Probabilistic Modeling with the Wolfram Language

Gerhard Hejc

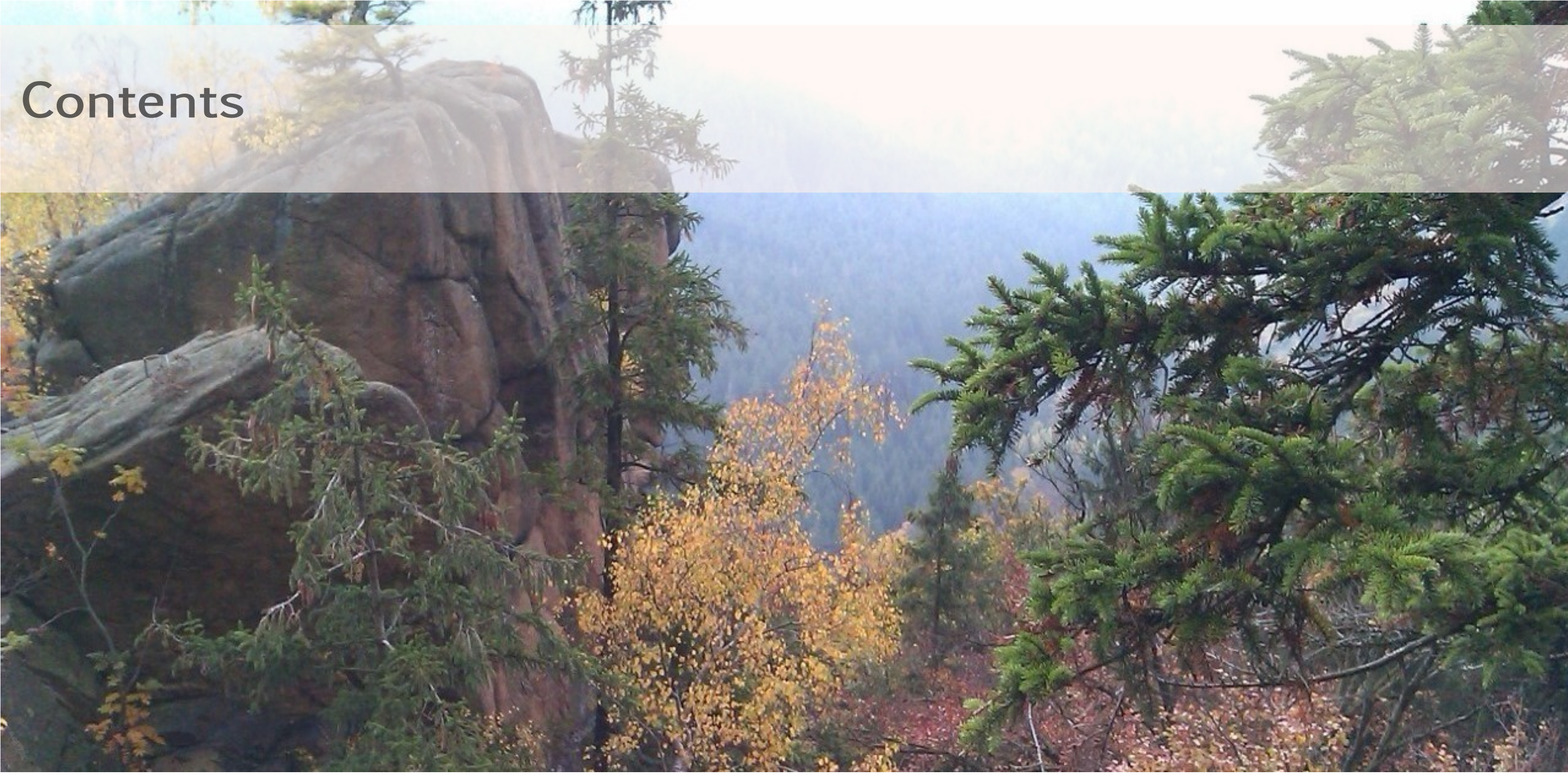
Publisher

Art. No xxxxx
ISBN xxx–xx–xxxx–xx–x
Edition 0.0

Cover design by Cover Designer

Published by Publisher
Printed in City

Contents



1 Basic Concepts 7

1.1 Probability Density 7

1.2 Probability 13

1.3 Conditional Probability Density 15

1.4 Deterministic Functions 17

1.5 Discrete Random Variables 19

1.6 Expectation, Mean and Variance 25

1.7 Examples 28

2 Density Estimation and Random Data 33

2.1 Histograms 33

2.2 Kernel Density Estimation 36

2.3 Gaussian Mixture Model 37

2.4 Mixture Density Networks 38

2.5 Normalizing Flows 39

2.6 Maximum Likelihood 40

2.7 Sampling 42

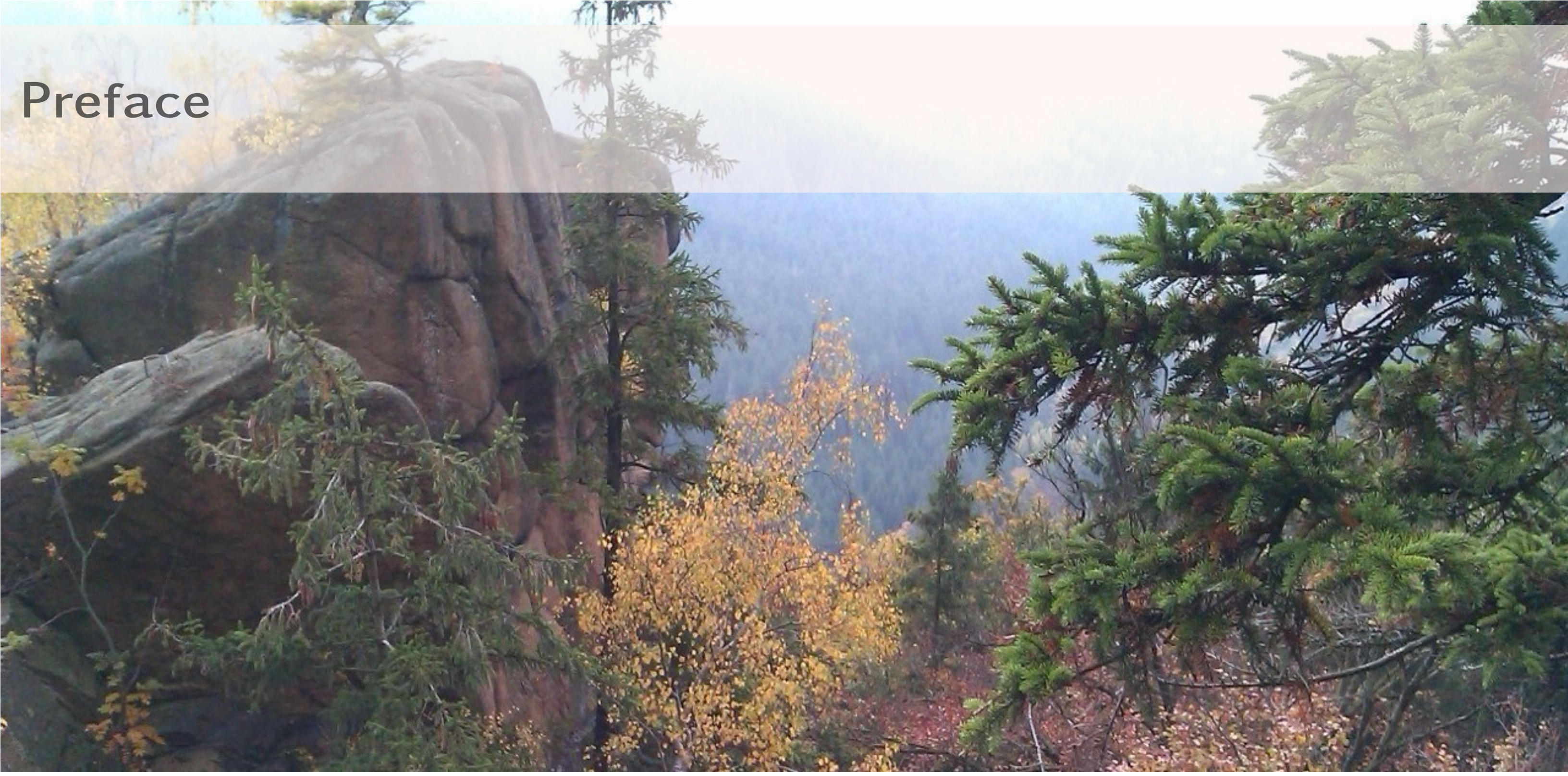
3 Probabilistic Models (Theory) 50

3.1 Graphical Models 50

3.2 Time Series 55

3.3 Particle Filter 63
3.4 Random Processes 70
4 Probabilistic Models (Applications) 74
4.1 Poisson Clock 74
4.2 Atomic Clock 74
4.3 Gaussian Lattice Model 75
4.4 Random Matrices 75
Literature 75

Preface



The book originally developed from talks and lecture notes given at the Fraunhofer institute in Nuremberg, Germany as an introduction to probabilistic modeling. The goal was primarily to provide the students with a minimal set of mathematical tools and concepts to do probabilistic computations by themselves and to apply it to real-world problems. This sounds easier than it actually is, because even conceptually easy problems can lead to results, which contradicts intuition like the famous Monty Hall problem.

There are basically two main branches of probabilistic modeling: simulation and inference. The first uses a probabilistic model based on a theory to make predictions or simulate the outcome of an experiment or a measurement, while the second one uses collected data to create a probabilistic model. The term probabilistic means the ability to include uncertainty into the model. Uncertainty in the first case is entering through only imprecisely known parameters, hidden variables and approximations, while in the second case it depends on the degree of knowledge about the noise which is inherently present in the data or the process how the data was generated or how many parameters are necessary to describe the data. Including uncertainty is not an option, but a necessity to make precise statements about the reliability of our results.

A lot of exciting developments took place in this field in the last years. Most of the non-trivial calculations are almost impossible to be done without a computer, so the question quickly arises what is the appropriate software tool for doing this kind of calculations. Because of the availability of a large number of good software, it is merely somebody’s preference towards one or the other, but in this book the Wolfram language is chosen for several reasons.

The Wolfram language [19, 6] has a clear and consistent design with a syntax close to the mathematical notation and has all the built-in knowledge to perform symbolic calculations where results in a closed-form are available and numerical evaluations in the case where this is not possible. Especially the Probability and Statistics part, which is used heavily in this book, has been improved significantly in the newer version of the language and the available functions can be easily applied to all kind of statistical problems. The visualization capabilities of the Wolfram language are outstanding and every figure in this book was generated with the version integrated into Mathematica 12.0 [10].

This is not the first book about this topic, but builds on the work of others. There are many great books available, and these recommendations are definitely not exhaustive, but only a subjective subset:

- David Barber's book **Bayesian Reasoning and Machine Learning** [1] is one of the best books, which covers extensively all topics in great detail and clarity.
- Bendat and Piersol's **Random Data: Analysis and Measurement Procedures** [3] is a classic book published first 1971, but still a clear and well-written reference about the analysis of random data, especially time series data.
- Simon Sirca's book **Probability for Physicists** [16] is a good introduction with many examples from physics.
- Glen Cowan's book **Statistical Data Analysis** [5] is a concise and well-written treatise targeting mainly readers of the particle physics community.

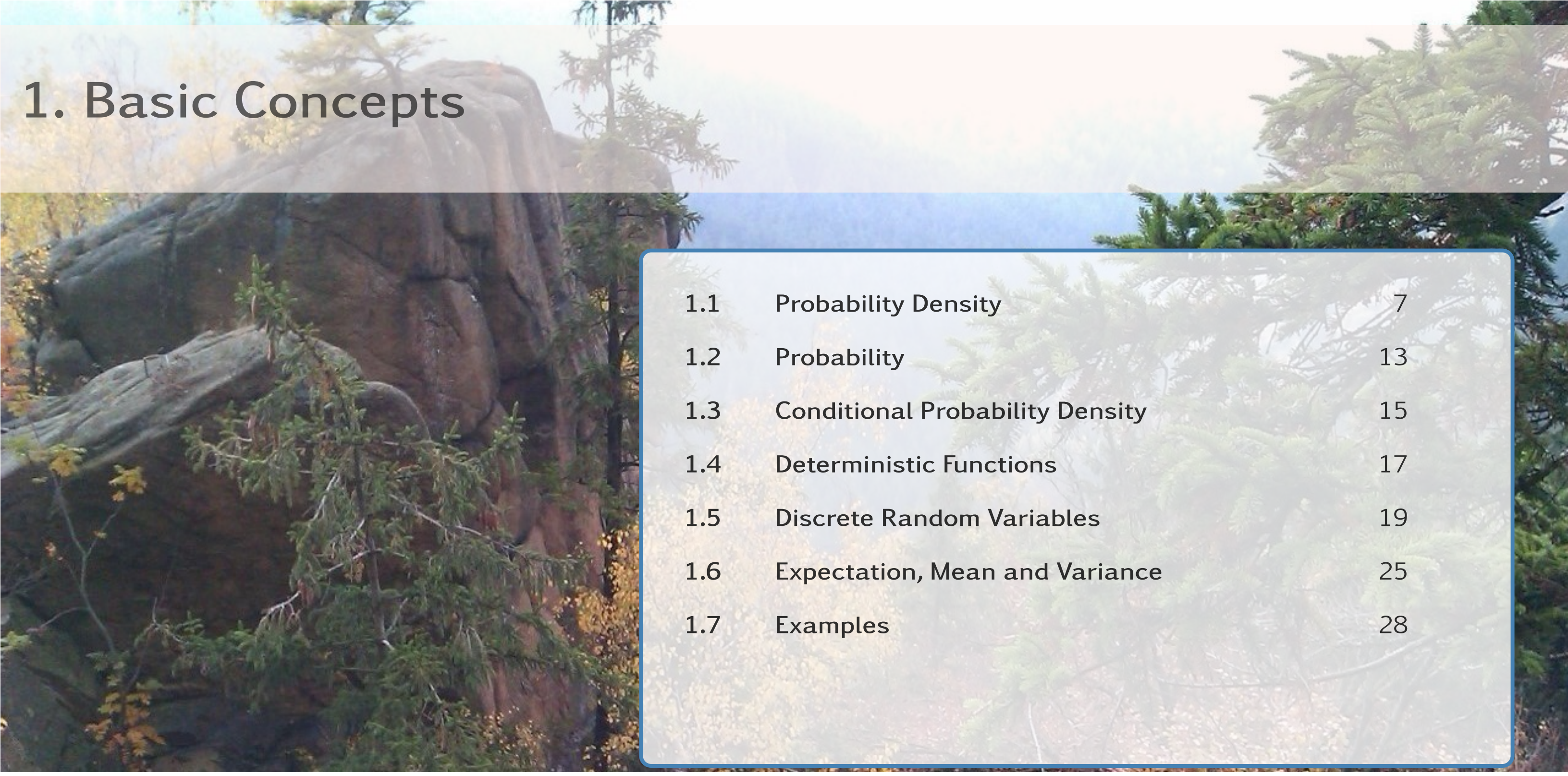
Some knowledge in Linear Algebra and Calculus is also required e.g. see Murray Spiegel's book **Schaum's Outline of Advanced Mathematics for Engineers and Scientists** [17] for an excellent coverage of all the mathematical methods and tools used by engineers and scientists. There is also another book by the same author completely dedicated to probability and statistics [18], which consists of 897 fully solved problems and 20 online videos.

A great tool for exploring univariate probability distributions built into the Wolfram Language is the **The Ultimate Univariate Probability Distribution Explorer**, which can be downloaded from [13].

There is already a very good book **Introduction to Probability with Mathematica** by Kevin J. Hastings [9] with a similar intention than this one. The second edition is based on Mathematica 7.0 and it is highly recommended as a complementary reading.

Nevertheless the goal of this book is a bit different. We want to cover only the relevant topics and concepts, which are necessary for building and analyzing probabilistic models and applying them to all kind of problems. Therefore it does not claim to be a complete treatise of the subject, but a rather concise introduction to bring the reader as fast as possible into a state to perform computations and to understand its results.

1. Basic Concepts



1.1 Probability Density 7

1.2 Probability 13

1.3 Conditional Probability Density 15

1.4 Deterministic Functions 17

1.5 Discrete Random Variables 19

1.6 Expectation, Mean and Variance 25

1.7 Examples 28

This first chapter introduces the basic concepts. The starting point in other text books about this topic is typically the definition of probability based on the Kolmogorov axioms. This book chooses a different approach and puts the main focus on the probability density. Everything else including probability is derived from this quantity. The reason for this choice is mainly that the result of almost every non-trivial probabilistic calculation in physics or mathematics is always a (conditional) probability density. It includes the complete information about a random process and plays a central role in the description of probabilistic systems. As we will see, it contains also deterministic systems as a special case and is therefore a generalization of a function for the case, when uncertainty comes into play.

1.1 Probability Density

The probability density is a generalized function, which describes the distribution of values of an random variable. Generalized means here that it can contain Dirac delta functions. It describes the outcome of an experiment or measurement, where the value of the random variable is determined. The exact value of an outcome is unpredictable, but the frequency of the occurrence of a value is described by the probability density. Even if the probability density is typically a function, the value at a specific point has no direct meaning, only the integral over a certain range of values can be interpreted as probability. The terms probability density function (pdf) or probability distribution or simply distribution are used synonymously for a probability density. So when we write $p(x)$, it means the probability density associated with the random variable x or x is distributed as $p(x)$. In the function $p(x)$ x is only a placeholder and can be replaced by another symbol e.g. u. In this case we write $p(x = u)$, which means that in the functional form of the pdf associated with the random variable x, we use u as an argument. This notation is also used when we set the random variable to a fixed value e.g. $p(x = x_0)$. Any dependence of the probability density on a set of parameters $\theta_1, \theta_2, \ldots, \theta_n$ will be written as $p(x|\theta_1, \theta_2, \ldots, \theta_n)$ or in short form $p(x|\boldsymbol{\theta})$.

The definition of a probability density is given below:

Definition 1.1 (Probability Density) A probability density $p(x)$ is a generalized non-negative function of a variable x with a support of $[-\infty,+\infty]$ such that

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \tag{1.1}$$

The dimension of a probability density $p(x)$ is the inverse of the dimension of the associated random variable x.

A random variable is defined as

Definition 1.2 (Random Variable) A real-valued continuous variable x is called a random variable if x is distributed as $p(x)$ (also written as $x \sim p(x)$). This means that the distribution of values of the random variable x is completely described by the probability density $p(x)$.

A random variable with a finite support $[a,b]$ can be described by a probability density, which is zero outside of the interval $[a,b]$.

A probability density of two or more random variables is called a joint probability density e.g. $p(x,y)$ in the case of two random variables x and y. The normalization condition is given by

$$\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} p(x,y)dxdy = 1 \tag{1.2}$$

Integrating only over one random variable e.g. y, the result is again a valid probability density $p(x)$. This operation is called marginalization. Therefore the simplest way to construct a joint (or multi-variate) probability density out of uni-variate probability densities is by multiplication e.g. $p(x,y) = p(x)p(y)$.

One word of caution: if we say that x is distributed as $p(x)$ and y is distributed as $p(y)$, that does not mean that the probability distributions have the same functional form, but it is kept open if they are identical or different. Whenever they are identical, we explicitly mention this and say that x and y are identically distributed.

For multi-variate probability densities we follow the notation of writing the argument in bold letters or with an index range subscript e.g. $p(\mathbf{x})$ or $p(x_{1:n})$ is the short form for the joint probability density $p(x_1,x_2,\ldots,x_n)$.

The Wolfram Language represents a probability density by the function `ProbabilityDistribution` e.g. the following code creates a pdf $\frac{\sqrt{2}}{\pi\alpha\left(1+\left(\frac{x}{\alpha}\right)^4\right)}$

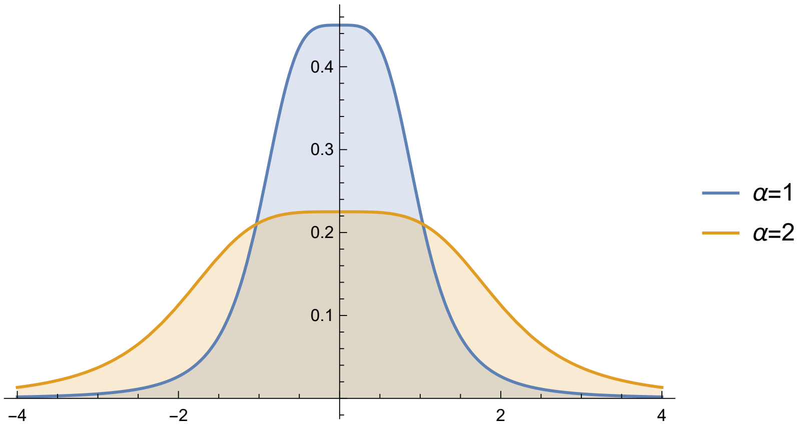
```
dist = ProbabilityDistribution[Sqrt[2]/(Pi*\[Alpha])/(1 + (x/\[Alpha])^4),
                               {x, -Infinity, Infinity},
                               Assumptions -> {\[Alpha] > 0}];
```

The `Assumptions` option $\alpha > 0$ is necessary to guarantee the non-negativity of the pdf. The function, which is provided as the first argument, must be normalized to unity or if not, the option `Method -> "Normalize"` must be explicitly set. An equivalent definition would be

```
dist = ProbabilityDistribution[1/(1 + (x/\[Alpha])^4),
                               {x, -Infinity, Infinity},
                               Method -> "Normalize",
                               Assumptions -> {\[Alpha] > 0}];
```

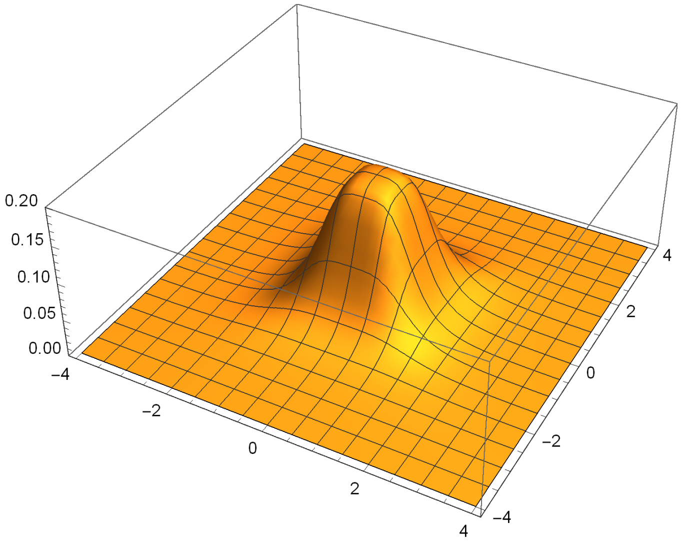
The distribution instance can then be used for plotting.

```
Plot[{PDF[dist, x] //. {\[Alpha] -> 1},
      PDF[dist, x] //. {\[Alpha] -> 2}},
      {x, -4, 4}, Filling -> Axis,
      PlotLegends -> Placed[{"\[Alpha]=1", "\[Alpha]=2"}, Right]]
```



ProbabilityDistribution can be also used for the creation of joint probability densities.

```
dist = ProbabilityDistribution[
          Gamma[3/4]/(Gamma[5/4]*Sqrt[2 Pi^3])/(1 + x^4 + y^4),
          {x, -Infinity, Infinity},
          {y, -Infinity, Infinity}];
Plot3D[PDF[dist, {x, y}], {x, -4, 4}, {y, -4, 4},
       PlotRange -> {0, 0.2}, PlotPoints -> {50, 50}]
```



Example 1.1 (The Univariate Normal Distribution) The most important probability density is the Normal (or Gauss) distribution $\mathcal{N}\left(x|\mu,\sigma^2\right)$, which has the functional form

$$\frac{1}{\sqrt{2\pi\sigma^2}}\exp-\frac{(x-\mu)^2}{2\sigma^2} \tag{1.3}$$

The function has two free parameters μ (mean) and σ (standard deviation) and fulfills the normalization condition for any value of μ and σ.

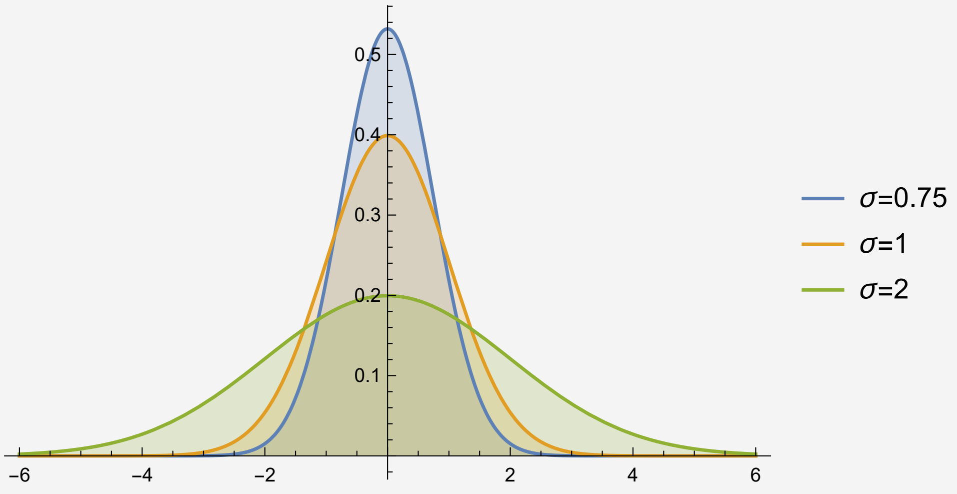
$$\frac{1}{\sqrt{2\pi\sigma^2}}\int_{-\infty}^{+\infty}\exp-\frac{(x-\mu)^2}{2\sigma^2}dx=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty}\exp-\frac{x^2}{2}dx=1 \tag{1.4}$$

where the variable transformation $x \rightarrow \frac{x-\mu}{\sigma}$ was used.

The notation $\mathcal{N}\left(x|\mu,\sigma^2\right)$ instead of $\mathcal{N}(x|\mu,\sigma)$ is motivated by the fact that the functional form only depends on σ^2 and by the generalization to the multi-variate Normal distribution, which is discussed in the next example.

The Wolfram Language implements the Normal distribution as `NormalDistribution`. Note that the second argument to `NormalDistribution` is the standard deviation σ, not the variance σ^2.

```
Plot[Table[PDF[NormalDistribution[0, \[Sigma]], x],
          {\[Sigma], {.75, 1, 2}}] // Evaluate,
     {x, -6, 6}, Filling -> Axis,
     PlotLegends -> Placed[{"\[Sigma]=0.75","\[Sigma]=1","\[Sigma]=2"}, Right]]
```



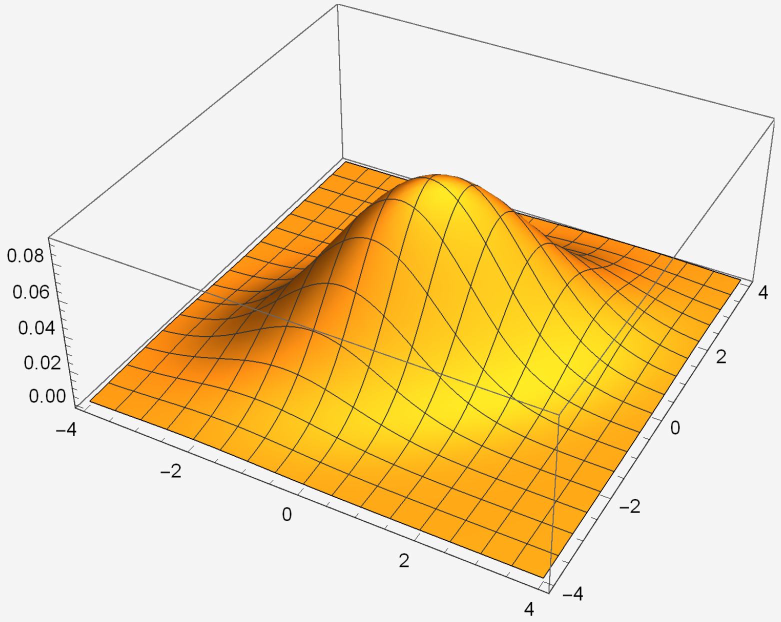
Example 1.2 (The Multivariate Normal Distribution) The multi-variate Normal (or Gauss) distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\Sigma)$ has the functional form

$$\frac{1}{\sqrt{2\pi\det(\Sigma)}}\exp-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{1.5}$$

with $\mathbf{x}=x_{1:n}, \boldsymbol{\mu}=\mu_{1:n}$ and $\Sigma=\Sigma_{1:n,1:n}$. Σ is called the covariance matrix and is symmetric $\Sigma^T=\Sigma$, so it has only $\frac{n(n+1)}{2}$ independent parameters. Therefore the multi-variate Normal distribution has $\frac{n(n+3)}{2}$ parameters.

