Gaussian noise added and the first few expansion time-series using SSAM. Figure 2.15 shows application of repeated SSA to the two input dimensions of the same data used in Fig. 2.14. Even though the expansion time-series seem very similar, they are not identical. We can confirm that by comparing the result of grouping the first expansion matrices corresponding to a total weight of 0.75 in both cases as shown in the bottom row of Fig. 2.16. Did SSAM improve the results of repeated SSA? To answer this question we calculated the Euclidean distance between the ground truth and both



**Fig. 2.14** Example time-series and its corresponding SSAM analysis. See text for details



**Fig. 2.15** Example time-series and its corresponding repeated applications of SSA analysis. The *two colors* represent two dimensions of the time-series. See text for details



**Fig. 2.16** Example time-series and its corresponding SSA analysis using SSAM and repeated applications of single dimensional SSA analysis. The *two colors* represent two dimensions of the time-series. See text for details

inputs and output of SSAM and repeated SSA. For the input, the distance was 22.45. It was 17.24 for repeated SSA and 16.39 for SSAM. This shows that SSAM had slightly better approximation capability compared with repeated application of SSA. This was true because the covariance matrix used to generate the data had nonzero off-diagonal numbers resulting in correlations between the two-time series. SSAM can utilize these correlation because of the combined  $V$  matrix while repeated SSA application cannot.

From this example, it is clear that SSAM is expected to work well when the dimensions of the input time-series are correlated otherwise, repeated application of SSA may achieve higher approximation accuracy.

## 2.4 Learning Time-Series Models from Data

In data mining applications in general and in social robotics in particular, we rarely have access to a generation model in one of the simple mathematical forms presented in Sect. 2.2. In most cases, we only have data generated—or can be assumed to be generated—from one of these models and are interested in estimating the generation model either for predicting future behavior or simply to understand the generation process. This section will focus on methods for learning the generation model assuming that it falls under one of the most important models presented in Sect. 2.4. Most of the information given here is standard knowledge for practitioners of time-series analysis, pattern recognition or machine learning and for this reason we only briefly describe the methods without delving into proofs or the correctness and soundness of the algorithms involved.

### 2.4.1 Learning an AR Process

Recall that an AR process is described by the following equation:

$$x_t = a_0 + \sum_{i=1}^m a_i x_{t-i} + \varepsilon_t, \quad (2.63)$$

where  $\varepsilon_t \sim \mathcal{N}(0, \sigma^2 I)$  is a Gaussian noise variable.

The constant value  $a_0$  can be estimated by the mean of the time-series and we will assume that it is zero without loss of generality hereafter.

The simplest way to find the set of parameters  $a_i$  is to use least squares. Let  $\theta = [-a_p, -a_{m-1}, \dots, -a_2, -a_1, 1]'$ ,  $\varepsilon = [\varepsilon_1, \varepsilon_2, \dots, \varepsilon_T]$  and  $A' = [x_{t-m:t}]_{t=m+1}^T$ , and  $B = [x_p : x_T]$ , then Eq. 2.63 can be summarized for every point in the time-series as:

$$A\theta = B. \quad (2.64)$$

This equation can be solved using standard least-squares to estimate  $\theta$ . Moreover, the residuals from comparing the signal generated by using the learned  $\theta$  in Eq. 2.63 to the original time-series can be used to estimate the variance of the white noise component ( $\sigma^2$ ).

This approach is used in `learnARLS()` to learn the parameters of an AR process given an input time-series and an estimate of the system order.

The problem with this approach is that it requires the solution of  $T$  equations on only  $m$  unknowns which may not scale well with the length of the time-series.

Least squares estimation in general reduces the squared distance between the prediction from the learned model and the input data. This can be captured by the following unconditional optimization problem:

$$\min_a J(a) = \min_a \sum_{t=-\infty}^{\infty} \left( x_t - \sum_{i=1}^m a_i x_{t-i} \right)^2. \quad (2.65)$$

To solve this problem for each parameter  $a_j$ , we simply find the point at which the partial derivative of  $J$  with respect to  $a_j$  vanishes.

$$\begin{aligned} \frac{\partial J}{\partial a_j} &= \frac{\partial}{\partial a_j} \sum_{t=-\infty}^{\infty} \left( x_t - \sum_{i=1}^m a_i x_{t-i} \right)^2, \\ \frac{\partial J}{\partial a_j} &= \sum_{t=-\infty}^{\infty} \frac{\partial}{\partial a_j} \left( x_t - \sum_{i=1}^m a_i x_{t-i} \right)^2, \\ \frac{\partial J}{\partial a_j} &= \sum_{t=-\infty}^{\infty} -2x_{t-j} \left( x_t - \sum_{i=1}^m a_i x_{t-i} \right). \end{aligned} \quad (2.66)$$

Setting  $\frac{\partial J}{\partial a_j} = 0$ , we get:

$$\begin{aligned} \frac{\partial J}{\partial a_j} &= \sum_{t=-\infty}^{\infty} 2x_{t-j} \left( x_t - \sum_{i=1}^m a_i x_{t-i} \right) = 0, \\ \sum_{t=-\infty}^{\infty} x_{t-j} x_t - \sum_{t=-\infty}^{\infty} x_{t-j} \left( \sum_{i=1}^m a_i x_{t-i} \right) &= 0, \\ \sum_{t=-\infty}^{\infty} x_{t-j} x_{t-0} - \sum_{i=1}^m a_i \sum_{t=-\infty}^{\infty} x_{t-i} x_{t-j} &= 0. \end{aligned} \quad (2.67)$$

