Loughborough University of Technology
Department of Electronic and Electrical Engineering
Loughborough LE11 3TU

STATISTICAL REPRESENTATION OF A HYBRID PHOTOVOLTAIC- WIND SYSTEM FOR CONTROLLER DESIGN

by

Joachim Baumgaertner

Supervisor:	Dr. David Infield
Director of Research:	Prof. I. Smith
Date:	29.07.1995

Submitted in partial fulfillment of the requirements for the award of A Master of Philosophy of the Loughborough University of Technology.

CONTENT

1. Introduction

This paper considers an autonomous, terrestrial energy supply plant applying renewable energy sources. It presents a mathematical model whose purpose is to gain an in-depth understanding of the impact of fluctuations of the wind speed and the intensity of the sun on the power supply of such an energy system. Results could then be used to design a controller that operates the system. The system with its four core elements is depicted in Fig. 1.1.

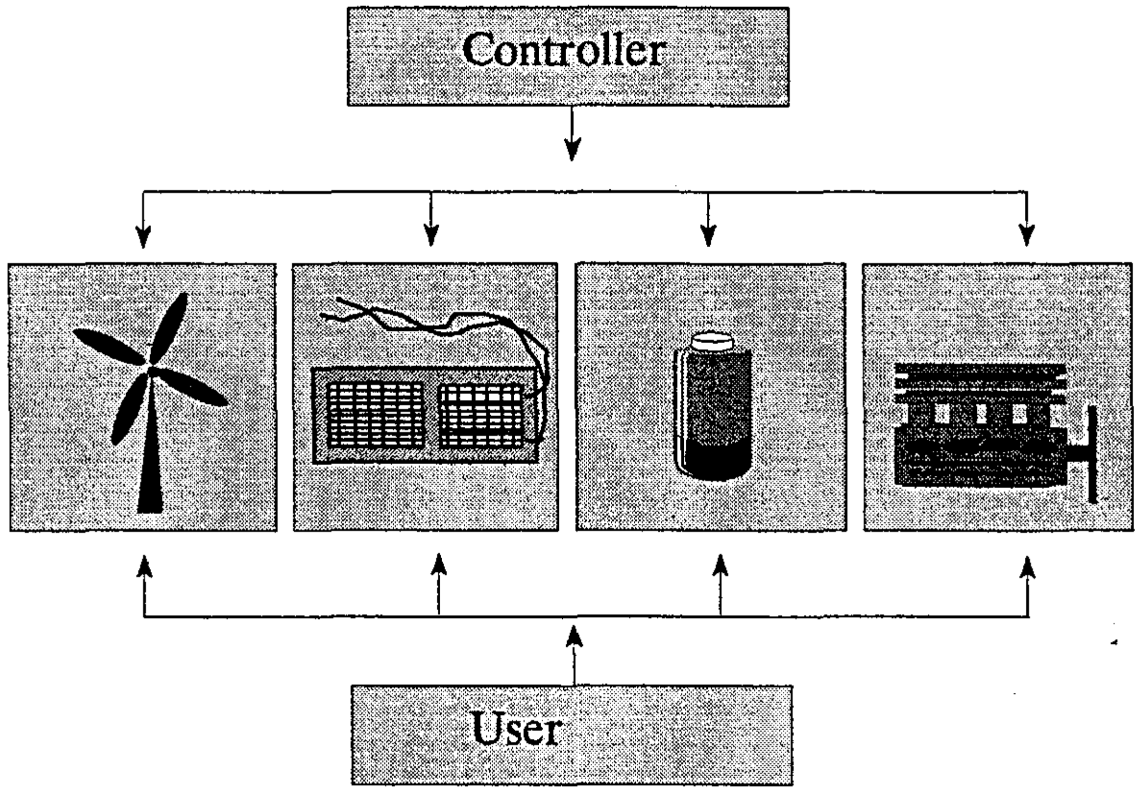

Fig. 1.1: Hybrid Energy System

They are a wind turbine, a photovoltaic array, a battery and a diesel engine. The controller receives data from these components and manages them. The electric energy generated by the system is provided for the user.

Combined Wind- PV- Diesel- systems do mainly compete with Diesel stand-alone systems, Wind- Diesel- systems and the connection to the mains. These island systems are typically designed for a rated power of up to several 10 kW. They are supposed to operate on remote sites where a connection to the mains is not given.

(1) **Diesel Stand- alone systems** are the most common systems for decentral energy supply. Eventhough they are the cheapest option - as far as the investment costs are concerned - they might not be the best. And this is for three reasons. First, a diesel uses an energy source with a limited range. Second, the combustion of crude oil products causes ecological problems. Third, in remote areas the price for fossil fuels might be significantly higher than in urban areas, thus leading to a steep increase of the actual cost of a KWh. Moreover, in remote areas the required regular service might either not be asserted or costly.

(2) **Wind- Diesel- systems** are one option to cut down on the fossil fuel consumption. Since the renewable energy supply (i.e. wind speed) fluctuates considerably, a diesel generator is necessary to ensure high reliability. As high wind speeds and high solar insolation are often complementary, it is supposed that the photovoltaic array may fill in the gap when the wind turbine does not produce enough energy and vice versa, thus justifying the additional investment of the photovoltaic array.

(3) **Connection to the national grid,** which is fed by conventional power plants. This option has to be ruled out for many a site such as islands far away from the mainland. Where possible at all however, the investment of the connection is likely to be fairly expensive as the costs for it increase with decreasing population density. Moreover, centrally fed mains with a large area extension are susceptible to faults.

Fig. 1.2 shows the system in more detail. It consists of a wind turbine and a photovoltaic array as the renewable energy sources, a battery as an energy storage unit and a fossil fuel generator (diesel engine) for backup in order to guarantee a power supply at all times. The battery is supposed to fill in short- term gaps in the energy supply by the renewable sources, thus smoothing the power supply function and reducing the number of diesel starts. Depending on the load that has to be supplied, the load might be directly connected to the DC- Bus or via a DC/AC- converter.

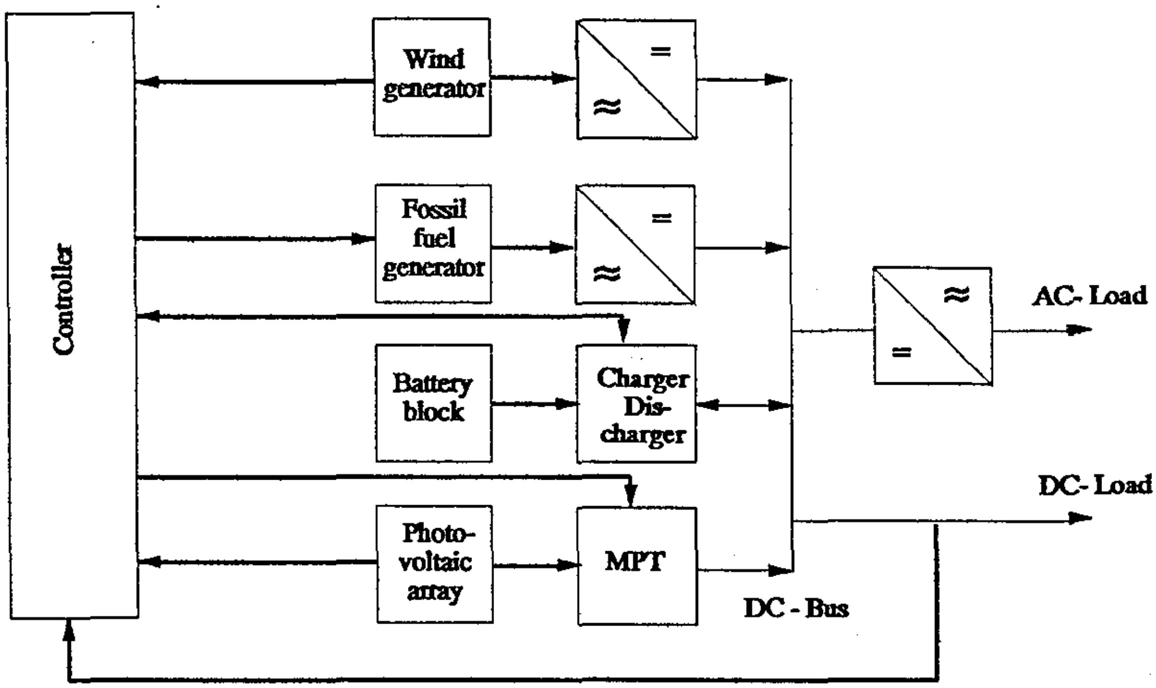

Fig. 1.2: Autonomous Wind- PV- System

Since both wind speed and solar intensity do vary considerably, the power supplied by the renewable energy sources will vary too. Therefore, the general problem in the performance of renewable energy systems is the matching of energy production and load. As far as the energy producing components, the PV array, the wind generator, the diesel and the battery, are concerned, it is assumed that standard components are used, thus restricting the controller to the interaction within the ensemble. The controller will therefore be in charge of the charging and discharging of the battery, the start- stop- policy for the fossil fuel generator, the maximum power tracking for the Photovoltaic array and its positioning. It is furthermore conceivable to switch on additional loads if there is a surplus energy in order to reduce the amount of dumped energy. These additional loads could produce storable goods as drinking or hot water. To assist the controller in its management data will be fed in from all components in regular time intervals. Hence, it will be informed of the current wind speed, current intensity of the sun, state of charge of the battery and the load demand.

The purpose of this paper is to provide a mathematical model that reflects this scenario and is able to support the controller in its decision making. The focus of this model is the mathematical formulation of the stochastic processes "wind speed" and "solar intensity". They can be transformed by applying simple models for the wind turbine and the photovoltaic array into the stochastic processes "wind turbine power" and "solar power". These algorithms allow to calculate time series, resulting in a short term prediction of the power supply, delivering data that can be used by the controller to decide on the best policy in order to minimize the operational costs of the system. The point that should be stressed here is that this model is a short term model which allows to plan ahead over time periods of the order of up to one hour by using hourly data from various sensors. This is supposed to enable the controller to operate the system in an efficient way. For the best sizing of the components, however, it is necessary to consider meteorological data of the site in question over a longer period.

Physical aspects of the energy sources which the model is based on are discussed in chapter 2, followed by the discussion of the energy converters (i.e. wind turbine, photovoltaic array, battery and diesel) in chapter 3. The statistical methods are then taken further in chapter 4. It will focus on the probability distribution of the power supplied by the renewable energy sources, followed by a section on the generation of synthetic time series of the power supply, including both renewable energy sources and the battery. The last section of this chapter discusses first passage time problems. The first passage time is the expected time when the power surpasses a certain passage level for the first time. This is useful for instance in the event that the renewable energy sources do not provide enough energy to meet the demand. If it is expected that this will be the case for a longer time period it might be worth switching on the diesel. If not, the power might as well be supplied by the battery in order to avoid switching the diesel on and off too often. Here, the first passage time provides useful information. Chapter 5, eventually, gives a summary by restating the main points.

The algorithms presented in this paper have been coded in C++ for a Windows 3.1 environment using the Borland C++ 3.1 compiler and the Borland Object Windows C++ 1.0 library. The relevant graphs in this paper have been created using Word Perfect Presentation to which a data interface is provided by the program. The mostly interactive program is described in the Appendix II, where a complete class reference and a description of global functions are given.

2. Energy Sources

2.1 Wind Energy

2.1.1 Wind Speed Power Spectrum - Empirical Results

The spectral density function of the horizontal wind speed is largely dependant on the location where the speed was monitored. The characteristics of different sites, however, reveal distinctive similarities. A generic spectrum ([19]) is shown in Fig. 2.1.