The corresponding function in the Wolfram Language is `MultinormalDistribution`.

```
Plot3D[PDF[MultinormalDistribution[{0, 0}, {{2, 1}, {1, 2}}], {x, y}],
       {x, -4, 4}, {y, -4, 4}, PlotPoints -> {50, 50}]
```



The function `MarginalDistribution` can be used to reduce the number of variables in a multi-variate distribution. The result of

```
MarginalDistribution[MultinormalDistribution[{0, 0}, {{2, 1}, {1, 2}}], 1]
```

is the uni-variate Normal distribution $\mathcal{N}\left(x|\mu=0,\sigma^2=2\right)$. The second argument can be a single index or a list of multiple indices of the variables, which should be present in the new distribution.

Theorem 1.1 **(Multivariate Normal Distribution under Affine Transformation)** An important property of the multi-variate Normal distribution is that the result of an affine transformation $\mathbf{x}' = \mathrm{M}\mathbf{x} + \mathbf{b}$ with a $n \times n$ matrix M and a $n \times 1$ vector $\mathbf{b}$ is again a multi-variate Normal distribution

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\Sigma) = |\det(M)|\,\mathcal{N}\left(\mathbf{x}'|\mathrm{M}\boldsymbol{\mu}+\mathbf{b},\mathrm{M}\Sigma\mathrm{M}^T\right) \tag{1.6}$$

Proof.

$$\mathcal{N}\left(\mathbf{x}'|\mathrm{M}\boldsymbol{\mu}+\mathbf{b},\mathrm{M}\Sigma\mathrm{M}^T\right) = \frac{1}{\sqrt{2\pi\det\left(\mathrm{M}\Sigma\mathrm{M}^T\right)}}\exp-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathrm{M}^T\left(\mathrm{M}\Sigma\mathrm{M}^T\right)^{-1}\mathrm{M}(\mathbf{x}-\boldsymbol{\mu}) \tag{1.7}$$

Using $\det\left(\mathrm{M}\Sigma\mathrm{M}^T\right) = \det(\mathrm{M})^2\det(\Sigma)$ and $\mathrm{M}^T\left(\mathrm{M}\Sigma\mathrm{M}^T\right)^{-1}\mathrm{M} = \mathrm{M}^T(\mathrm{M}^T)^{-1}\Sigma^{-1}\mathrm{M}^{-1}\mathrm{M} = \Sigma^{-1}$, the right hand side of the expression simplifies to

$$\mathcal{N}\left(\mathbf{x}'|\mathrm{M}\boldsymbol{\mu}+\mathbf{b},\mathrm{M}\Sigma\mathrm{M}^T\right) = \frac{1}{|\det(M)|\sqrt{2\pi\det(\Sigma)}}\exp-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{1.8}$$

The normalization condition is also fulfilled.

$$\int_{-\infty}^{+\infty} d\mathbf{x}'\mathcal{N}\left(\mathbf{x}'|\mathrm{M}\boldsymbol{\mu}+\mathbf{b},\mathrm{M}\Sigma\mathrm{M}^T\right) = \frac{|\det(M)|}{|\det(M)|}\int_{-\infty}^{+\infty} d\mathbf{x}\mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu},\Sigma\right) = 1 \tag{1.9}$$

The short-hand notation $\int_{-\infty}^{+\infty} d\mathbf{x}$ is used for the n-dimensional integral $\prod_{i=1}^{n}\int_{-\infty}^{+\infty} dx_i$. □

Example 1.3 (The Uniform Distribution) The simplest probability density is the uniform distribution $\mathcal{U}(x|\alpha,\beta)$, which has the functional form

$$\frac{1}{\beta-\alpha}\Theta(\beta-x)\Theta(x-\alpha) \tag{1.10}$$

with $\beta>\alpha$. $\Theta(x)$ is the Heaviside step function, which is defined as

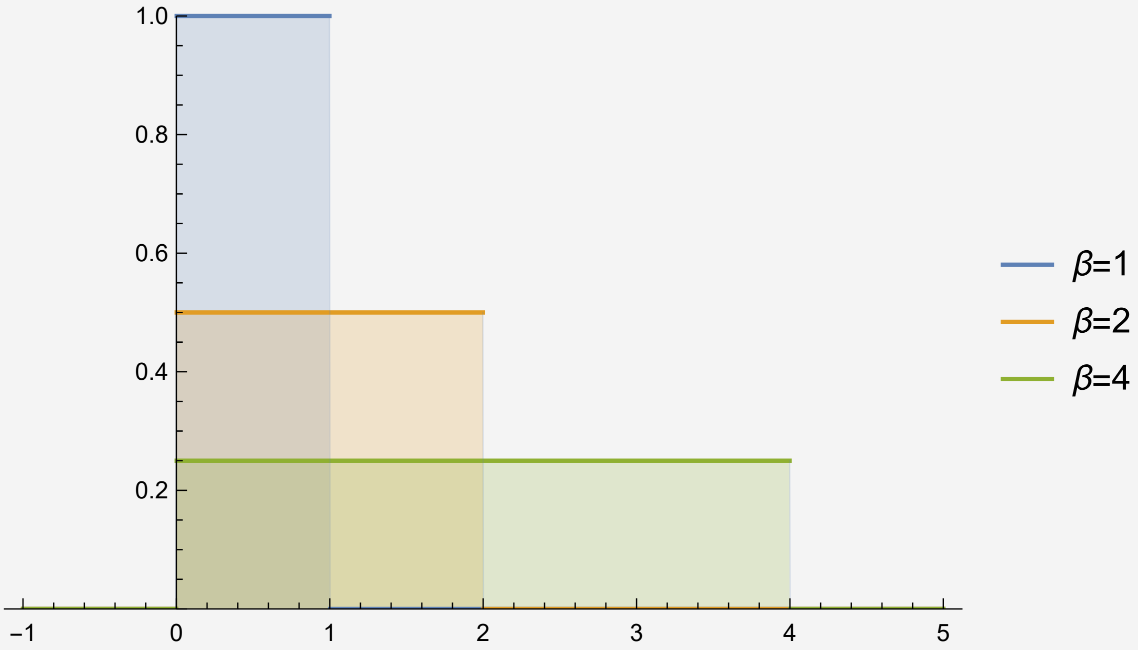
$$\Theta(x)=\begin{cases}0 & x<0\\ 1 & x\geqq 0\end{cases} \tag{1.11}$$

The two Θ-functions guarantee that the probability density is zero outside the interval $[a,b]$. The normalization condition can be easily verified by

$$\frac{1}{\beta-\alpha}\int_{-\infty}^{+\infty}\Theta(\beta-x)\Theta(x-\alpha)dx = \frac{1}{\beta-\alpha}\int_{\alpha}^{\beta} dx = \frac{\beta-\alpha}{\beta-\alpha} = 1 \tag{1.12}$$

Wolfram Language has a function `UniformDistribution` with a list argument $\{\alpha,\beta\}$.

```
Plot[Table[PDF[UniformDistribution[{0, \[Beta]}], x],
          {\[Beta], {1, 2, 4}}] // Evaluate,
     {x, -1, 5}, Filling -> Axis, PlotRange -> {0, 1},
     PlotLegends -> Placed[{"\[Beta]=1", "\[Beta]=2", "\[Beta]=4"}, Right]]
```



Exercise 1.1 Show that the sum of two random variables distributed as $\mathcal{U}(x|\alpha,\beta)$ is given by a triangular distribution between the two end points 2α and 2β and a peak at $\alpha+\beta$, which is built into the Wolfram Language as `TriangularDistribution`.

1.2 Probability

Probability is defined in terms of the probability density in the following way.

Definition 1.3 (Probability) The probability of x taking a value between x_1 and x_2 is given by

$$P(x_1 \leqq x \leqq x_2) = \int_{x_1}^{x_2} p(x)dx \tag{1.13}$$

In the case of a joint probability density, the probability of x taking a value between x_1 and x_2 and y taking a value between y_1 and y_2 is given by

$$P(x_1 \leqq x \leqq x_2 \ \& \, y_1 \leqq y \leqq y_2) = \int_{x_1}^{x_2} \int_{y_1}^{y_2} p(x,y)dxdy \tag{1.14}$$

The generalization to the case of a multi-variate probability density with more than 2 random variables is self-explanatory.

The convention to write probabilities with an upper case P and probability densities with a lower case p is adapted throughout the book.

We will now show that all the properties of a probability (also known as Kolmogorov axioms) follow from this definition.

Theorem 1.2 (Probability Axioms) The probability satisfies the following 3 axioms:

1. The probability is a non-negative real number.
2. The probability of x taking a value between $[-\infty,+\infty]$ is 1.
3. A set of disjoint intervals $\mathcal{I}_i$ with $i=1,\ldots,n$ has the probability $\sum_{i=1}^{n} P(\mathcal{I}_i)$.

Proof. Based on the definition of the probability as integral over a probability density, the proof is rather trivial. The first axiom follows directly from the non-negativity of the probability density, the second axiom from the normalization condition and the third axiom from the additivity of integrals over disjoint integration intervals, which simply follows from the interpretation of an integral as an area under a curve. □

Because probability is related to the integral of a probability density, a cumulative distribution function is defined as

Definition 1.4 (Cumulative Distribution Function) A cumulative distribution function (cdf) of a random variable x is given by

$$\Phi(x) = \int_{-\infty}^{x} p(x = x')dx' \tag{1.15}$$

The probability of x taking a value between x_1 and x_2 can then be written as the difference of two cumulative distribution functions evaluated at x_2 and x_1.

$$P(x_1 \leqq x \leqq x_2) = \Phi(x_2) - \Phi(x_1) \tag{1.16}$$

Wolfram Language has built in the functions `Probability` and `CDF`, which can be applied to any distribution e.g.

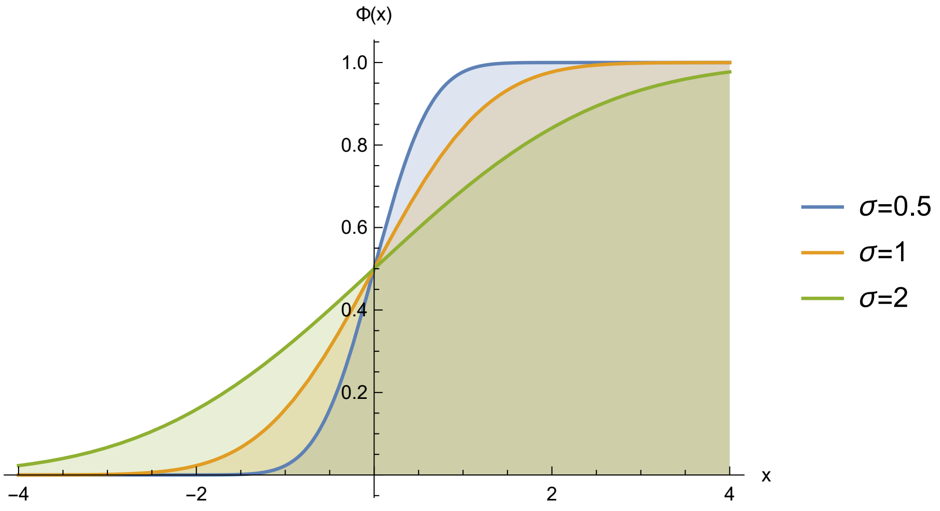
```
Probability[x <= 0, x \[Distributed] NormalDistribution[]]
```

with the result $\frac{1}{2}$, which can be also obtained using

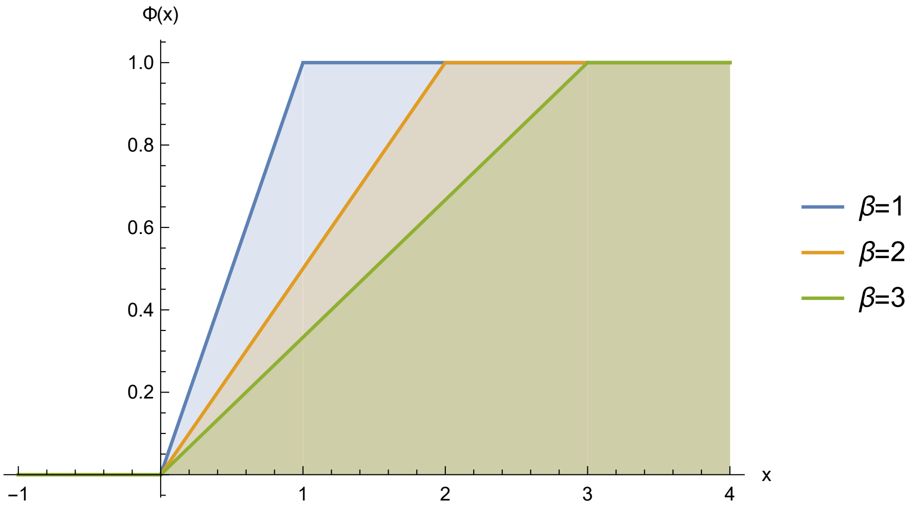
```
CDF[NormalDistribution[], 0]
```

The cumulative distribution function of the Normal and the Uniform distribution is shown in the next figures

```
Plot[Table[CDF[NormalDistribution[0, \[Sigma]], x],
     {\[Sigma], {0.5, 1, 2}}] // Evaluate,
     {x, -4, 4}, AxesLabel -> {"x", "\[CapitalPhi](x)"},
     Filling -> Axis, PlotLegends ->
     Placed[{"\[Sigma]=0.5", "\[Sigma]=1", "\[Sigma]=2"}, Right]]
```



```
Plot[Table[CDF[UniformDistribution[{0, \[Beta]}], x],
     {\[Beta], {1, 2, 3}}] // Evaluate,
     {x, -1, 4}, AxesLabel -> {"x", "\[CapitalPhi](x)"},
     Filling -> Axis, PlotLegends ->
  Placed[{"\[Beta]=1", "\[Beta]=2", "\[Beta]=3"}, Right]]
```



1.3 Conditional Probability Density

Almost all probability densities are conditional probability densities, because they depend on various parameters and/or other random variables. We will treat parameters and random variables behind the condition symbol in the same way, which means that we assume that they have a fixed value, when the probability density is evaluated. The only difference is that a random variable can be in front of or behind the condition symbol, whereas parameters can only be placed behind it.

A conditional probability density $p(x|y,\theta)$ means a probability density of x given y (a random variable) and θ (a parameter).

Definition 1.5 (Conditional Probability Density) The conditional probability $p(x|y,\theta)$ is defined by

$$p(x|y,\theta) = \frac{p(x,y|\theta)}{p(y|\theta)} = \frac{p(x,y|\theta)}{\int_{-\infty}^{+\infty} p(x,y|\theta)dx} \tag{1.17}$$

A similar equation holds for $p(y|x,\theta)$

$$p(y|x,\theta) = \frac{p(x,y|\theta)}{p(x|\theta)} = \frac{p(x,y|\theta)}{\int_{-\infty}^{+\infty} p(x,y|\theta)dy} \tag{1.18}$$

Eliminating the joint probability density $p(x,y|\theta)$ leads to the Bayes rule

$$p(x|y,\theta) = \frac{p(y|x,\theta)p(x|\theta)}{p(y|\theta)} \tag{1.19}$$

A joint probability density $p(\mathbf{x}|\mathbf{y},\boldsymbol{\theta})$ can be decomposed into univariate conditional probability densities by iteratively applying Bayes rule.

$$p(\mathbf{x}|\mathbf{y},\boldsymbol{\theta}) = p(x_{1:n}|\mathbf{y},\boldsymbol{\theta}) = p(x_1|x_{2:n},\mathbf{y},\boldsymbol{\theta})\underbrace{p(x_2|x_{3:n},\mathbf{y},\boldsymbol{\theta})\overbrace{\dots p(x_{n-1}|x_n,\mathbf{y},\boldsymbol{\theta})p(x_n|\mathbf{y},\boldsymbol{\theta})}^{p(x_{3:n}|\mathbf{y},\boldsymbol{\theta})}}_{p(x_{2:n}|\mathbf{y},\boldsymbol{\theta})} \tag{1.20}$$

This decomposition is not unique. Any permutations of **x** is possible here.

$$p(x_{1:n}|\mathbf{y},\boldsymbol{\theta}) = p(x_1|\mathbf{y},\boldsymbol{\theta})p(x_2|x_1,\mathbf{y},\boldsymbol{\theta})p(x_3|x_1,x_2,\mathbf{y},\boldsymbol{\theta})\ldots p(x_n|x_{1:n-1},\mathbf{y},\boldsymbol{\theta}) \tag{1.21}$$

Both sides of eq. 1.21 are equivalent in terms of the number of states and therefore contain the same amount of information. Let's assume that each random variable x_i with $i = 1,2\ldots n$ can take q values. Then the number of states of the rhs is $q^n - 1$, where the -1 comes from the normalization condition. Each univariate conditional probability density conrtibutes $q-1$ from the variable before the condition sign and q^m from the m variables after the condition sign. Therefore the number of states on the lhs is of the form of a geometric series $(q-1)\left(1+q+q^2+\ldots+q^{n-1}\right)$ with the sum $(q-1)\frac{q^n-1}{q-1}$, which is identical to the result of the rhs.

Exercise 1.2 Show that in the general case where each random variable x_i can take values q_i the number of states of the rhs is $q_1q_2\ldots q_n - 1$ and the lhs is $q_1 - 1 + (q_2-1)q_1 + (q_3-1)q_1q_2 + \ldots (q_n-1)q_1q_2\ldots q_{n-1}$ and that both sides are equal.

The question is: what is the best choice for this decomposition?

There is no general answer here, but the strategy would be to exploit the conditional independence of random variables as often as possible by applying

$$p(x|y,\theta) = p(x|\theta) \tag{1.22}$$

The statement x is conditionally independent from y means that the functional form of the probability density of x is the same regardless of the value of y. This requires additional or prior knowledge, which may or may not be available.

For example, using the Markov property that x_i only depends on x_{i-1} in eq. 1.21 simplifies the expression significantly to

$$p(x_{1:n}|\mathbf{y},\boldsymbol{\theta}) = p(x_1|\mathbf{y},\boldsymbol{\theta})p(x_2|x_1,\mathbf{y},\boldsymbol{\theta})p(x_3|x_2,\mathbf{y},\boldsymbol{\theta})\ldots p(x_n|x_{n-1},\mathbf{y},\boldsymbol{\theta}) \tag{1.23}$$

The number of states is reduced to $(q-1)(1+(n-1)q)$ and is therefore only quadratically in q compared to q^n.

Expressions with a product of probability densities will occur very often and are subject to numerical underflow. It is therefore recommended to perform numerical evaluations always with the logarithm of a probability density. It does not matter if it is the natural logarithm (we use ln *or* log *interchangeably) or the logarithm with base 2 or 10 (in this case we write explicitely* $\log_2$ *or* $\log_{10}$*). The crucial point is that a product of pdf's is converted into a sum of logarithms of pdf's, which is numerically more stable.*

Example 1.4 (A Normal Distribution with an Uncertain Parameter) Let's assume that a random variable x is distributed as $\mathcal{N}\left(x|\mu,\sigma^2\right)$, but we are uncertain about its mean μ. We can only say that its value is between α and β. So this uncertainty turns the parameter μ into a random variable distributed as $\mathcal{U}(\mu|\alpha,\beta)$ and we can ask what is the joint probability density $p(x, \mu)$.

$$p\left(x, \mu | \sigma^2, \alpha, \beta\right) = \mathcal{N}\left(x | \mu, \sigma^2\right) \mathcal{U}(\mu | \alpha, \beta) \tag{1.24}$$

Even if μ is now an random variable, the functional form of $\mathcal{N}\left(x | \mu, \sigma^2\right)$ is the same as before. This is the reason why no distinction is made between parameters and random variables behind the condition bar.

The Wolfram Language has no special function for conditional pdf's, because basically every probability density with parameters is a conditional pdf. Nevertheless it may be necessary to go from a joint pdf $p(x, y)$ to a conditional pdf $p(x|y)$.

```
joint = BinormalDistribution[{\[Mu]1, \[Mu]2}, {\[Sigma]1, \[Sigma]2}, \[Rho]];
dist2 = MarginalDistribution[joint, 2];
cond = ProbabilityDistribution[
          PDF[joint, {x1, x2}]/PDF[dist2, x2],
          {x1, -Infinity, Infinity}]
```

The result is

$$p\left(x_1 | x_2, \mu_1, \mu_2, \sigma_1, \sigma_2, \rho\right) = \frac{\exp\left(\frac{(x_2-\mu_2)^2}{2\sigma_2^2} - \frac{\frac{(x_1-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2} + \frac{(x_2-\mu_2)^2}{\sigma_2^2}}{2(1-\rho^2)}\right)}{\sqrt{2\pi}\sqrt{1-\rho^2}\,\sigma_1} \tag{1.25}$$

The normalization condition can be checked with

```
Integrate[PDF[cond, x], {x, -Infinity, Infinity},
   Assumptions -> {Element[\[Sigma]1, NonNegativeReals],
                   Element[\[Sigma]2, NonNegativeReals],
                   Element[\[Rho], NonNegativeReals],
                   \[Rho] < 1, \[Sigma]1 > 0}]
```

The result is exactly 1 if the constraints $\rho < 1$ and $\sigma_1 > 0$ are taken into account, because eq. 1.25 is singular at $\rho = 1$ and $\sigma_1 = 0$.

1.4 Deterministic Functions

We will now show that every function can be written as a conditional probability density. This will allow us to treat everything as probability density and mix deterministic and random behavior in our models.

Let's study the Normal distribution in the limit $\sigma^2 \to 0$, where the uncertainty of observing the value μ changes to certainty and the outcome of the random variable x is always μ.

Example 1.5 (The Normal Distribution with zero variance)

$$\lim_{\sigma^2 \to 0} \mathcal{N}\left(x | \mu, \sigma^2\right) = \lim_{\sigma^2 \to 0} \frac{1}{\sqrt{2\pi\sigma^2}} \exp - \frac{(x-\mu)^2}{2\sigma^2} = \delta(x - \mu) \tag{1.26}$$

As we can see from the example, the probability density becomes a Dirac delta function with the functional relation $x = \mu$ as argument.

Definition 1.6 (Deterministic Probability Density) Any function $y = f(x)$ can be written as conditional probability density in the following way.

$$p(y|x) = \delta(y - f(x)) \tag{1.27}$$

A common use-case is that we have a conditional probability density, which will be become deterministic by including hidden (or latent) random variables. The random character of a quantity could be completely originate from another random variable and as soon as this other random variable has been identified and included, the conditional probability density can be replaced by an expression of the form shown above if the functional dependency on the hidden variable is known.

Another consequence is that an integration over y can be carried out immediately using

$$\int_{-\infty}^{+\infty} \delta(y - f(x))g(y)dy = g(f(x)) \tag{1.28}$$

Another use-case are variable transformations from a random variable x with a known probability density $p(x)$ to another variable $y = f(x)$.

The question is: what is the pdf of y?

The starting point for the case with two random variables x and y is always the joint probability density $p(x,y)$. Because we are only interested in the pdf of y, we marginalize over x.

$$p(y) = \int_{-\infty}^{+\infty} p(x,y)dx = \int_{-\infty}^{+\infty} p(y|x)p(x)dx = \int_{-\infty}^{+\infty} \delta(y - f(x))p(x)dx \tag{1.29}$$

Using the following identity for the Dirac delta function

$$\delta(y - f(x)) = \sum_{i=1}^{n} \frac{1}{|f'(x_i)|}\delta(x - x_i(y)) \tag{1.30}$$

where $x_i(y)$ are all n solutions of $y = f(x)$ and $f'(x_i)$ is the derivative of $f(x)$ with respect to x evaluated at $x = x_i(y)$, one can carry out the integration over x with the result

$$p(y) = \sum_{i=1}^{n} \frac{1}{|f'(x_i)|}p(x = x_i(y)) \tag{1.31}$$

Example 1.6 (The Square of a Random Variable) What is the pdf of $y = x^2$ if $x \sim p(x)$? The equation $y = x^2$ has two solutions $x_1 = +\sqrt{y}, x_2 = -\sqrt{y}$. The derivative of x^2 is $2x$.

$$p(y) = \sum_{i=1}^{2} \frac{1}{|2x_i|}p(x = x_i(y)) = \frac{1}{2\sqrt{y}}\left(p\left(x = +\sqrt{y}\right) + p\left(x = -\sqrt{y}\right)\right) \tag{1.32}$$

y is only defined for values in the range $[0,\infty]$. Therefore we have to add $\Theta(y)$.

$$p(y) = \frac{1}{2\sqrt{y}}\left(p\left(x = +\sqrt{y}\right) + p\left(x = -\sqrt{y}\right)\right)\Theta(y) \tag{1.33}$$

Example 1.7 (The Square of a Gaussian Zero-Mean Random Variable) What is the pdf of $y = x^2$ if $x \sim \mathcal{N}(x|0,\sigma^2)$? The result from the previous example can be used to obtain

$$p(y) = \frac{1}{\sqrt{2\pi\sigma^2 y}} \exp\left(-\frac{y}{2\sigma^2}\right)\Theta(y) \tag{1.34}$$

The probability density is infinite at $y = 0$, but it is still a valid distribution, because the normalization condition is fulfilled and it is non-negative for all values of y. A probability density can be singular at some points, as long as the integral over it is finite.

In the Wolfram Language the same result can be obtained with

```
dist = TransformedDistribution[x^2, x \[Distributed] NormalDistribution[0,\[Sigma]]];
PDF[dist, y]
```

The singular value at $y = 0$ will be shown as `Indeterminate`.

Example 1.8 (A Deterministic Function with Additive Gaussian Noise) What is the pdf of a random variable $y = f(x) + \eta$ with $\eta \sim \mathcal{N}\left(\eta|\mu,\sigma^2\right)$? A typical use-case is the measurement of the state x of a system. The outcome of this measurement is y, where the function f describes the measurement process, and η is the measurement noise. As in the previous examples, the starting point is the joint probability density $p(y,\eta|x)$. Because we are only interested in y, we marginalize over η.

$$p(y|x) = \int_{-\infty}^{\infty} p(y,\eta|x)d\eta \tag{1.35}$$

In the next step, we apply Bayes rule.

$$p(y|x) = \int_{-\infty}^{\infty} p(y|x,\eta)\mathcal{N}\left(\eta|\mu,\sigma^2\right)d\eta \tag{1.36}$$

$p(\eta|x) = p(\eta) = \mathcal{N}\left(\eta|\mu,\sigma^2\right)$ was used here, because η is independent of x. $p(y|x,\eta)$ is a deterministic function and can therefore be written as $\delta(y - f(x) - \eta)$. Carrying out the η - integration gives the following result

$$p\left(y|x,\mu,\sigma^2\right) = \mathcal{N}\left(y - f(x)|\mu,\sigma^2\right) = \mathcal{N}\left(y|\mu + f(x),\sigma^2\right) \tag{1.37}$$

The probability density of y is also a Normal distribution with $\mu \rightarrow \mu + f(x)$.

1.5 Discrete Random Variables

So far, we have studied continuous random variables and their associated probability densities. Next, we look at discrete random variables, which only take values from a discrete set x_i with $i \in \mathbb{Z}$.

How can we construct a probability density for discrete random variables?

We assign a probability P_i to each x_i and the corresponding discrete probability density is given by

Definition 1.7 (Discrete Probability Density)

$$p(x) = \sum_{i=-\infty}^{+\infty} P_i \delta(x - x_i) \tag{1.38}$$

Note that x is still a continuous variable, but due to the Dirac delta functions it is restricted to the discrete values $x_i = f(i)$, which can be an arbitrary function of the integer-valued index i. We write here the coefficients with a capital P to indicate that these numbers are indeed probabilities. The normalization condition simplifies to a sum over all probabilities.

$$\int_{-\infty}^{+\infty} p(x)dx = \sum_{i=-\infty}^{+\infty} P_i = 1 \tag{1.39}$$

The definition shows that in the case of discrete random variables, probability densities $p(x)$ reduce to probabilities P_i and integrals to sums over all the possible values of x. If the index range is limited to a finite set e.g. $i = 0, 1, \ldots, n$, $P_i = 0$ for $i < 0$ and $i > n$.

While eq. 1.38 is useful to express continuous and discrete probability densities in the same way, it has its limitations e.g. the logarithm or powers of $p(x)$ don't make sense due to the presence of Dirac delta-functions and are ill-defined. Only convolution $\int_{-\infty}^{+\infty} \delta(x-y)\delta(y-z)\,dy = \delta(x-z)$ is a valid operation. The next example highlights some of the issues you have to deal with.

Example 1.9 (Entropy) The entropy for a continuous random variable $x \sim p(x)$ (or differential entropy) is defined as

$$h(x) = -\int_{-\infty}^{+\infty} p(x) \log_2(p(x)) \tag{1.40}$$

From a physical perspective this definition has a flaw: a probability density $p(x)$ has a physical unit if x has a physical unit. Taking the logarithm of non-dimensionless quantity is a forbidden operation. To avoid the problem, an arbitrary factor Λ with the same unit as $p(x)$ has to be added.

$$h(x) = -\int_{-\infty}^{+\infty} p(x) \log_2\left(\frac{p(x)}{\Lambda}\right) = -\int_{-\infty}^{+\infty} p(x) \log_2(p(x)) + \log_2(\Lambda) \tag{1.41}$$

Also the choice of the base of the logarithm is arbitrary. One could also use the natural logarithm (preferred in physics) or any other base in the definition of the entropy. Base 2 is the preferred choice in information theory. The difference is again an additive constant, which leads to the conclusion that differential entropy is only defined up to an additive constant.