Defining  $A_i = -a_i$  for  $1 \leq i \leq m$  and  $A_0 = 1$ , and defining  $l = i - j$  we get:

$$\sum_{i=0}^m A_i \sum_{t=-\infty}^{\infty} x_t x_{t+l} = 0. \quad (2.68)$$

Now we notice that the internal summation is the autocorrelation coefficient  $\rho$  for different delays  $l$ . This means that the Eq. 2.68 can be written as:

$$\sum_{i=0}^m A_i \rho_l = 0. \quad (2.69)$$

We do not have access to  $\rho_l$  in Eq. 2.69 but we can estimate it using:

$$r_l = \sum_{t=1}^{T-l} x_t x_{t+l}. \quad (2.70)$$

Substituting  $r_l$  for  $\rho_l$  in Eq. 2.69, we get the famous Yule–Walker equations. These are  $M$  equations in  $M$  unknowns and their solution scales well with the length of the time-series and can be written as:

$$RA = B, \quad (2.71)$$

where:

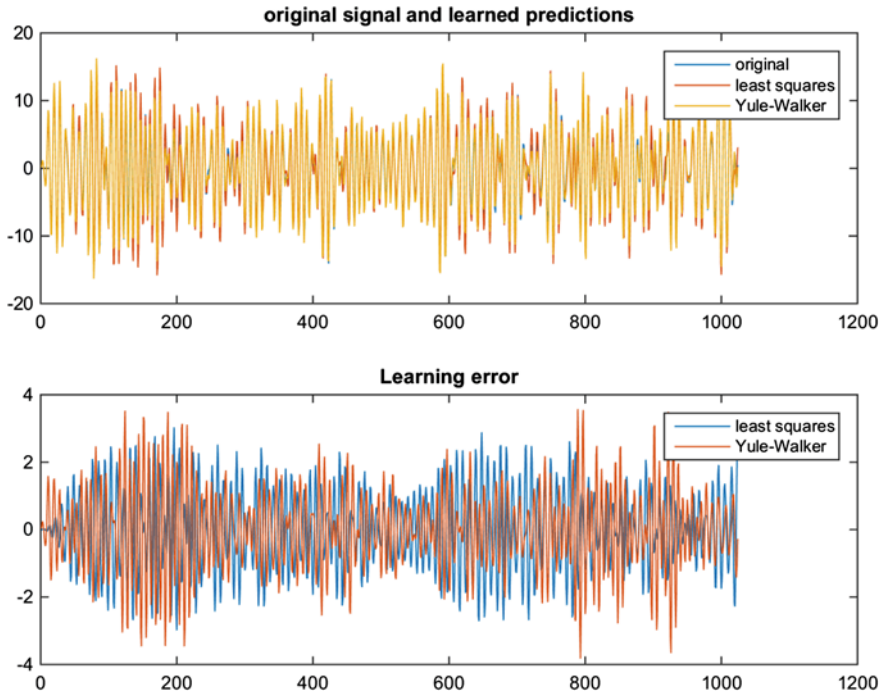
$$R = \begin{bmatrix} r_0 & r_{-1} & \cdots & r_{1-m} \\ r_1 & r_0 & \cdots & r_{2-m} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m-1} & r_{m-2} & \cdots & r_1 \end{bmatrix},$$

$$A = [a_1, a_2, \dots, a_m]',$$

$$B = [r_1, r_2, \dots, r_m].$$

This system of equations can be solved efficiently using the Levinson–Durbin recursion leading to an estimate for  $a$ . This procedure is implemented in the function `learnARYuleWalker()` in the toolbox.

Figure 2.17 shows the results of applying least squares and Yule–Walker approaches to learning the parameters of a time-series generated from an AR model. The original model had the parameter vector:  $a = [2.7607, -3.8106, 2.6535, -0.9238]$ . The model learned from least squares was  $a_{ls} = [2.7746, -3.8419, 2.6857, -0.9367]$ , while the model learned from Yule–Walker equations was:  $a_{yw} = [2.7262, -3.7296, 2.5753, -0.8927]$ . Comparing the parameter vectors directly we get  $a \cdot a_{ls} = 0.0487$  and  $a \cdot a_{yw} = 0.1218$ . It appears that the least-squares estimator could provide smaller error in terms of parameter values. Figure 2.17 shows also that the Yule–Walker approach gives higher error *in terms of the ability to predict the actual values of the time-series* which is what we usually care about. This is quantified by finding the Euclidean distance between the original time-series and the one predicted using the learned parameters (with the same *random* Gaussian noise) which leads to 39.5113 for the Yule–Walker approach compared with only 38.7442 for the least squares solution.



**Fig. 2.17** Results of learning the parameters of an AR process using both least squares and Yule–Walker approaches then generating an estimate of the time-series from the learned model

### 2.4.2 Learning an ARMA Process

Learning the parameters of an AR process using least squares or the Yule–Walker approach was a simple exercise. ARMA processes on the other hand provide a more challenging problem because we need to fit not only the parameter vector  $a$  but also  $b$  and the MA part of the process makes noise values at different time-steps correlated which renders least squares like solutions inappropriate for the problem.

The simplest approach to solve this problem is two-stages regression. We start by assuming that the data is from an  $AR(\hat{m})$  process instead of an  $ARMA(m, n)$  process where  $\hat{m} > m$ . This means that we assume that information about the Gaussian noise part is encoded in the longer AR process directly in the values of the time-series. This can be seen by rearranging Eq. 2.9 as follows:

$$\theta_t = \sum_{i=1}^m a_i x_{t-i}^{arma} - x_t^{arma} + \sum_{i=1}^n b_i \theta_{t-i}, \quad (2.72)$$

$$\theta_{t-1} = \sum_{i=1}^m a_i x_{t-i-1}^{arma} - x_{t-1}^{arma} + \sum_{i=1}^n b_i \theta_{t-i-1}, \quad (2.73)$$

$$\vdots$$

$$\theta_{t-n} = \sum_{i=1}^m a_i x_{t-i-n}^{arma} - x_{t-n}^{arma} + \sum_{i=1}^n b_i \theta_{t-i-n}. \quad (2.74)$$

This set of equations show that past values of  $x$  carries information about the Gaussian noise. We use  $\hat{m} = m + n$  for our implementation.