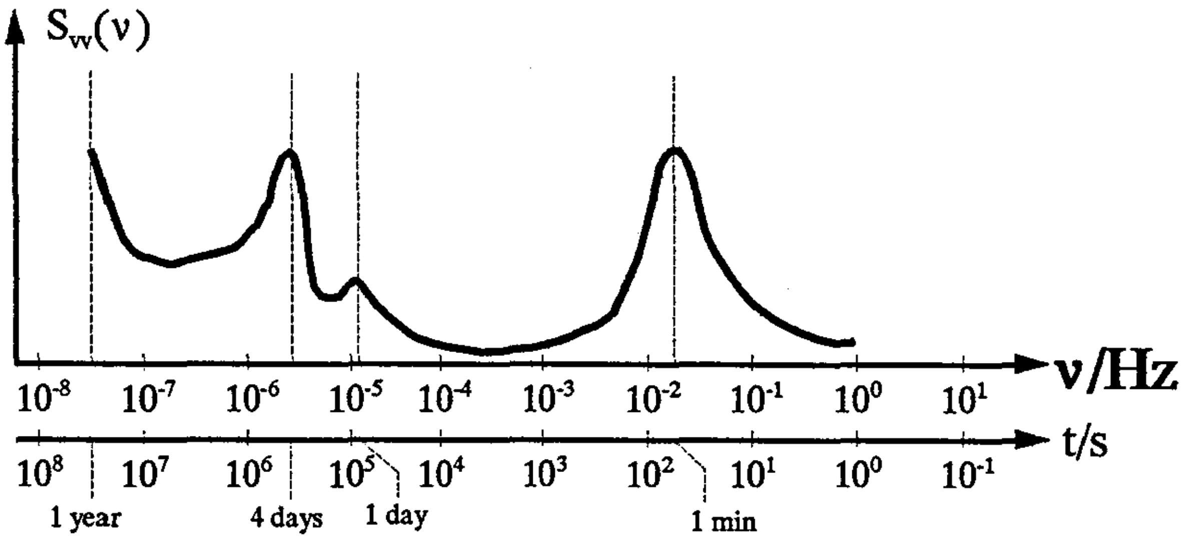

Fig. 2.1: Generic Wind Speed Spectrum

(1) Micrometeorological range

The peak in the high frequency range is caused by fluctuations called atmospheric turbulence. The energy of the fluctuations is centered around a period of around 1 minute. They can be approximated by the Ornstein- Uhlenbeck process ([25]), a stochastic model process. The micrometeorological range will be discussed in more detail detail in 2.1.2.

(2) Spectral gap

A striking phenomenon of a typical wind speed spectrum is a spectral gap between time periods of 10 minutes and 2 hours ([19]).

(3) Macrometeorological range

Large- scale movements of air masses account for three peaks on the macrometeorological side of the spectrum. The relative maximum at a diurnal time period is due to different temperature gradients at day and night. This effect is likely to be more distinctive at coastal sites as the air temperature on shore decreases more rapidly during night time than off shore. Depressions and anti- cyclones usually occur with periods of about four days which explains the second maximum of the spectrum. Again the pattern here is that the peak will be more distinctive in oceanic climates rather than continental. The peak at the one- year period in contrast is likely to vary with the degree of latitude. It will vanish at sites in close proximity to the equator. Some aspects of the macrometeorological range will be discussed in more detail in chapter 2.1.3

The peak in the micrometeorological range allows a short term prediction of the wind speed. Here, "short term" indicates time periods that fall into the spectral gap, i.e. between 10 minutes and one hour. Within this short term model a constant average hourly wind speed and standard deviation are assumed. These macrometeorological, hourly data can be derived from measured data. So far, what is said here, only applies to the wind speed distribution.
In chapter 2.2.4 it will be shown, however, that the solar power spectrum too, can be seperated into a short term and a long term range. Hence, it will follow the same pattern: For short term considerations a statistical model will be used, whereas hourly values for the beam intensity are taken from a data feeder. Usually, the data feeder will hold current data. For optimization purposes, however, it could as well hold historical data taken from a specific site over a week or a month.

2.1.2 Turbulence: The Micrometeorological Range

2.1.2.1 Definitions

Turbulence includes all fluctuations with frequencies higher than the quasi- steady mean wind speed variation. If we assume the mean wind speed to be constant over a sufficiently short time period, $\bar{v}(t) = \bar{v}$, the wind speed of the fluctuation will be defined by ([19], 2.15)

$$v_f(t) = v(t) - \bar{v} \quad , \tag{2.1}$$

the difference between the instantaneous wind speed v(t) and the mean wind speed $\bar{v}$. The variance of the turbulence will then be

$$Var(V) = \int_{-\infty}^{\infty} (v-\bar{v})^2 f_v(v)\,\mathrm{d}v \tag{2.2}$$

where $f_v(v)$ is the probability density function with respect to the wind speed v. The index v signals that V is the random variable. It is worth noting that the argument of the variance operator in (2.2) is capital V. Throughout this paper random variables will be referred to by capital letters, their realizations by small ones[1]. Given n realizations of the instantenous speed, v_j (j=1..n), the empirical variance of the turbulence can be estimated from

$$\sigma_v^2 = \frac{1}{n-1}\sum_{j=1}^{n}(v_j - \bar{v})^2 \tag{2.3}$$

The turbulence intensity is defined as the quotient ([19], 2.17)

$$I_v = \frac{\sigma_v}{\bar{v}} \tag{2.4}$$

2.1.2.2 Turbulence and the Ornstein- Uhlenbeck Process

Wind fluctuations over a restricted time interval can be represented by the Ornstein-Uhlenbeck process, which also describes the velocity of free particles in Brownian motion. The random variable related to the velocity will be called V. In order to condense and simplify the formulas involved let us introduce the normalizations of the time axis,

---

[1]Refer to chapter 6 for further discussion of random variables and distribution functions.

$$\tau = \beta\, t \tag{2.5}$$

with the time constant β_v, and the normalization of v,

$$\xi(t) = \frac{v(t) - \bar{v}}{\sigma} \tag{2.6}$$

with the deviation σ. Both parameters τ and ξ are thus dimensionless and their significance will prove to be self- explanatory after the following remarks. The random variable that stands for the normalized process will be Ξ. It is beyond the scope of this paper to elaborate on the physical details of the Ornstein- Uhlenbeck process. The O.U. - process is a continuous time Markov process whose probability density function $\varrho(\xi,\tau)$ has to satisfy the Fokker- Planck equation, which has the form

$$\frac{\partial \varrho(\xi,\tau)}{\partial \tau} = \frac{\partial^2 \varrho(\xi,\tau)}{\partial \xi^2} + \frac{\partial}{\partial \xi}\left[\, \xi \varrho(\xi,\tau) \,\right] \tag{2.7}$$

in the special case of the O.U. - process. The value $\varrho(\xi,\tau)d\xi$ is the probability that, at time τ, the wind speed lies in the interval $[\xi,\xi+d\xi]$ subjected to an initial condition $\varrho(\xi,0) = h(\xi)$ at time $\tau = 0$. A solution will be given later.

It may be noted that a discrete realization of an Ornstein- Uhlenbeck process is the Ehrenfest model of diffusion ([14], p.343), which can be interpreted as a diffusion with a central force. That is a random walk in which the probability of a step in one direction varies with the position.

(i) Power Spectrum and Autocorrelation Function

The power spectrum of the Ornstein- Uhlenbeck process as a function of the angular frequency ω,

$$S_{\xi\xi}(\omega) = \frac{2}{\omega^2 + \beta^2} \tag{2.8}$$

is Lorenzian with the corresponding autocorrelation function[2]

$$R_{\xi\xi}(\tau) = \exp(-|\tau|) \tag{2.9}$$

Please bear in mind that τ in (2.9) is normalized via (2.5). In the frame of the description of wind turbulence it is sometimes referred to as Dryden spectrum. For the sake of simplicity we will usually refer to the autocorrelation function (2.9) via the short hand $r = R_{\xi\xi}(\tau)$ or in its unnormalized form $r_v = R_{\xi\xi}(\beta_v t)$.

(ii) The Probability Density Function

The probability density function $\varrho(\xi,\tau)$ is the solution of the Fokker- Planck equation (2.7). In this section we assume boundary conditions to satisfiy $\varrho(\infty,\tau) = \varrho(-\infty,\tau) = 0$. These are two physically sensible conditions to avoid infinite wind speeds. In the first step the special initial condition $\varrho(\xi,0) = \delta(\xi - \xi_0)$ is considered. In this case, $\varrho(\xi,\tau) = \varrho(\xi,\tau;\xi_0)$, is the probability density under the condition that a wind speed ξ_0 has been observed at time $\tau = 0$. The solution is ([20], eq.3.40) given by

$$\varrho(\xi,\tau;\xi_0) = \frac{1}{\sqrt{2\pi(1-r^2)}}\exp\left[-\frac{1}{2}\frac{(\xi-\xi_0 r)^2}{1-r^2}\right] \tag{2.10}$$

This is identical to the probability density function of a bivariate standard normal probability density function with correlation coefficient r (compare with equation 6.23). In fact, $\varrho(\xi,\tau;\xi_0)$ can be thought of as a Gaussian curve whose peak wanders with τ towards $\xi = 0$ while becoming broader. Other methods of solving the Fokker- Planck equation are discussed for example in [34]. Actually, (2.10) can be interpreted as Green's function of the given boundary problem. Consequently, the probability density function for any initial condition $\varrho(\xi,0) = h(\xi)$ can be obtained by convoluting Green's function with the initial condition:

$$\varrho(\xi,\tau) = \langle \varrho(\xi,\tau;\xi_0) \mid h(\xi) \rangle = \int_{-\infty}^{\infty} \varrho(\xi,\tau;\xi_0)\, h(\xi_0)\, d\xi_0 \tag{2.11}$$

[2]Refer to chapter 6 for a discussion of the relationship between autocorrelation function and power spectrum of a stochastic process.

Equation (2.11) is actually generally valid: Green's function gives the solution of a boundary value problem for the special initial condition $h(\xi) = \delta(\xi - \xi_0)$. The system response for another initial condition can then easily evaluated via the convolution integral. Hence, Green's function depends on both the partial differential equation and the boundary values. It is worth pointing out that are different types of Green functions, depending on the type of differential equation and on the formulation of the boundary conditions, thus restricting the generality of (2.11). In this paper, however, we only come across the type described above.

We might as well expand $\varrho(\xi,\tau;\xi_0)$ as (using a generating formula in [26], p.252)

$$G_1(\xi,\xi_0,\tau) = \frac{1}{\sqrt{2\pi}} \sum_{n=0}^{\infty} \left[\frac{H_n\left(\frac{\xi_0}{\sqrt{2}}\right) H_n\left(\frac{\xi}{\sqrt{2}}\right)}{2^n n!} e^{-n\tau} e^{-\frac{\xi^2}{2}} \right] \tag{2.12}$$

where H_n is the Hermitian polynom ([26], p. 249). The dependencies revealed by this formula are characteristic for diffusion processes: The time τ appears as a linear term in the exponent, a fact that makes clear that the process is irreversible, as it does not produce the same values for negative times. In contrast, solutions of the well known wave equation, where a second time derivative occurs, are invariant under time reversal.

(iii) Equilibrium Distribution

The equilibrium distribution,

$$\varrho(\xi) = \lim_{\tau\to\infty} \varrho(\xi,\tau;\xi_0) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\xi^2\right) \tag{2.13}$$

is simply the standard normal distribution (equation 6.20). Bearing the normalization in mind we conclude that the stationary process V is normally distributed with variance σ^2 and mean wind speed $\bar{v}$. If $\Phi(x)$ denotes the Gaussian distribution function (equation 6.20) the underlying distribution function is simply $F_\xi(\xi) = \Phi(\xi)$. Hence, the expected time fraction τ_{ex} when the wind speed $\xi(\tau)$ exceeds a given value ξ_{ex} can be determined by

$$\tau_{ex} = p(\Xi > \xi_{ex}) = \Phi(-\xi_{ex}) \tag{2.14}$$

where p stands for "probability for".

(iv) Level Crossing

The level crossing analysis of the O.U.- process gives an answer to the question of how frequently a stochastic process crosses a given level. The situation is illustrated in Fig. 2.2 for the normalized process Ξ.