If we insert eq. 1.38 in this expression, we get

$$\begin{aligned} h(x) &= -\sum_{i=-\infty}^{+\infty} \int_{-\infty}^{+\infty} P_i \delta(x-x_i) \log_2(p(x)) = -\sum_{i=-\infty}^{+\infty} P_i \log_2(p(x_i)) \\ &= -\sum_{i=-\infty}^{+\infty} P_i \log_2(P_i \delta(0)) = -\sum_{i=-\infty}^{+\infty} P_i \log_2(P_i) - \log_2(\delta(0)) \end{aligned} \tag{1.42}$$

The step, where we replaced $p(x_i)$ by $P_i\delta(0)$ treated $\delta(x)$ as a function, which is zero for any non-zero argument, therefore the sum inside the logarithm is reduced to only one term.

The last term $\log_2(\delta(0))$ is divergent, but independent of the random variable x and therefore plays no role, because entropy is anyhow defined up to an additive factor due to the arguments presented above.

So the conclusion is that in this case the result for the entropy of a discrete random variable comes out right if you renormalize the divergent term away (as the physicists would call it).

The next paragraphs will discuss important examples of discrete probability densities, which are encountered in many different contexts.

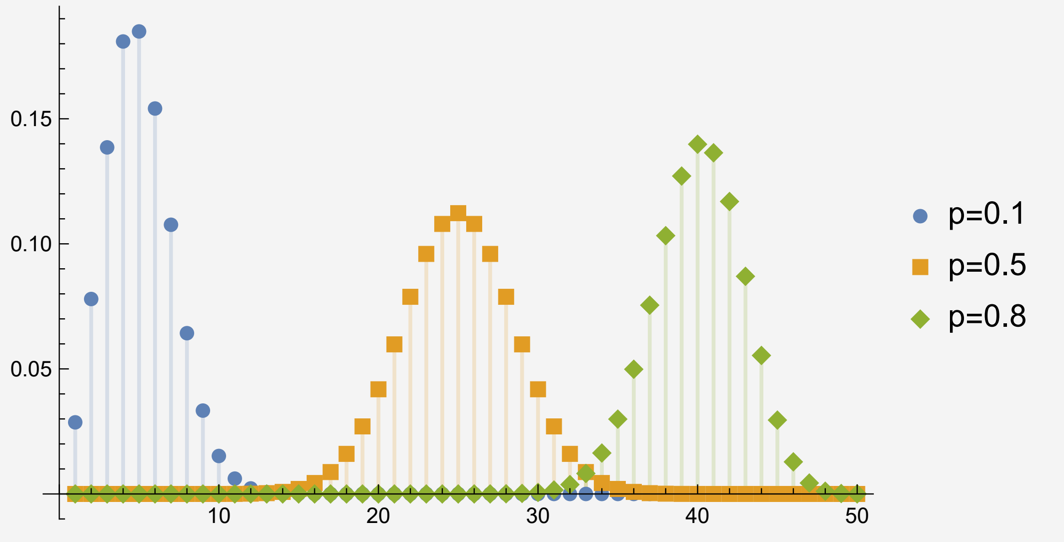
Example 1.10 (The Binomial Distribution) The binomial distribution is one of the most important discrete probability densities. x_k is e.g. the outcome of a sequence of n coin flips with k heads and $n-k$ tails. The probability of head in a coin flip is p, while the probability of tail is $1-p$.

$$\mathcal{B}(x|\, n, p) = \sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} \delta(x - x_k) \tag{1.43}$$

Let' say that we flip a fair coin with $p = \frac{1}{2}$ $n = 3$ times and we want to calculate the probability that we have $k = 2$ heads and $n - k = 1$ tails. All coin flip sequences of the form hht, hth, thh contribute to it. Therefore the probability is $\frac{3}{8}$. The same probability is obtained in the case $k = 1$ heads and $n - k = 2$ tails. For the case $k = 3$ heads and 0 tails there is only one possibility: hhh. Same for $k = 0$ heads and $n - k = 3$ tails. Therefore the probability in these cases is $\frac{1}{8}$. The sum of all probabilities is $\frac{1+3+3+1}{8} = 1$.

Wolfram Language has the built-in function `BinomialDistribution`.

```
DiscretePlot[Table[PDF[BinomialDistribution[100, p], k],
            {p, {0.1, 0.5, 0.8}}] // Evaluate,
            {k, 90}, PlotRange -> All, PlotMarkers -> Automatic,
            PlotLegends -> {"p=0.1", "p=0.5", "p=0.8"}]
```



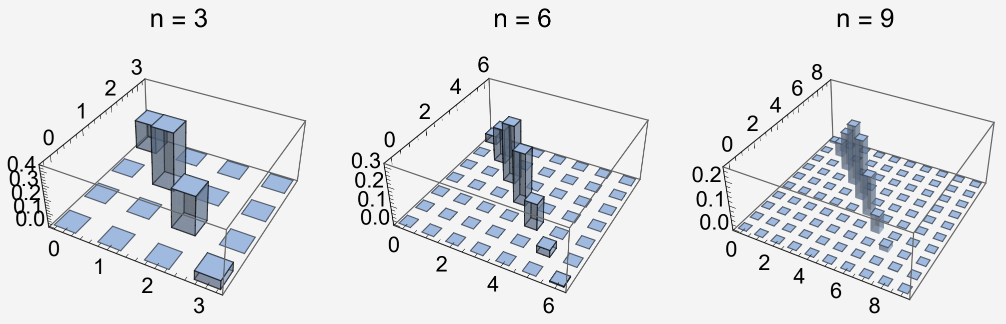
Example 1.11 (The Multinomial Distribution) The multinomial distribution $\mathcal{M}(\mathbf{x}|n,\mathbf{p})$ is a generalization of the binomial distribution, where the number of possible outcomes is also a variable m.

$$\mathcal{M}(x_{1:m}|n,p_{1:m}) = \sum_{n_1=0}^{n}\sum_{n_2=0}^{n}\ldots\sum_{n_m=0}^{n}\frac{n!}{n_1!n_2!\ldots n_m!}p_1^{n_1}p_2^{n_2}\ldots p_m^{n_m}\prod_{i=1}^{m}\delta(x_i - x_{n_i}) \qquad (1.44)$$

with the constraints $\sum_{i=1}^{m} n_i = n$ and $\sum_{i=1}^{m} p_i = 1$. An application of the multinomial distribution in the context of histograms will be discussed in 2.1.

Wolfram language provides the function `MultinomialDistribution`

```
GraphicsRow[Table[DiscretePlot3D[
   PDF[MultinomialDistribution[n, {0.6, 0.4}], {x, y}],
   {y, 0, n}, {x,0, n}, PlotLabel -> Row[{"n = ", n}],
   ExtentSize -> 0.5], {n, {3, 6, 9}}], ImageSize -> Full]
```



Example 1.12 (The Poisson Distribution) In the limit $n \to \infty$ and $\lambda = np$ remains finite,

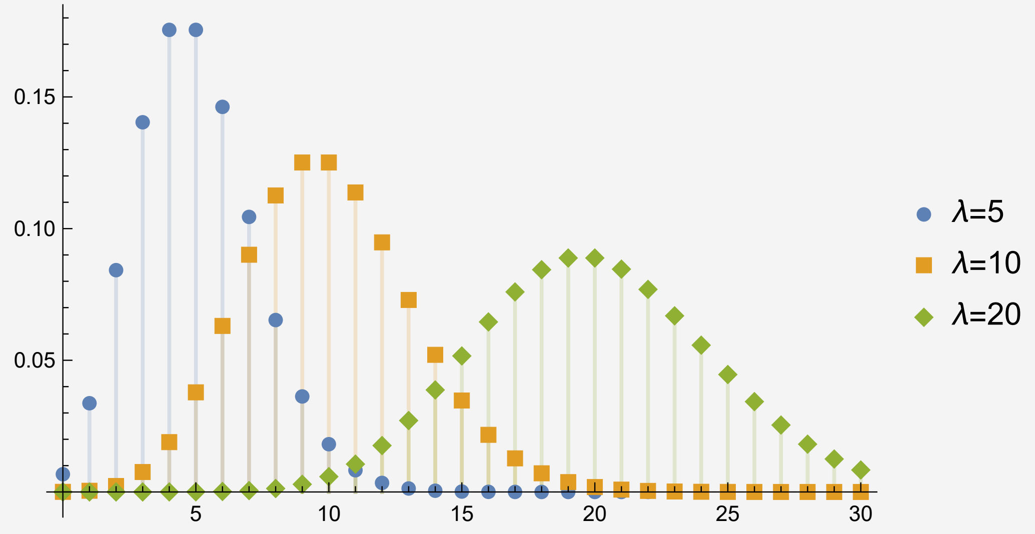
the binomial distribution becomes the Poisson distribution given by

$$\mathcal{P}(x|\lambda) = \sum_{k=0}^{\infty} \frac{\lambda^k}{k!} \exp(-\lambda)\delta(x - x_k) \tag{1.45}$$

This distribution describes the probability of observing k events given a mean event count of λ.

Wolfram Language has the built-in function `PoissonDistribution`.

```
DiscretePlot[Table[PDF[PoissonDistribution[\[Lambda]], k],
             {\[Lambda], {5, 10, 20}}] // Evaluate,
             {k, 0, 30}, PlotRange -> All, PlotMarkers -> Automatic,
             PlotLegends -> {"\[Lambda]=5", "\[Lambda]=10", "\[Lambda]=20"}]
```



Theorem 1.3 **(Sum of two Poisson distributions)** Show that $z = x + y$ is distributed as a Poisson-distribution $P(z|\lambda_1 + \lambda_2)$ if $y \sim \mathcal{P}(x|\lambda_1)$ and $y \sim \mathcal{P}(y|\lambda_2)$.

Proof.

$$p(z) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} p(z,x,y)dxdy = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} p(z|x,y)\mathcal{P}(x|\lambda_1)\mathcal{P}(y|\lambda_2)dxdy \tag{1.46}$$

Bayes rule was used in last step. $p(z|x,y)$ is a deterministic function $z = x + y$ and is given by $\delta(z - x - y)$. The y-integration can be carried out and gives the following result.

$$p(z) = \int_{-\infty}^{+\infty} P(x|\lambda_1)P(z - x|\lambda_2)dx \tag{1.47}$$

Inserting the expressions for the Poisson distribution and carrying out the x-integration leads to

$$p(z) = \exp(-\lambda_1 - \lambda_2)\sum_{k=0}^{\infty}\frac{\lambda_1^k}{k!}\sum_{l=0}^{\infty}\frac{\lambda_2^l}{l!}\delta(z - x_k - y_l) \tag{1.48}$$

We set now $z_n = x_k + y_l$ with $n = k + l$ by introducing a third sum $\sum_{n=0}^{\infty} \delta_{n,k+l} = 1$ and carrying out the summation over l

$$p(z) = \exp(-\lambda_1 - \lambda_2) \sum_{n=0}^{\infty} \frac{1}{n!} \underbrace{\sum_{k=0}^{n} n! \frac{\lambda_1^k}{k!} \frac{\lambda_2^{n-k}}{(n-k)!}}_{(\lambda_1+\lambda_2)^n} \delta(z - z_n) \tag{1.49}$$

We used here $\delta_{n,k+l} = \delta_{n-k,l}$, which replaces l by $n-k$ and limits the sum over k to n, because l is positive and $n-k$ is negative for $k > n$ and the Kronecker delta is therefore always zero. □

In the Wolfram Language the proof is rather simple.

```
TransformedDistribution[u + v,
                        {u \[Distributed] PoissonDistribution[\[Lambda]1],
                         v \[Distributed] PoissonDistribution[\[Lambda]2]}]
```

A custom discrete probability function from a list of values x_i with their associated probabilities P_i can be defined with `EmpiricalDistribution`.

```
values = {0, 1, 2, 3};
probabilities = {0.2, 0.3, 0.3, 0.2};
p = EmpiricalDistribution[probabilities -> values];
```

`PDF[p,x]` returns

```
   0.2 Boole[0 == x] + 0.3 Boole[1 == x] + 0.3 Boole[2 == x] + 0.2 Boole[3 ==
x]
```

where the `Boole` function is 1 if the argument is `True`, otherwise 0.

Exercise 1.3 Show that $z_n = \sum_{i=1}^{n} x_i$ is distributed as a Poisson-distribution $P\left(z_n \,|\, \sum_{i=0}^{n} \lambda_i\right)$ if $x_i \sim \mathcal{P}(x_i \,|\, \lambda_i)$. Proof this statement by induction using $z_n = z_{n-1} + x_n$.

Exercise 1.4 Show that the probability density of $z = x + y$ can be written as the convolution of the probability densities of x and y.

$$p(z) = \int_{-\infty}^{+\infty} p(x)\, p(y = z - x)\, dx \tag{1.50}$$

Exercise 1.5 Show that $z = x + y$ is distributed as $\mathcal{N}\left(z | \mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2\right)$ if $y \sim \mathcal{N}(x | \mu_1, \sigma_1^2)$ and $y \sim \mathcal{N}(y | \mu_2, \sigma_2^2)$. The first part of the derivation is similar to the one for the Poisson distribution with the result

$$p(z) = \int_{-\infty}^{+\infty} \mathcal{N}\left(x | \mu_1, \sigma_1^2\right) \mathcal{N}\left(z - x | \mu_2, \sigma_2^2\right) dx \tag{1.51}$$

The x integration can be simplified using the variable transformation $x \to \frac{x-\mu_1-\mu_2}{\sqrt{\sigma_1^2+\sigma_2^2}}$.

1.6 Expectation, Mean and Variance

The expectation value of a function $f(x)$ of a random variable x, which is distributed as $p(x)$, is defined as

Definition 1.8 (Expectation)

$$\mathbb{E}_{x \sim p(x)}[f(x)] = \int_{-\infty}^{+\infty} f(x)p(x)dx \tag{1.52}$$

While the probability density contains all the information needed to perform calculations, it is often useful to characterize its shape by two values: mean and variance.

Definition 1.9 (Mean)

$$\mu = \mathbb{E}_{x \sim p(x)}[x] = \int_{-\infty}^{+\infty} xp(x)dx \tag{1.53}$$

Definition 1.10 (Variance)

$$\sigma^2 = \mathbb{E}_{x \sim p(x)}[(x-\mu)^2] = \int_{-\infty}^{+\infty} (x-\mu)^2 p(x)dx \tag{1.54}$$

The normal distribution $\mathcal{N}\left(x|\mu,\sigma^2\right)$ is completely characterized by these values, while other distributions have non-zero expectation value of higher powers of $x-\mu$.

Wolfram Language provides the three functions `Expectation`, `Mean` and `Variance`, which can be applied to a distribution e.g. the multi-variate Normal distribution

```
Expectation[{x1, x2}, {x1, x2} \[Distributed]
  MultinormalDistribution[{\[Mu]1, \[Mu]2},
                          {{\[CapitalSigma]11, \[CapitalSigma]12},
                           {\[CapitalSigma]12, \[CapitalSigma]22}}]]
Mean[MultinormalDistribution[{\[Mu]1, \[Mu]2},
                             {{\[CapitalSigma]11, \[CapitalSigma]12},
                              {\[CapitalSigma]12, \[CapitalSigma]22}}]]
Variance[MultinormalDistribution[{\[Mu]1, \[Mu]2},
                                 {{\[CapitalSigma]11, \[CapitalSigma]12},
                                  {\[CapitalSigma]12, \[CapitalSigma]22}}]]
```

The results of these three commands is $\{\mu_1, \mu_2\}$, $\{\mu_1, \mu_2\}$ and $\{\Sigma_{11}, \Sigma_{22}\}$.

`Variance` returns only the diagonal elements of the covariance matrix. The function `Covariance` returns the full covariance matrix.

```
Covariance[MultinormalDistribution[{\[Mu]1, \[Mu]2},
                                   {{\[CapitalSigma]11, \[CapitalSigma]12},
                                    {\[CapitalSigma]12, \[CapitalSigma]22}}]]
```

The result is $\begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{12} & \Sigma_{22} \end{pmatrix}$.

Example 1.13 (Mean and Variance of Poisson Distribution) What is the mean and variance of a random variable x distributed as a Poisson distribution $\mathcal{P}(x|\lambda)$ if the possible values x can take are $x_k = k$?
The mean μ is given by

$$\begin{aligned} \mu(\lambda) &= \sum_{k=0}^{\infty} P_k k = \sum_{k=1}^{\infty} k \frac{\lambda^k}{k!} \exp(-\lambda) \\ &= \exp(-\lambda)\lambda \sum_{k=1}^{\infty} \frac{\lambda^{k-1}}{(k-1)!} = \exp(-\lambda)\lambda \underbrace{\sum_{k=0}^{\infty} \frac{\lambda^k}{k!}}_{\exp(\lambda)} = \lambda \end{aligned} \tag{1.55}$$

The summation range was changed from $k = 0$ to $k = 1$, because the $k = 0$ term is identically zero.
The variance σ^2 can be calculated as

$$\begin{aligned} \sigma(\lambda)^2 &= \sum_{k=0}^{\infty} P_k (k-\lambda)^2 = \sum_{k=0}^{\infty} P_k k^2 - 2\lambda \underbrace{\sum_{k=0}^{\infty} P_k k}_{\lambda} + \lambda^2 \underbrace{\sum_{k=0}^{\infty} P_k}_{1} \\ &= \exp(-\lambda) \sum_{k=0}^{\infty} k^2 \frac{\lambda^k}{k!} - \lambda^2 \end{aligned} \tag{1.56}$$

Applying $\lambda \frac{\partial}{\partial \lambda}$ to $\mu(\lambda)$ gives the following identity

$$\lambda \frac{\partial \mu(\lambda)}{\partial \lambda} = \sum_{k=0}^{\infty} k^2 \frac{\lambda^k}{k!} \exp(-\lambda) - \lambda \underbrace{\sum_{k=0}^{\infty} k \frac{\lambda^k}{k!} \exp(-\lambda)}_{\lambda} = \lambda \tag{1.57}$$

This leads to the final result for σ^2.

$$\sigma(\lambda)^2 = \underbrace{\exp(-\lambda) \sum_{k=0}^{\infty} k^2 \frac{\lambda^k}{k!}}_{\lambda^2+\lambda} - \lambda^2 = \lambda \tag{1.58}$$

Example 1.14 (The Central Limit Theorem) This theorem states that the mean μ of an independent random sample $x_1, x_2, ... x_N$ with x_i being distributed according to an arbitrary distribution $p(x)$ tends to a normal distribution in the limit $N \to \infty$. We choose for $p(x)$ the uniform distribution $\mathcal{U}(x| -1, +1)$, which has zero mean and $\sigma^2 = \frac{1}{3}$ (see next exercise).
Starting point as usual is the joint probability density $p(\mu, x_1, x_2, \ldots, x_N)$, which we marginalize over $x_1, x_2, ... x_N$ to get $p(\mu)$.

$$p(\mu) = \int_{-\infty}^{+\infty} dx_1 \int_{-\infty}^{+\infty} dx_2 \ldots \int_{-\infty}^{+\infty} dx_N \, p(\mu, x_1, x_2, \ldots, x_N) \tag{1.59}$$

The expression can be further simplified

$$p(\mu) = \int_{-\infty}^{+\infty} dx_1 \int_{-\infty}^{+\infty} dx_2 \ldots \int_{-\infty}^{+\infty} dx_N p(\mu|x_1, x_2, \ldots, x_N) \prod_{i=1}^{N} p(x_i) \tag{1.60}$$

Because μ is the sample mean, the conditional pdf $p(\mu|x_1, x_2, \ldots, x_N)$ can be replaced by $\delta\left(\mu - \frac{1}{N}(x_1 + x_2 + \ldots + x_N)\right)$. We also use the following integral representation of the delta function.

$$\delta(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ikx} \tag{1.61}$$

Then the expression for $p(\mu)$ can be rewritten as

$$p(\mu) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} \prod_{i=1}^{N} \int_{-\infty}^{+\infty} dx_i p(x_i) e^{-ikx_i/N} \tag{1.62}$$

Now we use the fact that $p(x_i)$ is a uniform distribution in the interval $(-1, +1)$.

$$p(\mu) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} \prod_{i=1}^{N} \frac{1}{2} \int_{-1}^{+1} dx_i e^{-ikx_i/N} \tag{1.63}$$

The integration over the N variables x_i can now be carried out.

$$\begin{aligned} p(\mu) &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} \left(\frac{-N}{2ik}\right)^N \left(e^{-ik/N} - e^{ik/N}\right)^N \\ &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} \left(\frac{N}{k} \sin(k/N)\right)^N \end{aligned} \tag{1.64}$$

For large N the sine function can be expanded as $\sin(x) = x - \frac{x^3}{3!} + \ldots$ ignoring higher-order terms.

$$\begin{aligned} p(\mu) &\approx \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} \left(1 - \frac{(k/N)^2}{3!}\right)^N \\ &\approx \frac{1}{2\pi} \int_{-\infty}^{+\infty} dk e^{ik\mu} e^{-k^2/(3!N)} \\ &= \frac{1}{\sqrt{2\pi}} \mathcal{F}\{e^{-k^2/(3!N)}\}[\mu] \end{aligned} \tag{1.65}$$

using $\left(1 + \frac{x}{N}\right)^N = e^x$ for $N \to \infty$ and the Fourier transform $\mathcal{F}$ in the last step.
Using the Wolfram Language to solve the integral

```
1/Sqrt[2*Pi]*FourierTransform[Exp[-k**2/(Factorial[3]*N)], k, \[Mu]]
```

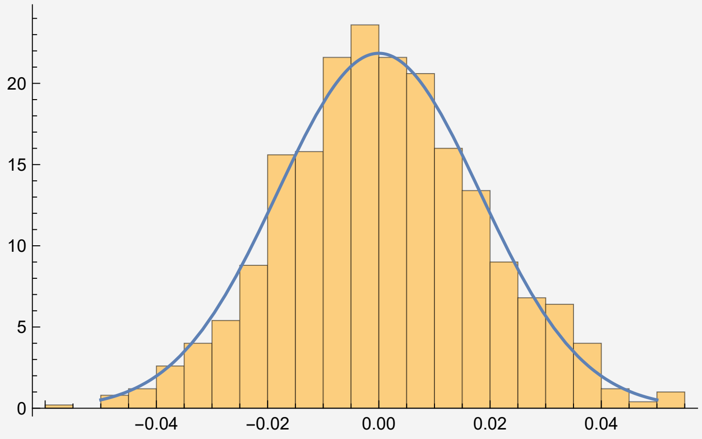
gives

$$p(\mu) \approx \sqrt{\frac{3N}{2\pi}} e^{-3N\mu^2/2} \tag{1.66}$$

which is a normal distribution with $\mu' = 0$ and $\sigma'^2 = \frac{\sigma^2}{N} = \frac{1}{3N}$.

The following code generates 10^6 uniformly distributed values, reshapes the array to a matrix and takes the mean over each column vector. The distribution of the mean values is compared to the expected Normal distribution.

```
NumPoints = 1000000;
x = RandomVariate[UniformDistribution[{-1, 1}], NumPoints];
x = ArrayReshape[x, {1000, 1000}];
Show[Histogram[Mean[x], 20, "PDF"],
Plot[PDF[NormalDistribution[0, 1/Sqrt[3000]], u], {u, -0.05, 0.05}]]
```



The mean value distribution has the expected Gaussian shape.

Exercise 1.6 Show that the mean and variance of a random variable x distributed as a uniform distribution $\mathcal{U}(x|\alpha,\beta)$ is given by $\mu = \frac{\alpha+\beta}{2}$ and $\sigma^2 = \frac{(\beta-\alpha)^2}{12}$.

1.7 Examples

In this chapter we apply the learned concepts to some interesting examples by using both analytical and computational approaches to solve the problem.

Example 1.15 (The Monty Hall Problem) This is a famous and non-trivial example of a probabilistic problem, where intuition will give you the wrong result. The problem statement is as follows:

Monty Hall, after which the problem is named, was a moderator of a game show and a participant has to choose between 3 doors. Behind one door a prize is hidden. After the choice has been made e.g. door 2, the moderator opens a door with no prize e.g. door 1, and asks the participant if he wants to choose the remaining door (in this case door 3) or if he wants to stay with his decision (door 2).

The question is now: What is the probability to win if the participant stays with his first decision or if he chooses the remaining door?

The difficult part is to cast the problem into a manageable form, which means, selecting proper random variables to make the computation as easy as possible. A discrete random variable x_1 can be associated with the action that the participant chooses a door randomly at the beginning and the choice is either a door with a prize behind it $x_1 = p$ or a door with no prize $x_1 = n$ (*note that this is not known, when* *the choice happens, but the crucial point is that the probability for its occurrence can be calculated easily with the result $P(x_1 = p) = \frac{1}{3}$ and $P(x_1 = n) = \frac{2}{3}$, because there are only 3 doors and one contains a prize and the two others not).*

Another discrete random variable x_2 will be associated with the outcome of the game (participant wins $x_2 = w$ or looses $x_2 = l$). This is the random variable, which is needed to answer the question above.

Furthermore we introduce a parameter θ, which describes the two possible ways: participant stays with his first decision ($\theta = 0$) or participant chooses the remaining door ($\theta = 1$).

After the random variables and parameters have been defined, the starting point is always the joint probability $P(x_2, x_1|\theta)$. Because we are only interested in the probability of winning the game, we can marginalize over x_1 and use Bayes rule to separate $P(x_1)$, which is known.

$$P(x_2 = w|\theta) = \sum_{x_1=p,n} P(x_2 = w, x_1|\theta) = \sum_{x_1=p,n} P(x_2 = w|x_1, \theta)P(x_1) \tag{1.67}$$

We only need to calculate the conditional probability $P(x_2|x_1, \theta)$. We will see that this quantity is completely deterministic.

Let's study first the case $\theta = 0$.

In this case if x_1 was already the right door with the prize, the participant will win, and if x_1 was the wrong door, he will loose regardless if the moderator opens another door without a prize.

$$P(x_2|x_1, \theta = 0) = \begin{cases} 1 & x_2 = w \ \& \ x_1 = p \\ 0 & x_2 = l \ \& \ x_1 = p \\ 1 & x_2 = l \ \& \ x_1 = n \\ 0 & x_2 = w \ \& \ x_1 = n \end{cases} \tag{1.68}$$

Next we handle the case $\theta = 1$.

In this case if x_1 was already the right door with the prize, the participant will loose when changing to the remaining door and if x_1 was the wrong door, he will win, because the moderator has opened the door without no prize, so the remaining door must contain the prize.

$$P(x_2|x_1, \theta = 1) = \begin{cases} 0 & x_2 = w \ \& \ x_1 = p \\ 1 & x_2 = l \ \& \ x_1 = p \\ 0 & x_2 = l \ \& \ x_1 = n \\ 1 & x_2 = w \ \& \ x_1 = n \end{cases} \tag{1.69}$$

So the final result is

$$P(x_2 = w|\theta) = \begin{cases} 1 \cdot \frac{1}{3} + 0 \cdot \frac{2}{3} = \frac{1}{3} & \theta = 0 \\ 0 \cdot \frac{1}{3} + 1 \cdot \frac{2}{3} = \frac{2}{3} & \theta = 1 \end{cases} \tag{1.70}$$

The remarkable result is that making a new choice increases your chances to win considerably. Naively one would have thought that the probability in this case is $\frac{1}{2}$, because after the moderator has opened a door without the prize, two doors are remaining and there is an equal chance to get the right door with the prize. But unfortunately this intuitively convincing argument is wrong.