After fitting the time-series using  $AR(\hat{m})$  and finding the parameter vector  $\hat{a}$ , we use the fitted model to predict the time-series using:

$$x_t^{fit} = \sum_{i=1}^{\hat{m}} \hat{a}_i x_{t-i}^{fit}. \quad (2.75)$$

An estimate of the Gaussian noise can then be found as:

$$\hat{\theta}_t = x_t^{arma} - x_t^{fit}. \quad (2.76)$$

We now solve another regression problem (similar to the one used to fit the AR model) but with estimates of  $\theta_{1:T}$  now incorporated into the linear model  $A\theta = B$  where:

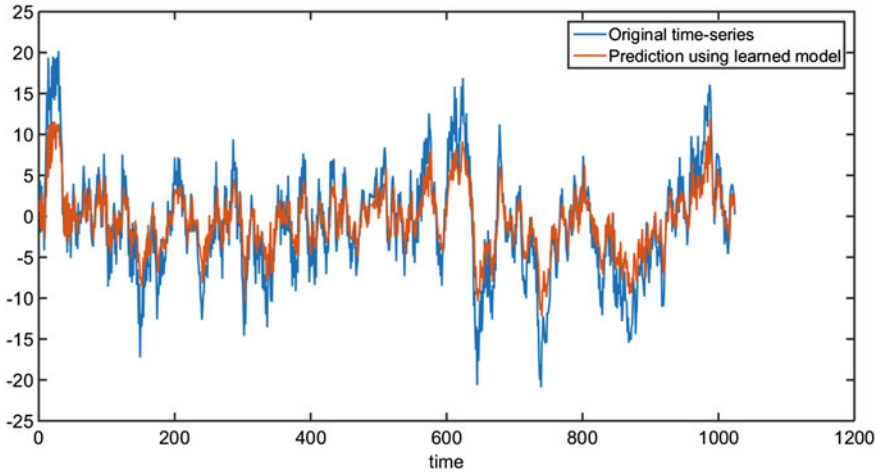
$$A = \begin{bmatrix} x_{r-1} & x_{r-2} & \cdots & x_{r-m} & \hat{\epsilon}_{r-1} & \hat{\epsilon}_{r-2} & \cdots & \hat{\epsilon}_{r-n} \\ x_r & x_{r-1} & \cdots & x_{r-m+1} & \hat{\epsilon}_r & \hat{\epsilon}_{r-1} & \cdots & \hat{\epsilon}_{r-n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{T-1} & x_{T-2} & \cdots & x_{T-m} & \hat{\epsilon}_{T-1} & \hat{\epsilon}_{T-2} & \cdots & \hat{\epsilon}_{T-m} \end{bmatrix}, \quad (2.77)$$

$$\theta = [a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n]', \quad (2.78)$$

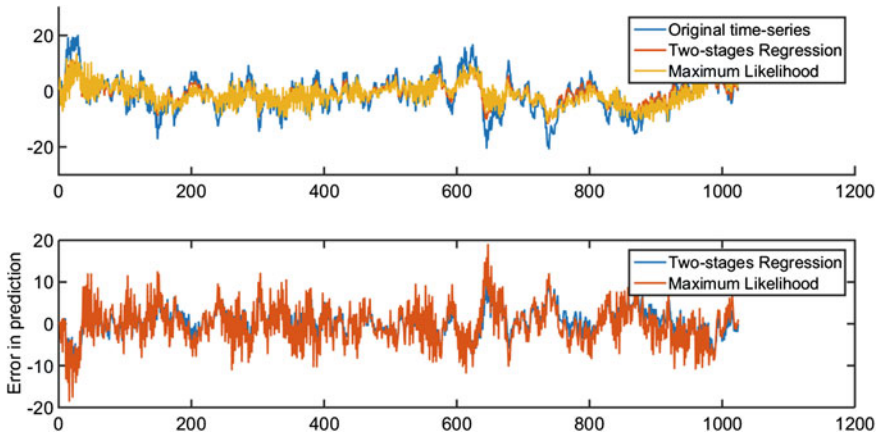
$$B = [x_r - \hat{\epsilon}_r, x_{r+1} - \hat{\epsilon}_{r+1}, \dots, x_T - \hat{\epsilon}_T]'. \quad (2.79)$$

Solving this linear system leads to an estimate of the parameter vectors  $a$  and  $b$ . Figure 2.18 shows the results of applying this procedure to a time-series generated from an  $ARMA(3, 5)$  model with  $a = [-0.7, 0.5, 0.9]$  and  $b = [1, 2, 3, 2, 1]$ .

Even though the accuracy is less than the case for the AR process shown in Fig. 2.17, this solution still captures the main characteristics of the time-series rising and falling with it with a correlation coefficient of 0.9356. This solution can also be used as an initial solution for a local search method (e.g. gradient descent on the log-



**Fig. 2.18** Results of learning the parameters of an ARMA process using two-stages regression then generating an estimate of the time-series from the learned model



**Fig. 2.19** Results of learning the parameters of an ARMA process using two-stages regression and maximum likelihood then generating an estimate of the time-series from the learned model

likelihood function) to find a better solution. Two-stages regression is implemented in the  $MC^2$  toolbox using the function *learnARMALS()*.

Matlab’s Econometrics toolbox has an implementation of the maximum likelihood estimator for ARMA model parameters in the function *estimate()*. Figure 2.19 shows the results of using two-stage regression and maximum likelihood to learn the same data presented in Fig. 2.18.

Another approach for estimating the parameters of an ARMA process is to convert it into a linear state-space model and use a Kalman filter to estimate the state of the

system (which corresponds to the time-series values without the MA part). These can then be used to estimate  $a$ . To estimate  $b$ , we use the residues of the first modeling step.