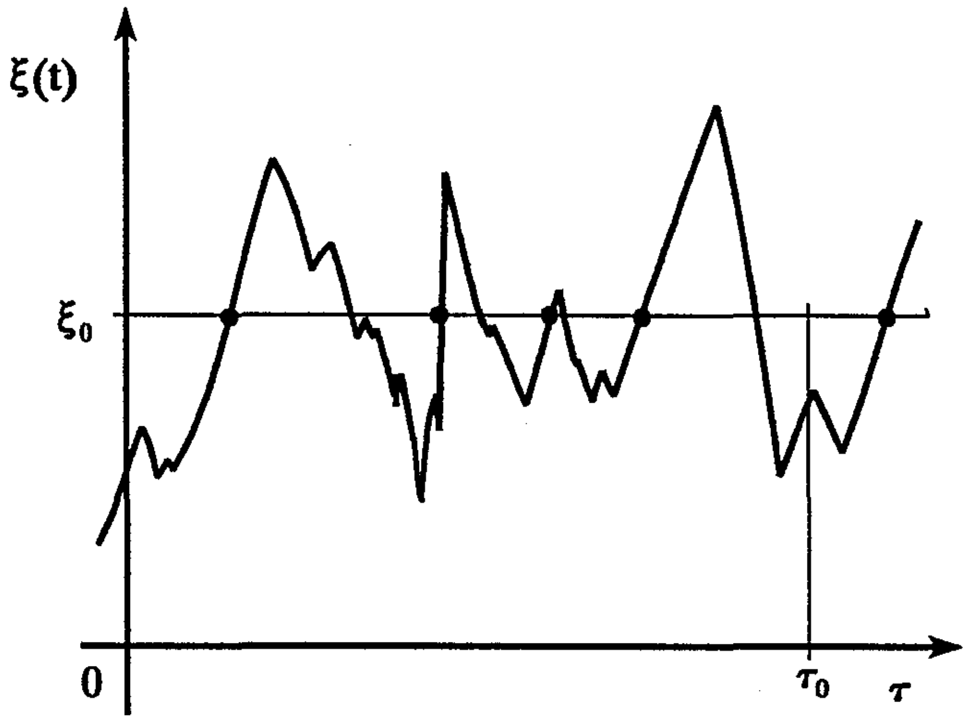

Fig. 2.2 Level Crossing

We will for the moment set $\xi_r = 0$, thus reducing the problem to a zero crossing problem. The probability $p_0(\tau_0)$ that the zero level will be crossed by the process Ξ in the time interval $\tau \in [0,\tau_0]$ at least once when only crossings from negative to positive values count (dots in Fig. 2.2), is equal to

$$p_0(\tau) = \frac{1}{2}p(Z(\tau) < 0) \quad , Z(\tau) = \frac{\Xi(0)}{\Xi(\tau)} \tag{2.15}$$

In (2.15) the random variable Z could as well be the product $Z(\tau) = \Xi(0)\Xi(\tau)$ as it is only the change in sign of Ξ from time 0 to τ which is of interest here. We prefer the quotient as in (2.15) since the necessary integration (compare with equations 6.12) is straightforward. The factor ½ in front of p_0 stems from the fact that only a half of the crossings are from a state below to a state above ξ_r. The distribution function $F_z(z)$ of the quotient Z of two normal processes is given by ([30], eq. 6.46)

$$F_z(z) = \frac{1}{2} + \frac{1}{\pi}\arctan\frac{z-r}{\sqrt{1-r^2}} \tag{2.16}$$

with autocorrelation coefficient r (2.9) , thus resulting in a zero crossing probability (now writing τ instead of τ_0)

$$p_0(\tau) = \frac{1}{2}F(0) = \frac{1}{2\pi}\arccos(r(\tau)) \tag{2.17}$$

Extending the theory to any ξ_r the crossing probability will be ([30], 11.119)

$$p_\xi(\tau) = p_0(\tau)\exp\left(-\frac{\xi_r^2}{2}\right) \tag{2.18}$$

Different approaches are presented in [30] (p. 345) and [25] (p. 346) reaching at the same results.

(v) Linear Prediction

Linear prediction gives an estimate for a future value $\xi(\tau + \lambda)$ of the O.U. - process, represented by the random variable Ξ, as a multiple of the instantaneous value $\xi(\tau)$. The estimator can be obtained by evaluating the Yule- Walker- equations ([30], eq. 13.6) and it is

$$\hat{\xi}(\tau+\lambda) = e^{-\lambda}\xi(\tau) \tag{2.19}$$

where $\hat{\xi}$ denotes the estimator of ξ. This reflects the fact that the process drifts towards the mean value at a rate proportional to the distance from the mean. Although it is a very simple method of prediction it will not be used in this paper as it can not be applied to time series or first passage times.

(vi) First Passage Time Problem

Suppose we want to determine the expected time $\bar{\tau}_1$ the O.U. - process needs to reach the state ξ_1 from the initial state ξ_0 at $\tau = 0$. The situation, which is called a first passage time problem, is illustrated in Fig. 2.3.

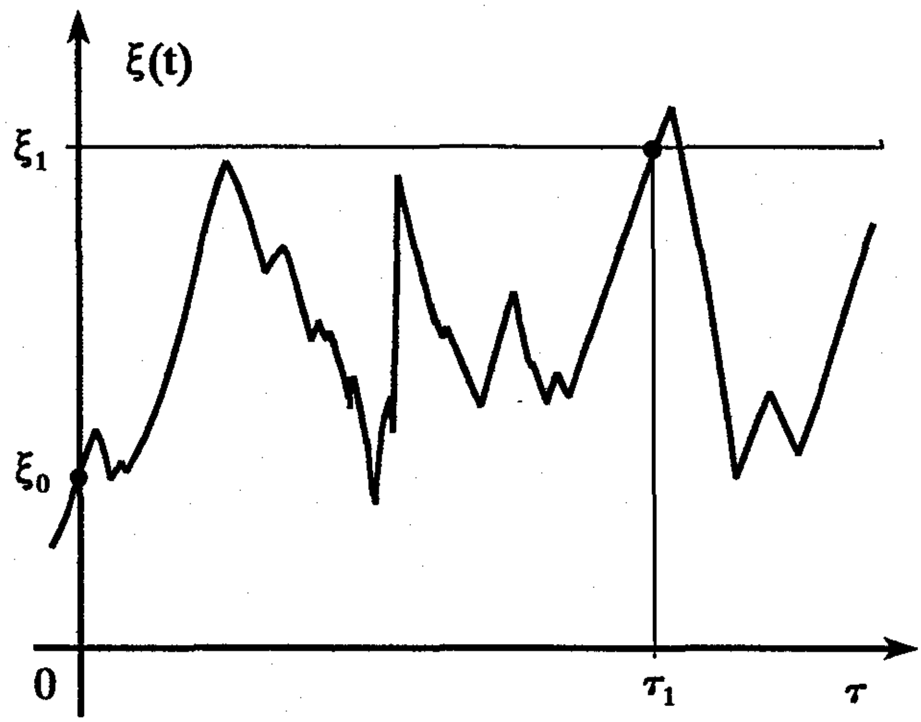

Fig. 2.3 First Passage Time Problem

Until further notice we will assume $\xi_1 > \xi_0$. Mathematically speaking we wish to calculate the distribution function $F_{T1}(\tau)$ of the random variable T_1 that stands for the time the process crosses the line ξ_1 for the first time after having started at level ξ_0. $F_{T1}(\tau)$ can be expressed by the conditional probability

$$F_{T1}(\tau) = p(T_1 \leq \tau) = p(\xi,\xi_0,\tau \;|\Xi(t) < \xi_1 \;\forall\; t \in (0,\tau)\,) \tag{2.20}$$

This problem can be solved in a very efficient way by examining the diffusion process in the half space. Here, one boundary condition will be $\varrho(\xi_1, \tau) = 0$, whereas the other remains in the infinite space, $\varrho(-\infty, \tau) = 0$. Hence, the boundary ξ_1 acts as an absorbing wall. Particles reaching the ξ_1 - level for the first time will be removed and will not appear anymore in the half space $\xi < \xi_1$. Green's function of this boundary problem is given by

$$G_2(\xi,\xi_0,\tau) = \frac{1}{\sqrt{2\pi(1-r^2)}}\left[\exp\left(-\frac{1}{2}\frac{(\xi-\xi_0 r)^2}{1-r^2}\right)-\exp\left(-\frac{1}{2}\frac{(\xi-(2\xi_1-\xi_0)r)^2}{1-r^2}\right)\right] \qquad (2.21)$$

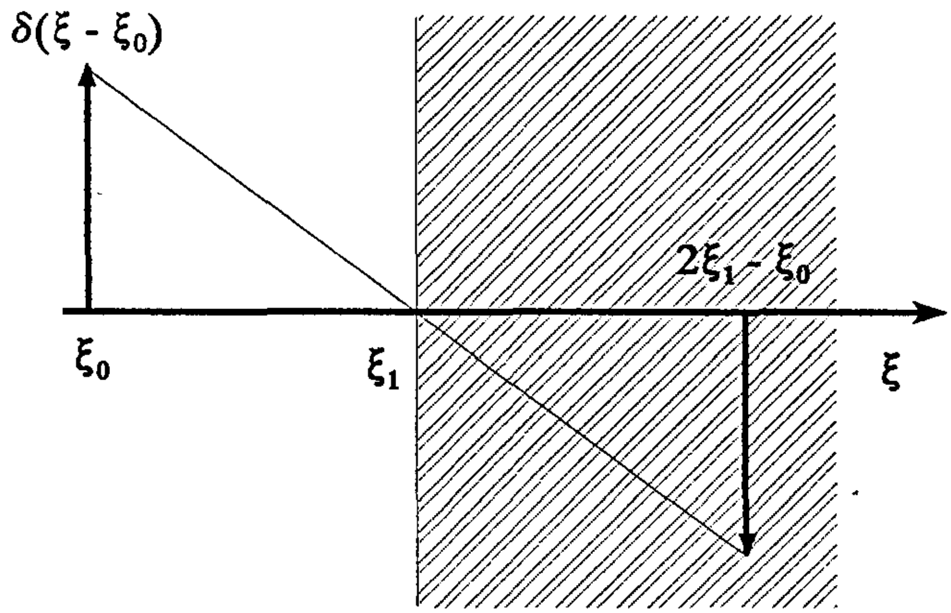

Fig. 2.4 Diffusion in the Half Space

The solution can be interpreted as the diffusion of two fields, punctual symmetric to the boundary $\xi = \xi_1$,where they compensate each other. (This way of calculating the first passage time has been applied to the Brownian process in [20] (p. 447)). This is illustrated in Fig. 2.4. In the space $\xi < \xi_1$ is the original field while the sub space $\xi > \xi_1$ is occupied by the imaginary field. As in (2.11) the solution for the special initial condition $\varrho(\xi,0) = \delta(\xi - \xi_0)$ is $\varrho(\xi,\tau) = G_2(\xi,\xi_0,\tau)$. The number of particles left in the ensemble at time τ can consequently be obtained via integration

$$N(\xi_1,\tau) = \int_{-\infty}^{\xi_1} \varrho(\xi,\tau)\, d\xi = \Phi\left(\frac{\xi_1 - \xi_0 r}{\sqrt{1-r^2}}\right) - \Phi\left(\frac{\xi_1 - (2\xi_1 - \xi_0) r}{\sqrt{1-r^2}}\right) \tag{2.22}$$

The distribution function in question, $F_{T1}(\tau)$, will then of course be

$$F_{T1}(\tau) = 1 - N(\xi_0,\tau) \tag{2.23}$$

Applying (2.21) we obtain the distribution function of T_1,

$$F_{T1}(\tau) = \Phi\left(\frac{\xi_0 r - \xi_1}{\sqrt{1-r^2}}\right) + \Phi\left(\frac{\xi_1 - (2\xi_1 - \xi_0) r}{\sqrt{1-r^2}}\right) \tag{2.24}$$

which conveys the limits $F_{T1}(0) = \delta_{\xi_0\, \xi_1}$ (δ denotes the Kronecker symbol) and $F_{T1}(\infty) = 1$, as it has to be. The density function $f_{T1}(\tau)$ with respect to τ will be attained via the time derivative, and it is

$$f_{T1}(\tau) = \frac{r}{\sqrt{2\pi}} \frac{1}{(1-r^2)^{\frac{3}{2}}} \left[(\xi_1 r - \xi_0) \exp\left(-\frac{1}{2} \frac{(\xi_0 r - \xi_1)^2}{1-r^2}\right) + (2\xi_1 - \xi_0 - \xi_1 r) \exp\left(-\frac{1}{2} \frac{(\xi_1 - (2\xi_1 - \xi_0) r)^2}{1-r^2}\right) \right] \tag{2.25}$$

The expected transition time (average mean time for the process to get from ξ_0 to ξ_1 for $\xi_1 > \xi_0$ is

$$E[T] = \int_0^\infty t\, f_T(t)\, dt \tag{2.26}$$

Looking at $f_T(\tau)$ it is obvious that the expected time exists as the integral (2.26) converges.