The following code shows a Monte Carlo simulation with the following variables and functions: `doorsWithPrize` is a randomly generated list of door numbers, which contain a prize, `firstChoice` is a randomly generated list of door numbers from the participant's first choice (this variable will be used to evaluate the case participant stays with his first choice), `doorsWithoutPrize` is a randomly generated list of the doors opened by the moderator after the first choice was made, `secondChoice` is a list of door numbers from the participant's second choice (this variable will be used to evaluate the case participant chooses the remaining door), `ProbabilityWinningPrize` is a function to calculate the probability to win a prize for a given choice (it simply counts the number of elements in the choice list, which agree with the corresponding item in the `doorsWithPrize` list).

```
doors = Range[3];
doorsWithPrize = RandomChoice[doors, 1000];
firstChoice = RandomChoice[doors, Length[doorsWithPrize]];
doorsWithoutPrize = MapThread[
                        RandomChoice[Complement[doors, {#1, #2}]] &,
                        {doorsWithPrize, firstChoice}];
secondChoice = MapThread[
                        RandomChoice[Complement[doors, {#1, #2}]] &,
                        {doorsWithoutPrize, firstChoice}];
ProbabilityWinningPrize[choice_] := Apply[Plus,
                                    MapThread[Boole[#1 == #2] &,
                                    {doorsWithPrize, choice}]
                                    ] / Length[doorsWithPrize] // N;
Print["Probability of winning by staying with your first choice = ",
      ProbabilityWinningPrize[firstChoice]];
Print["Probability of winning by choosing the remaining door = ",
      ProbabilityWinningPrize[secondChoice]];
```

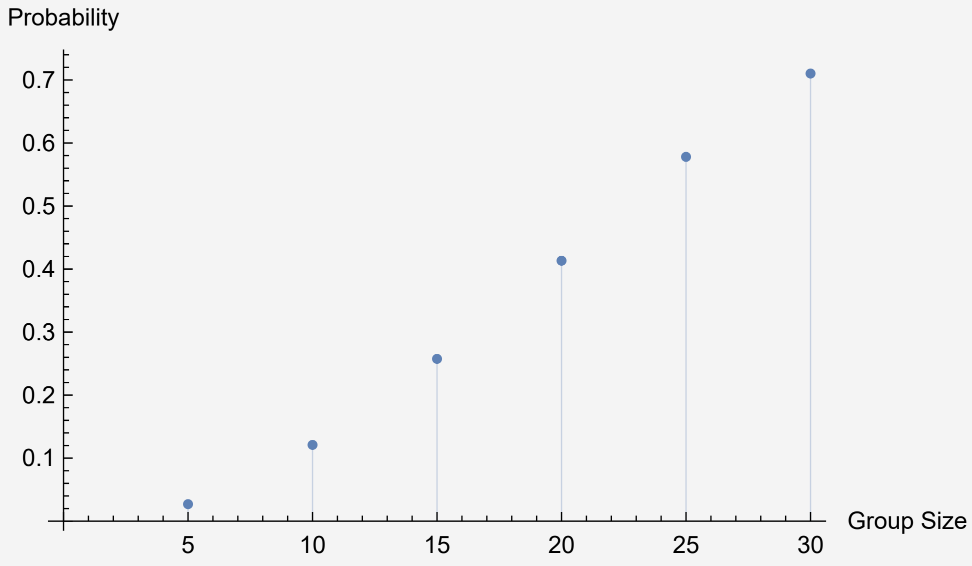
The probability of winning with 1000 samples is `0.32` in the case of staying with the first choice and `0.68` in the case of choosing the remaining door. Using 1000000 samples the values are `0.333` and `0.667`, respectively.

Example 1.16 (The Birthday Problem) The Birthday problem is another classic probabilistic example. Let's assume that each day of the year (we ignore leap years) is equally likely a birthday with probability $p = \frac{1}{365}$. What is the probability that in a group with n distinct individuals (no twins) at least two persons have the same birthday?

We want to solve this problem with a simulation and plot the probability as a function of the group size.

We create a variable birthdays containing all possible birthdays from 1 to 365. The main part is the function sameBirthdaysInGroup, which calculates the probability of having at least two persons with the same birthday in a group of *n* people. A random matrix groupSamples is generated with sampleSize rows and n columns with elements chosen randomly from the list birthdays. The next line evaluates each row of groupSamples if there are same birthdays present. Each row is first sorted and then the difference of each element is calculated. If two birthdays are the same, at least one difference is zero and MemberQ tests if a zero is present. Boole converts the boolean to numerical values (True to 1 and False to 0) and the result is stored in the variable sameBirthDay. The return value is simply the number of ones in the list sameBirthDay divided by the number of samples.

```
birthdays = Range[365];
SameBirthdaysInGroup[n_Integer] := Module[
   {sameBirthday, groupSamples, sampleSize = 1000},
   groupSamples = RandomChoice[birthdays, {sampleSize, n}];
   sameBirthday = Map[Boole[MemberQ[Differences[Sort[#]], 0]] &, groupSamples];
   Return[Apply[Plus, sameBirthday] / sampleSize // N]
   ];
groupSizes = {5, 10, 15, 20, 25, 30};
probs = Map[SameBirthdaysInGroup, groupSizes];
ListPlot[Transpose[{groupSizes, probs}],
         Filling -> Axis,
         AxesLabel -> {"n", "Probability"}]
```



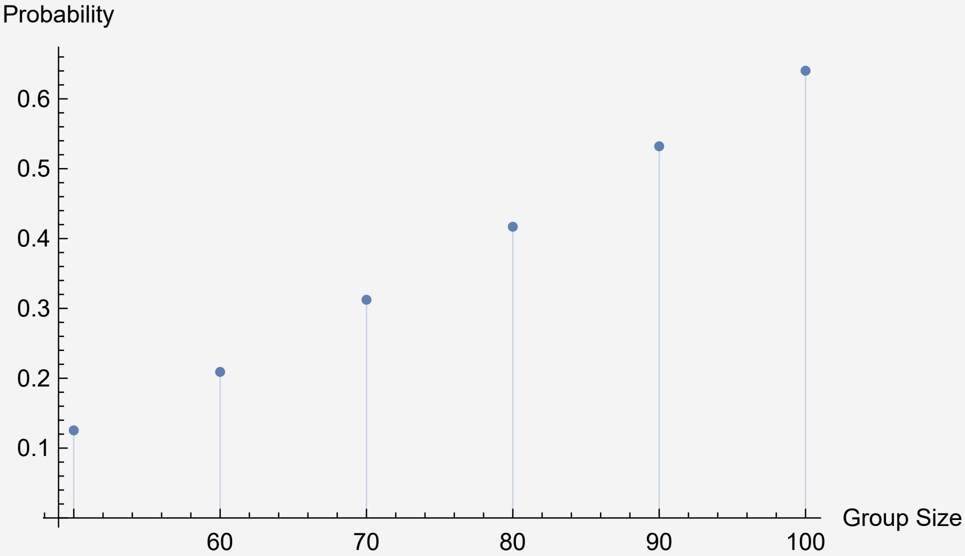
The probability is larger than 50% for a group size > 22.

What is the probability that at least three persons have the same birthday?

Only one line of the code above has to be changed to answer this question.

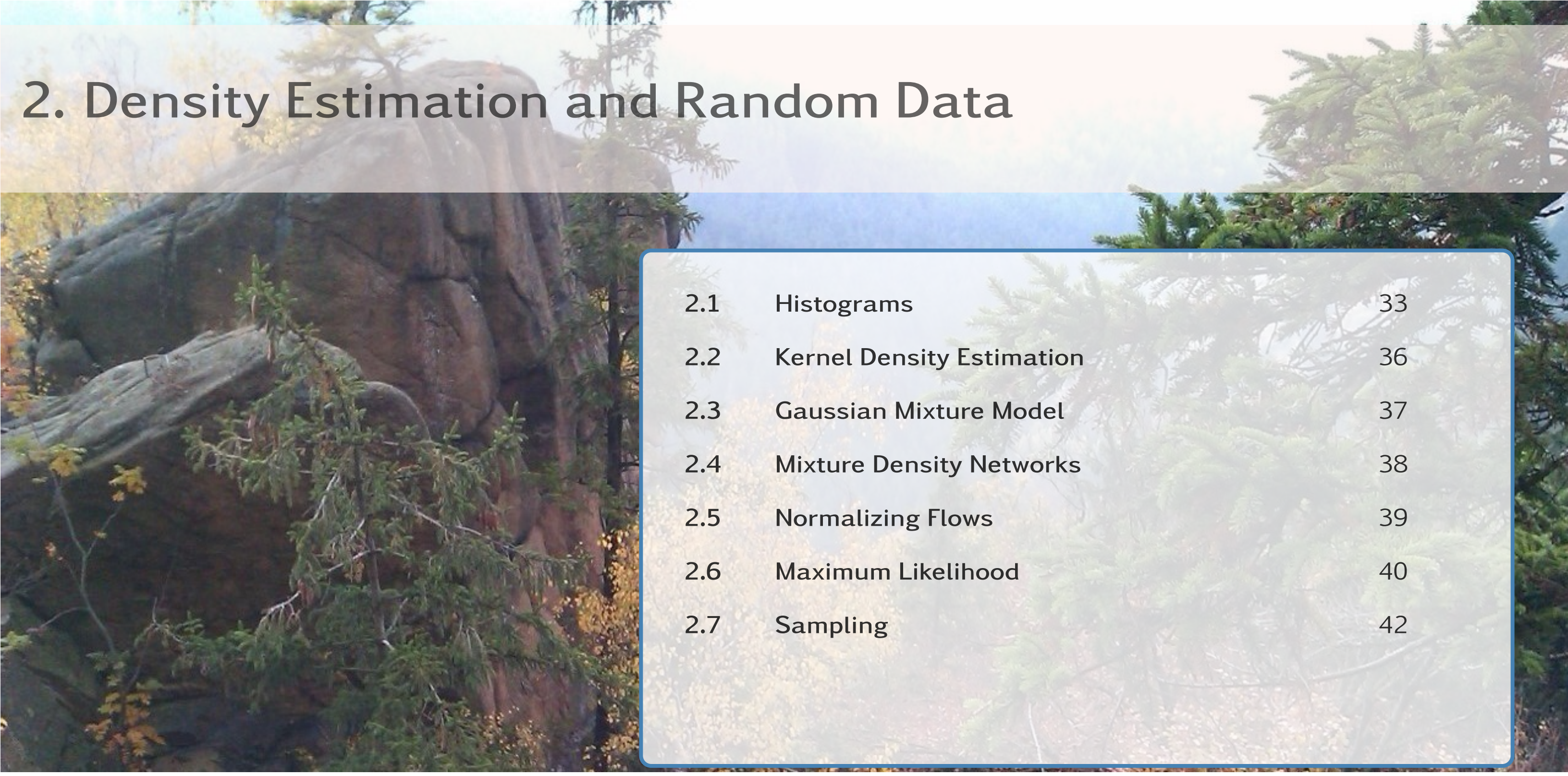
```
sameBirthday =
    Map[Boole[MemberQ[Partition[Differences[Sort[#]], 2, 1], {0, 0}]] &,
    groupSamples];
```

The function `Partition` decomposes the differences list into overlapping pairs and any pair with only zeros in it is a triple birthday.



So a group of 88 people is needed to have a 50% chance of having at least three persons with the same birthday.

2. Density Estimation and Random Data



2.1 Histograms 33

2.2 Kernel Density Estimation 36

2.3 Gaussian Mixture Model 37

2.4 Mixture Density Networks 38

2.5 Normalizing Flows 39

2.6 Maximum Likelihood 40

2.7 Sampling 42

This chapter deals with the problem how a probability density can be estimated from data and how data can be generated from a distribution.

2.1 Histograms

A histogram divides the support of a random variable x into disjunct bins and counts the number of outcomes of x in each bin. If $x \sim p(x)$, the normalized count in bin $[x_i, x_{i+1}]$ is given by

$$P_i = P(x_i \leqq x \leqq x_{i+1}) = \int_{x_i}^{x_{i+1}} p(x)dx = \lim_{N\to\infty} \frac{N_i}{N} \tag{2.1}$$

The count N_i can only take integer values, therefore $N_i = [P_i N]$, where the symbol $[x]$ means nearest integer of x and N is the total number of outcomes.

Let's assume that we have collected data about the random variable x as $\mathcal{D} = \{x_{1:N}\}$. We assign each data point x_k to a bin and count the number of data points in each bin. If the bins do not cover the whole range of x, there will be data points, which can not be assigned to a bin. These data points are called outliers.

If we represent the bins in terms of a center point $\tilde{x}_i = \frac{x_i + x_{i+1}}{2}$ and the width $w_i = \frac{x_{i+1} - x_i}{2}$, we can write the bin b_i as $[\tilde{x}_i - \frac{w_i}{2}, \tilde{x}_i + \frac{w_i}{2}]$.

Then the N_i can be expressed as

$$N_i = \sum_{k=1}^{N} \mathcal{I}(x_k - \tilde{x}_i; w_i) \tag{2.2}$$

where $\mathcal{I}$ is the indicator function defined as

$$\mathcal{I}(x; w) = \begin{cases} 1 & x \in [-\frac{w}{2}, +\frac{w}{2}] \\ 0 & otherwise \end{cases} \tag{2.3}$$

Then the probability density estimate $\hat{p}(x)$ can be written as

$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{m} \frac{N_i}{w_i} \mathcal{I}(x - \tilde{x}_i; w_i) \tag{2.4}$$

where m is the number of bins.

There are some best practice rules for choosing the number of bins m

$$m = [1 + \log_2(N)] \tag{2.5}$$

and the bin width w

$$w = 3.5 s N^{-\frac{1}{3}} \tag{2.6}$$

where s is the standard deviation of the data sample.

If we create n histograms from datasets $\mathcal{D}_{1:n}$ sampled from the same distribution $p(x)$ and with the same size N, then the bin entries $N_1, N_2, \ldots, N_m$ would be distributed according to a multinomial distribution

$$P(N_{1:m} | N, P_{1:m}) = \frac{N!}{N_1! N_2! \ldots N_m!} P_1^{N_1} P_2^{N_2} \ldots P_m^{N_m} \tag{2.7}$$

with $\sum_{i=1}^{m} N_i = N$ and P_i are the probabilities of x to be assigned to bin $[x_i, x_{i+1}]$.

The Wolfram Language provides the function `Histogram` e.g.

```
Histogram[data, {{1,2,3,4}}]
```

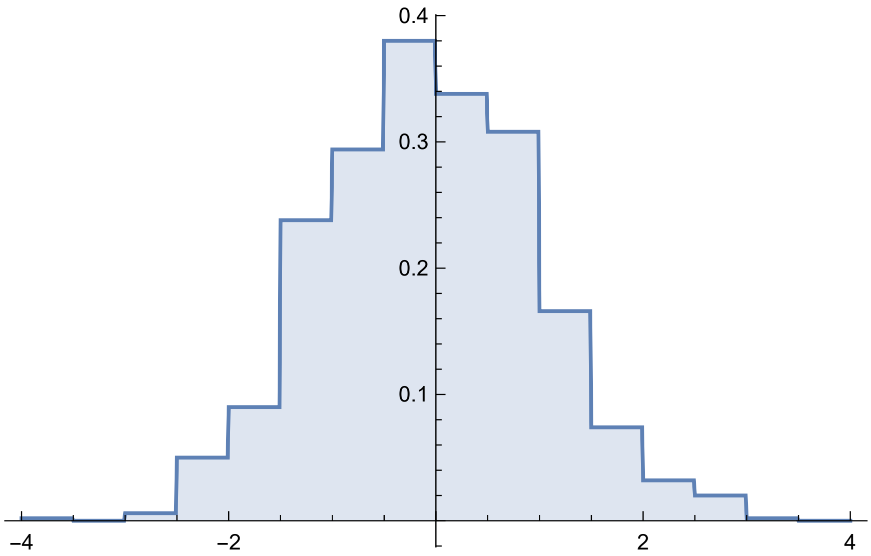
creates a histogram with 3 bins $[1,2]$, $[2,3]$ and $[3,4]$ or

```
Histogram[data, {-10,10,1}, "Probability"]
```

creates a histogram of normalized counts $\frac{N_i}{N}$ from -10 to 10 with a bin width of 1. Using the option `"PDF"`, a probability density estimate $\hat{p}(x)$ can be plotted. If the bin values and histogram heights are only required numerically, the function `HistogramList` can be used.

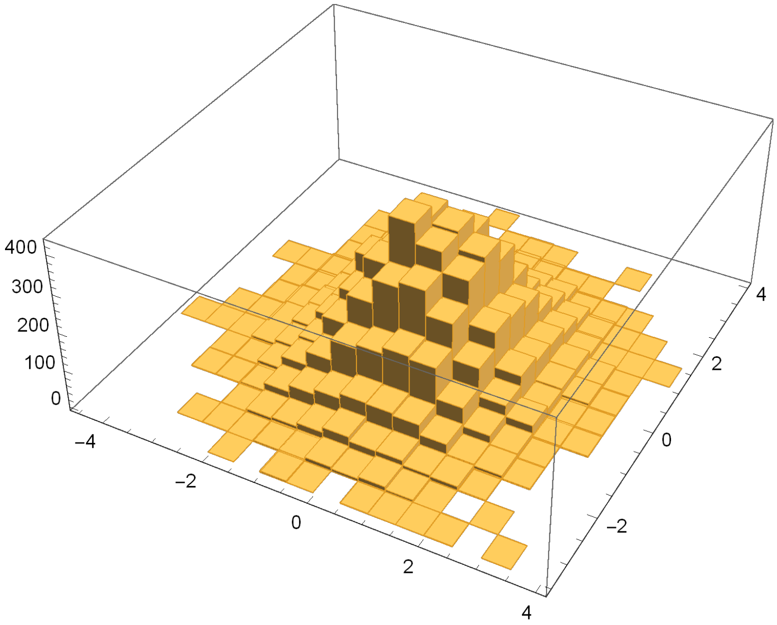
A probability density estimate can be constructed with the function `HistogramDistribution` according to eq. 2.4.

```
data = RandomVariate[NormalDistribution[], 1000];
dist = HistogramDistribution[data];
DiscretePlot[PDF[dist, x], {x, -4, 4, .01}]
```

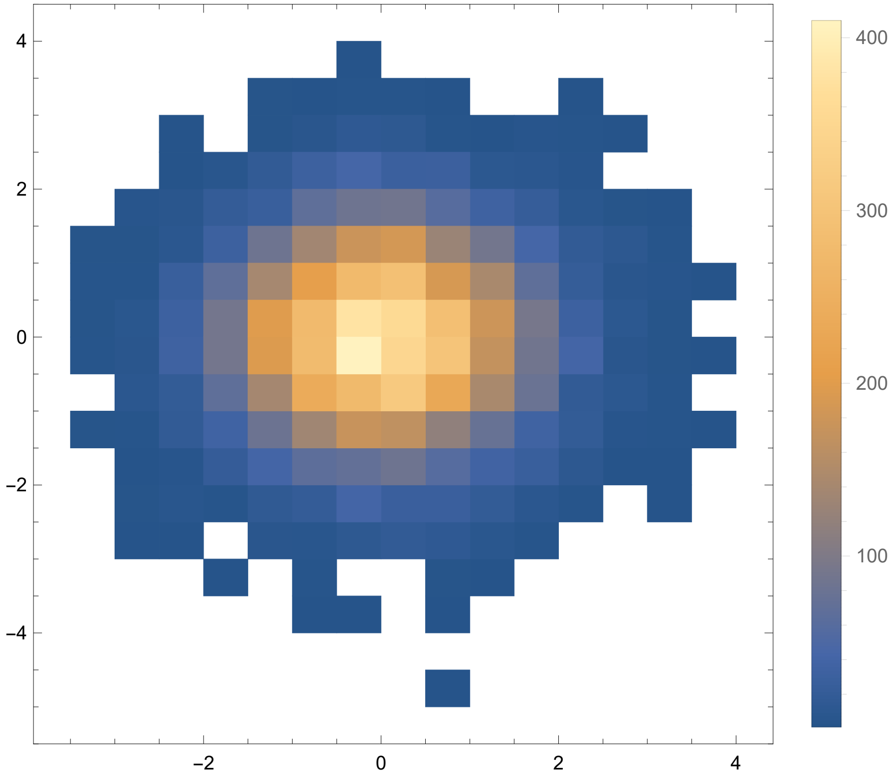


In addition, Histogram3D and DensityHistogram are available for bi-variate random data. In the following examples the bi-variate data samples are generated with the BinormalDistribution, which is a special case of the multinormal distribution for $n = 2$.

```
Histogram3D[RandomVariate[BinormalDistribution[0], 10000]]
```



```
DensityHistogram[RandomVariate[BinormalDistribution[0], 10000]]
```



Exercise 2.1 Verify that $\hat{p}(x)$ given by eq. 2.4 is normalized.

$$\int_{-\infty}^{+\infty} \hat{p}(x) dx = 1 \tag{2.8}$$

2.2 Kernel Density Estimation

For some applications a smooth version of a histogram is preferred, which can be constructed by assigning a Gaussian distribution $\mathcal{N}\left(x_i, \sigma^2\right)$ to each data point x_i and summing up the area under the pdf in each bin from each contribution to yield the count value of the corresponding histogram. The result depends very much on the choice of σ. The histogram is exactly reproduced in the limit $\sigma \to \infty$.

Definition 2.1 **(Binned Kernel Density Estimator)** A binned KDE (Kernel Density Estimator) is defined by

$$p_{bKDE}(b|\sigma^2, b_{1:M}) = \begin{cases} \sum_{i=1}^{N} \int_{b_j}^{b_{j+1}} dx \mathcal{N}\left(x|x_i, \sigma^2\right) & b \in [b_j, b_{j+1}] \\ 0 & otherwise \end{cases} \tag{2.9}$$

where b_j is an ordered sequence of M bin values.

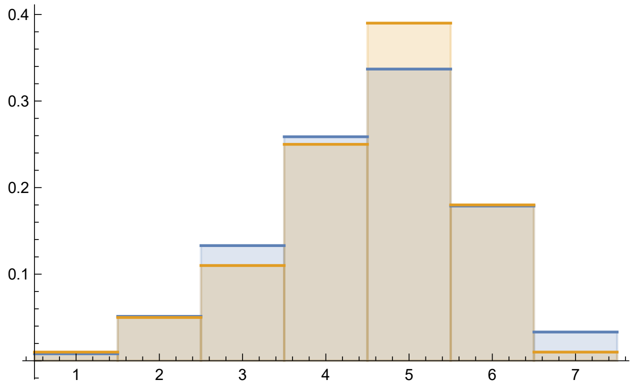
A Wolfram Language implementation `BinnedKernelDensityEstimate` uses the fact that the integral over the normal distribution can be expressed as the difference of two error functions.

$$\int_{b_j}^{b_{j+1}} dx \mathcal{N}\left(x|x_i, \sigma^2\right) = \frac{1}{2}\left[\mathtt{erf}\left(\frac{b_{j+1} - x_i}{\sqrt{2}\sigma}\right) - \mathtt{erf}\left(\frac{b_j - x_i}{\sqrt{2}\sigma}\right)\right] \tag{2.10}$$

The function returns a normalized count for each bin.

```
BinnedKernelDensityEstimate[data_List, bins_List, bandwidth_Real] := Module[
   {d, w, counts},
   w = Sqrt[2] * bandwidth;
   counts = Sum[MapThread[
                Erf[(#2 - d)/w] - Erf[(#1 - d)/w] &, {bins[[;; -2]], bins[[2 ;;]]}],
                {d, data}];
   Return[counts / Apply[Plus, counts]]];

data = RandomVariate[NormalDistribution[], 100];
h = HistogramList[data];
counts1 = BinnedKernelDensityEstimate[data, h[[1]], 0.5];
counts2 = h[[2]] / Apply[Plus, h[[2]]] // N;
DiscretePlot[{counts1[[i]], counts2[[i]]}, {i, 1, Length[counts1]},
            ExtentSize -> Full]
```

A comparison with the function `HistogramList` shows a good agreement for the chosen bandwidth of 0.5.

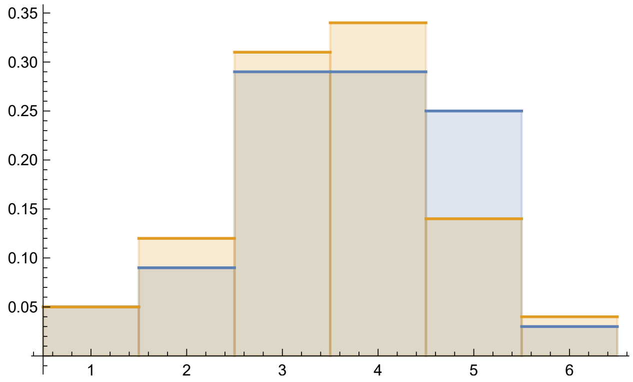


The integration over each bin in eq. 2.9 is useful to get directly a probability value, but it is not necessary. A meaningful kernel density estimation can be done using a Gaussian pdf as kernel and therefore yields a function instead of sequence of values.

$$p_{KDE}(x|\sigma^2) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}\left(x|x_i,\sigma^2\right) \tag{2.11}$$

The Wolfram Language provides such a function under the name `SmoothKernelDistribution[data,bw,ker]` with the kernel argument `ker` set to `"Gaussian"`, which returns a probability distribution.

```
data = RandomVariate[NormalDistribution[], 100];
dist = SmoothKernelDistribution[data, 0.5, "Gaussian"];
h1 = HistogramList[RandomVariate[dist, 100]];
(* Apply the binning of histogram 1 to histogram 2 *)
h2 = HistogramList[data, {h1[[1]]}];
counts1 = h1[[2]] / Apply[Plus, h1[[2]]] // N;
counts2 = h2[[2]] / Apply[Plus, h2[[2]]] // N;
DiscretePlot[{counts1[[i]], counts2[[i]]}, {i, 1, Length[counts1]},
            ExtentSize -> Full]
```



2.3 Gaussian Mixture Model

Histograms are typically a coarse-grained representation of a probability density, which depends very much on the choice of binning. A better representation is a Gaussian Mixture Model (GMM).

Definition 2.2 (Gaussian Mixture Model) The Gaussian Mixture Model is defined by

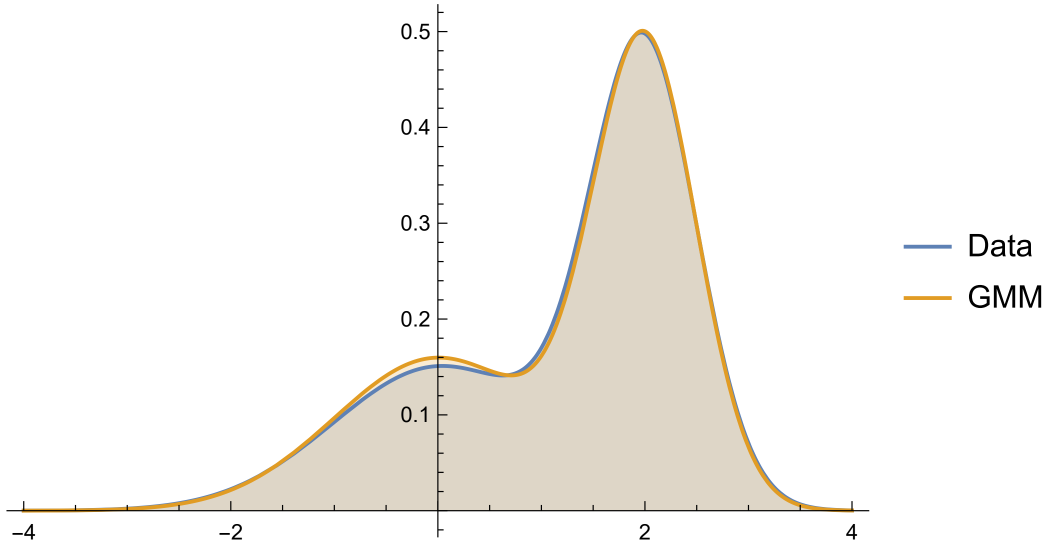
$$p_{GMM}(\mathbf{x}|\boldsymbol{\alpha}, \boldsymbol{\mu}_{1:N}, \Sigma_{1:N}) = \sum_{i=1}^{N} \alpha_i \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_i, \Sigma_i) \tag{2.12}$$

with $\sum_{i=1}^{N} \alpha_i = 1$.

A Gaussian Mixture Model can approximate any continuous probability density as long as N is big enough.

Wolfram Language can learn GMM's from data using the high-level function `LearnDistribution` with the option `Method -> "GaussianMixture"`.

```
mixture = MixtureDistribution[{0.4, 0.6},
            {NormalDistribution[], NormalDistribution[2, 0.5]}];
data = RandomVariate[mixture, 1000];
dist = LearnDistribution[data, Method -> "GaussianMixture"];
Plot[{PDF[dist, x], PDF[mixture, x]}, {x, -4, 4},
     Filling -> Axis,
     PlotLegends -> Placed[{"Data", "GMM"}, Right]
```



2.4 Mixture Density Networks

An extension of Gaussian Mixture Models for conditional distribution estimation are given by Mixture Density Networks.