### 2.4.3 Learning a Hidden Markov Model

Hidden Markov Models (HMM) are widely utilized in speech recognition, gesture recognition and—as we will see in Chap. 13—in learning from demonstration. In this section we will focus on HMMs with Gaussian observation distributions (GHMMs) defined as in Sect. 2.2.8.

A GHMM is completely specified by a tuple  $\{\pi, A, \mu_1, \mu_2, \dots, \mu_N, \Sigma_1, \Sigma_2, \dots, \Sigma_N\}$  where  $\pi$  is a  $N \times 1$  vector specifying prior probabilities for the first hidden state,  $A$  is a  $N \times N$  matrix where  $A_{ij}$  specifying the transition probabilities from state  $i$  to state  $j$ , and  $\mu_n$  and  $\Sigma_n$  specify a Gaussian distribution from which the time-series value is sampled when the GHMM is in state  $n$  for  $1 \leq n \leq N$ . The full specification of a GHMM is given in Sect. 2.2.8 and repeated in Eq. 2.80 for convenience.

$$\begin{aligned} s_0 &\sim p(s_0) \equiv \pi, \\ s_t &\sim p(s_t | s_{t-1}) \equiv A_{s_{t-1}}^T, 0 < t \leq T, \\ x_t^{ghmm} &\sim p(x_t^{ghmm} | s_t) \equiv \mathcal{N}(\mu_{s_t}, \Sigma_{s_t}). \end{aligned} \quad (2.80)$$

Now, given a time-series  $X = (x_0, x_1, \dots, x_{T-1})$ , we would like to learn the parameters of the GHMM generating it. Collecting all the parameters in a single vector  $\theta$ , this problem can be casted as a maximum likelihood problem of the form:

$$\max_{\theta} J(\theta) = \max_{\theta} p(X|\theta).$$

This problem can be solved efficiently using the Baum–Welch forward-backward algorithm which is a form of expectation maximization that takes advantage of the independence relationships implicit in the definition of HMMs to achieve linear time estimation of model parameters.

Because it is an Expectation Maximization algorithm, it requires an initial model  $\lambda = (\pi, A, \mu_{1:N}, \Sigma_{1:N})$ . Using this model, we will find  $p(s_t = n | X, \lambda)$  for  $0 \leq t \leq T-1$  and  $1 \leq n \leq N$  which is the probability of having every possible state  $n$  at every possible time-step  $t$  given that we have observed  $X$  and assuming the model  $\lambda$ . This is the expectation step. Given these probabilities, it is straightforward to find an estimate of the model parameters  $\lambda_+$  that achieves higher likelihood in the maximization step. Iterating these two steps, it is guaranteed that  $\lambda_+$  will converge to a local maximum of the likelihood function  $J$ .

Let's consider the expectation step. Our goal is to calculate  $p(s_t = n | X, \lambda)$  efficiently. Firstly we notice that:

$$p(s_t = n|X, \lambda) \equiv \gamma_t(n) = p(s_t = n, x_{0:t}|\lambda) p(x_{t+1:T}|s_t = n, \lambda). \quad (2.81)$$

Equation 2.81 is true due to the Markovian property of HMMs (i.e.  $s_{t+1}$  is independent of everything given  $s_t$ ). Defining  $\alpha_t(n) \equiv p(s_t = n, x_{0:t}|\lambda)$  and  $\beta_t(n) \equiv p(x_{t+1:T}|s_t = n, \lambda)$ , Eq. 2.81 can be written as:

$$\gamma_t(n) \propto \alpha_t(n) \circ \beta_t(n), \quad (2.82)$$

where  $\circ$  is the element wise multiplication operator. Now the expectation step reduces to the problem of calculating  $\alpha$  and  $\beta$ .

Consider  $\alpha$ , we know that:

$$\alpha_0(n) = p(s_0 = n|x_0, \lambda) = \frac{p(s_0 = n, x_0|\lambda)}{p(x_0|\lambda)}, \quad (2.83)$$

$$\therefore \alpha_0(n) = \frac{p(s_0 = n|\lambda) p(x_0|s_0 = n, \lambda)}{p(x_0|\lambda)}, \quad (2.84)$$

$$\therefore \alpha_0(n) = \frac{\pi_n \mathcal{N}(x_0; \mu_n, \Sigma_n)}{p(x_0|\lambda)}. \quad (2.85)$$

Defining  $\rho_t(n) \equiv \mathcal{N}(x_t; \mu_n, \Sigma_n)$ , Eq. 2.85 can be written as:

$$\alpha_0(n) = \frac{\pi_n \rho_t(n)}{p(x_0|\lambda)}. \quad (2.86)$$

Now consider the general case (i.e.  $\alpha_t(n)$ ):

$$\begin{aligned} \alpha_t(n) &= p(s_t = n|x_{0:t}, \lambda) = \frac{p(s_t = n, x_{0:t}|\lambda)}{p(x_{0:t}|\lambda)}. \\ \therefore \alpha_t(n) &\propto p(s_t = n, x_t, x_{0:t-1}|\lambda), \\ \therefore \alpha_t(n) &\propto p(s_t = n, x_{0:t-1}|x_t, \lambda) p(x_t|s_t = n, x_{0:t-1}, \lambda), \\ \therefore \alpha_t(n) &\propto p(s_t = n, x_{0:t-1}|x_t, \lambda) p(x_t|s_t = n, \lambda), \\ \therefore \alpha_t(n) &\propto \beta_t(n) \sum_{i=1}^N p(s_t = n, s_{t-1} = i, x_{0:t-1}|x_t, \lambda), \\ \therefore \alpha_t(n) &\propto \beta_t(n) \sum_{i=1}^N p(s_t = n, s_{t-1} = i, \lambda) p(s_{t-1} = i, x_{0:t-1}|\lambda). \end{aligned}$$