To simplify the numerical evaluation the substitution $r(\tau) = \exp(-\beta_v \tau)$ helps to extract the representation

$$E[T] = \frac{-1}{\sqrt{2\pi}} \left\{ \lim_{\epsilon \to 0} \int_{\epsilon}^{1} \frac{\ln(r)}{(1-r^2)^{\frac{3}{2}}} \left[(\xi_1 r - \xi_0) \exp\left(-\frac{1}{2} \frac{(\xi_0 r - \xi_1)^2}{1-r^2} \right) + (2\xi_1 - \xi_0 - \xi_1 r) \exp\left(-\frac{1}{2} \frac{(\xi_1 - (2\xi_1 - \xi_0) r)^2}{1-r^2} \right) \right] dr \right\} \tag{2.27}$$

The emergence of the small value ϵ is necessary as the integral is an improper one. It reminds one that the above proposed substitution is not permitted at the singularity $r = 0$. The results for $\xi_0 > \xi_1$ are dual to the above results as the same Green function holds true. The number of particles left in the ensemble is accordingly

$$N(\xi_1, \tau) = \int_{\xi_1}^{\infty} \varrho(\xi, \tau)\, d\xi = -\int_{-\infty}^{\xi_1} \varrho(\xi, \tau)\, d\xi \tag{2.28}$$

In analogy to the first case we denote the random variable that stands for the transition time with T_2. Its distribution function is

$$F_{T2}(\tau) = 2 - F_{T1}(\tau) \tag{2.29}$$

and therefore the expected value $E[T_2] = -E[T_1]$. It is actually not only formally necessary to split up in two parts depending on the sign of $(\xi_1 - \xi_0)$. The physical background of this is that diffusion processes are not time reversible. This finds its expression in the time derivative of only first order in the Fokker- Planck equation. In the case of wind speeds the very result was expected anyway. Suppose the wind speeds ξ_0 and ξ_1 are both positive. The equations developed here now say that it takes longer (on average) to get from a smaller ξ to a bigger one than in the opposite direction. Summarizing the results in a closed representation we can note the expected average transition time from ξ_0 to ξ_1

$$E[T] = \operatorname{sign}(\xi_1 - \xi_0)\, E[T_1] \tag{2.30}$$

by applying the well-known signum function.

A different approach to the expected average transition time has been carried out in [32] where Markov chains were used to determine the expected value. The technique described above is only applicable if the random variable is normal distributed, which is true in the case of wind speed fluctuations. For other distributions this method seems not to be feasible. The Markov- Chain- technique on the other hand is more general and adaptable to any type of distribution. We benefit from this in chapter 4.3 where two calculation techniques are presented, which are generally valid. As far as the wind speed distribution is concerned, however, the evaluation of integral (2.27) promises to be more efficient than the Markov-chain- algorithm. It can, however, not be extended to the wind turbine power. The analytical approach is therefore not further pursued.

(vii) Two Sided Boundary Value Problem

Suppose we want to calculate the mean time $\tau_b = E[T_b]$ the O.U.- process Ξ will stay within the boundaries $\xi_1 < \xi_0 < \xi_2$ starting at ξ_0 at $\tau = 0$. The random variable that represents the time the process lasts within the band is denoted T_b.

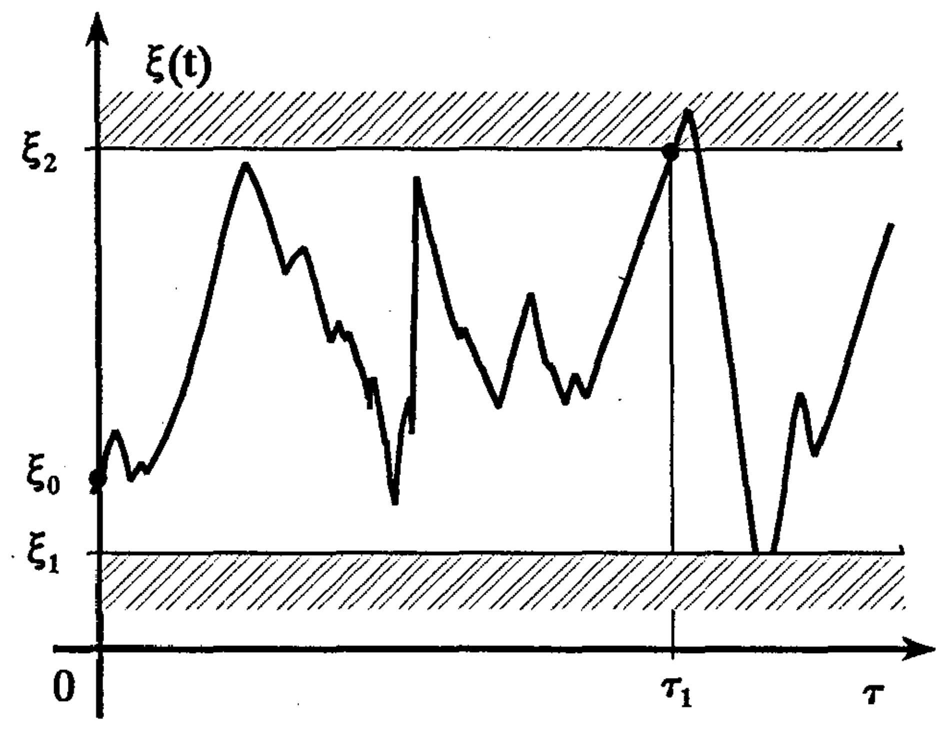

Fig. 2.5 Two Sided First Passage Time Problem

The situation is shown in Fig. 2.5. Formally we can take the same way as before, assuming now two boundary conditions, $\varrho(\xi_1,\tau) = 0$ and $\varrho(\xi_2,\tau) = 0$, and the initial condition $\varrho(\xi,0) = \delta(\xi - \xi_0)$. Again, the problem will be solved by Green's function, $G_3(\xi,\xi_0,\tau)$. The expected transition time can then be computed by applying the same method as before. Green's function G_3 however cannot be obtained as easily as in the case of a diffusion in the half space. The two- boundary values problem results in a discrete eigenvalue spectrum and Green's function is to be expected of the form

$$G_3(\xi,\xi_0,\tau) = \frac{2}{\xi_2 - \xi_1} \sum_{n=1}^{\infty} \sin\left(n\pi \frac{\xi - \xi_1}{\xi_2 - \xi_1}\right) \sin\left(n\pi \frac{\xi_0 - \xi_1}{\xi_2 - \xi_1}\right) \exp[-\tau \Theta_n(\xi,\xi_0,\tau)] \tag{2.31}$$

This statement satisfies both boundary conditions and the initial condition which can be easily verified by bearing the completeness of the sine- function

$$2 \sum_{n=1}^{\infty} \sin(n\pi\zeta) \sin(n\pi\zeta') = \delta(\zeta - \zeta') \tag{2.32}$$

in the interval $\zeta \in (0,1)$ (n integer) in mind. Obviously, this is not an efficient method of calculating the expected time. The methods discussed in chapter 4.3, however, can be easily adapted to this problem.

2.1.2.3 The Kaimal Spectrum

Empirical results show that the Kaimal spectrum ([25], eq. 16.15)

$$S_{KI}(\omega) = \frac{c_1 \sigma_{kl}^2}{1 + c_2 \omega^b} \tag{2.33}$$

with the coefficients

$$\begin{aligned} c_1 &= a\zeta \\ c_2 &= a\left(\frac{\zeta}{2\pi}\right)^b \\ b &= 1.67 \\ a &= 0.164 \\ \zeta &= \frac{L}{0.041\,\bar{v}} \end{aligned} \tag{2.34}$$

is a better representation of wind turbulence than the Dryden Spectrum (2.8). Its autocorrelation function

$$R_{KI}(t) = \frac{1}{\pi}\int_0^\infty S_{KI}(\omega)\cos(\omega t)\,d\omega \tag{2.35}$$

can be obtained via Wiener- Chintchin transform (eq. 6.16), where the time axis is not normalized. This equation is used in order to determine the constants σ_v^2 and β_v in the autocorrelation function of the O.U. - process, which is in the unnormalized form

$$R_{vv}(t) = \sigma_v^2\, e^{-\beta_v t} \qquad ,t>0 \tag{2.36}$$

It is worth pointing out that the Kaimal spectrum was empirically found. The above developed theory however only holds for a Lorenzian spectrum with autocorrelation (2.35). In order to use the results of the statistical theory based on the Lorenzian spectrum we approximate its parameters σ_v^2 and β_v as functions of the Kaimal parameter σ_{KI}^2 and ζ. As the autocorrelation function at t = 0 represents the power of the process, both autocorrelation functions (2.36) and (2.35) have to return the same value at t = 0, thus leading to the equation

$$\begin{aligned} \sigma_v^2 &= \frac{c_1\sigma_{KI}^2}{\pi}\int_0^\infty \frac{d\omega}{(1+c_2\omega^b)}\,d\omega \\ &= \frac{c_1\sigma_{KI}^2}{\pi\, b\sqrt[b]{c_2}}\int_0^\infty \frac{dy}{y^{1-\frac{1}{b}}(1+y)} \end{aligned} \tag{2.37}$$

The integrand in the second expression is not dependent on any parameters. This integral can be solved analytically ([8], 1.1.3.4), thus leading to the surprising result

$$\sigma_v^2 = 1.735\ \sigma_{kl}^2 \tag{2.38}$$

To estimate the coefficient β_v , the autocorrelation of the Kaimal spectrum is to be calculated at another point t,

$$\beta_v = -\frac{1}{t} \ln\left[\frac{c_1 \sigma_{kl}^2}{\pi \sigma_v^2} \int_0^\infty \frac{\cos(\omega t)}{1 + c_2 \omega^b} d\omega \right] \tag{2.39}$$

The integral has to be numerically calculated for a given t and c_2. In [25], p. 347 it is suggested to select t = 2s, as we are interested in short term fluctuations.

2.1.3 Macrometeorological Range

2.1.3.1 Mean Wind Speed Distribution

The horizontal hourly mean wind speed $\bar{v}$ is said to be Weibull- distributed with the distribution function ([19], eq. 2.14)

$$F(\bar{v}) = p(\bar{V} \le \bar{v}) = 1 - \exp\left[-\left(\frac{\bar{v}}{c}\right)^k \right] \tag{2.40}$$

which can be adapted to a given wind site by varying the shape parameter k and the scale parameter c. These parameters typically hover in the range of $k \in [1.7, 2.5]$ and $c \in [1.15, 1.18]$ respectively.

2.1.3.2 Mean Wind Speed Profiles

The horizontal wind speed varies with height. If the mean wind speed $\bar{v}$ is monitored at height $\hat{z}$ the mean wind speed at height z can be concluded from the formula ([19], eq. 2.5)

$$\frac{\bar{v}(z)}{\bar{v}(\hat{z})} = \frac{\ln\left(\frac{z}{z_0}\right) + 5.75\frac{z}{h}}{\ln\left(\frac{\hat{z}}{z_0}\right) + 5.75\frac{\hat{z}}{h}} \qquad h = \frac{u_*}{6f} \tag{2.41}$$

Here, h is the gradient height, f the Coriolis parameter, u_* the friction velocity and z_0 the roughness length. The Coriolis parameter depends on the location. It is f = 11.5E-5 s^{-1} for the UK. Values for z_0 are given in [19]. The friction velocity varies with surface roughness and with overall wind speed. If the friction velocity u_* is unknown the simpler form ([19], eq. 2.4)

$$\frac{\bar{v}(z)}{\bar{v}(\bar{z})} = \frac{\ln\left(\frac{z}{Z_0}\right)}{\ln\left(\frac{\hat{z}}{z_0}\right)} \tag{2.42}$$

may be applied.