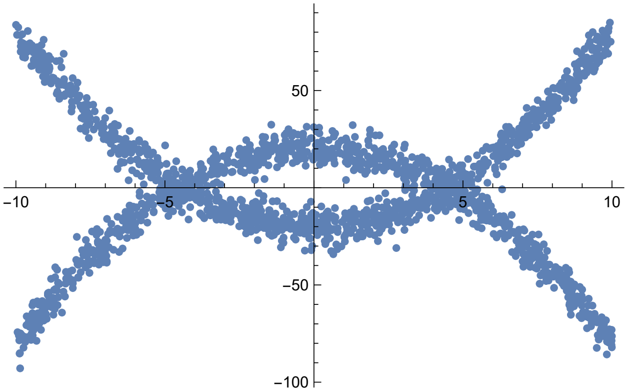
Definition 2.3 (Mixture Density Network) A Mixture Density Network is defined by

$$p_{MDN}(\mathbf{x}|\mathbf{z}) = \sum_{i=1}^{N} \alpha_i(\mathbf{z}) \mathcal{N}(\mathbf{x}|\mu_i(\mathbf{z}), \sigma_i(\mathbf{z})) \tag{2.13}$$

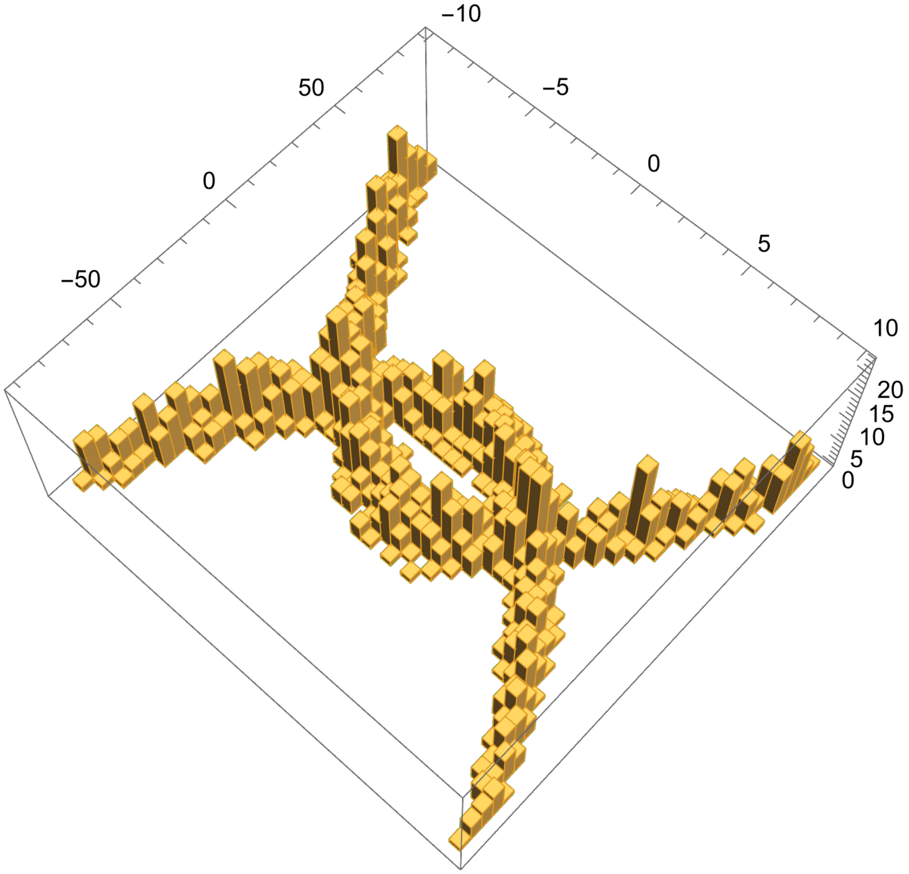
with $\sum_{i=1}^{N} \alpha_i(\mathbf{z}) = 1$.

The parameters $\alpha_{1:N}$, $\mu_{1:N}$ and $\sigma_{1:N}$ are functions of $\boldsymbol{z}$ and are approximated by neural networks.

For demonstrating the capabilities of this method, we take the following two-dimensional data. The density of points is the probability measure.



The corresponding histogram gives a rough approximation of the distribution described by these data.



2.5 Normalizing Flows

Normalizing flows are another way to construct a probability distribution from its samples. A Normalizing Flow is a mapping of a simple probability distribution e.g. a Normal distribution into a more complex distribution by a sequence of invertible transformations f_i. The complete transformation f is given by $f_1 \circ f_2 \circ \dots f_{N-1} \circ f_N$ and is also invertible, which means that f^{-1} exists. For such a function to be useful, it must also be necessary to differentiate f with respect ot its arguments.

2.6 Maximum Likelihood

If the physical process how data is generated is understood, the functional form of the probability density $p(\mathbf{x}|\boldsymbol{\theta})$ is typically known. In these cases the goal is to estimate the parameters $\boldsymbol{\theta}$ of a probability density from data.

Definition 2.4 (Likelihood) The likelihood is defined by

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{i=1}^{N} p\left(x_i \,|\, \boldsymbol{\theta}\right) \tag{2.14}$$

where $x_i \in \mathcal{D} = \{x_{1:N}\}$. It is a function of the parameter of the probability density, not a distribution.

Using the logarithm of the likelihood is better from the view point of numerical stability.

Definition 2.5 (Loglikelihood) The loglikelihood is defined by

$$\log(\mathcal{L}(\boldsymbol{\theta})) = \sum_{i=1}^{N} \log\left(p\left(x_i \,|\, \boldsymbol{\theta}\right)\right) \tag{2.15}$$

An optimal estimate of the parameters $\hat{\boldsymbol{\theta}}$ can be found by solving $\frac{\partial \log(\mathcal{L}(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} = 0$. The probability density $p(\mathbf{x}|\hat{\boldsymbol{\theta}})$ is then called the Maximum Likelihood estimate.

Example 2.1 (Parameter Estimation of Normal Distribution) The loglikelihood function for the Normal distribution is given by

$$\log(\mathcal{L}(\mu, \sigma^2) = \sum_{i=1}^{N} \log\left(\mathcal{N}\left(x_i \,|\, \mu, \sigma^2\right)\right) \tag{2.16}$$

Inserting the function form of $\mathcal{N}\left(x \,|\, \mu, \sigma^2\right)$ gives the following expression

$$\log(\mathcal{L}(\mu, \sigma^2) = \frac{1}{2\sigma^2} \sum_{i=1}^{N} (x_i - \mu)^2 + \frac{N}{2} \log\left(2\pi\sigma^2\right) \tag{2.17}$$

The derivative with respect to μ gives

$$\frac{\partial \log(\mathcal{L}(\mu, \sigma^2)}{\partial \mu} = -\frac{1}{\sigma^2} \sum_{i=1}^{N} (x_i - \mu) = -\frac{1}{\sigma^2} \left(\sum_{i=1}^{N} x_i - N\mu \right) = 0 \tag{2.18}$$

and has the solution $\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x_i$ (sample mean).

The derivative with respect to σ^2 gives

$$\frac{\partial \log(\mathcal{L}(\mu,\sigma^2))}{\partial \sigma^2} = -\frac{1}{2\sigma^4}\sum_{i=1}^{N}(x_i-\mu)^2 + \frac{N}{2\sigma^2} = -\frac{1}{2\sigma^4}\left(\sum_{i=1}^{N}(x_i-\mu)^2 - N\sigma^2\right) = 0 \tag{2.19}$$

and has the solution $\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i-\hat{\mu})^2$ (biased sample variance). An unbiased sample variance is given by $s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i-\hat{\mu})^2$.

Wolfram Language has built in Maximum Likelihood parameter estimation using the function `EstimatedDistribution` with the option `ParameterEstimator -> "MaximumLikelihood"`.

```
data = RandomVariate[NormalDistribution[0, 1], 1000];
dist = EstimatedDistribution[data, NormalDistribution[\[Mu], \[Sigma]],
                                   ParameterEstimator -> "MaximumLikelihood"]
```

has the result `NormalDistribution[0.0400075, 0.974344]`.

```
{Mean[data], StandardDeviation[data], RootMeanSquare[data - Mean[data]]}
```

has the result `{0.0400075, 0.974832, 0.974344}`. The function `StandardDeviation` implements the square root of the unbiased sample variance, while `RootMeanSquare[data - Mean[data]]` calculates the square root of the biased sample variance, which agrees with the result from the Maximum Likelihood calculation.

The function `FindDistribution` tries to find the best distribution for the given data.

```
FindDistribution[data]
```

returns exactly the same result as the Maximum Likelihood estimator without specifying a distribution as an argument.

Example 2.2 (Parameter Estimation of Poisson Distribution) In the case of a discrete probability density like the Poisson distribution, the likelihood is calculated with the probabilities evaluated on the data samples $x_{1:N}$.

$$\mathcal{L}(\lambda) = \prod_{i=1}^{N}\frac{\lambda^{x_k}}{x_k!}\exp(-\lambda) = \frac{\lambda^{\sum_{i=1}^{N}x_k}}{\prod_{i=1}^{N}x_k!}\exp(-N\lambda) \tag{2.20}$$

Differentiating with respect to λ gives

$$\frac{\partial \mathcal{L}(\lambda)}{\partial \lambda} = \left(\frac{\sum_{i=1}^{N}x_k}{\lambda} - N\right)\frac{\lambda^{\sum_{i=1}^{N}x_k}}{\prod_{i=1}^{N}x_k!}\exp(-N\lambda) = 0 \tag{2.21}$$

The result is $\hat{\lambda} = \frac{1}{N}\sum_{i=1}^{N}x_k$ (sample mean).

Exercise 2.2 The likelihood of a uniform distribution evaluated on the data samples

$x_{1:N}$ is given by

$$\mathcal{L}(\alpha,\beta) = \prod_{i=1}^{N} \frac{1}{\beta-\alpha}\Theta(x_k - a)\Theta(b - x_k) = \frac{1}{(\beta-\alpha)^N}\Theta(x_{min}-\alpha)\Theta(\beta - x_{max}) \tag{2.22}$$

with $x_{min} = \min(x_{1:N})$ and $x_{max} = \max(x_{1:N})$. Show that eq. 2.22 has a maximum at $\alpha = x_{min}$ and $\beta = x_{max}$.

So far we applied the method to all available data. It is often better to use Maximum Likelihood on small batches of the data and update the parameters $\boldsymbol{\theta}$ iteratively by gradient descent

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{\lambda}{\mathcal{L}^{(i)}(\boldsymbol{\theta})}\frac{\partial \mathcal{L}^{(i)}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}|_{\theta=\theta_i} \tag{2.23}$$

where λ is the small positive number so that $\mathcal{L}^{(i)}(\boldsymbol{\theta_i}) > \mathcal{L}^{(i)}(\boldsymbol{\theta_{i+1}})$ is guaranteed and $\mathcal{L}^{(i)}$ is the likelihood of the i-th batch given by

$$\log\left(\mathcal{L}^{(i)}(\boldsymbol{\theta})\right) = \sum_{k=1}^{M} \log\left(p\left(x_{k+iM}\,|\,\boldsymbol{\theta}\right)\right) \tag{2.24}$$

where M is the batch size and i goes from 0 to the integer part of N/M. The initial values of $\boldsymbol{\theta_0}$ have to chosen properly so that a global minimum is found.

2.7 Sampling

In the previous sections we studied the reconstruction of a probability density from data. Here we investigate the inverse problem: the generation of data from a given probability density. This process is called sampling.

It is usually quite easy to generate random numbers r from a uniform distribution $\mathcal{U}(r|\alpha = 0, \beta = 1)$. Almost every programming language has a function for it (in the Wolfram Language the function is called `RandomReal`) . The goal is now to transform the sequence $r_{1:N}$ to a sequence $x_{1:}$ so that x is distributed as $p(x)$.

One way to do this is to find a transformation function $x(r)$ for $0 \leqq r \leqq 1$.

$$\Phi(x(r)) = \int_{-\infty}^{x(r)} p(x)dx = \int_{-\infty}^{r} \mathcal{U}(r = r'|\alpha = 0, \beta = 1)dr' = r \tag{2.25}$$

$\Phi(x)$ is the cumulative distribution function (see 1.4) evaluated at $x(r)$. Therefore the function $x(r)$ is the inverse of the cdf evaluated at r.

$$x(r) = \Phi^{-1}(r) \tag{2.26}$$

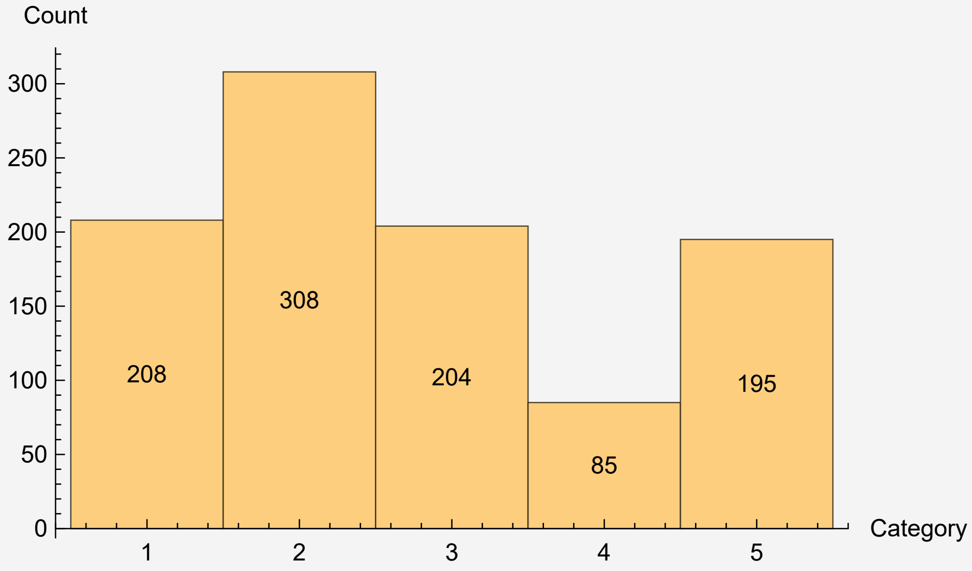
Example 2.3 (Sampling from a Discrete Distribution) Let's assume that we have an array of probabilities $P_{1:N}$ with $\sum_{i=1}^{N} P_i = 1$ and we want to sample an index variable $j \in [1, 2, \ldots, N]$ from this distribution.

In this case the cumulative distribution is given by $\Phi(j) = \sum_{i=1}^{j} P_i$, where the index j has a range from 0 to N and $\Phi(0) = 0$. We iterate the index j from N down to 0 and stop as as soon as the condition $\Phi(j) < r$ is fulfilled. Finally we increment the index $j \rightarrow j+1$, because the possible value range of j is from 0 to $N-1$ and we need to map them to the range from 1 to N. This procedure generates an index sequence with the desired distribution.

The Wolfram Language provides the function `EmpiricalDistribution` to construct a pdf from a list of weights, which need not to be normalized. Using `RandomVariate` 1000 samples are drawn from this pdf and are used to fill a histogram.

```
weights = {200, 300, 200, 100, 200};
categories = Range[1, Length[weights]];
dist = EmpiricalDistribution[weights -> categories];
data = RandomVariate[dist, 1000];
Histogram[data,
         LabelingFunction -> Center,
         AxesLabel -> {"Category", "Count"}]
```

The histogram shows the counts of the data samples generated from an empirical distribution for each category. The counts are indeed numerically close to the given weights.



Example 2.4 (Sampling from a Histogram) If the probability density is given as a histogram, the sampling procedure is the following:

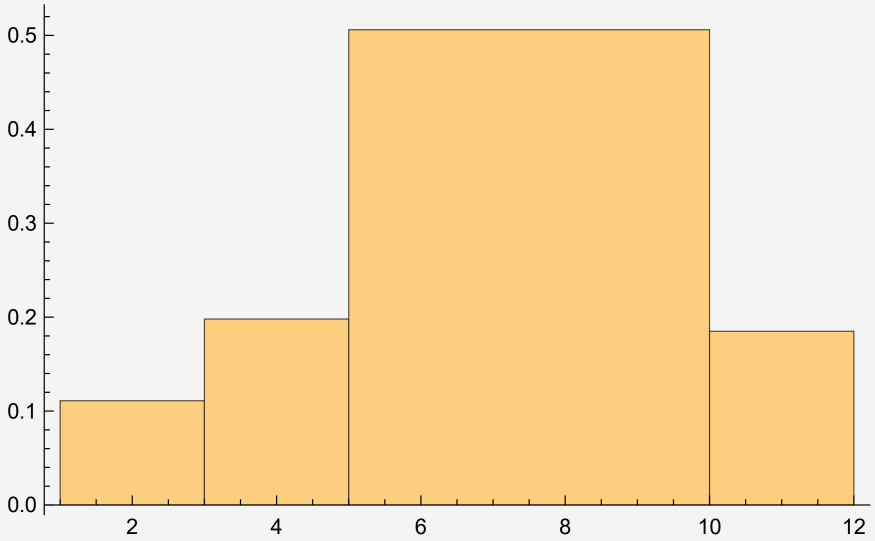
- Sample a bin index j using the bin heights $\frac{N_i}{N}$ as you would sample from a discrete distribution.
- Sample x from a uniform distribution $\mathcal{U}\left(x \,|\, \alpha = \tilde{x}_j - \frac{w_j}{2}, \beta = \tilde{x}_j + \frac{w_j}{2}\right)$.

where $\tilde{x}_j$ is the center coordinate and w_j is the width of bin j.

The Wolfram Language provides the function `HistogramDistribution`, which can be constructed from given data and can be also used to generate new data samples using `RandomVariate`, but we first we want to implement our own function based on the sampling procedure described above.

The function `SampleFromHistogram` has three arguments: the bin interval limits `bins` as a list of $N+1$ elements, the normalized bin heights `binHeights` as a list with N elements and the number of samples `numSamples`. So the number of bins N is provided implicitely by the length of the list arguments. It calculates first the cumulative sum of the bin heights, where zero is appended at the beginning of the list. `numSamples` random numbers are generated to sample a list of bin indices. This is done by subtracting the random number from each element of the cumulative sum and keep only the largest element of the non-positive results. The index is then used to generate uniformly distributed random numbers inside the corresponding bin.

```
SampleFromHistogram[bins_, binHeights_, numSamples_] := Module[
  {cumSum, randomNumbers, binIndices, histSamples},
  cumSum = Prepend[Accumulate[binHeights], 0];
  randomNumbers = RandomReal[1, numSamples];
  binIndices = Flatten[Map[Ordering[Select[cumSum - #, NonPositive], -1] &,
                           randomNumbers]];
  histSamples = Map[RandomReal[{bins[[#]], bins[[# + 1]]}] &, binIndices];
  Return[histSamples]]

data = SampleFromHistogram[{1, 3, 5, 10, 12},
                           {0.1, 0.2, 0.5, 0.2},
                           1000];
Histogram[data, {{1, 3, 5, 10, 12}}, "Probability"]
```



The same result can be obtained with

```
dist = HistogramDistribution[data, {{1, 3, 5, 10, 12}}];
data2 = RandomVariate[dist, 1000];
Histogram[data2, {{1, 3, 5, 10, 12}}, "Probability"]
```

but the performance is much better (almost a factor 100 using `Timing` to measure the runtime of `SampleFromHistogram` and `RandomVariate` generating 1000000 samples).

Example 2.5 (Normal-Distributed Random Numbers) The trick here is to perform the transformation on the joint pdf of x and y, where both random variables are identically distributed according to $\mathcal{N}(\_|\mu = 0, \sigma^2 = 1)$.

$$\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} \mathcal{N}(x|\mu = 0, \sigma^2 = 1)\mathcal{N}(y|\mu = 0, \sigma^2 = 1)dxdy = 1 \tag{2.27}$$

Using the transformation $x = \rho\cos(\phi)$ and $y = \rho\sin(\phi)$, the double integral can written as

$$\frac{1}{2\pi}\int_0^{2\pi} d\phi \int_0^{+\infty} \exp\left(-\frac{\rho^2}{2}\right)\rho d\rho = \int_0^1 d\phi' \int_0^{+\infty} \exp(-\rho')\, d\rho' = 1 \tag{2.28}$$

where the new variables ρ' and ϕ' are given by $\rho' = \frac{\rho^2}{2}$ and $\phi' = \frac{\phi}{2\pi}$. So ϕ' is uniformly distributed in the range $[0, 1]$ and ρ' is exponentially distributed with $\xi = 1$.

We leave it as an exercise to show that $\rho'(r) = -\log(r)$ and therefore $\rho(r) = \sqrt{-2\ln(r)}$.

The transformation function for ϕ is given by $\phi(r) = 2\pi r$. Because ϕ and ρ are independent random variables, we must also use independent random number sequences r and s to evaluate them.

Therefore the transformation for x is given by

$$x(r, s) = \sqrt{-2\ln(r)}\cos(2\pi s) \tag{2.29}$$

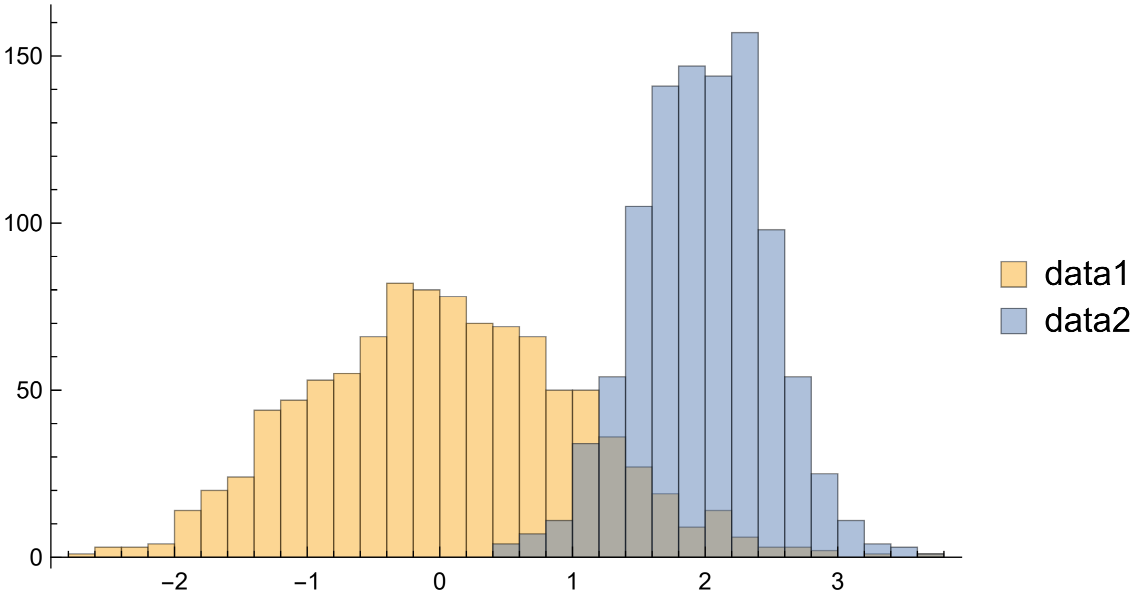
and for y by

$$y(r, s) = \sqrt{-2\ln(r)}\sin(2\pi s) \tag{2.30}$$

The algorithm is called Box-Muller method [4].

Random numbers can be generated with the Wolfram Language using `RandomVariate` for any built-in distribution e.g.

```
data1 = RandomVariate[NormalDistribution[0, 1], 1000];
data2 = RandomVariate[NormalDistribution[2, 0.5], 1000];
Histogram[{data1, data2}]
```



Exercise 2.3 Show that the function $x(r)$ for the exponential distribution $p(x) = \frac{1}{\xi}\exp\left(-\frac{x}{\xi}\right)$ is given by $x(r) = -\xi\ln(1-r)$ or $x(r) = -\xi\ln(r)$.

Exercise 2.4 Show how the method above can be extended to generate random numbers distributed as $\mathcal{N}(\_|\mu,\sigma^2)$.

Finding a function $x(r)$ analytically is not always possible. Another approach, which is applicable to a wider range of distributions, is rejection sampling [15].

Definition 2.6 **(Rejection Sampling)** The algorithm works in the following way:

- Choose a number M and a distribution $p(y)$ in such a way that the condition $p(x=u) \leqq Mp(y=u)$ holds everywhere on the support of x and drawing samples from $p(y)$ is known.
- Draw samples y_k from a distribution $p(y)$ and r_k from a uniform distribution $\mathcal{U}(r|\alpha=0,\beta=1)$ for $k=1:N$.
- Check for all k the condition $r_k < \frac{p(x=y_k)}{Mp(y=y_k)}$. If it is true, accept y_k as a sample from $p(x)$, otherwise reject the sample.

If $p(x)$ is bounded to a finite interval $[a,b]$, the algorithm can be simplified:

- Choose p_{max} as the maximum value of $p(x)$ for $x \in [a,b]$
- Draw samples $x_k = (b-a)s_k + a$ from a uniform distribution $\mathcal{U}(s|\alpha=0,\beta=1)$ and r_k from a uniform distribution $\mathcal{U}(r|\alpha=0,\beta=1)$ for $k=1:N$.
- Check for all k the condition $r_k < \frac{p(x=x_k)}{p_{max}}$. If it is true, accept x_k as a sample from $p(x)$, otherwise reject the sample.

The rejection rate determines the efficiency of the algorithm.

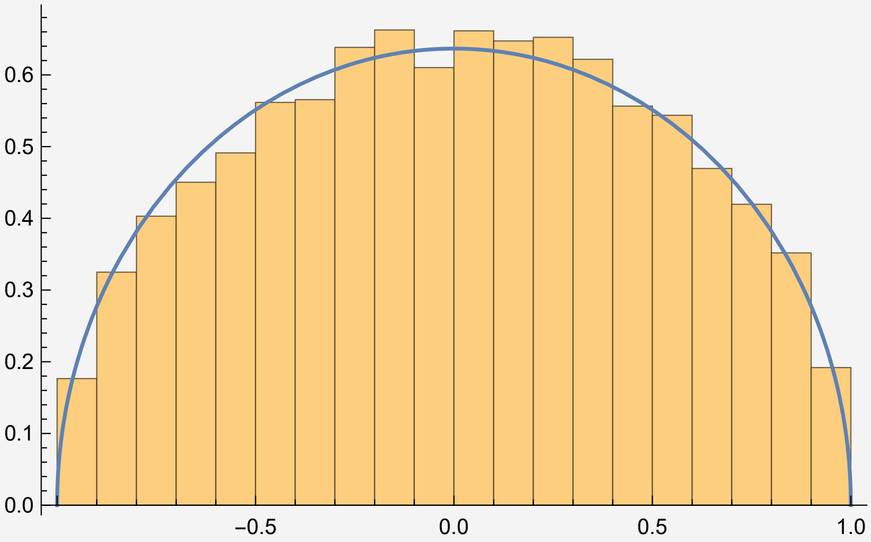
Example 2.6 (Sampling from a SemiCircle Distribution) We want to sample data from a semicircle distribution

$$p(x) = \begin{cases} \frac{2}{\pi}\sqrt{1-x^2} & -1 \leqq x \leqq +1 \\ 0 & otherwise \end{cases} \tag{2.31}$$

The maximum value of the distribution is given by $p_{max} = \frac{2}{\pi}$.

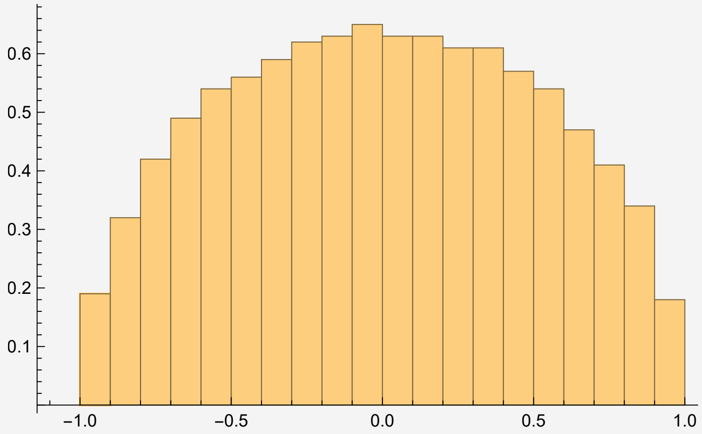
The following Wolfram language code implements rejection sampling for this distribution.

```
pmax = 2/Pi;
SemiCircleDistribution[x_] := pmax * Sqrt[1 - x^2];
xk = RandomVariate[UniformDistribution[{-1, 1}], 10000];
rk = RandomVariate[UniformDistribution[{0, 1}], 10000];
data = Pick[xk, MapThread[#1 < (SemiCircleDistribution[#2]/pmax) &, {rk, xk}]];
Show[{Histogram[data, 20, "PDF"],
      Plot[SemiCircleDistribution[x], {x, -1, 1}]}]
```



The SemiCircle pdf describes the distribution of eigenvalues of an infinite symmetric random matrix with elements distributed according to a Gaussian probability density.