This can be written as:

$$\alpha_t(n) \propto \beta_t(n) \sum_{i=1}^N A_{in} \alpha_{t-1}(i). \quad (2.87)$$

But we know that  $\sum \alpha_t(n)$  must equal 1. This leads to the final estimation equation for  $\alpha_t(n)$ :

$$\alpha_t(n) = \frac{\beta_t(n) \sum_{i=1}^N A_{in} \alpha_{t-1}(i)}{\sum_j \beta_t(j) \sum_{i=1}^N A_{ij} \alpha_{t-1}(i)}. \quad (2.88)$$

This derivation shows that we can find  $\alpha_t(n)$  for any  $t$  and  $n$  within their respective ranges given  $A$ ,  $\alpha_{t-1}(1:N)$ , and  $\rho_t(1:N)$ . This suggests a simple recursion starting by finding  $\alpha_0(n)$  for  $1 \leq n \leq N$  using Eq. 2.86. We then use Eq. 2.88 for  $1 \leq t \leq T$  and  $1 \leq n \leq N$ . This is called the forward recursion.

The second part of Eq. 2.81 can be estimated using a similar procedure but now going backward in the time-series according to the following two equations:

$$\beta_T(n) = 1, \quad (2.89)$$

$$\beta_t(n) = \sum_{i=1}^N A_{ni} \rho_{t+1}(i) \beta_{t+1}(i). \quad (2.90)$$

Using these two equations,  $\beta_t(n)$  can be calculated backward starting from  $\beta_T(n)$ . Now having calculated both  $\alpha$  and  $\beta$ , it is easy to calculate  $\gamma$  as follows:

$$\gamma_t(n) = \frac{\alpha_t(n) \beta_t(n)}{\sum_{i=1}^N \alpha_t(i) \beta_t(i)}. \quad (2.91)$$

Now  $\gamma_t(n)$  gives us an estimate of the probability of being at state  $n$  at time step  $t$  given the observed time-series  $X$  and the initial model  $\lambda$ . Now that we have an estimate of the *hidden* state at every time-step, we can try to re-estimate the model parameters.

A useful quantity for the maximization step is the probability of transiting from state  $i$  at time-step  $t$  to state  $j$  at time-step  $t+1$  (notice that marginalizing this gives an estimate of  $A_{ij}$ ). This quantity is defined as:

$$\zeta_t(i, j) \equiv p(s_t = i, s_{t+1} = j | x_{0:T-1}, \lambda).$$

It can easily be shown that  $\zeta$  can be calculated as:

$$\zeta_t(i, j) = \frac{\alpha_t(i) A_{ij} \rho_{t+1}(j) \beta_{t+1}(j)}{\sum_{l=1}^N \sum_{k=1}^N \alpha_t(k) A_{kl} \rho_{t+1}(l) \beta_{t+1}(l)}. \quad (2.92)$$

Given these estimates of  $\zeta$  and  $\gamma$ , we can easily estimate the model  $\lambda^+ = (\pi^+, A, \mu_{1:N}^+, \Sigma_{1:N}^+)$  using:

$$\pi^+(n) = \gamma_0(n), \quad (2.93)$$

$$A_{ij}^+ = \frac{\sum_{t=0}^{T-2} \zeta_t(i, j)}{\sum_{t=0}^{T-2} \gamma_t(i)}, \quad (2.94)$$

$$\mu_i^+ = \frac{\sum_{t=0}^{T-1} x_t \gamma_t(i)}{\sum_{t=0}^{T-1} \gamma_t(i)}, \quad (2.95)$$

$$\Sigma_i^+ = \frac{\sum_{t=0}^{T-2} \gamma_t(i) (x_t - \mu_i^+) (x_t - \mu_i^+)^T}{\sum_{t=0}^{T-1} \gamma_t(i)}. \quad (2.96)$$

This process can be repeated until  $\lambda^+$  does not differ much from  $\lambda$  or its likelihood is not different from that of  $\lambda$  or until a predefined number of iterations is reached. The aforementioned procedure is implemented in the function *learnHMM()* in the *MC<sup>2</sup>* toolbox. When multiple time-series are available that are believed to be from the same GHMM, a slightly modified version of this procedure can be implemented to learn the HMM parameters from all of the input time-series. This procedure is implemented in the function *learnHMMMMulti()*.

#### 2.4.4 Learning a Gaussian Mixture Model

Learning the parameters of a GMM from input time-series is conceptually very similar to learning the parameters of GHMMs using Expectation Maximization. The main idea is to estimate the responsibility of every Gaussian for the time-series values at every sample in the expectation step  $r_t(k)$  using:

$$r_t(k) = \frac{\pi_k \mathcal{N}(x_t; \mu_k, \Sigma_k)}{\sum_{i=1}^K \pi_i \mathcal{N}(x_t; \mu_i, \Sigma_i)}. \quad (2.97)$$

The maximization step can be summarized as:

$$\pi_k^+ = \frac{\sum_{t=1}^T r_t(t)}{K}, \quad (2.98)$$

$$\mu_k^+ = \frac{\sum_{t=1}^T r_t(t) x_t}{\sum_{t=1}^T r_t(t)}, \quad (2.99)$$

$$\Sigma_k^+ = \frac{\sum_{t=1}^T r_t(t) (x_t - \mu_k^+) (x_t - \mu_k^+)^T}{\sum_{t=1}^T r_t(t)}. \quad (2.100)$$

These two steps are repeated a predefined number of times or until the likelihood is not changing anymore. This algorithm is implemented in the *learnGMM()* function of the toolbox.

### 2.4.5 Model Selection Problem

Learning the parameters of a generating model (despite the type of this model) usually requires an assumption about the model *complexity*. For example, to learn the parameters of an  $ARMA(m, n)$  process we need some assumption about  $m$  and  $n$  and to learn a GHMM, we need to know the number of states  $N$ . Selecting the complexity of the model is a ubiquitous problem in data mining and there are several known approaches to deal with it.