2.2 Solar Energy

The intensity of the solar irradiation directly outside the earth's atmosphere is almost constant at around 1350 Wm^{-2}. Eventhough this value varies up to ± 3% due to eccentricities in the earth's orbit and fluctuating sunspots, it is stable enough to justify the name *solar constant.* On the earth's surface the peak solar intensity hovers around 1 kWm^{-2} on a horizontal surface, provided the sun is at its apex on a sunny day. In case the latter conditions are not fullfilled, the solar radiation experienced on a surface will not be as big. In general, it will depend on the position of the sun and the clarity of the atmosphere. These geometrical aspects will be covered in 2.2.1. The actual solar power on a tilted surface as a function of the clearness of the sky and the geometry will be calculated in 2.2.2. Chapter 2.2.3 is devoted to a brief discussion of the optimum surface orientation. It is worth noting that the solar power evaluated in 2.2.2 is a value, averaged over a longer time period. These values are good to estimate the solar energy received over a whole year at a selected site. They are, however, not suitable for on-line control schemes. Though, the introduced terminology and techniques will form the starting-point for the discussion of the statistical characteristics of short term fluctuations in chapter 2.2.4.

2.2.1 Geometrical Aspects

2.2.1.1 Determination of the sun' s position

The angle under which the sun is observed from a point on the earth' s surface is affected by the earth's daily rotation, expressed by the solar hour angle, and the annual rotation of the tilted earth, expressed by the declination angle and the observer's latitude. The orientation of the sun can then phrased in terms of the solar altitude and azimuth.

(i) The solar hour angle

The solar hour angle Ω expresses the daily rotation of the earth. As the earth rotates 360° within 24 hours, every hour adds another 15° to the solar hour angle. When the sun is in its highest point in the sky, the solar hour angle is zero ("Solar noon"). Angles before noon count negative, after noon positive. It is worth bearing in mind that the solar angle is not identical with the local time. For a conversion from solar hour angle values to the local time the longitude of the site in question and the local standard time have to be considered.

(ii) The declination angle

The declination angle δ is the angular position of the sun at solar noon with respect to the plane of the equator, and it varies because of the earth's tilt of 23.45° from -23.45° to +23.45°. Hence, the declination angle depends on the day of the year, $n \in [1, 365]$, and it is ([9], eq. 3-8)

$$\delta = 23.45 \left(\frac{\pi}{180}\right) \sin\left[2\pi \frac{284 + n}{365}\right] \tag{2.43}$$

on the northern hemisphere (in rad - not degrees). The declination angle reaches its peak at summer solistice and drops to its negative peak at winter solistice. It is converse on the southern hemisphere.

(iii) The latitude

If the sun is observed from a site other than the equator, the observer's latitude θ has to be considered, as the sun's highest altitude decreases with θ. The resulting solar-noon altitude angle is $\Omega_\theta = \frac{1}{2}\pi - \theta + \delta$.

(iv) Solar altitude, azimuth and zenith angle

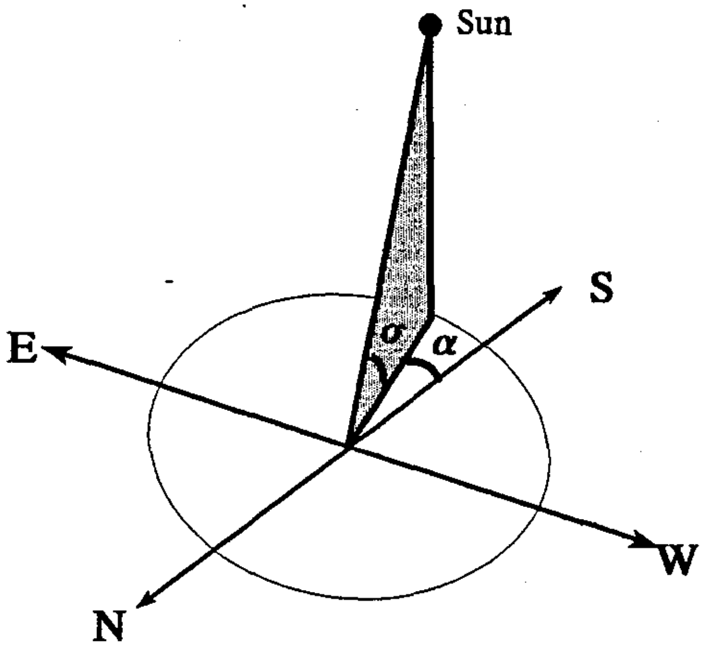

Fig. 2.6 Solar Altitude, Azimuth and Zenith Angle

The orientation of the sun in the sky can be phrased in terms of the solar altitude σ and the azimuth angle of the sun α. The altitude angle measures the angle between the line from the observer to the sun and the line to the horizon (compare Fig. 2.6). The solar azimuth angle gives the sun's angular distance from due south. An orientation to the East (as in Fig. 2.6) counts negative, West counts positive. Hence, azimuth angles from sunrise to solar noon are negative, while angles from solar noon to sunset are positive. The azimuth angle is obtained from ([9], eq. 3-4)

$$\sin\sigma = \sin\theta \sin\delta + \cos\theta \cos\delta \cos\Omega \tag{2.44}$$

The altitude is calculated from ([9], eq. 3-5)

$$\sin\alpha = -\frac{\cos\delta \sin\Omega}{\cos\sigma} \tag{2.45}$$

The complement of the solar altitude angle, the zenith angle, is defined as

$$\theta_z = \frac{\pi}{2} - \sigma \tag{2.46}$$

2.2.1.2 Sunrise and sunset

As the solar altitude angle is restricted to values $\sigma \in [-90°,90°]$ equation (2.45) is only valid for solar hour angles in the interval $\Omega \in [\Omega_{sr}, \Omega_{ss}]$ where Ω_{sr} denotes the sunrise angle and Ω_{ss} the sunset angle. Substituting $\sigma = \pm 90°$ into (2.45) leads to the sunrise angle $\Omega_{sr,h} = \Omega_s$ and sunset angle $\Omega_{ss,h} = -\Omega_s$ for horizontal surfaces, where

$$\Omega_s = \arccos(-\tan\theta \tan\delta) \tag{2.47}$$

For a tilted surface, however, equation (2.45) does not hold true.

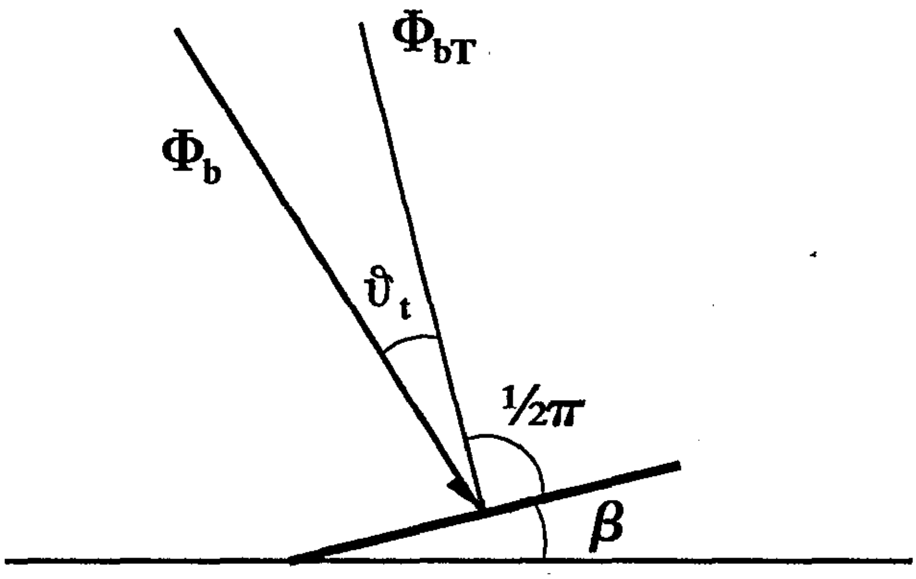

Fig. 2.7 Tilted Surface

Suppose we have an array that is inclined to the horizontal by an angle β (compare Fig. 2.7). The angle between the projection of the normal of the plane on the horizontal and South is α, the azimuth angle as introduced above, so that $\alpha = 0$ is due South, $\alpha > 0$ an orientation towards the West and $\alpha < 0$ an orientation towards the East. In contrast to the horizontal surface, the magnitudes of the solar angle for sunrise and sunset are not equal. They can be calculated by evaluating ([12], 2.2.15)

$$\begin{aligned} \Omega_{sr} &= -\min\left\{\Omega_s\,,\ \arccos\left(\frac{-ab+sign(\alpha)\sin(\alpha)\sin(\beta)\sqrt{a^2-b^2+1}}{a^2+\sin^2(\alpha)\sin^2(\beta)}\right)\right\} \\ \Omega_{ss} &= \min\left\{\Omega_s\,,\ \arccos\left(\frac{-ab-sign(\alpha)\sin(\alpha)\sin(\beta)\sqrt{a^2-b^2+1}}{a^2+\sin^2(\alpha)\sin^2(\beta)}\right)\right\} \end{aligned} \tag{2.48}$$

with the abbreviations

$$\begin{aligned} a &= \cos\theta\ \cos\beta\ +\ \sin\theta\ \cos\alpha\ \sin\beta \\ b &= \tan\delta\ (\sin\theta\ \cos\beta\ -\ \cos\theta\ \cos\alpha\ \sin\beta) \end{aligned} \tag{2.49}$$

In case the surface faces due south (β = 0), the magnitudes of sunset and sunrise angle will be the same. Substituting β = 0 into (2.48) leads to a sunset angle

$$\Omega_s = \min\left\{\Omega_s\,,\ \arccos(-\tan(\theta-\beta)\ \tan\delta)\right\} \tag{2.50}$$

2.2.2 Average Daily Solar Energy

Empirical solar radiation data is mostly data for horizontal surfaces. That is, the monthly average daily total radiation on a horizontal surface, H, is measured. If H_0 denotes the monthly average daily total radiation directly outside the earth's atmosphere (i.e. the insolation that would be experienced without the earth's atmosphere), the clarity index K can be defined by

$$K = \frac{H}{H_0} \tag{2.51}$$

which is the quotient of H and H_0. This coefficient is based on measured data depending on the location and the month. The sunlight received by a horizontal surface can be divided into two parts. First, the direct beam radiation, which strikes the surface from one angle only - directly from the sun. Second, the diffuse light, which is the proportion of light that is absorbed or scattered by air molecules, water vapor dust while passing the earth's atmosphere. Diffuse light approaches the horizontal surface from almost any angle. Hence, the monthly average daily total radiation on a horizontal surface can be written as a superposition of H_b, the direct or beam radiation, and H_d, the diffuse radiation:

$$H = H_b + H_d \tag{2.52}$$

Light which approaches a tilted surface may as well be light reflected upon the ground (other than the array surface). The conversion of the monthly average daily energy on a horizontal surface, H, can be converted to the monthly average daily energy on a tilted surface, H_T in two steps. This is in so far important as only values for the horizontal surface are available.

(1) Estimating the diffuse light

Given an observed value of H, the diffuse radiation term in (2.52) can be separated by a specific correlation function. For latitudes θ between 43°N and 54°N the transformation ([29], eq.3)

$$K_d = \begin{cases} 1.557 - 1.84K, & 0.35 \le K \le 0.75 \\ 0.177, & K > 0.75 \\ 1.0 - 0.249K, & 0 \le K < 0.35 \end{cases} \tag{2.53}$$

is supposed to be accurate, where

$$K_d = \frac{H_d}{H} \quad . \tag{2.54}$$

is called diffusion index in analogy to the clarity index defined in (2.51). For other latitudes similar formulas have been developed (for instance [17]). Having calculated the diffusion term H_d, the beam radiation H_b can be worked out from (2.52).