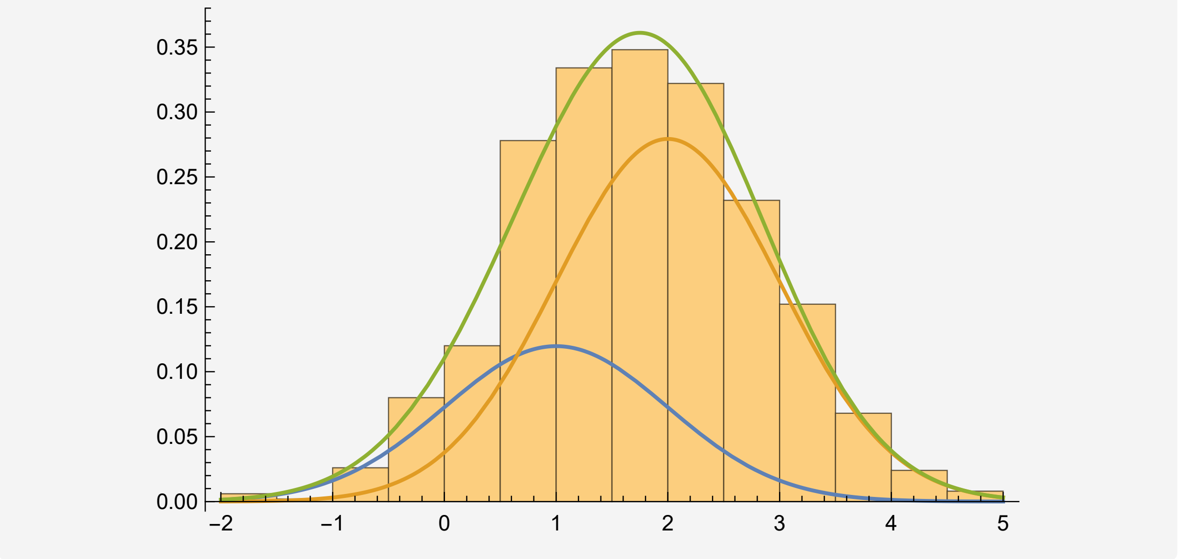
```
mat = ArrayReshape[RandomVariate[NormalDistribution[], 1000000], {1000, 1000}];
eigenvals = Eigenvalues[Symmetrize[mat]];
Show[Histogram[eigenvals/Max[Abs[eigenvals]], 20, "PDF"]]
```

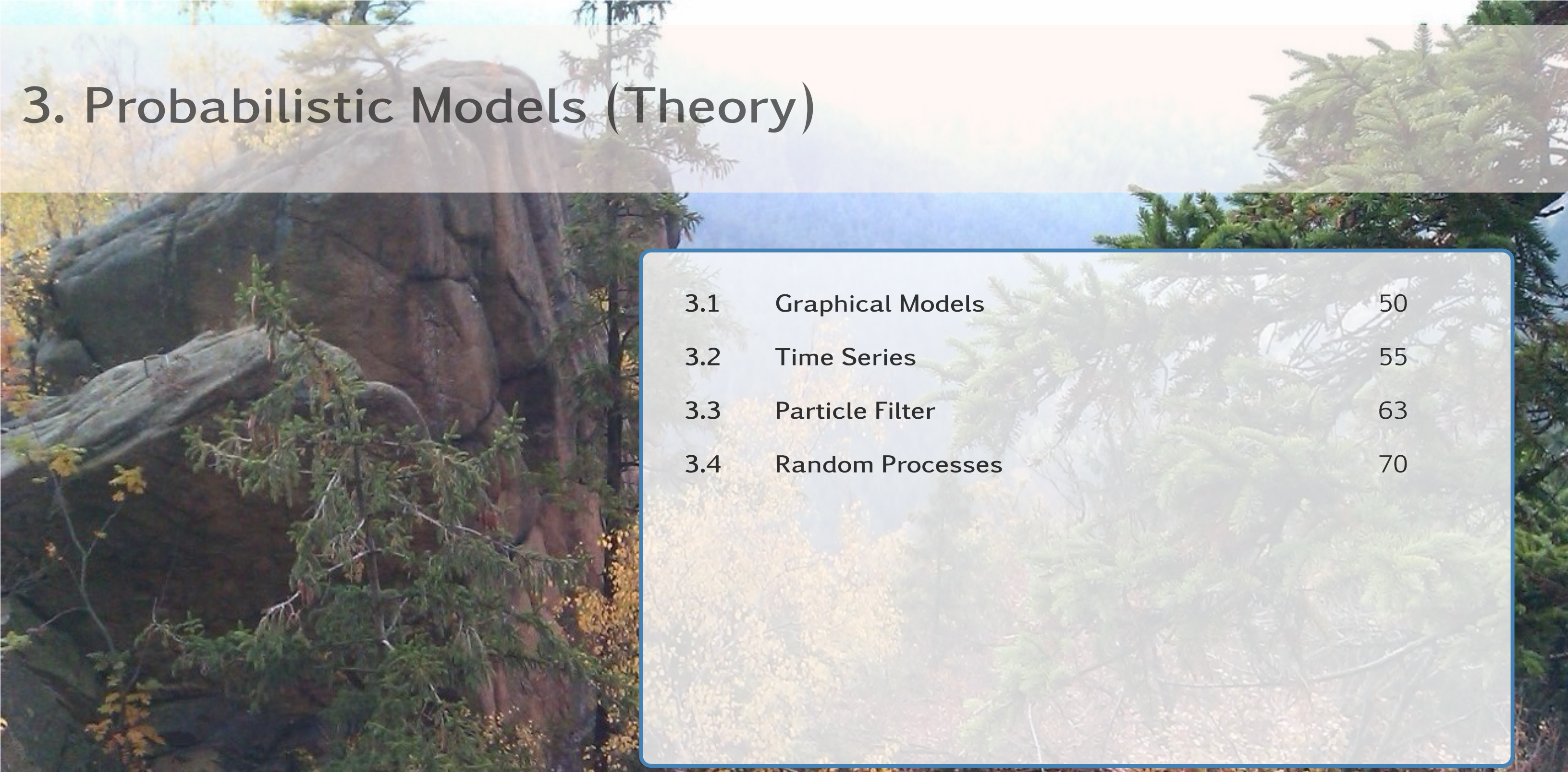


The corresponding function in the Wolfram Language is WignerSemicircleDistribution[a,r], which describes a semi circle of radius r centered at a.

Example 2.7 (Sampling from a Gaussian Mixture Model) Sampling from a Gaussian Mixture Model is easy to implement in the Wolfram Language. We write a function SampleFromGaussianMixtureModel with four arguments: weights, which are the normalized coefficients as a list, means, which is a list of mean vectors or numbers, covariances, which is a list of covariance matrices or standard variances and numSamples, which is the number of samples, which should be returned. In the case of a multi-variate distribution, the returned samples are a matrix with numSamples rows and as many columns as there are variables. The first step is to create a list of normal distributions with the parameters specified by means and covariances, which will be the input for the MixtureDistribution together with weights. This gives already a distribution, where we can sample from using RandomVariate. The output will be either a list of samples in the uni-variate case or a matrix with numSamples and as many columns as there are variables in the multi-variate case.

```
SampleFromGaussianMixtureModel[weights_,
                               means_,
                               covariances_,
                               numSamples_] := Module[
  {distributions, mixture, samples},
  distributions = If[NumberQ[means[[1]]],
                     MapThread[NormalDistribution[#1, #2] &,
                               {means, covariances}],
                     MapThread[MultinormalDistribution[#1, #2] &,
                               {means, covariances}]];
  mixture = MixtureDistribution[weights, distributions];
  samples = RandomVariate[mixture, numSamples];
  Return[samples]]

data = SampleFromGaussianMixtureModel[{0.3, 0.7},
                                      {1, 2}, {1, 1},
                                      1000];
Show[{Histogram[data, 20, "PDF"],
      Plot[{0.3 * PDF[NormalDistribution[1, 1], x],
            0.7 * PDF[NormalDistribution[2, 1], x],
            0.3 * PDF[NormalDistribution[1, 1], x] +
            0.7 * PDF[NormalDistribution[2, 1], x]}, {x, -2, 5}]}]
```


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−2
−1
0
1
2
3
4
5



3. Probabilistic Models (Theory)

3.1 Graphical Models 50

3.2 Time Series 55

3.3 Particle Filter 63

3.4 Random Processes 70

This chapter introduces the theoretical foundations of probabilistic models, especially how models can be decomposed into simpler parts and how these parts are connected with each other. Such structures can be best represented by graphs or networks, which allows us to identify important relations between random variables. These graphical representations help us also to deal systematically with more complex and diverse systems and allow us to formulate and implement everything in a common graphical way built on top of graph theory, which provides us with mathematical algorithms to perform efficient calculations.

3.1 Graphical Models

Probabilistic models are described by a number of random variables $x_{1:n}$, a number of parameters $\theta_{1:m}$ and a joint probability density $p(x_{1:n}|\theta_{1:m})$. It is often favorable to introduce further hidden or latent variables $h_{1:k}$, which are not directly observable and therefore do not show up as inputs or outputs of the system, but are hidden inside the model.

$$p(x_{1:n}|\theta_{1:m}) = \int_{-\infty}^{+\infty} dh_1 \dots \int_{-\infty}^{+\infty} dh_k p(x_{1:n}, h_{1:k}|\theta_{1:m}) \tag{3.1}$$

These new variables $h_{1:k}$ encode some static or dynamic properties of the system state and often simplify the overall structure of the model by decoupling the outputs from the inputs, which reduces the dependencies between the random variables and parameters in the model. Dependency or causal relations can be very vividly expressed through a graph. Such a graph is also called a belief or Bayesian network [1] [11].

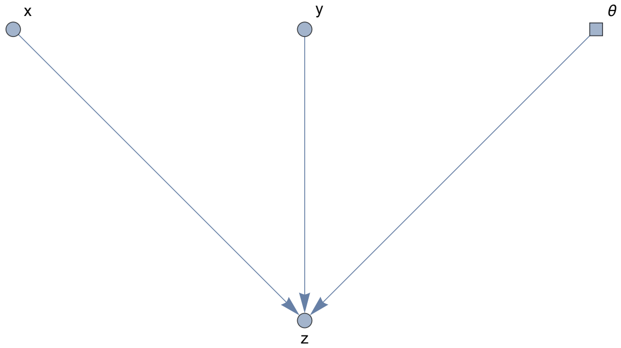
Definition 3.1 (Belief Network) A belief network is a directed graph of random variables $x_{1:n}$, which represents a probability distribution of the form

$$p(x_{1:n}) = \prod_{i=1}^{n} p\left(x_i|pa(x_i)\right) \tag{3.2}$$

where $pa(x_i)$ are all random variables x_j (parents), which are connected to the random variable x_i (child) with a directed edge pointing from node x_j to node x_i.

The following Wolfram Language code draws a graph with 3 random variables x, y, z and one parameter θ. The function `Graph` gets an adjacency list as the first argument, which contains a list of edges. The nodes or vertices are numbered from 1 to 4 and `i \[DirectedEdge] j` means that there is a directed edge from node i to j. This expresses a conditional dependence of node j on node i. The other options are only necessary to customize the graphical representation.

```
Graph[{1 \[DirectedEdge] 3, 2 \[DirectedEdge] 3, 4 \[DirectedEdge] 3},
     VertexLabels -> {1 -> "x", 2 -> "y",
                      3 -> Placed["z", Below], 4 -> \[Theta]},
     VertexShapeFunction -> {4 -> "Square"}, VertexSize -> Tiny]
```



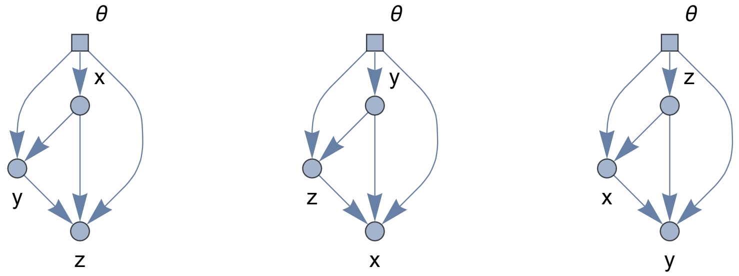
From this graph we can immediately derive the joint probability density as

$$p(x,y,z|\theta) = p(z|x,y,\theta)p(x)p(y) \tag{3.3}$$

The multivariate probability density is decomposed into univariate probability densities. We saw such a decomposition already in eq. 1.20. We can iteratively apply Bayes rule to $p(x,y,z|\theta)$ with the result

$$\begin{aligned}p(x,y,z|\theta) &= p(z|x,y,\theta)p(y|x,\theta)p(x|\theta)\\ p(x,y,z|\theta) &= p(x|y,z,\theta)p(z|y,\theta)p(y|\theta)\\ p(x,y,z|\theta) &= p(y|z,x,\theta)p(x|z,\theta)p(z|\theta)\end{aligned} \tag{3.4}$$

and the corresponding graphs



All these decompositions are equivalent and differ only by a reordering of the random variables x, y and z. The first one can be rewritten to eq. 3.3 using $p(y|x,\theta) = p(y)$ and $p(x|\theta)$ assuming that y is conditionally independent of x and θ and x is conditionally independent of θ, but the other two decompositions seem to express contradictory conditional independence relations, so what is now correct?

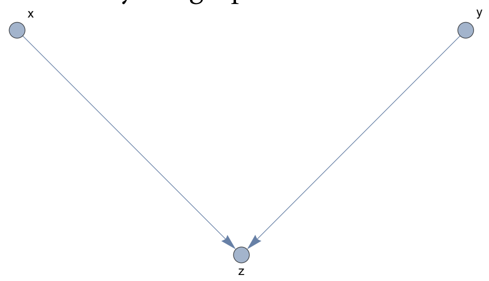
One has to be careful to interpret these equations as conditional independence relations e.g. the third equation has a term $p(z|\theta)$, which indicates that z is only conditionally dependent on θ, which could be right or wrong. This can be only decided if the probability density is known. If it is right, $p(z|\theta)$ can indeed be written as a function of the parameter θ only. If it is wrong e.g. z depends on x and y, $p(z|\theta)$ is the result of a marginalization of $\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} p(z|x,y,\theta)p(x)p(y)dxdy$.

The graph does exactly that: it specifies conditional independence relations through the lack of edges between certain nodes (in the example above the edges $\theta \rightarrow x$, $\theta \rightarrow y$ and $x \rightarrow y$) , which can be used to simplify the joint probability density. This is basically an assumption or belief based on some prior knowledge. If this assumption is valid or sufficient to describe the observations or data, has to be verified.

Using eq. 3.3 it can be easily shown that x and y are independent.

$$p(x,y|\theta) = \int_{-\infty}^{\infty} dzp(x,y,z|\theta) = \int_{-\infty}^{\infty} dzp(z|x,y,\theta)p(x)p(y) = p(x)p(y) \tag{3.5}$$

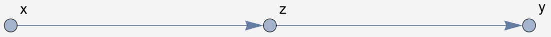
This independence relation holds for any subgraph of the form



which is also called v-structure.

Note that x and y in the v-structure are not conditionally independent given z.

$$p(x,y|z) = \frac{p(x,y,z)}{p(z)} = \frac{p(z|x,y)p(x)p(y)}{p(z)} \tag{3.6}$$

Example 3.1 Show that the graph  implies the independence of x and y given z.

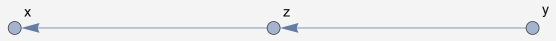
The joint probability density for this graph is given by

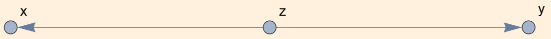
$$p(x,y,z) = p(y|z)p(z|x)p(x) = p(y|z)p(z,x) = p(y|z)p(x|z)p(z) \tag{3.7}$$

Therefore we get for $p(x,y|z)$

$$p(x,y|z) = \frac{p(x,y,z)}{p(z)} = p(x|z)p(y|z) \tag{3.8}$$

There is no way for x to influence y if we fix z. Because the result is symmetric in x and y, the same conditional independence relation holds if the nodes x and y in the graph are exchanged



Exercise 3.1 Show that the graph  also implies the independence of x and y given z.

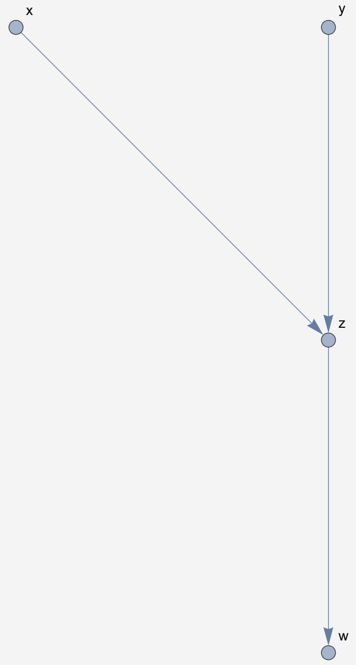
Example 3.2 **(Conditional Independence Test)** We want to implement a function `IsConditionalIndependent`, which returns `True` if `sourceNode` is conditionally independent of `targetNode` given a list of `conditionalNodes` for `graph`, otherwise `False`.

The implemented algorithm works as follows:

- If `sourceNode` and `targetNode` are identical, then they are dependent.
- If there exist undirected paths between `sourceNode` and `targetNode` and one of these paths has only two nodes, `sourceNode` and `targetNode` are directly connected and dependent.
- If there is no undirected path between `sourceNode` and `targetNode`, they are independent.
- If all paths contain at least one segment with 3 nodes, which is not a v-structure and the center node is in `conditionalNodes`, `sourceNode` and `targetNode` are independent.
- If all paths contain at least one segment with 3 nodes, which is a v-structure and the center node and all its descendants are not in `conditionalNodes`, `sourceNode` and `targetNode` are independent, else they are dependent.

```
IsConditionalIndependent[g_Graph,
                         sourceNode_Integer,
                         targetNode_Integer,
                         conditionalNodes_List]:= Module[
    {paths, blockedPathSegments},
    (* source and target node are identical *)
    If[sourceNode == targetNode,
       Return[False]];
    paths = FindPath[UndirectedGraph[g], sourceNode, targetNode,
                     Infinity, All];
    (* there is a direct path between source and target node *)
    If[AnyTrue[paths, Length[#] == 2 &],
       Return[False]];
    (* there is no path between source and target node *)
    If[Length[paths] == 0,
       Return[True]];
    (* partition each path into overlapping segments of 3 nodes *)
    paths = Map[Partition[#, 3, 1] &,
                paths];
    (* get all paths, which are not blocked by a a conditional node *)
    blockedPathSegments = Map[Not[EdgeQ[g, #[[1]] \[DirectedEdge] #[[2]]] &&
                                  EdgeQ[g, #[[3]] \[DirectedEdge] #[[2]]]] &&
                                  MemberQ[conditionalNodes, #[[2]]]  &,
                              paths, {2}];
    paths = Pick[paths, Map[Not[AnyTrue[#, TrueQ]] &, blockedPathSegments]];
    If[Length[paths] == 0,
       Return[True]];
    (* get all paths, which are not blocked by a v-structure *)
    blockedPathSegments = Map[EdgeQ[g, #[[1]] \[DirectedEdge] #[[2]]] &&
                              EdgeQ[g, #[[3]] \[DirectedEdge] #[[2]]] &&
                              Not[MemberQ[conditionalNodes, #[[2]]]] &&
                              Length[Intersection[VertexOutComponent[g, #[[2]]],
                                                  conditionalNodes]] == 0  &,
                              paths, {2}];
    paths = Pick[paths, Map[Not[AnyTrue[#, TrueQ]] &, blockedPathSegments]];
    If[Length[paths] == 0,
       Return[True],
       Return[False]];
    ]

g = Graph[{1 \[DirectedEdge] 3, 2 \[DirectedEdge] 3, 3 \[DirectedEdge] 4},
          VertexLabels -> {1 -> "x", 2 -> "y", 3 -> "z", 4 -> "w"}];
IsConditionalIndependent[g, 1, 2, {4}]
```

returns `False` for the graph



which states that x is not independent of y given w even if x, y and z are nodes of a v-structure.

To verify this statement, we write down the joint distribution for this graph

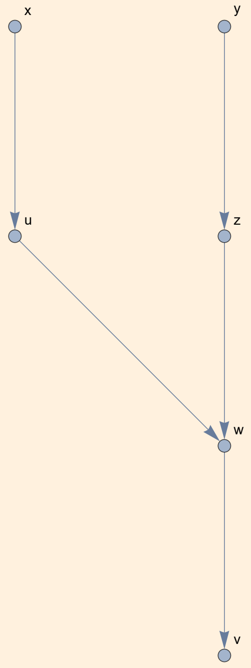
$$p(x,y,z,w) = p(w|z)p(z|x,y)p(x)p(y) \tag{3.9}$$

We marginalize over z and condition on w.

$$p(x,y|w) = \int_{-\infty}^{+\infty} dz \frac{p(x,y,z,w)}{p(w)} = p(x)p(y) \int_{-\infty}^{+\infty} dz \frac{p(w|z)p(z|x,y)}{p(w)} \tag{3.10}$$

which does not factor into a product of x and y. For some special probability densities it could happen that the factorization occurs, but in general x and y are dependent, which agrees with the result of the function `IsConditionalIndependent`. The independence relation is broken in the case of a v-structure if any descendants of the center node is conditioned on.

Exercise 3.2 Apply the function `IsConditionalIndependent` to the graph



and check under which conditions node x and y are independent when conditioned on a certain subset of the remaining nodes.

3.2 Time Series

We now want to apply graphical models to the analysis of time series data [2]. A time series can be represented by a sequence of random variables $x_{1:T}$, which are associated with a sequence of time points $t_{1:T}$. A time point can be e.g. 01 Jan 1970 00:00:00.000, which has a resolution of milliseconds, but depending on your application a higher resolution may be necessary such as nanoseconds and therefore more digits after decimal point. We also use the notation x_t, where t stands for the index associated with current time t, and x_{t-1} means the random variable at the previous time point and x_{t+1} the random variable at the next future time point and so on.

The Wolfram Language has the built-in symbol `TimeSeries` as a wrapper around a list of time-value pairs $\{t_i, x_i\}$, where x_i can be a scalar or an array and t_i can be numeric or `Ab-`

`soluteTime` e.g. `AbsoluteTime["01 Jan 1900 00:00:00.000"]`, which returns the result `0` as the number of seconds since the beginning of January 1, 1900 in your local timezone. Local time has typically an offset to GMT in multiple of hours including daylight savings, which is given by `$TimeZone`. Therefore the number of seconds returned by `AbsoluteTime` is the same on each computer in the world, where the code is executed.

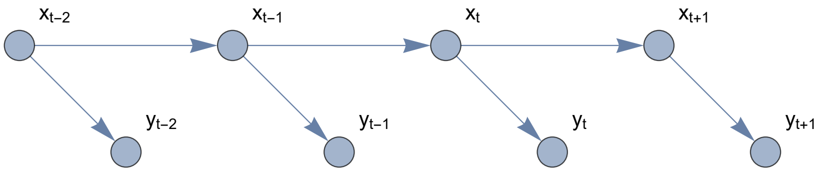
If you specify a timezone explictly e.g.

```
AbsoluteTime["01 Jan 1900 00:00:00.000", TimeZone -> 0],
```

the returned number is non-zero and a multiple of 3600, because it calculates the difference between January 1, 1900 00:00:00 GMT and January 1, 1900 00:00:00 in your local timezone.

`AbsoluteTime` does not take into account leap seconds. Therefore the difference between two calendar dates calculated by `AbsoluteTime` will differ by the amount of leap seconds, which were introduced in this time interval.

We want to study a probabilistic model based on the following subgraph



The idea behind this model is that the system is fully described by the variable x and there are observations y, which determine x completely or partially depending on the measurement process. The system state x is not directly observable, nevertheless we want to predict the future system state based on all its observations and some prior knowledge of x.

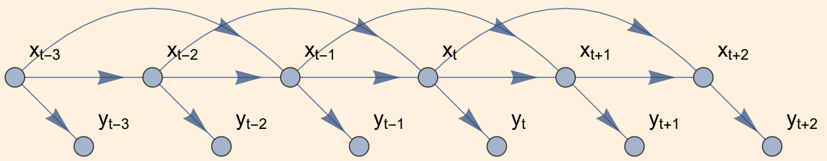
The system state x also fulfills the Markov property.

Definition 3.2 (Markov Property) The system state x_t is completely determined by the previous state x_{t-1} only, which can be written as

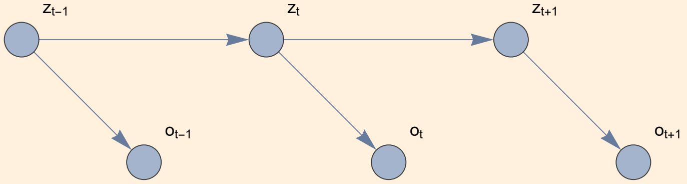
$$p(x_t|x_{t-1},x_{t-2},\ldots,x_{t-n}) = p(x_t|x_{t-1}) \qquad (3.11)$$

The property can be directly read off the graph, because if we condition on x_{t-1}, x_t become conditionally independent of its complete past history $x_{t-2},\ldots$ as we have shown in example 3.1.

Exercise 3.3 A graph like



does not fulfill the Markov property. Show that by merging two x nodes and two y nodes to new nodes z and o, the new graph takes the form



with $z_{t-1} = \{x_{t-3}, x_{t-2}\}$, $o_{t-1} = \{y_{t-3}, y_{t-2}\}$, $z_t = \{x_{t-1}, x_t\}$, $o_t = \{y_{t-1}, y_t\}$, $z_{t+1} = \{x_{t+1}, x_{t+2}\}$, $o_{t+1} = \{y_{t+1}, y_{t+2}\}$ and fulfills the Markov property.

The joint probability density can be immediately written down as a recursive equation

$$p(x_{t+1}, y_{t+1}, x_t, y_t, \ldots) = p(x_{t+1}|x_t)p(y_{t+1}|x_{t+1})p(x_t, y_t, x_{t-1}, y_{t-1}, \ldots) \tag{3.12}$$

We are only interested to predict x_{t+1}, so we marginalize over all previous system states $x_t, x_{t-1}, \ldots$.

$$\begin{aligned} p(x_{t+1}, y_{t+1}, y_t, \ldots) &= \int_{-\infty}^{+\infty} dx_t \int_{-\infty}^{+\infty} dx_{t-1} \ldots p(x_{t+1}, y_{t+1}, x_t, y_t, \ldots) \\ &= p(y_{t+1}|x_{t+1}) \int_{-\infty}^{+\infty} dx_t p(x_{t+1}|x_t) \int_{-\infty}^{+\infty} dx_{t-1} \ldots p(x_t, y_t, x_{t-1}, y_{t-1}, \ldots) \\ &= p(y_{t+1}|x_{t+1}) \int_{-\infty}^{+\infty} dx_t p(x_{t+1}|x_t) p(x_t, y_t, y_{t-1}, \ldots) \end{aligned} \tag{3.13}$$

We now condition on all the observations $y_{t+1}, y_t, \ldots$

$$\begin{aligned} p(x_{t+1}|y_{t+1}, y_t, \ldots) &= \frac{p(x_{t+1}, y_{t+1}, y_t, \ldots)}{p(y_{t+1}, y_t, \ldots)} = \frac{p(x_{t+1}, y_{t+1}, y_t, \ldots)}{p(y_{t+1}|y_t, \ldots)p(y_t, \ldots)} \\ &= \frac{1}{p(y_{t+1}|y_t, \ldots)} p(y_{t+1}|x_{t+1}) \int_{-\infty}^{+\infty} dx_t p(x_{t+1}|x_t) \underbrace{\frac{p(x_t, y_t, y_{t-1}, \ldots)}{p(y_t, \ldots)}}_{p(x_t|\ y_t, y_{t-1}, \ldots)} \end{aligned} \tag{3.14}$$

This recursive equation is the mathematical fundament of the Bayes filter [8].

Definition 3.3 **(Bayes Filter)** The probability density $p(x_{t+1}|y_{t+1}, y_t, \ldots)$ can be calculated iteratively from the last known $p(x_t|\ y_t, y_{t-1}, \ldots)$ by a prediction step

$$p(x_{t+1}|y_t, \ldots) = \int_{-\infty}^{+\infty} dx_t p(x_{t+1}|x_t) p(x_t|y_t, y_{t-1}, \ldots) \tag{3.15}$$

followed by a correction step

$$p(x_{t+1}|y_{t+1},y_t,\ldots) = \frac{p(y_{t+1}|x_{t+1})}{p(y_{t+1}|y_t,\ldots)} p(x_{t+1}|y_t,\ldots) \tag{3.16}$$

$p(x_{t+1}|x_t)$ describes the time evolution of the system and $p(y_{t+1}|x_{t+1})$ the measurement process of the system state. If the system state is controlled by an external input u_t, it is only necessary to modify $p(x_{t+1}|x_t)$ by $p(x_{t+1}|x_t,u_t)$.

The Bayes filter estimates the state at time $t+1$ from the all the previous states and observations, but how do we start the iteration or what can we say about the system state without any observations?

It turns out that the prior state estimate $p(x_0)$ before any observations are made can be chosen as a constant. After the first correction step when the normalization is performed the constant will anyhow be cancelled out and the state estimate will improve with each update.

The Bayes filter framework is the conceptual basis of a large number of existing models like Kalman and Particle filters (see 3.3), Hidden Markov Models (see next example), Dynamic Bayesian Networks (basically the introduction to this chapter) and Markov Decision Processes with a wide range of applications.

Example 3.3 (Graph World) We want to study a simple system called Graph World, which is a generalization of the Grid World system often used as an introductory example in reinforcement learning.

The system consists of a finite number of states $1,2,\ldots,k$ and a $k \times k$ transition matrix P_{ij}, which describes the probabilities of a transition from state i to state j. P_{ij} is also an adjacency matrix with an associated graph structure, where the vertices are the system states and the directed edges with an assigned weight corresponding to the probability that a transition from a source to a target node happens (a probability 0 means that such a transition never happens and therefore no edge is drawn in this case).