The simplest approach is K-fold cross validation. The training data (the time-series in our case) is divided to  $K$  equally sized partitions. The system is trained on  $K - 1$  partitions and its performance is tested on the remaining one. This process is repeated  $K$  times with a different testing partition each time. This process is repeated for the set of complexities to be tested (e.g. values for  $K$  in HMM learning) and the value the achieves best predictive performance is then selected.

Another approach that is used widely is Bayesian Information Criteria (BIC). In this case, a statistic is calculated by adding the negative log-likelihood of different models to another term measuring the complexity of the system. Rather than selecting the system complexity that maximizes the likelihood (which is prone to overfitting), we take the fact that more complex systems can in general achieve higher likelihood values due to their tendency to overfit the data and model not only the system dynamics but the noise corrupting it. The complexity level that minimizes the BIC statistic is selected instead of the maximum likelihood statistic.

## 2.5 Time Series Preprocessing

Before any mining algorithm is applied to time-series data, preprocessing may be required to remove artifacts, or enhance the quality of the data in some way. Several preprocessing operations exist and for each of which there can be several alternative algorithms. Here, we discuss the most useful of these operations for our purposes in this book and some simple algorithms to achieve them keeping in mind application to social robotics.

### 2.5.1 Smoothing

A very common problem with collected time-series specially from real-world sensors is the contamination with high frequency noise that may cause problems to modeling algorithms. For example, piecewise linear approximations of time series (that we will use several times in this book) may suffer from over-segmentation (i.e. generating too many lines) in the face of such noise. A simple approach to reduce the effect of this kind of noise is smoothing. Several algorithms exist for smoothing of time-series data but in most cases the simple moving average approach will do.

### 2.5.2 Thinning

Thinning is the process of keeping only local maxima of the time-series. This process can be used as a compression technique by keeping a sparse representation of the time-series in question. In this book we use thinning as a postprocessing step in the *RSST* change point discovery algorithm (See Sect. 3.5).

A very simple thinning algorithm can be implemented by noticing the first difference of the data. Two rules are applied in order given some positive small number  $\delta$ : if  $x_t - x_{t-1} > \delta$  then set  $x_{t-1}$  to zero. if  $x_t - x_{t-1} < -\delta$  then set  $x_t$  to zero.

More sophisticated approaches exist. For example, a piecewise linear approximation of the time series can be generated then only the points at which the line slopes changes from positive to negative are kept.

### 2.5.3 Normalization

In many cases, it is necessary to keep the range of values in a time-series within some range or make these ranges similar for different dimensions of the time-series. Again several approaches can be thought of for this problem but we will focus on two simple and widely used approaches.

First of all, we may just want to remove any constant component of the time-series because for example it does not contribute to the information content. This can easily be achieved by removing the mean of the time-series from each point. This approach is applicable to both single dimensional and multidimensional time-series. Defining  $\mu(X)$  to be the mean of a time-series along its independent dimension, we can state this simply as:

$$\bar{x}_t = x_t - \mu(X). \quad (2.101)$$

If the scale is also to be removed, we can use the following general formula:

$$\bar{x}_t = \frac{x_t - \mu(X)}{S}, \quad (2.102)$$

where  $S$  represents the scale which can be either the range of the time-series (i.e.  $\max(X) - \min(X)$ ) or its standard deviation  $\sigma(X)$ . Equation 2.102 assumes that  $X$  has a single dimension. A generalization to the multidimensional case can be defined as:

$$\bar{x}_t = C^{-1} (x_t - \mu(X)), \quad (2.103)$$

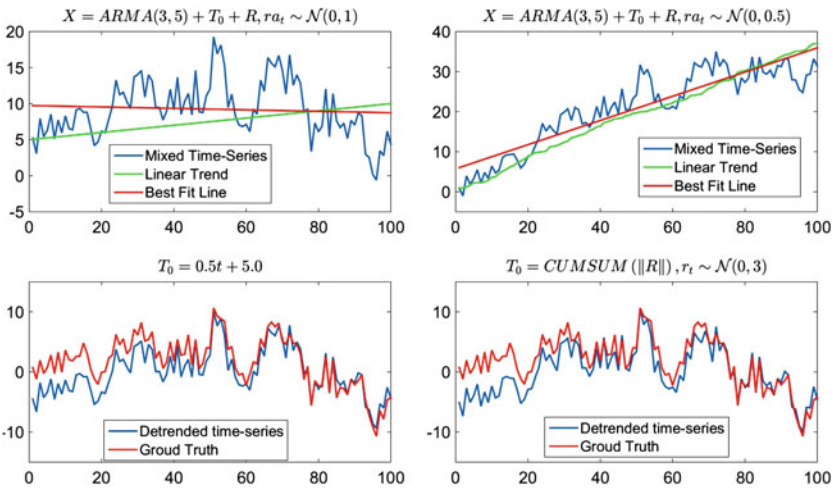
where  $C$  is the covariance matrix calculated as  $XX^T$  assuming  $x_t$  are column vectors.