(ii) Radiation on a tilted surface

The total hourly radiation on a titled surface is (the index T connotes "tilted")

$$H_T = H_{bT} + H_{dT} + H_{rT} \tag{2.55}$$

It differs from (2.52) only in the additional term H_r, representing the reflected light. In the following we express these terms as functions of H, the hourly total radiation on a horizontal surface, and the introduced geometrical magnitudes.

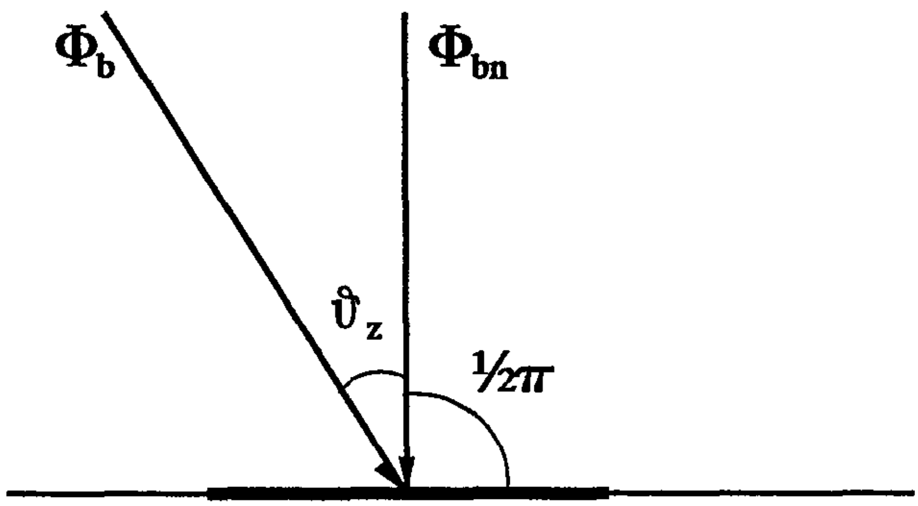

Fig. 2.8 Radiation on a Tilted Surface

We will first deal with the direct radiation term. The normal component $\Phi_{bT,n}$ of the intensitiy Φ_b of the incoming light beam (compare with Fig. 2.8) on a tilted surface can be obtained from ([12], eq. 2.2.9, 2.2.10)

$$\Phi_{bT,n} = \Phi_b \cos \vartheta_T = \Phi_{bn} \frac{\cos \vartheta_T}{\cos \vartheta_z} \tag{2.56}$$

with

$$\cos \vartheta_T = \sin\delta \, \sin(\theta - \beta) + \cos\delta \, \cos(\theta - \beta) \, \cos\Omega \tag{2.57}$$

Here, Φ_{bn} is the normal component on the horizontal surface. Equation (2.56) is a good approximation unless large differences between ϑ_T and ϑ_z have to be considered. Otherwise the Bădescu- formula ([2], eq. 10) should be used. Let R_b denote the ratio of the average daily beam radiation on a tilted surface to that on a horizontal surface,

$$R_b = \frac{H_{bT}}{H_b} \tag{2.58}$$

With the different solar hour angles for sunrise and sunset, (2.48) and (2.47), the ratio R_b for any tilted surface with slope angle β and azimuth angle α is obtained from ([12], eq. 2.2.14)

$$\begin{aligned} R_b &= \frac{a\,(\Omega_{ss} - \Omega_{sr}) + b\,(\sin(\Omega_{ss}) - \sin(\Omega_{sr}) - c\,(\cos(\Omega_{ss}) - \cos(\Omega_{ss}))}{2(\cos\theta \, \cos\delta \, \sin\Omega_s + \Omega_s \sin\theta \, \sin\delta\,)} \\ a &= \sin\delta \, (\cos\beta \, \sin\theta - \cos\alpha \, \sin\beta \, \cos\theta) \\ b &= \cos\delta (\cos\theta \, \cos\beta + \sin\theta \, \sin\beta \, \cos\alpha) \\ c &= \cos\delta \, \sin\beta \, \sin\alpha \end{aligned} \tag{2.59}$$

In the preferrable situtation that the solar array is facing due south (α = 0) R_b can be evaluated from the simpler representation (with Ω'_s as in (2.50))

$$R_b = \frac{\cos(\theta - \beta)\cos\delta \, \sin\Omega_s + \Omega_s \sin(\theta - \beta)\sin\delta}{\cos\theta \, \cos\delta \, \sin\Omega_s + \Omega_s \sin\theta \, \sin\delta} \tag{2.60}$$

As far as the diffuse radiation on a tilted surface is concerned, an isotropic distribution of the diffuse radiation over the hemisphere is assumed. The diffusion term can be attained from ([12], eq. 3.23)

$$H_{dT} = H_d \frac{(1 + \cos\beta)}{2} \tag{2.61}$$

which takes into account that the tilted slope sees only a portion of the hemisphere. H_d is the diffusion term of the horizontal surface.

The last term in (2.55) is the reflected light portion. The energy of the reflected light is dependant on the ground's abilitiy to reflect, a property which may be represented by the albedo factor ϱ. The albedo usually ranges from 0.1 (asphalt paved roads) up to 0.9 (snow). Given the albedo, the diffusion term can be calculated from

$$H_{rT} = \varrho\ (H_b + H_d) \left(\frac{1 - \cos\beta}{2}\right) \tag{2.62}$$

Substituting equations (2.59), (2.61) and (2.62) into (2.55) results in the monthly daily total radiation on a tilted surface:

$$H_T = H_b R_b + H_d \frac{(1 + \cos\beta)}{2} + \varrho (H_b + H_d) \frac{(1 - \cos\beta)}{2} \tag{2.63}$$

Finally, the ratio of monthly average daily total radiation on a tilted surface to that on a horizontal surface can be defined as

$$R = \frac{H_T}{H} = (1 - K_d) R_b + K_d \left(\frac{1 + \cos\beta}{2}\right) + \varrho \left(\frac{1 - \cos\beta}{2}\right) \tag{2.64}$$

At the end of this section it is worth pointing out that the calculus presented here applies to monthly averages. It is assumed that clouds are uniformly distributed over the sky. Drifting clouds are not considered in this technique.

2.2.3 Optimum Surface Orientation

Apparently, the maximum amount of direct-beam insolation is experienced by a surface whose normal is parrallel to the incoming light. In order to achieve this optimum orientation it must be possible to rotate the surface around two axes, namely the tilt and the azimuth angle, which requires two motors. Usually, the additional energy obtained by a two- motor option is marginal and does not pay off. Hence, the second best option is to fix the surface, so that it faces due south and keep the slope angle flexible. In case that there is no possibility to move the array at all, the surface would obtain the optimum amount of direct-beam solar radiation over a year, if the tilt angle was equal to the site's latitude. Tilting the surface up, on the other hand, causes the diffuse light portion to decrease. The annual optimum surface at sites with humid climates is therefore about 10% - 25% less than the latitude ([9]). The last statement is backed by an experimental investigation ([23]), in which a tilt angel of 30° is suggested for a location at 48° north.

2.2.4 Short- term Global Irradiance

2.2.4.1 Probability Density Function

Similar to the wind, the solar insolation is a stochastic process that reveals a distinctive short- term irradiance process, a phenomenon we might call turbulence by borrowing the word from the analysis of the wind. The short- term (5 minutes time average values) solar irradiance has been modelled in a paper by A. Skartveit ([40]). We will cite from this paper throughout this section unless otherwise specified. The objective is a probability density function with the same functionality as in the case of wind turbulence, now for the intra-hour radiation. Again the pattern here is that we have a stochastic model of the radiation for a time period of an hour.

For the purpose of the short- term solar irradiance model the average root squared deviation

$$\sigma_k = \sqrt{\frac{(K_j - K_{j-1})^2 + (K_j - K_{j+1})^2}{2}} \tag{2.65}$$

will be defined. The coefficient K_j is the clearneass index as defined in (2.51) at the hour with index j. The average root squared deviation is hence a weight function that takes into account the changes of the clearness index from the precedent hour to the hour in question and further on to the subsequent hour. Within the 5 minutes developmental sample the (i.e. for 5 minutes time average values) observed distribution of the intrahour standard deviation σ_k is Weibull- distributed with the density function

$$p(s) = \alpha \gamma (\alpha s)^{\gamma - 1} \exp\left(-(\alpha s)^{\gamma}\right) \tag{2.66}$$

corresponding distribution function

$$F(s) = 1 - \exp\left(-(\alpha s)^{\gamma}\right) \tag{2.67}$$

and the coefficients

$$\begin{aligned} \alpha &= \Gamma\left(1+\frac{1}{\gamma}\right) \\ s &= \frac{\sigma_k}{\sigma^*} \\ \sigma^* &= 0.87K^2(1-K) + 0.39\tilde{\sigma}\sqrt{K} \\ \gamma &= 0.88 + 42(\sigma^*)^2 \end{aligned} \tag{2.68}$$

Here, $\Gamma(x)$ is the well known gamma function. The coefficient σ_k must be estimated by chosing a random number ζ, which is supposed to be evenly distributed between 0 and 1, instead of F(s). Then solve (2.67) for s,

$$s = \frac{1}{\alpha}\sqrt[\gamma]{-\ln(1-\zeta)} \tag{2.69}$$

and eventually determine σ_k with (2.68). Given the hourly mean clearness index K (capital K) and the standard deviation σ_k, the distribution of short term k- values (lower case k) is phrased in terms of a scaled clearness index x,

$$x = \frac{k - k_{min}}{k_{max} - k_{min}} \tag{2.70}$$

and standard deviation σ_x,

$$\sigma_x = \frac{\sigma_k}{k_{max} - k_{min}} \tag{2.71}$$

The minimum and maximum values of k are given by the empirical formulas

$$\begin{aligned} k_{min} &= \max\left\{0\ ,\ (K-0.03)\exp(-11\sigma_k^{1.4}) - 0.09\right\} \\ k_{max} &= (K-1.5)\exp(-9\sigma_k^{1.3}) + 1.5 \end{aligned} \tag{2.72}$$

The probability density function of the scaled index x is now described by a linear combination of two Beta- distributions[3]. To clarify the following formalism we state the definition of the incomplete Beta- function ([41], def. 58:3:1)

$$B(\alpha,\beta,x) = \int_0^x t^{\alpha-1}(1-t)^{\beta-1}\,dt \qquad , 0<x<1 \tag{2.73}$$

and its normalized form ([41], def. 58:1:1)

$$I(\alpha,\beta,x) = \frac{B(\alpha,\beta,x)}{B(\alpha,\beta)} \qquad , B(\alpha,\beta) = B(\alpha,\beta,1) \tag{2.74}$$

$B(\alpha,\beta)$ is called Beta- function. Applying this notation the probability density function of the scaled index x is

$$f_x(x) = w C_1 t^{a_1-1}(1-x)^{b_1-1} + (1-w)\, C_2 x^{a_2-1}(1-x)^{b_2-1} \tag{2.75}$$

with the coefficients

$$\begin{aligned} a_j &= \max\left\{1 \; , \; (1-\kappa_j)\left(\frac{\kappa_j}{\sigma_j}\right)^2 - \kappa_j\right\} \\ b_j &= \max\left\{1 \; , \; \frac{1-\kappa_j}{\sigma_j^2}\left(\kappa_j(1-\kappa_j)-\sigma_j^2\right)\right\} \\ C_j &= (B(a_j,b_j))^{-1} \end{aligned} \tag{2.76}$$

and

$$w = \frac{\kappa_2 - T}{\kappa_2 - \kappa_1} \tag{2.77}$$

with

[3]A random variable X is said to be beta- distributed with the parameters α and β if the corresponding probability distribution function is $F(x) = I(\alpha,\beta,x)$.

$$\begin{aligned} \kappa_1 &= \hat{K}\left(0.01 + 0.98\exp\left(-60\sigma_t^{3.3}\right)\right) \\ \kappa_2 &= (\hat{K}-1)\left(0.01 + 0.98\exp\left(-11\sigma_t^2\right)\right) + 1 \\ \sigma_1^2 &= 0.014 \\ \sigma_2^2 &= 0.006 \end{aligned} \tag{2.78}$$

Here, $\hat{K}$ is the hourly average clearness index normalized as in (2.70). The probability distribution function of the process X will then be written as

$$F_x(x) = w\ I(a_1, b_1, x) + (1-w)\ I(a_2, b_2, x) \tag{2.79}$$

and consequently the distribution function of the short term k- values (clearness index) as[4]

$$F_k(k) = F_x\left(\frac{k - k_{min}}{k_{max} - k_{min}}\right) \tag{2.80}$$

At the end of this section, let us throw the main points into relief: Within a reasonable time interval, the clearness index k is a stochastic process whose distribution function is described by $F_k(k)$ (2.80), which is a function of the hourly mean clearness index K and the standard deviation σ_k. In practice, the latter parameter can be estimated from previous observations (eq. (2.68)).