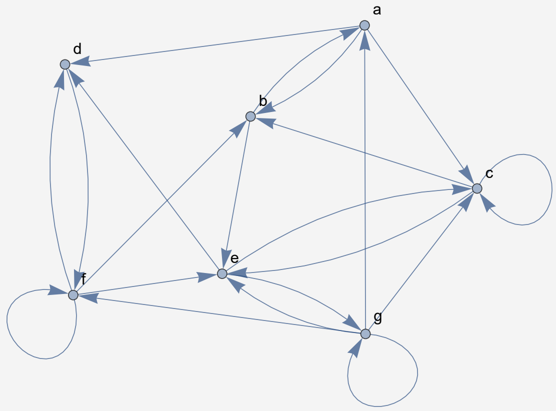
We further introduce a measurement process, which measures the distance between an arbitrarily selected observer node o and the current system state i in terms of the number of edges between them. Because there can be many possible paths between two nodes in the graph, the one with a minimum number of edges is chosen as a measure of distance. However, this measurement does not determine the system state uniquely, because there may be several other nodes with the same distance. Only for distance 0 the system state will be known without doubt.

Let's assume that the system is in state i at t_0 and in the next time step t_1, it will transition to another state j, where $P_{ij} \neq 0$. It is also possible that the system remains in the same state at t_1 if $P_{ii} \neq 0$. At the beginning the first state is unknown and we start with a prior state estimate of $p(x_0) = \frac{1}{k}\sum_{i=1}^{k} \delta(x-i)$.

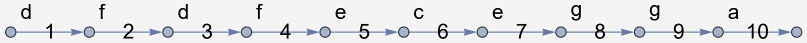
We write first a function `GraphWorld`, which constructs a random graph world from a list of node names, which provide a textual label for each node, and the length of history of states. It returns key-value pairs with 4 pre-defined keys: a key "graph" with an associated Graph object, a key "transition_probabilities" with the matrix P_{ij}, a key history with a sequence of states, which are generated from a random initial state, and a graphical representation of the state history with key "timeline".

```
GraphWorld[nodeLabels_List, historyLength_Integer] := Module[
   {graph, transitionMatrix, history, world, numNodes, labels,
   initialState, nodes, dists, edges, historyGraph},

   numNodes = Length[nodeLabels];
   nodes = Range[numNodes];
   labels = MapThread[#1 -> #2 &, {nodes, nodeLabels}];
   transitionMatrix = RandomChoice[{0, 1}, {numNodes, numNodes}];

   (* There could be a row with only zeros, where no transitions to
      another node occurs. To avoid this situation, a random unit
      vector will replace such a row.
    *)
   transitionMatrix = Map[If[Total[#] == 0,
                          UnitVector[Length[#], RandomChoice[nodes]], #] &,
                          transitionMatrix];

   (* A graphical representation of all possible state transitions, which can occur
      in this world.
    *)
   graph = AdjacencyGraph[transitionMatrix, VertexLabels -> labels];

   (* Normalize each row of the transition matrix *)
   transitionMatrix = Map[#/Total[#] &, transitionMatrix];
   initialState = RandomChoice[nodes];

   (* Create an empirical distribution for each node from the transition matrix *)
   dists = Map[EmpiricalDistribution[# -> nodes] &,
               transitionMatrix];

   (* Starting with a random initial state, sample a new state from the old state
      using the empirical distributions and collect them in a list.
    *)
   history = NestList[RandomVariate[dists[[#]]] &,
                      initialState, historyLength];

   (* Create a graphical representation of the state transitions as directed graph,
      where each edge represents a time step from node at time t to the node at
      time t + 1.
    *)
   edges = Partition[history, 2, 1];
   historyGraph = Graph[Table[Labeled[i \[DirectedEdge] i + 1, i],
                              {i, 1, Length[edges]}],
    VertexLabels -> Append[Table[i -> nodeLabels[[edges[[i, 1]]]],
                                 {i, 1, Length[edges]}],
                    Length[edges] + 1 -> nodeLabels[[edges[[Length[edges], 2]]]]]];
   world = Association["graph" -> graph,
                       "transition_probabilities" -> Transpose[transitionMatrix],
                       "history" -> history ,
                       "timeline" -> historyGraph];
   Return[world]]
```

```
SeedRandom[123456789];
world = GraphWorld[Characters["abcdefg"], 10];
world["graph"]
```



```
world["timeline"]
```



The timeline graph shows the sequence of states at $t_0, t_1, \ldots$ and the edge i represents a transition from state at t_{i-1} to the state at t_i. We assume that there are no measurements before t_1.

In the next step we implement a measurement function MeasurementProcess, which gets a world as the first argument, on which the measurement is performed, and an observer node observerNode and returns a list of distributions for each time step of the world history. Each distribution has as many elements as number of nodes in the graph and the values represent the probability that the graph distance of a node to the observer node is compatible with the distance measurement of the system state. If the graph distance would unambiguously determine the system state, only one element would be 1 and the others would be 0, but there are typically more nodes with the same graph distance. Because the measurement cannot distinguish between these nodes, they have equal probability to be the actual system state.

Wolfram Language provides the function GraphDistance, which returns the number edges of the shortest path between two nodes in a graph. If two nodes are not connected, the graph distance will be Infinity.

```
MeasurementProcess[world_, observerNode_] := Module[
   {measurements, distances, size, distributions},

   (* Calculate a list of measurements of the graph distance between the
      observer node and all the system states in the world’s history
      excluding the initial system state.
    *)
   measurements = Map[GraphDistance[world["graph"], observerNode, #]&,
                      world["history"][[2 ;; All]]];

   (* Calculate the graph distances between the observer node and all other
      graph nodes. *)
   distances = GraphDistance[world["graph"], observerNode];
   size = Length[distances];

   (* Calculate the measurement distribution for each time step by checking,
      which node has the same graph distance than the measurement.
    *)
   distributions = Boole[Map[MapThread[Equal, {distances, Table[#, size]}] &,
                             measurements]];

   (* Normalize each row so that the values can be interpreted as
      probabilities. *)
   distributions = Map[# / Total[#]&, distributions];
   Return[distributions]]

measurements = MeasurementProcess[world, 2];
```

The observer node is chosen to be b.

The Bayes filtering starts with an initial distribution of states reflecting our prior knowledge about the system state at t_0. Because we have no prior knowledge at this point, we choose a uniform distribution over all possible states or nodes of the graph.

```
nodes = Characters["abcdefg"];
stateDistribution = AssociationThread[nodes -> Table[1/Length[nodes],
                                                     Length[nodes]]];
```

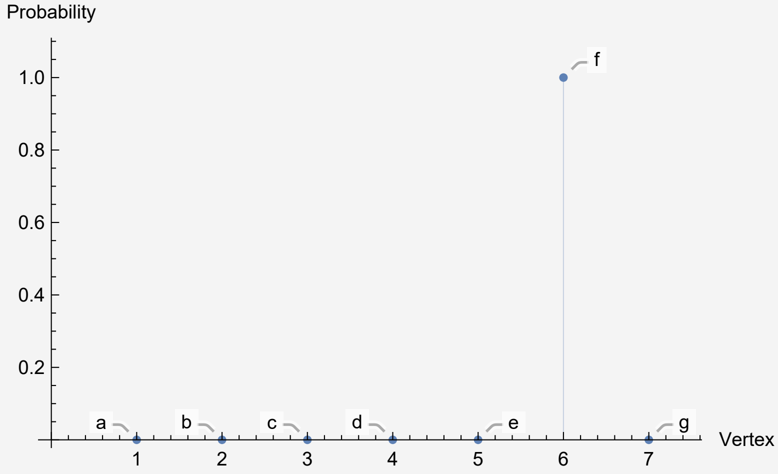
Then we apply the transition matrix to this distribution in the prediction step.

```
stateDistribution = AssociationThread[nodes -> world["transition_probabilities"].
                                          Values[stateDistribution]];
```

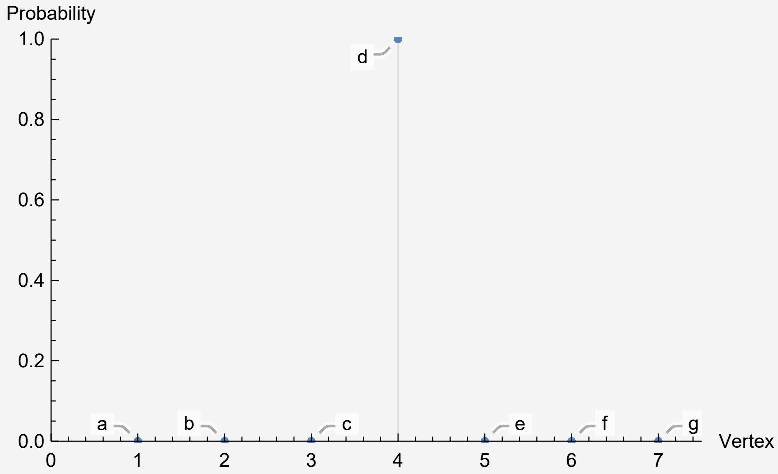
In the correction step we multiply with the first element of the distributions returned by `MeasurementProcess` and normalize the result.

```
stateDistribution = AssociationThread[nodes -> measurements[[1]] *
                                          Values[stateDistribution]];
stateDistribution = If[Total[stateDistribution] == 0,
                       AssociationThread[nodes -> Table[1/Length[nodes],
                                                        Length[nodes]]],
                       stateDistribution / Total[stateDistribution]];
```

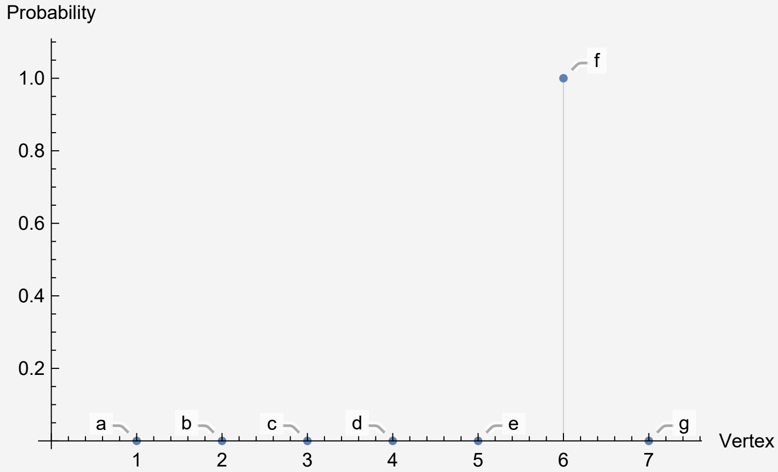
This gives the estimated state distribution at time step t_1.



We repeat the steps for t_2 with the result

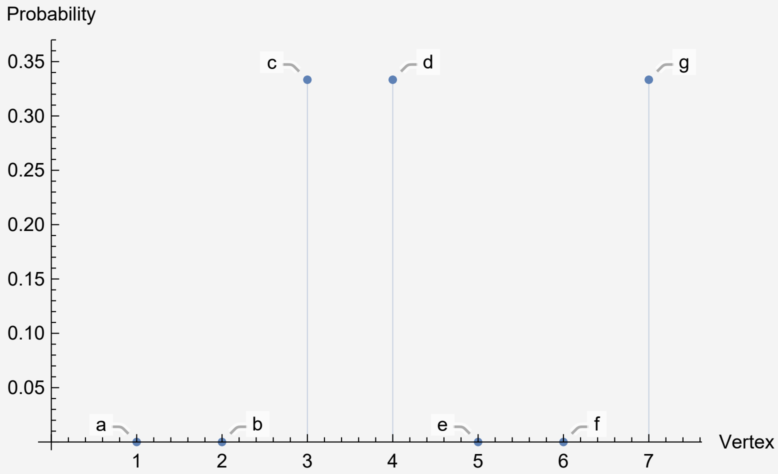


and for t_3

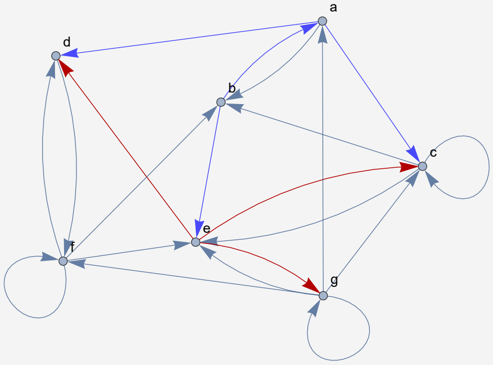


If we take the state with the maximum probability, the state estimates at t_1, t_2 and t_3 are f, d and f, which agrees perfectly with the timeline graph.

Also at t_4 the system state is correctly predicted as e, but at t_5 the states c, d and g have all the same probability and the correct state cannot be unambiguously estimated.



The reason for this ambiguity is shown in the next figure, where the red edges mark all transitions from the previous state e at t_4 to the possible next states at t_5. The blue edges highlight the shortest paths from the observer node b to these states and they all have the same graph distance 2.



Therefore it is not possible to identify the correct state at t_5 based on this measurement.

Exercise 3.4 Modify the parameter of the last example e.g. the observer node or the number of nodes in the graph to get an deeper understanding how the Bayes filter works.

3.3 Particle Filter

The Bayes filter can be easily and efficiently applied to all problems, where the state space is discrete and not too large. For continuous state spaces there exist closed-form solutions only for special cases like the Kalman filter for linear models with additive Gaussian noise. In the general non-linear case with arbitrary additive noise distributions, only approximate solutions are possible.

We represent the distribution $p(x_t|y_t, y_{t-1}, \ldots)$ by a set of N particles with a weight $w_t^{(i)}$

and a state $x_t^{(i)}$

$$p(x_t|y_t, y_{t-1}, \dots) = \sum_{i=1}^{N} w_t^{(i)}(y_t, y_{t-1}, \dots)\,\delta\left(x_t - x_t^{(i)}(y_t, y_{t-1}, \dots)\right) \tag{3.17}$$

with $\sum_{i=1}^{N} w_t^{(i)} = 1$. The density of particles in the state space is a measure for the probability of finding the system in a particular state.

Algorithm 1 Particle filter algorithm

```
1: Sample a new particle $x_{t+1}^{(i)}$ from $p\left(x_{t+1}|x_t^{(i)}\right)$ for $i \in 1, 2, \dots, N$.
2: Set the new particle weight to $w_{t+1}^{(i)} = p\left(y_{t+1}|x_{t+1}^{(i)}\right)$ for $i \in 1, 2, \dots, N$.
3: Resample the new particles based on their weights $w_{t+1}^{(i)}$ with replacement.
```

This algorithm generates a new set of particles at time $t+1$ from a given set of particles at time t through sampling. It depends on the functional form of $p\left(x_{t+1}|x_t^{(i)}\right)$ if this first sampling step can be done directly or using a proposal distribution $q\left(x_{t+1}|x_t^{(i)}\right)$, which is easy to sample from. In the last case the particle filter algorithm has to modified to include an importance sampling step, which uses the proposal distribution for sampling and takes into account the difference between actual and proposal distribution by multiplying the weights with a factor $p\left(x_{t+1}^{(i)}|x_t^{(i)}\right)/q\left(x_{t+1}^{(i)}|x_t^{(i)}\right)$.

Algorithm 2 Particle filter algorithm with importance sampling

```
1: Sample a new particle $x_{t+1}^{(i)}$ from $q\left(x_{t+1}|x_t^{(i)}\right)$ for $i \in 1, 2, \dots, N$.
2: Set the new particle weight to $w_{t+1}^{(i)} = p\left(y_{t+1}|x_{t+1}^{(i)}\right)\frac{p\left(x_{t+1}^{(i)}|x_t^{(i)}\right)}{q\left(x_{t+1}^{(i)}|x_t^{(i)}\right)}$ for $i \in 1, 2, \dots, N$.
3: Resample the new particles based on their weights $w_{t+1}^{(i)}$ with replacement.
```

The Resampling step will either add the same particle multiple times if the weight is high or remove it completely if the weight is small. Therefore several copies of the same particle will be present if the corresponding state has a high probability.

Example 3.4 (A moving ball in a two-dimensional non-transparent box) To demonstrate how the particle filter works, we choose the following scenario:

we have a two-dimensional system consisting of a non-transparent box and a ball moving inside the box with a constant speed. When the ball hits a wall, there is a sound, which tells us that the ball is now at a specific wall. In this case we know one coordinate of the ball exactly. The ball starts a random initial position (x_0, y_0) and with a random initial velocity (v_x, v_y). The magnitude of the velocity $v = \sqrt{v_x^2 + v_y^2}$ remains constant and is known, but the direction changes as soon as the ball hits a wall in such a way that one or both components of the velocity changes its sign.

The system state at time t is $(\vec{r},\vec{v})$, where $\vec{r}$ is the coordinate and $\vec{v}$ is the velocity of the ball. The initial state is specified by the arguments `initialTime`, `initialPosition` and `initialSpeed` and `finalTime` determines the length of the trajectory. The origin of the coordinate system is the center of the box. The default system boundaries are at $x = \pm 1$ and $y = \pm 1$.

We implement a function `Trajectory`, which returns the trajectory of the ball.

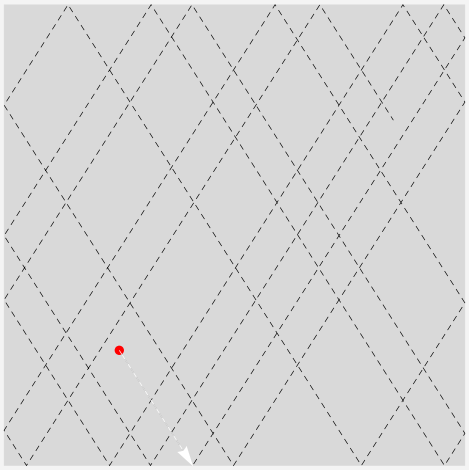
```
Trajectory[initialPosition_, initialSpeed_, initialTime_, finalTime_,
   boundaries_List:{+1,-1,+1,-1}] := Module[
   {trajectory, Propagate, lastState},

   (* The internal function gets the current state of the ball and returns the next
      state when the ball hits a wall. *)
   Propagate[state_] := Module[
      {collisionTimes, nextCollisionTime, t, newState, verticalWallCollision,
       horizontalWallCollision},
      (* The variable collisionTimes are the times, when the ball collides with
         the four walls of the box. Some of these time intervals will be negative,
         because the ball is moving away from these walls. A positive time means
         that the ball will hit the wall in the future. Because the ball changes
         direction after a collision with the wall, only the smallest time interval
         must be considered. *)
      collisionTimes = Association[
         1 -> (boundaries[[1]] - state["position"][[1]]) / state["velocity"][[1]],
         2 -> (boundaries[[2]] - state["position"][[1]]) / state["velocity"][[1]],
         3 -> (boundaries[[3]] - state["position"][[2]]) / state["velocity"][[2]],
         4 -> (boundaries[[4]] - state["position"][[2]]) / state["velocity"][[2]]];

      (* The next collision time is the one, where the time when the ball hits
         a wall is non-zero and smaller than the others after removing the negative
         collision times. *)
      nextCollisionTime = MinimalBy[Value] @ Select[collisionTimes, Positive];
      t = Values[nextCollisionTime][[1]];

      (* The next state of the ball is obtained by propagating the ball to the
         event of collision with the wall. *)
      newState = Association["time" -> state["time"] + t,
                             "position" -> state["position"] + state["velocity"]*t,
                             "velocity" -> state["velocity"],
                             "wall" -> Keys[nextCollisionTime][[1]]];
      verticalWallCollision = Boole[IntersectingQ[Keys[nextCollisionTime],
                                                  {1, 2}]];
      horizontalWallCollision = Boole[IntersectingQ[Keys[nextCollisionTime],
                                                    {3, 4}]];

      (* The velocity change by bouncing back from the wall is calculated
         by changing the sign of the first velocity component if the collision
         happens with a vertical wall or of the second velocity component if the
         collision occurs on a horizontal wall. The magnitude of the velocity
         remains constant and identical to the initial speed. If the ball hits
         an edge of the box and therefore a vertical and horizontal wall at the
         same time, the sign of both velocity components are changed and the ball
         is reflected back on the same trajectory where it is coming from. *)
      newState["velocity"] = {newState["velocity"][[1]] *
                                    (-1)^verticalWallCollision,
                              newState["velocity"][[2]] *
```

```
                                       (-1)^horizontalWallCollision};
      Return[newState]];

   (* The trajectory is calculated by applying the function Propagate
      several times starting with the initial state. The last state
      is dropped, because it has a time greater than the final time. *)
   trajectory = Drop[NestWhileList[Propagate,
                              Association["time" -> initialTime,
                                          "position" -> initialPosition,
                                          "velocity" -> initialSpeed,
                                          "wall" -> 0],
                              #["time"] < finalTime &], -1];
   lastState = Last[trajectory];

   (* Adds the final state to the trajectory by propagating the last state
      to the final time. *)
   AppendTo[trajectory,
            Association["time" -> lastTime,
                        "position" -> lastState["position"] +
                        lastState["velocity"] *
                        (finalTime - lastState["time"]),
                        "velocity" -> lastState["velocity"],
                        "wall" -> 0];
   Return[trajectory]]
```

The function can be used to generate a trajectory of the ball inside the box.

```
boundaries = {1, -1, 1, -1};
initialVelocity = {1/Pi, -0.5};
trajectory = Trajectory[{-0.5, -0.5}, initialVelocity, 0, 50, boundaries];
TrajectoryPlot[trajectory_, boundaries_]:= Module{box, ball, arrow, track},
    box = Polygon[{{boundaries[[1]], boundaries[[3]]},
                   {boundaries[[1]], boundaries[[4]]},
                   {boundaries[[2]], boundaries[[4]]},
                   {boundaries[[2]], boundaries[[3]]}}];
    ball = Point[trajectory[[1]]["position"]];
    arrow = Arrow[{trajectory[[1]]["position"], trajectory[[2]]["position"]}];
    track = Line[Map[#["position"] &, trajectory]];
    Graphics[{{LightGray, box},
              {Dashed, track},
              {PointSize[Large], Red, ball},
              {White, arrow}}]
]
TrajectoryPlot[trajectory, boundaries]
```

The result can be seen in the next figure, where the red point marks the initial position, the white arrow the direction of movement and the dashed line the complete trajectory of the ball.

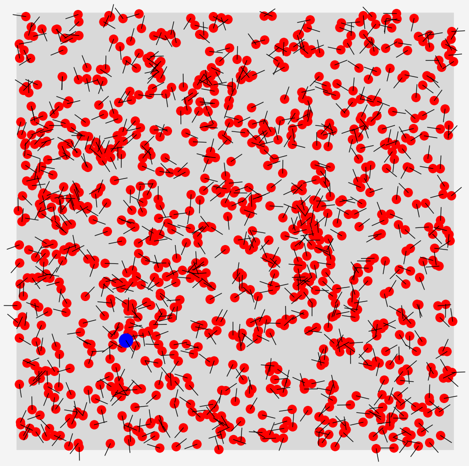


The function ParticleFilterInit generates a set of particles with random positions inside the box and assign a velocity vector to each particle, which has the same magnitude as the initial velocity of the ball, but a random direction.

```
ParticleFilterInit[numberOfParticles_, velocity_, boundaries_] := Module[
    {pos, v, vel, particles},
    (* The particle positions are uniformly distributed within the boundaries
       of the box. *)
    pos = Transpose[{RandomVariate[
                        UniformDistribution[{boundaries[[1]], boundaries[[2]]}],
                        numberOfParticles],
                     RandomVariate[
                        UniformDistribution[{boundaries[[3]], boundaries[[4]]}],
                        numberOfParticles]}];

    (* The particle moving directions are uniformly distributed, but the magnitude
       of the velocity is always the same for each particle. *)
    v = Sqrt[velocity.velocity];
    vel = Map[v * {Cos[#], Sin[#]} &,
              RandomVariate[UniformDistribution[{0, 2*Pi}], numberOfParticles]];
    particles = MapThread[Association["time" -> 0,
                                      "position" -> #1,
                                      "velocity" -> #2] &,
                          {pos, vel}];
    Return[particles];
]
```

Furthermore a function ParticleFilterPlot draws the particles inside the box and adds a short line to each particle to indicate the direction of motion.

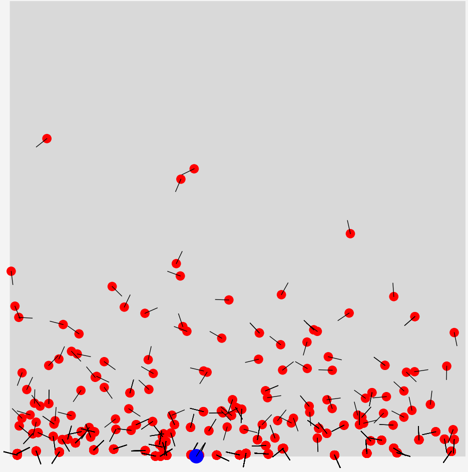
```
ParticleFilterPlot[particles_, currentState_, boundaries_] := Module[
  {points, directions, box},
  box = Polygon[{{boundaries[[1]], boundaries[[3]]},
                 {boundaries[[1]], boundaries[[4]]},
                 {boundaries[[2]], boundaries[[4]]},
                 {boundaries[[2]], boundaries[[3]]}}];
  points = Point[Map[#["position"] &, particles]];
  directions = Map[Line[{#["position"], #["position"] + #["velocity"]*0.1}] &,
                   particles];
  Graphics[{{LightGray, box},
            {PointSize[Large], Red, points},
            directions,
            {PointSize[0.03], Blue, Point[{currentState["position"]}]},
            Line[{currentState["position"],
                  currentState["position"] + currentState["velocity"]*0.1}]}]
]

particles = ParticleFilterInit[1000, initialVelocity, boundaries];
ParticleFilterPlot[particles, trajectory[[1]], boundaries]
```



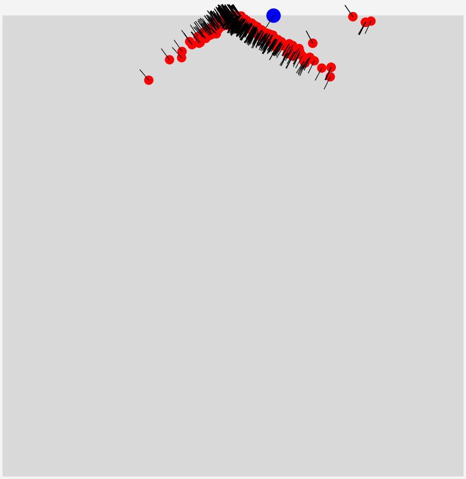
The variable `trajectory` contains the collision times, which are used to calculate the position of each particle at the instant when the ball hits the wall. The motion of each particle is deterministic, but we must add a certain degree of randomness to the state, otherwise the particle filter would not work properly. We use the function `Trajectory` to calculate the state at each collision time and then add a random deviation in the direction of the velocity.

The particle weight is calculated from a decreasing function of the perpendicular distance of the particle to the wall. There are many possible choices, but in this example we use $\frac{1}{1+1000x^2}$. We have to normalize the weights before the resampling step is performed.

```
ParticleFilterUpdate[particleStates_, observation_, boundaries_] := Module[
    {angles, velocities, distances, weights, dist, indices, particles},

    (* Predict the state of the particles at the next measurement time. *)
    particles = Map[Last[Trajectory[#["position"], #["velocity"],
                               #["time"], observation["time"]]] &,
                    particleStates];
    angles = RandomVariate[NormalDistribution[0, 0.05], Length[particles]];
    velocities = MapThread[{{Cos[#1], Sin[#1]},
                            {-Sin[#1], Cos[#1]}}.#2["velocity"] &,
                           {angles, particles}];
    particles = MapThread[Association["time" -> #1["time"],
                                      "position" -> #1["position"],
                                      "velocity" -> #2,
                                      "wall" -> #1["wall"]] &,
                          {particles, velocities}];

    (* Calculate the particle weights based on the distance
       of the particle to the wall, the ball has collided with. *)
    distances = Map[#["position"][[If[observation["wall"] <= 2, 1, 2]]] -
                    boundaries[[observation["wall"]]] &, particles];
    weights = Map[1/(1 + 1000 * #^2)&, distances];
    (* We do not need an explicit normalization step here, because
       the function EmpiricalDistribution normalizes the weights
       automatically. *)

    (* Resample the particles according to their weight. *)
    dist = EmpiricalDistribution[weights -> Range[1, Length[weights]]];
    indices = RandomVariate[dist, Length[particles]];
    particles = Map[particles[[#]] &, indices];
    Return[particles];
]

particles = ParticleFilterUpdate[particles,
                                 trajectory[[2]],
                                 boundaries];
ParticleFilterPlot[particles, trajectory[[2]], boundaries]
```



After 20 updates the particles are close to the actual position of the ball, but there is still a large spread in the distribution of particles. The observations provide only partial information about the ball state, which is not sufficient to determine the position and velocity completely. Therefore a large number of particles still move in the wrong directions.