### 2.5.4 De-Trending

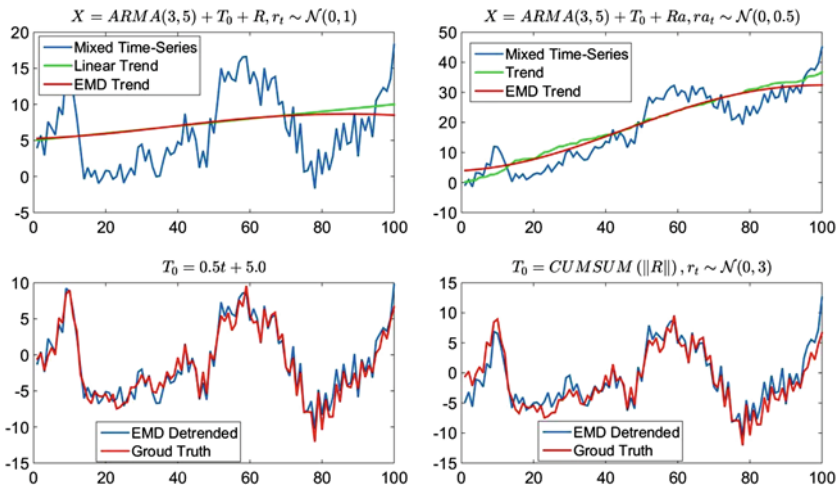
Referring to the xLAT model of time-series discussed in Sect. 2.2.1, the time-series can be modeled by the addition of four factors one of them is the trend  $T_0$  which is a monotonically increasing/decreasing time-series. A common pre-processing step in many time-series mining applications is called de-trending and involves the removal of this trend from the time-series. This is analogous to the removal of DC component in signal processing applications. The simplest case of de-trending happens when we can assume that  $T_0$  is linear. In this case, a line is fit to the original time-series then subtracted from each point in it. Figure 2.20 shows two examples of de-trending when the original trend is linear. As the figure shows, when this assumption holds, this method can recover the original signal up to the added noise level while when the linear assumption fails, the results can differ widely from the ground truth. Notice though that the error at every point is dependent only on the difference between the fitted line and the actual trend. In  $MC^2$ , the function `detrendLinear()` implements this linear de-trending procedure.

When the trend is nonlinear, SSA can be used to find it by manually grouping monotonically increasing or decreasing expansion time-series during the grouping step (Sect. 2.3.5). The problem here is that a test is needed to decide whether a given expansion time-series is a trend component.

Empirical Mode Decomposition (EMD) is another adaptive expansion technique that has the advantage of always finding the trend component as its last component by construction. The  $MC^2$  has another de-trending routine called `detrendEMD()` that uses EMD for finding the trend component and removing it. Figure 2.21 shows the results of applying this routine to two time-series with a linear trend (left) and



**Fig. 2.20** Linear de-trending of a time-series. On the *left* the case where the trend is linear and on the *right* the case when it is nonlinear



**Fig. 2.21** EMD based de-trending of a time-series. On the *left* the case where the trend is linear and on the *right* the case when it is nonlinear

a nonlinear trend (right). In both cases, EMD based de-trending provides superior performance over linear de-trending.

### 2.5.5 Dimensionality Reduction

In many cases, we need to convert a multidimensional time-series to a single-dimensional time-series. Several methods for dimensionality reduction can be used including linear methods like Independent Component Analysis (ICA), Principle Component Analysis (PCA), and nonlinear methods including IsoMap. This section introduces one of the simpler approaches using PCA.

The following notation will be used in this section:  $x_{t,i}$  is the  $i$ th dimension of  $x_t$ ,  $x_{t,l,i}$  is the  $i$ th component of the subsequence  $x_{t,l}$ , and  $X_{t,i}$  is the  $i$ th dimension of the time-series  $X$  which is a single-dimensional time-series.

To apply PCA to a time-series, we start by creating the covariance matrix  $A$  which is defined as:

$$A_{i,j} = \sum_{t=1}^T x_{t,i} x_{t,j}. \quad (2.104)$$

This matrix can easily be found by treating  $X$  as a  $n \times T$  matrix:

$$A_{n \times n} = X_{n \times T} X_{T \times n}^T. \quad (2.105)$$

We then find the first Eigen vector of  $A$  ( $v_1$ ) and its corresponding Eigen value  $\lambda_1$ . The output 1D time-series can then be found by projecting every time-series value on that vector using the dot product operation:

$$y_t = v_1^T x_t. \quad (2.106)$$

Calculating  $A$  in the aforementioned procedure requires  $O(n^2T)$  operations which may be too slow for some applications. We can speedup the operation by selecting  $K < T$  vectors from the time-series and use them to find  $A$ . Both of the exact and approximate versions of this process are implemented in the function `tspca()` in the `MC2` toolbox.

### 2.5.6 Dynamic Time Warping

Sometimes, we receive for mining a set of time-series of different lengths and would like to use an algorithm that assumes that all inputs have the same length. This can simply be achieved by re-sampling/interpolating the shorter sequences. For example if we have two time-series  $X, Y$  of lengths  $T_x$  and  $T_y$  where  $T_x > T_y$ , we assume that the sample  $y_t$  represents the value of  $Y$  at time  $\frac{t \times (T_x - 1)}{(T_y - 1)}$ . Linear or polynomial interpolation can then be used to estimate values of  $\hat{Y}$  at times  $0, 1, \dots, T_x$ . A similar approach can be used to change the length of the longer time-series to equal the length of the shorter time-series.

This simple approach assumes that the difference in the length between the two time-series comes from a difference in the sampling frequency used to collect them. In many cases, the difference in length is not related to the sampling frequency but represents genuine difference between the two time-series. One such example happens in learning from multiple demonstrations (Chap. 13). In these problems, we have multiple demonstrations of the same action conducted by a teacher from which a learner is expected to generate a model representing the generation process of these demonstrations. In this case, the difference in length of the time-series cannot be assumed to originate from a difference in sampling rate or approximated by such difference. The difference here is most likely a genuine difference in how fast did the teacher perform different phases of the motion compared to one another. This means that the relation between the time-series to be equalized in length cannot be assumed to be a constant scaling that can be handled by the simple re-sampling procedure described above. A common solution to these cases is to use the Dynamic Time Warping algorithm (DTW).