2.2.4.2 Conditional Probability

The objective of this subsection is to develop a technique to calculate the conditional distribution function $F_x(x(t)|X(0) = x_0)$ of X(t) subject to the condition $X(0) = x_0$. We will often use the abbreviation $G_x(x) = F_x(x(t)|X(0) = x_0)$. For the purpose of this section we assume an autocorrelation coefficient in the form

$$r_x = r_x(t) = \exp(-\beta_x t) \tag{2.81}$$

for the scaled clearness index x. At time t = 0 the conditional distribution function should

[4]Refer to chapter 6.2, for discussion of functions of random variables

yield $F_x(x(0) \mid X(0) = x_0) = s(x - x_0)$ [5] and its probability density function $f_x(x(0) \mid X(0) = x_0) = \delta(x - x_0)$ since the probability to observe the process X(t) at time t = 0 in x_0 is equal to 1. One way to work out the conditional probability would be to construct the joint probability density function $f_x(x(t), x(0))$ of the stochastic processes X(t) and X(0) from the given marginal distributions $F_x(x(t))$ and $F_x(x(0))$ and the autocorrelation coefficient. A technique to construct the joint probability density function from the marginal distributions is presented in [18]. Given the joint probability density function, the conditional probability density could be concluded from equation (6.14). In [18], the joint density function is known, which is not the case here. Hence, the problem is being solved in a different manner. First, the (non-conditional) distribution function $F_x(x)$ (2.79) will be approximated by a superposition of normal distributions with their peaks shifted along the x- axis. The expansion has the form

$$\hat{F}_x(x) = \sum_{q=1}^{Q} u_q \, v_q(x) \approx F_x(x) \tag{2.82}$$

with the generating functions

$$v_q(x) = \Phi\left(\frac{x - \frac{q}{Q+1}}{\sigma_q}\right) \tag{2.83}$$

In (2.82), u_q are coefficients which will be subsequently determined. The generating functions $v_q(x)$ are normal distributions (definition equation (6.19)) along x with their means centered at x = 0.5 and equidistantly distributed. The standard variation coefficients σ_q will be chosen as

$$\sigma_q = \frac{\varepsilon}{Q}\left[\max\left\{1,\ f_x\left(\frac{q}{Q+1}\right)\right\} + 1\right] \tag{2.84}$$

with a single coefficient ϵ. The standard variation of each of the normal distributions will thus be smaller if Q is larger or - in other words - if more functions are taken into account and hence the distance between two peaks becomes smaller. The division by Q in (2.84) is

[5] $s(x - x_0)$ denotes the unit step function with the step at $x = x_0$.

not imperative but intended to ensure that ϵ lies in the same order of magnitude irrespective of Q. The term in brackets in equation (2.84) is a number between 1 and 2 and has the following effect: Whenever the density function $f_x(x)$ is small (or the increments in $F_x(x)$) the variance of the normal distribution with its peak at this point will be smaller and vice versa. This correction term permits a more sensitive adaption in low- probability regions. The limitation of the correction term to values in the interval [1,2] seems to be appropriate to the range of $f_x(x)$. In order to optimize the approximation a least square problem is introduced with the merit function

$$V(u_q) = \sum_{m=1}^{M} \left[\sum_{q=1}^{Q} \left(u_q \, v_q\!\left(\frac{m}{M+1}\right)\right) - F_x\!\left(\frac{m}{M+1}\right) \right]^2 \tag{2.85}$$

as a function of the coefficients u_q. Here, we assume that M trial points are taken into account. It is worth pointing out that the generating functions $v_q(x)$ do not form an orthogonal or complete function system. Therefore the choice of Q, M and ϵ has to be carefully considered. As $F_x(x)$ is a superposition of two incomplete Beta- functions its derivative $f_x(x)$ may have up to 2 relative maxima over $x \in [0,1]$. Hence, Q must be greater than 2, better 8 or 12. Numerical results have shown that $Q > 12$ is not beneficial. For a condensed representation we note the abbreviation

$$\alpha_{mq} = \Phi\left(\frac{\frac{m}{M+1} - \frac{q}{Q+1}}{\sigma_q} \right) \tag{2.86}$$

To find the minimum of (2.85) its gradient with respect to u_q is to be set equals 0. Rearranging this condition yields

$$\sum_{m=1}^{M} \sum_{q=1}^{Q} u_q \, \alpha_{mq} \, \alpha_{mj} = \sum_{m=1}^{M} F_x\!\left(\frac{m}{M+1}\right) \alpha_{mj} \qquad j=1...Q \tag{2.87}$$

This is a system of linear equations, and we can arrange it into the matrix representation

$$\boldsymbol{A}\, \boldsymbol{u} = \boldsymbol{d} \tag{2.88}$$

with a symmetric coefficient matrix **A** and a right hand vector **d**. The elements of **A** and **d** are

$$d_j = \sum_{m=1}^{M} F_x\left(\frac{m}{M+1}\right) \alpha_{mj}$$
$$A_{ij} = \sum_{m=1}^{M} \alpha_{mi}\, \alpha_{mj} \qquad (2.89)$$

Hence, solving (2.88) for **u** minimizes the merit function V (2.85) with respect to the coefficients u_q for a fixed standard deviation parameter ϵ. The whole algorithm that considers ϵ as well is then as follow:

1. Set initial $\epsilon = 0.4$
2. Calculate **u** from (2.88) and the merit function V (2.85)
3. Repeat step 2 for different values of ϵ until a minimum of V along the ϵ - axis has been found. The line search for ϵ is carried out in two steps: First, a bracket will be searched for, in which the minimum lies in. Second, a golden section search[6] ([15]) follows to determine the minimum with a higher accuracy. High accuracy on the other hand is counterproductive to the computing time. Note that for each ϵ, ½MQ evaluations of $\Phi(x)$ are required. We will therefore quit the algorithm as soon as a V - value has been found which is below a specific value (e.g. 0.003). In case Q was selected as 5 and V at the initial point $\epsilon = 0.4$ is above 0.1, Q will be set to 8 and the algorithm restarted. Otherwise the algorithm will be aborted if the minimum of V has been determined to lay in an interval along the ϵ- axis which is smaller than 0.02 .
4. Function values of $F_x(x)$ can then be worked out from $\hat{F}_x(x)$ (eq. (2.82)).

The quality of the approximation can be checked by calculating the difference between the object function $F_x(x)$ ((2.79)) and its approximation (2.82),

$$\Delta(x) = F_x(x) - \hat{F}_x(x) \qquad (2.90)$$

[6] Golden section search is after the Fibonacci routine the most efficient routine to find a minimum of a function of one variable, when an initial bracketing of the minimum is given.

In Fig. 2.9 $\Delta(x)$ has been calculated for typical values for k, K_0 and σ_k. Here, the number of coefficients is set to Q = 8, with the number of trials, M, as parameter. For M = Q the trial points coincide with the peaks of the Gaussian functions. The figure of merit in this case was V = 2.4E-30. Increasing the number of trials to 16 does not improve the performance. It is actually quite the reverse. Hence, it is recommended to set Q = M.

In Fig. 2.10, Q and M have been simultaneously changed so that Q = M. Obviously, Q = 6 is not sufficient as the maximum difference Δ is 0.159 for the chosen parameters of k, K_0 and σ_k.

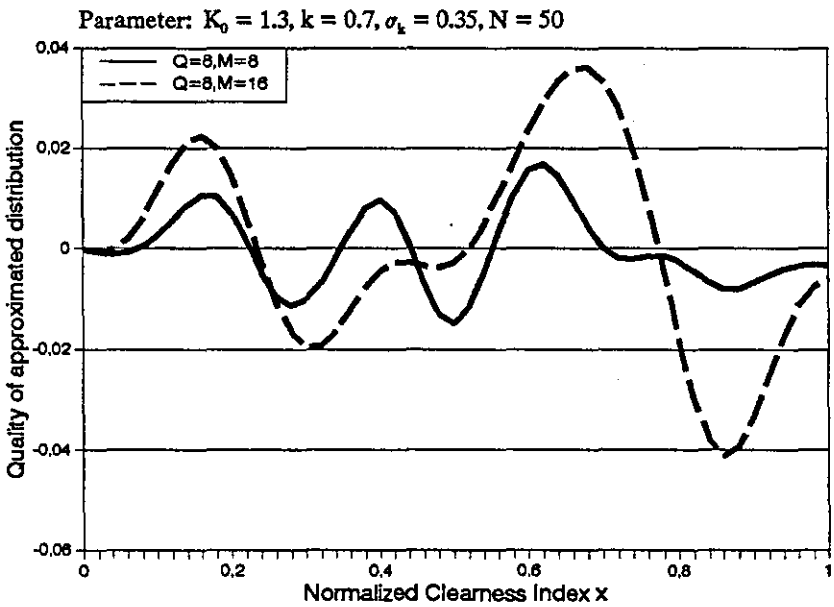

Fig. 2.9 Quality of the Approximation

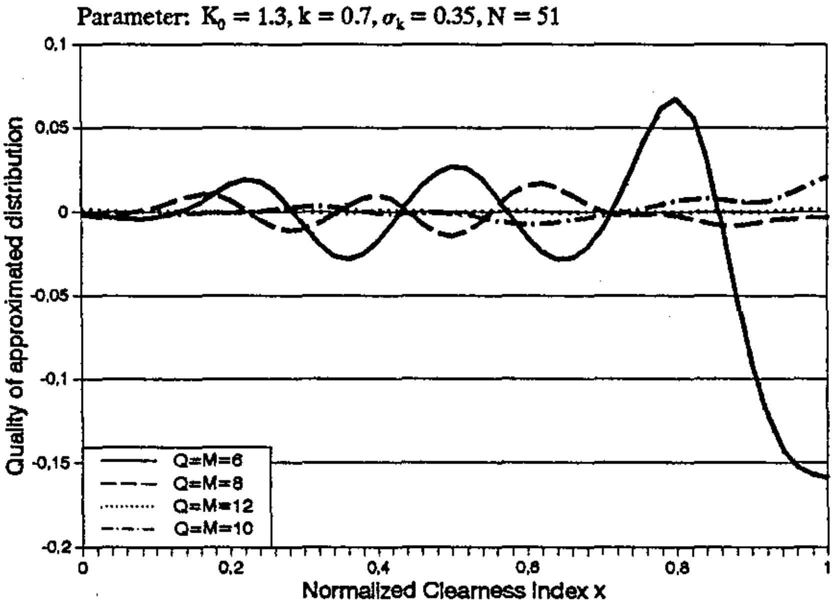

Fig. 2.10 Quality of the Approximation

For Q = 12, a maximum difference Δ of 0.001 has been observed. Larger Q- values will further improve the approximation. The associated calculation time, however, will increase as well, thus forcing to strike a balance between expenditure and accuracy. As the probability function is an empirical function, Q = 12 seems to be a good choice and will be used in all calculations carried out in this paper unless otherwise explicitly stated.

Having determined u_q and ϵ the distribution function can now be worked out from (2.82). As the conditional distribution of a normal distributed random variable is known (with density function as in 6.23), the conditional distribution function of X as the superposition of normal distributions can be easily concluded. It is

$$\hat{F}_x(x \mid X(0) = x_0) = \sum_{q=1}^{Q} u_q \, \Phi\left(\frac{x - \left(\frac{q}{Q+1} + \left(x_0 - \frac{q}{Q+1} \right) r \right)}{\sigma_q \sqrt{1 - r^2}} \right) \tag{2.91}$$

which is the superposition of weighted, conditional normal distributions with autocorrelation coefficient r_x (2.81). Equation (2.91) satisfies the stated initial conditions and it goes over into (2.82) for $t \to \infty$ when $r \to 0$.