To get an estimate of the ball state, we average over all particles

```
Association["time" -> Mean[Map[#["time"] &, particles]],
            "position" -> Mean[Map[#["position"] &, particles]],
            "velocity" -> Mean[Map[#["velocity"] &, particles]]]
```

Exercise 3.5 Due to numerical inaccuracies several problems can occur. One problem are collision times, which are positive and very close to zero. Another problem is the detection of an edge collision, when the times hitting the two walls are very close to each other, but not equal. In these situations the velocity change is incorrectly calculated and the ball can escape the box.

```
boundaries = {0.5, -0.1, 1, -1};
trajectory = Trajectory[{0, 0}, {0.1, 0.2}, 0, 40, boundaries];
TrajectoryPlot[trajectory, boundaries]
```

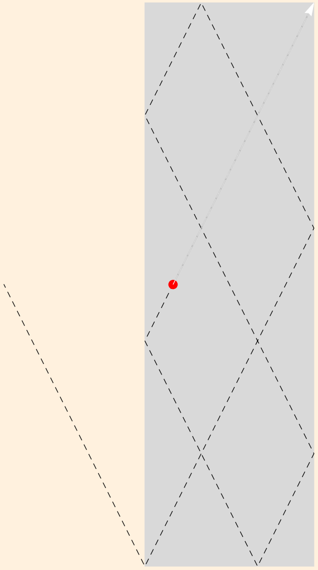


Figure out how to deal with these exceptional cases programmatically.

Exercise 3.6 Assume that the sound when the ball hits the wall gives a rough measurement of velocity component normal to the wall. Modify the measurement update in such a way that the weight calculation includes a term, which measures the difference between this velocity component of the particles and the observation. Study the influence on the distribution of particles.

3.4 Random Processes

Random processes have some similarity with time series, but without the association of time points with the random data. A random process $x_{1:n}$ is described by a functional relationship of the form

$$x_{n+1} = f(x_n, z_n) \tag{3.18}$$

where z_n is a random sample from a distribution $p(z)$ and the probability density of x_0 is known.

Example 3.5 (One-dimensional Random Walk) In this example we analyze the one-dimensional motion of a particle under a random displacement, which is described by the following equation

$$x_{n+1} = x_n + z_n \tag{3.19}$$

where x_n is the displacement of the particle after n steps and z_n is $+1$ with probability α and -1 with probability $1-\alpha$. The initial distribution $p\,(x_0)$ is $\delta\,(x_0)$, which means that the particle starts at position 0 and then moves either to the right with probability α or to the left with probability $1-\alpha$. What is the probability density after $n+1$ steps?

$$p\,(x_{n+1}) = \int_{-\infty}^{+\infty} dx_n \int_{-\infty}^{+\infty} dz_n p\,(x_{n+1}, x_n, z_n) = \int_{-\infty}^{+\infty} dx_n \int_{-\infty}^{+\infty} dz_n p\,(x_{n+1}|x_n, z_n)\,p\,(x_n, z_n) \tag{3.20}$$

Because z_n cannot influence x_n anymore, $p\,(x_n, z_n)$ can be replaced by $p\,(x_n)\,p\,(z_n)$. Furthermore x_{n+1} is completely determined given x_n and z_n and can be written as $p\,(x_{n+1}|x_n, z_n) = \delta\,(x_{n+1} - x_n - z_n)$. Therefore the z_n integral can be carried out with the result

$$p\,(x_{n+1}) = \int_{-\infty}^{+\infty} dx_n p\,(x_n)\,p\,(z_n = x_{n+1} - x_n) \tag{3.21}$$

Using $p\,(z_n) = \alpha\delta(z_n - 1) + (1-\alpha)\delta(z_n + 1)$, the x_n integral can be carried out too.

$$p\,(x_{n+1}) = \alpha p\,(x_n = x_{n+1} - 1) + (1-\alpha)p\,(x_n = x_{n+1} + 1) \tag{3.22}$$

Using eq. 3.22 $p\,(x_{n+1})$ can be determined iteratively starting with the initial distribution $p\,(x_0) = \delta\,(x_0)$.
The first three and the n-th iterations are shown below:

$$\begin{aligned}
p\,(x_1) &= \alpha\delta\,(x_1 - 1) + (1-\alpha)\delta\,(x_1 + 1) \\
p\,(x_2) &= \alpha^2\delta\,(x_2 - 2) + 2\alpha(1-\alpha)\delta\,(x_2) + (1-\alpha)^2\delta\,(x_2 + 2) \\
p\,(x_3) &= \alpha^3\delta\,(x_3 - 3) + 3\alpha^2(1-\alpha)\delta\,(x_3 - 1) \\
&\quad + 3\alpha(1-\alpha)^2\delta\,(x_3 + 1) + (1-\alpha)^3\delta\,(x_3 + 3) \\
&\dots \\
p\,(x_n) &= \sum_{i=0}^{n} \binom{n}{i} \alpha^{n-i}(1-\alpha)^i\delta\,(x_n - n + 2i)
\end{aligned} \tag{3.23}$$

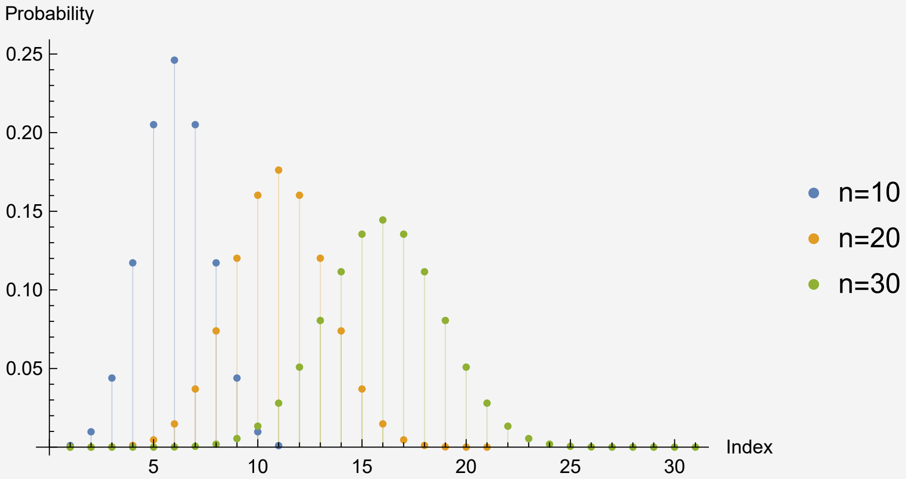
If we assume that $n = 2m$ is an even number, the coefficient of the term $\delta\,(x_n)$ tells us the probability to find the particle at its starting position after n steps. In the limit $m \to \infty$, the probability goes to zero.

We can check this with the following code, where we set α to $\frac{1}{2}$.

```
Limit[Binomial[2*m, m]/4^m, m -> Infinity]
```

If n is an odd number, it is impossible to find the particle at its starting position, because it requires the same number of steps to the right and left.

The next figure shows the probability as a function of the index i after n steps (note that the displacement of the particle is $2i-n$). The maximum is always at displacement 0, but becomes smaller for each increasing n.



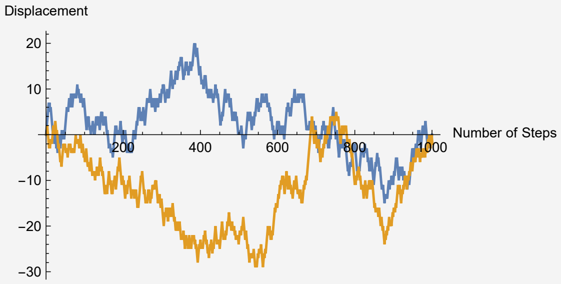
For large n the displacement can be described by a Normal distribution $\mathcal{N}\left(0,\sigma^2\right)$, where σ^2 is proportional to the number of steps n.

In the Wolfram Language a random walk sequence generator can be implemented with the following function

```
RandomWalkSequence[alpha_, n_]:= NestList[# + RandomChoice[{alpha, 1 - alpha}->
                                                {1, -1}] &, 0, n];
```

The function `RandomChoice` selects with probability `alpha` the value $+1$ and with probability `1-alpha` the value -1 and adds it to the last evaluation, which is performed iteratively `n` times using `NestList` starting with 0.
Wolfram Language has already a built-in functions `RandomWalkProcess` and `RandomFunction` for that purpose.

```
data1 = RandomFunction[RandomWalkProcess[0.5], {0, 1000}];
data2 = RandomWalkSequence[0.5, 1000];
ListLinePlot[{data1, data2}]
```

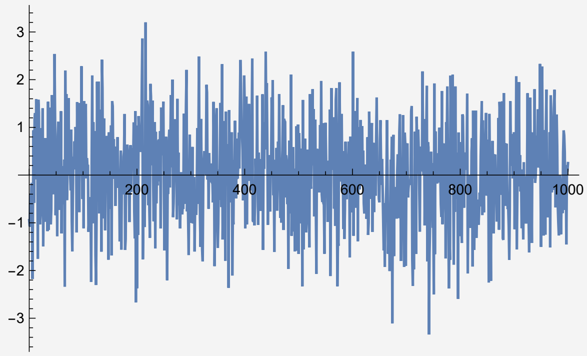


Example 3.6 (White Noise Process) Another important random process is White Noise. The sequence consists simply of samples from a Normal distribution.

$$x_n \sim \mathcal{N}\left(0,\sigma^2\right) \tag{3.24}$$

The built-in Wolfram Language function is `WhiteNoiseProcess`.

```
data = RandomFunction[WhiteNoiseProcess[], {0, 1000}];
ListLinePlot[data]
```



Exercise 3.7 Study the Wiener process, which is given by

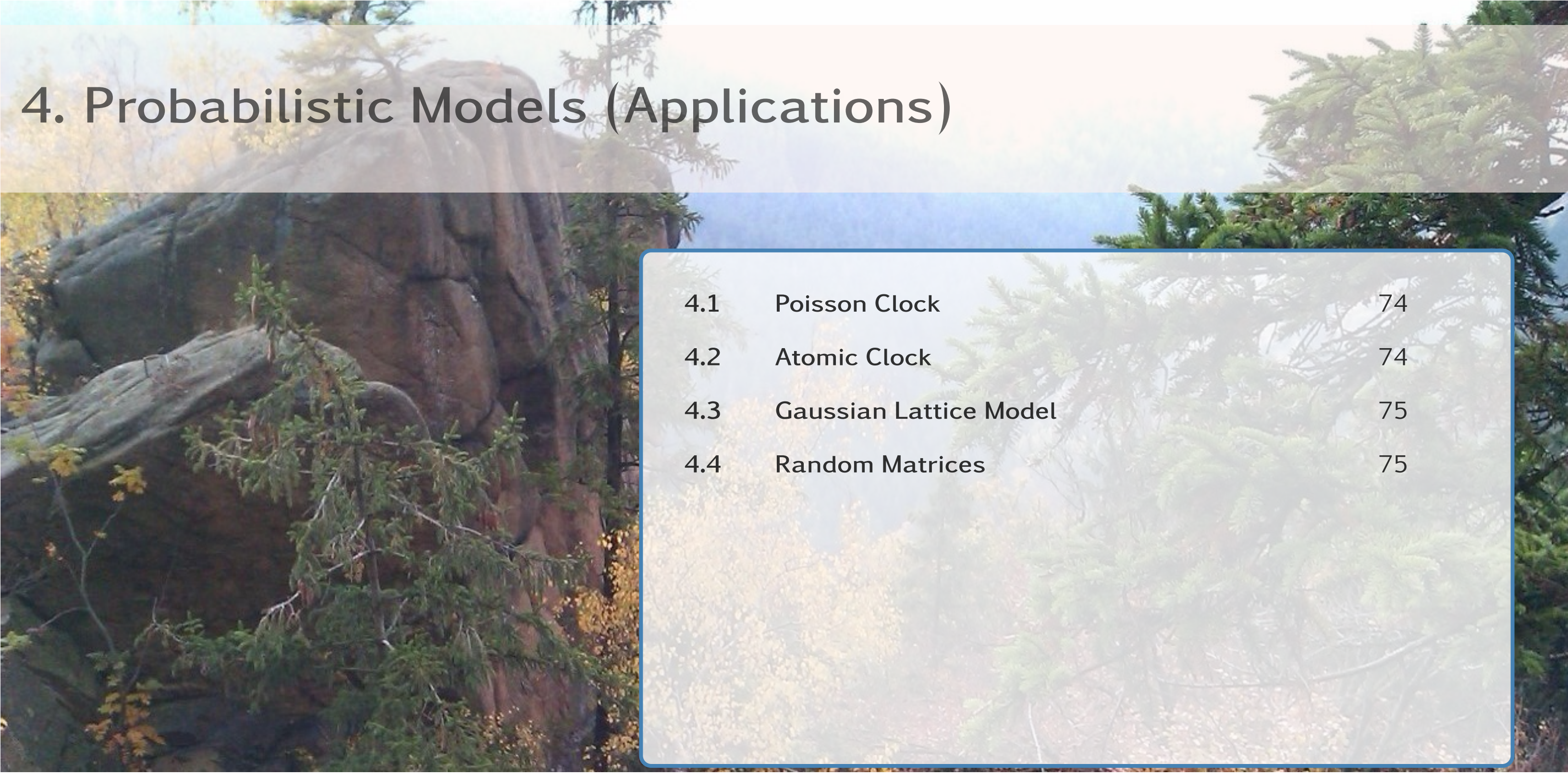
$$x_{n+1} = x_n + z_n \tag{3.25}$$

with $z_n \sim \mathcal{N}\left(\mu, \sigma^2\right)$. Use the function `WienerProcess` to generate paths starting at $x_0 = 0$ and show that for non-zero μ the maximum of the distribution of endpoints after n steps is shifted by a factor of μ.

Exercise 3.8 Revisit the Random Walk process using the equation

$$x_{n+1} = x_n + z_n \tag{3.26}$$

with $z_n \sim \alpha \mathcal{N}\left(+1, \sigma^2\right) + (1-\alpha)\mathcal{N}\left(-1, \sigma^2\right)$. Show that the results from example 3.5 are reproduced in the limit $\sigma^2 \to 0$.



4. Probabilistic Models (Applications)

4.1 Poisson Clock 74

4.2 Atomic Clock 74

4.3 Gaussian Lattice Model 75

4.4 Random Matrices 75

This chapter covers the whole range of probabilistic models from basic to real-world applications. We show that the simple models can be used as building blocks for creating gradually more complex systems, which can then serve as reusable components in an even larger simulation.

4.1 Poisson Clock

A very simplified model of a clock is the Poisson clock, where the tick counts x are Poisson-distributed.

Definition 4.1 (Poisson Clock)

$$\mathcal{P}(x|\lambda \to \nu t) = \sum_{k=0}^{\infty} \frac{\nu^k t^k}{k!} \exp(-\nu t)\delta(x-k) \tag{4.1}$$

ν is the clock rate (number of ticks in a certain time interval) and t is the elapsed time. Because the expectation value is $\mu = \nu\ t$, the tick count is a measure of time.

However, this is not a good model of a real clock, because there is only one parameter ν, which controls the clock rate, and the clock noise is completely determined by this choice. In a real clock model you can control the rate and the noise independently.

Nevertheless, the Poisson clock is used very often for e.g. simulating the arrival of costumers in a shop or the number of decays of a radioactive material. It is also a building block of the Ptolemy II simulation framework [7].

4.2 Atomic Clock

We want to create a parametric generic model of an atomic clock, which is able to generate data indistinguishable from measurement data of a passive hydrogen maser if the parameters are chosen properly. The treatment is based on the article **A mathematical model for the atomic clock error in case of jumps** [20].

An atomic clock is a frequency source, which provides typically the following two outputs: a sinusoidal signal with a nominal frequency of f_0 and one pulse per second signal, which marks the start of second with its rising edge. The 1pps signal is generated from the high-frequency signal by counting the cycles and outputting a pulse when the cycle count reaches f_0. We focus on the high-frequency signal, which is characterized by its amplitude, the frequency deviation from the nominal frequency given by $y(t) = \frac{f(t)-f_0}{f_0}$ and phase time offset given by $x(t) = \frac{\phi(t)}{2\pi f_0}$. For simplicity we ignore amplitude variations and add the frequency drift or ageing $z(t)$ as a third parameter to describe the state of an atomic clock.

The starting point for our analysis is a set of parametric equations, which describe the time evolution of the state $x(t)$, $y(t)$, $z(t)$ of an atomic clock including noise sources and artifacts like phase and frequency jumps. The parameters makes it possible to fit the model to actual phase data from measurements with a real clock.

$$\begin{aligned}\frac{dx(t)}{dt} &= y + w_1 + a_1 n_1 \\ \frac{dy(t)}{dt} &= z + w_2 + a_2 n_2 \\ \frac{dz(t)}{dt} &= w_3 + a_3 n_3\end{aligned} \tag{4.2}$$

with $w_i \sim \mathcal{N}\left(\mu_i, \sigma_i^2\right)$, $n_i \sim \mathcal{P}(\lambda_i)$ and $a_i \sim \mathcal{N}\left(0, \sigma_i^2\right)$.

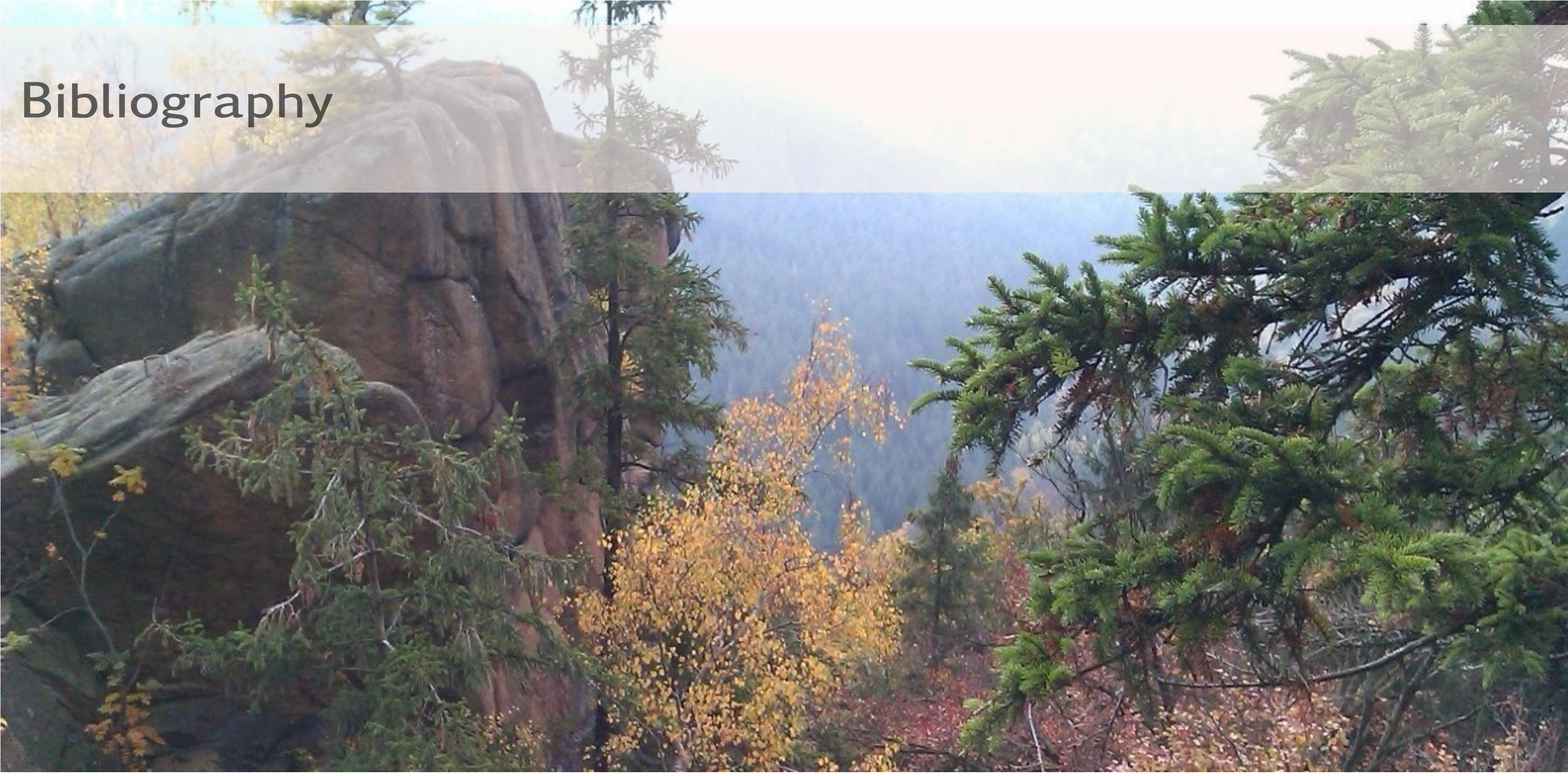
One must consider that it is not possible to measure the phase of a single clock directly, because a phase comparison requires two inputs: the device under test and the reference clock. For simplicity, we assume that the contribution of the reference clock can be ignored, which is the good approximation for the phase data, we will use in this chapter and which comes from a frequency stability measurement between a passive hydrogen maser (the clock, which we are interested in) and an active hydrogen maser (the reference clock, which is more stable than its passive counter part). The measurements were conducted in a thermally-controlled environment, so that the effect of temperature-induced phase variations can be also neglected.

4.3 Gaussian Lattice Model

The Gaussian Lattice Model is a variant of the famous Ising Model [14]. We want to study this model in the one-dimensional case, where it can be solved exactly.

4.4 Random Matrices

A random matrix m is $N \times N$ matrix with its elements $m_{ij} \sim p(x)$ [12]. We want to answer the questions how the spectral properties of the matrix like the distribution of eigenvalues in the limit $N \to \infty$ behaves. We have already seen an example for the case of a symmetric matrix m with $m_{ij} \sim \mathcal{N}(0, 1)$ and have found that the eigenvalues are distributed according to a SemiCircle pdf. We will now generalize the result by studying other distributions.



Bibliography

[1] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2011. URL: http://www.cs.ucl.ac.uk/staff/d.barber/brml.

[2] David Barber and A. Taylan Cemgil. "Graphical Models for Time Series". In: *IEEE Signal Processing Magazine* 27.6 (Nov. 2010), pp. 18–28.

[3] Julius S. Bendat and Allan G. Piersol. *Random Data: Analysis and Measurement Procedures*. Wiley Series in Probability and Statistics. John Wiley & Sons, 2010.

[4] George Edward Pelham Box and Mervin Edgar Muller. "A note on the generation of random normal deviates". In: *Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611.

[5] Glen Cowan. *Statistical data analysis*. Oxford University Press, USA, 1998.

[6] Eugene Don. *Schaum's Outline of Mathematica and the Wolfram Language*. Schaum's Outline Series. McGraw-Hill Education, 2018.

[7] Johan Eker et al. "Taming heterogeneity - the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144.

[8] Fredrik Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur AB, 2012. ISBN: 9144077327.

[9] Kevin J. Hastings. *Introduction to Probability with Mathematica*. CRC Press, 2001.

[10] Wolfram Research Inc. *Mathematica, Version 12.0*. Champaign, IL, 2019. URL: https://www.wolfram.com/mathematica.

[11] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, 2009. ISBN: 9780262013192. URL: https://books.google.co.in/books?id=7dzpHCHzNQ4C.

[12] Giacomo Livan, Marcel Novaes, and Pierpaolo Vivo. *Introduction to Random Matrices - Theory and Practice*. Springer Briefs in Mathematical Physics 26. Springer, 2017.

[13] Oleg Marichev and Michael Trott. *The Ultimate Univariate Probability Distribution Explorer*. Feb. 2013. URL: https://blog.wolfram.com/data/uploads/2013/02/ProbabilityDistributionExplorer.zip.

[14] Giuseppe Mussardo. *Statistical Field Theory. An Introduction to Exactly Solved Models of Statistical Physics*. Oxford University Press, 2010.

[15] John von Neumann. “Various techniques used in connection with random digits. Monte Carlo methods”. In: *Nat. Bureau Standards* 12 (1951), pp. 36–38.

[16] Simon Sirca. *Probability for physicists*. Graduate texts in physics. Berlin: Springer, 2016.

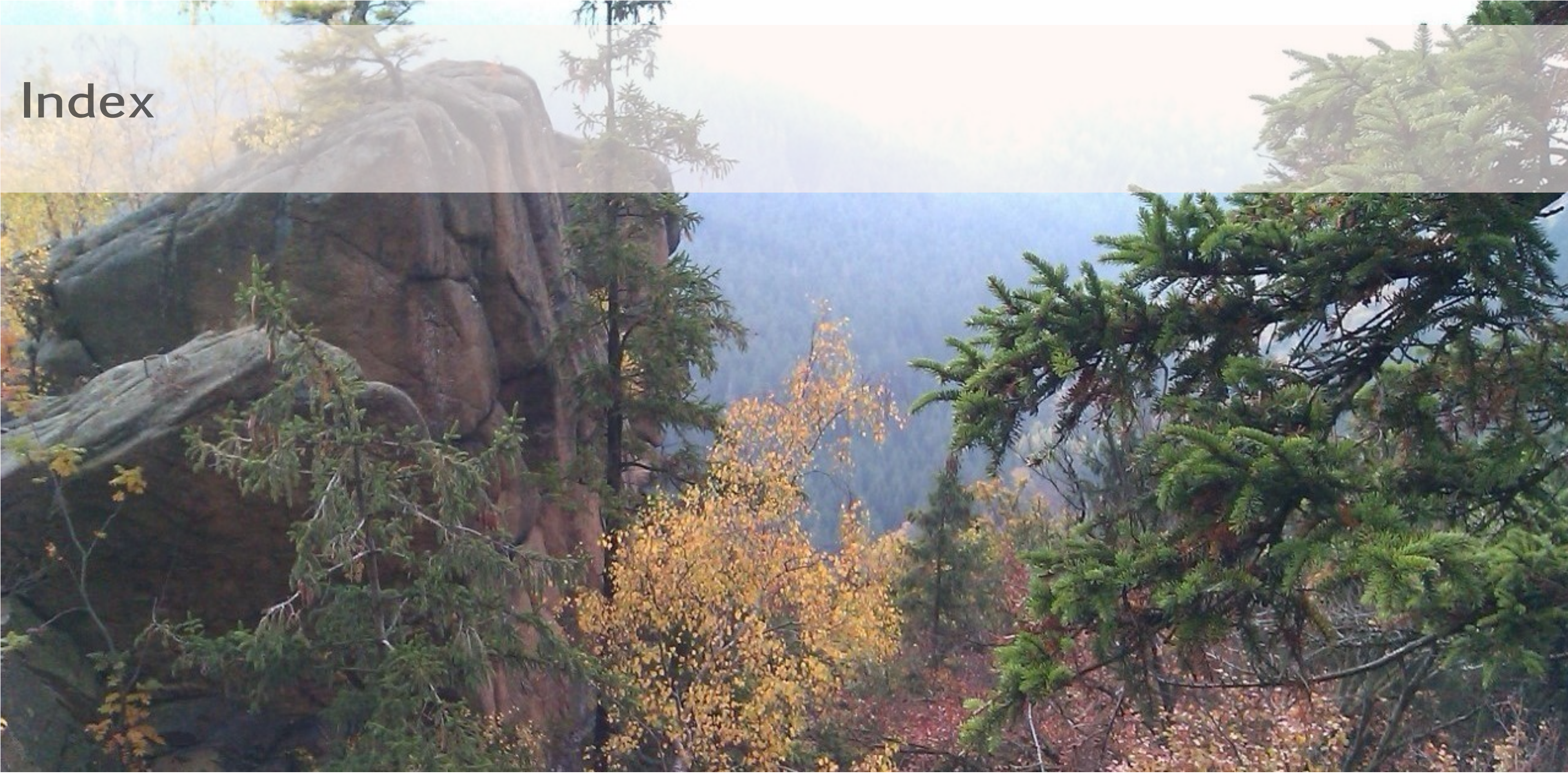
[17] Murray Spiegel. *Schaum’s Outline of Advanced Mathematics for Engineers and Scientists*. Schaum’s Outline Series. McGraw-Hill Education, 2009.

[18] Murray Spiegel. *Schaum’s Outline of Probability and Statistics*. Schaum’s Outline Series. McGraw-Hill Education, 2012.

[19] Stephen Wolfram. *An Elementary Introduction to the Wolfram Language*. Wolfram Media, Inc., 2015.

[20] Cristina Zucca and Patrizia Tavella. “A mathematical model for the atomic clock error in case of jumps”. In: *Metrologia* 52 (June 2015). doi: 10.1088/0026-1394/52/4/514.

Index



bayes filter, 57
belief network, 50
binned kernel density estimator, 36
binomial distribution, 21
birthday problem, 30

central limit theorem, 26
chain graph, 52
conditional probability density, 15
cumulative distribution function, 13

deterministic probability density, 18
differential entropy, 20
discrete probability density, 20

expectation, 25

gaussian kde, 37
gaussian mixture model, 38
graph world, 58

histogram, 33

likelihood, 40
linear transformation, 11
loglikelihood, 40

markov property, 56
mean, 25
mixture density networks, 38
monty hall problem, 28
multinomial distribution, 22

normalizing flows, 39

particle filter, 63
poisson clock, 74
poisson distribution, 22
probability, 13
probability axioms, 13
probability density, 8

random variable, 8
random walk, 71
rejection sampling, 46

sampling, 42
semicircle distribution, 47
sum of two Poisson distributions, 23

time series, 55

variance, 25