DTW appeared as a distance function that can be used to estimate distances between time-series more accurately than the standard Euclidean distance when the data points are slightly temporally displaced compared with one another (See Ding et al. 2008 for an experimental evaluation). It reuses all the points in the two time-series to be aligned. The main idea is to find for each point in the shorter time-series

a corresponding point in the longer one with the constraint that the first and last two points of the two time-series must align (this is called the boundary constraint). The algorithm can be visualized on a 2D grid  $\mathcal{M}$  where the points of time-series  $X$  are represented by the rows and the points of the time-series  $Y$  are represented by the column. The boundary condition translates to the statement that the path representing the correspondences between these two time-series must start at the bottom left cell (representing  $(x_0, y_0)$ ) and end at the top right cell (representing  $(x_{T_x-1}, y_{T_y-1})$ ). Each cell in this grid contains the distance between the corresponding points of  $X$  and  $Y$  (i.e.  $\mathcal{M}_{ij} = d(x_i, y_j)$  for some distance function  $d$ ). Now, DTW finds the path through  $\mathcal{M}$  that minimizes the sum of these distances. This corresponds to the best possible match between the two time-series. Other than the boundary constraint, DTW optimization is constrained in two other ways:

**Monotonicity constraint:** The indices of both  $X$  and  $Y$  must be monotonically increasing which means that for the optimal path ( $P = (p^1, p^2, \dots, p^{\max(T_x, T_y)})$ ), where  $p^i$  is an ordered pair  $(j^i, k^i)$  and  $0 \leq j^i \leq T_x, 0 \leq k^i \leq T_y, i_1 > i_2$ ; implies that  $j^{i_1} \leq j^{i_2}$  and  $k^{i_1} \leq k^{i_2}$ .

**Continuity constraint:** Continuous points in the path correspond to adjacent cells both horizontally and vertically. This means that for  $p^i$  and  $p^{i+1}$ ,  $|j^{i+1} - j^i| \leq 1$  and  $|k^{i+1} - k^i| \leq 1$ .

Other constraints are usually used to speed up the calculation of DTW and prevent the path from wandering around leading to meaningless wraps. A *warping window* condition limits the maximum wandering distance allowed from the diagonal. A *slop* constraints limits the number of steps that can be taken in the same direction by the path horizontally or vertically. The two most common constraints in the literature are the Sakoe-Chiba Band and the Itakura Parallelogram. Sakoe-Chiba Band restricts the path to lie within a band around the diagonal. The width of this band is usually taken to be 10 % of the shorter time-series' length. Ratanamahatana and Keogh (2005) showed that this limit is not only useful for speeding up the DTW calculations but that it is even less stringent than necessary for real world data mining applications. Itakura Parallelogram limits the path to lie within a parallelogram with two vertices at the starting and ending points of the path (limited by the boundary constraint). This means that the path is allowed to wander more near the middle of the time-series and less near the boundaries.

DTW is usually used to align single dimensional time-series but it can easily be extended to multidimensional time-series by modifying the distance function used. This algorithm is implemented in the function `dtw()` in the *MC<sup>2</sup>* toolbox.

## 2.6 Summary

This chapter introduced basic time-series analysis techniques that will be used throughout this book. The focus of the chapter was on techniques that are of direct relevance to the ideas presented in the following chapters, rather than on providing

an exhaustive treatment of time-series analysis (which requires a much larger volume by itself). We presented several models of generating processes for time-series data. These generation models will be used for generating test sets for algorithms developed later and some of them (e.g. GMM/GMR and GP) will be of direct use in learning from demonstration. We also presented five transformations for representing time-series that will form the basis of algorithms for change point discovery and motif discovery (Chaps. 2 and 3) specially the Singular Spectrum Analysis method that will be a common ingredient of several algorithms later in this book. The chapter also gave a brief treatment of preprocessing techniques that are employed everywhere in this book including smoothing, thinning, normalization and de-trending. The following three chapters will focus on specific time-series analysis problems that use the aforementioned generation models and transformations to create the building blocks for our autonomous learning system to be introduced in the second part of the book.

## References

- Calinon S, Guenter F, Billard A (2006) On learning the statistical representation of a task and generalizing it to various contexts. In: ICRA'06: IEEE International conference on robotics and automation. IEEE, pp 2978–2983
- Ding H, Trajcevski G, Scheuermann P, Wang X, Keogh E (2008) Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc VLDB Endow* 1(2):1542–1552. doi:[10.14778/1454159.1454226](https://doi.org/10.14778/1454159.1454226)
- Elsner JB, Tsonis AA (2013) Singular spectrum analysis: a new tool in time series analysis. Springer
- Hassani H (2007) Singular spectrum analysis: methodology and comparison. *J Data Sci* 5(2): 239–257
- Larsen RJ, Marx ML (1986) Introduction to mathematical statistics and its applications, 2nd edn. Prentice Hall
- Lin J, Keogh E, Lonardi S, Chiu B (2003) A symbolic representation of time series, with implications for streaming algorithms. In: The 8th ACM SIGMOD workshop on research issues in data mining and knowledge discovery. ACM, pp 2–11
- Mohammad Y, Nishida T (2014) Robust learning from demonstrations using multidimensional SAX. In: ICCAS'14: 14th international conference on control, automation and cystems. IEEE, pp 64–71
- Rasmussen CE, Williams CK (2006) Gaussian processes for machine learning. MIT Press
- Ratanamahatana CA, Keogh E (2005) Three myths about dynamic time warping data mining. In: SDM'05: SIAM international conference on data mining. SIAM, pp 506–510
- Ratanamahatana CA, Lin J, Gunopulos D, Keogh E, Vlachos M, Das G (2010) Mining time series data. In: *Data Mining and Knowledge Discovery Handbook*. Springer, pp 1049–1077

Data Mining for Social Robotics

Toward Autonomously Social Robots

Mohammad, Y.; Nishida, T.

2015, XII, 328 p. 74 illus. in color., Hardcover

ISBN: 978-3-319-25230-8