So far, statistical models for the short term behaviour of both wind speed and clearness index have been presented. We will continue this discussion in chapter 4.1, where the short term statistical models will be unified and extended to the total power supplied by the renewable energy sources. In order to include the power in the statistical theory, models for the wind generator and the photvoltaic array are needed. They will be the focus of the discussion in the following chapter 3.

2.3 Battery

2.3.1 Storage Technolgies

A storage unit in a hybrid wind- pv- system is used to deposit any surplus in the energy supplied by the renewable energy sources. In times, when the energy demand exceeds the available renewable energy, it is supposed to deliver the stored energy in order to avoid starting the fossil fuel generator. This could be for a short period of seconds as well as for a period of days. Out of all possible technologies the one should be selected, that fullfills the following criteria best:

- High charging- and recharging efficiency as well as a high storage efficiency
- Speed at which the storage system can be brought into in order to absorb or deliver energy.
- High lifetime expectancy
- High reliability
- Low cost
- Low ecologically harmful emission during both production and operation. Possibility of recycling after reaching the lifetime limit.
- Small size

In the following a brief outline of different storage technologies will be given and the above criteria will be addressed.

Mechanical storage systems reveal a high energy conversion efficiency. A drawback is their large size.

Chemical storage systems, in general, have a lower efficiency for energy conversion. The most prominent example for this category is the hydrogen production ([31]). Hydrogen is versatile in its application and an environment friendly storage medium. The costs, however, are considerably (~ 1000 ECU/kW).

Electrical storage systems, for instance in form of an electrolyte capacitor, are only suitable for the storage of energy for a few seconds.

Electrochemical storage systems (batteries) are systems where the chemical energy is translated into electric energy, which is produced when the chemicals in the system react with one another. Rechargeable systems allow the reverse process as well. Lead- Acid is the most commonly used battery type in PV applications due to its competitive price. NiCd batteries tend to have a higher energy density and may last longer in very cold areas. They are, however, more expensive [21].

For this paper a lead- acid battery has been chosen as energy storage system, which seems to be a good compromise between cost and life expectancy. Compare discussion in [21] and [7].

2.3.2 Lead- acid battery

2.3.2.1 Chemical Reaction

The energy stored in a battery is a chemical energy that is translated into electrical energy. The latter one is produced when the chemicals in the battery react with one another. Rechargeable batteries as the lead- acid battery allow the reverse process as well. In case of the lead- acid battery the chemical reaction can be written as ([37])

$$\mathbf{Pb + PbO_2 + H_2SO_4 \leftrightharpoons 2PbSO_4 + 2H_2O}\ .$$

The rate of the chemical reaction varies with

- state of charge,
- battery storage capacity,
- rate of charge and discharge,
- environmental temperature and
- the age and the shelf life of the battery.

2.3.2.2 State of Charge

The electric charge, $Q_0(t)$, in a battery can be thought of as the sum of the available charge

$Q_1(t)$ and the bound charge $Q_2(t)$. They all vary with the time. At the beginning, however, the electric charge $Q_0(0) = Q_{1,0}(t) + Q_{2,0}(t) = Q_b$ coincides with the battery storage capacity (i.e. the rated charge). The state of charge is defined as

$$SOC(t) = \frac{Q_1(t)}{Q_b} \tag{2.92}$$

the quotient of the residual capacity $Q_1(t)$ and the battery storage capacity. The depth of discharge, DOD, is then simply

$$DOD = 1 - SOC \tag{2.93}$$

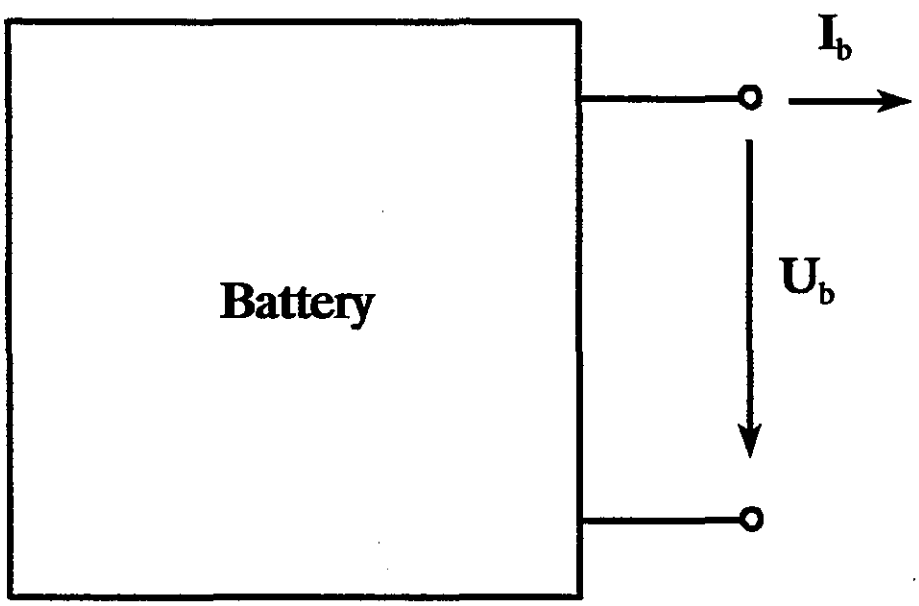

Fig. 2.11 Battery as a Two-Pole Device

If the battery is viewed as a two- pole electrical device (Fig. 2.11) with output current I_b and voltage V_b three states of the battery, dependant on the sign of I_b, can be defined as follows:

(1) $I_b < 0$ **:** The battery will be charged.

(2) $I_b = 0$ **:** The battery will be exposed to an internal discharge, idle discharge. A typical value for self discharge is 0.1% per day ([42]).

(3) $I_b > 0$ **:** The battery will be discharged.

In Fig. 2.12 the SOC is sketched for the three phases as a function of time.

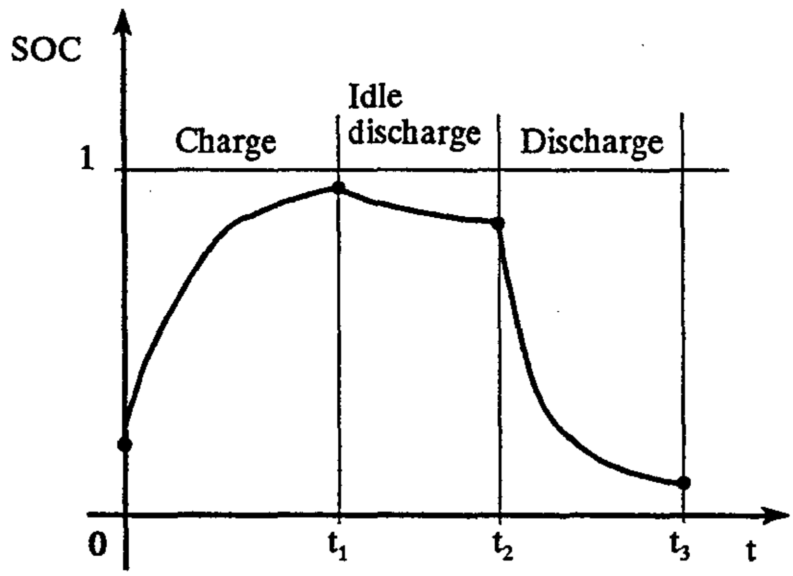

Fig. 2.12 State of Charge

Knowing the state of charge of the battery is very important for the energy management as it directly represents the energy that is available in the battery. As the battery charging or discharging current is in most cases not constants and varies according to changes in solar insolation and wind speed, a reliable state-of-charge determining on-line method is needed. For the purpose of this paper it is assumed that the state of charge can be determined. An on-line algorithm is described in [43] for instance.

2.3.2.3 Battery Modelling

The purpose of a battery model in this context is to provide a relationship between the state of charge, current and voltage. Below follows the brief discussion of three battery models. The first two, the Shepherd and the Salameh- Model, are electric models, the third is a storage model. The electric models can be described in terms of an electric circuit with various elements. They permit us to calculate the voltage and the current. Given the electric current the available charge can be concluded from the differential

$$\frac{\partial Q}{\partial t} = \eta_b \, I \tag{2.94}$$

where η_b is a charge/ discharge efficiency factor.

(i) The Shepherd Model

A simple electric model was devised by Shepherd ([38], [24]). The electric circuit is illustrated in Fig. 2.13.

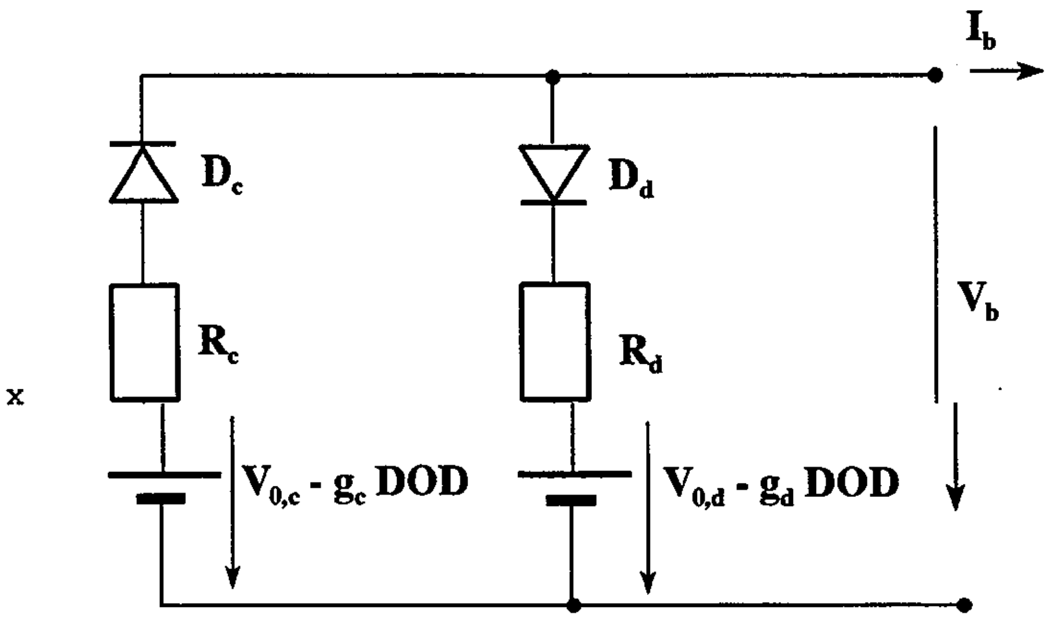

Fig. 2.13 Battery Equivalent Circuit: Shepherd Model

It consists of a series of a resistance, R_0, a fixed voltage, V_0, and a charge dependant voltage, g DOD. The diodes are for directional purposes only, with the index 'c' for 'charge' and 'd' for 'discharge'. The discharge voltage is ([24], eq. 6)

$$V_b = V_{0,d} - g_d\, DOD + I_b R_d \tag{2.95}$$

and the charge voltage accordingly with index 'c' instead. The equation does not take into account the diodes which modify the model slightly at very low currents. The resistance R_b is defined as ([24], eq. 7)

$$R_d = R_{0,d} \left(1 + \frac{m_d DOD}{\frac{Q_{m,d}}{Q_b} - DOD} \right) \tag{2.96}$$

Here, m_d denotes a parameter describing the cell type, $R_{0,d}$ the internal resistance at full charge and $Q_{m,d}$ a capacity paramter. Again, the same formula applies for charging the battery with index 'c'. In this form the model requires 5 parameters for each process, charging and discharging. This model can be easily extended to accommodate temperature dependancy by declaring parameters as functions of the temperature. Facinelli ([13], eq. 4a) assumes a quadratic relationship, whereas Khouzam ([24], eq. 9) employs linear functions.

(ii) The Salameh Model

The Salameh Model ([37]) is a further development of the Shepherd model, as it takes internal discharge and overvoltage into account. The electric circuit is shown in Fig. 2.14.

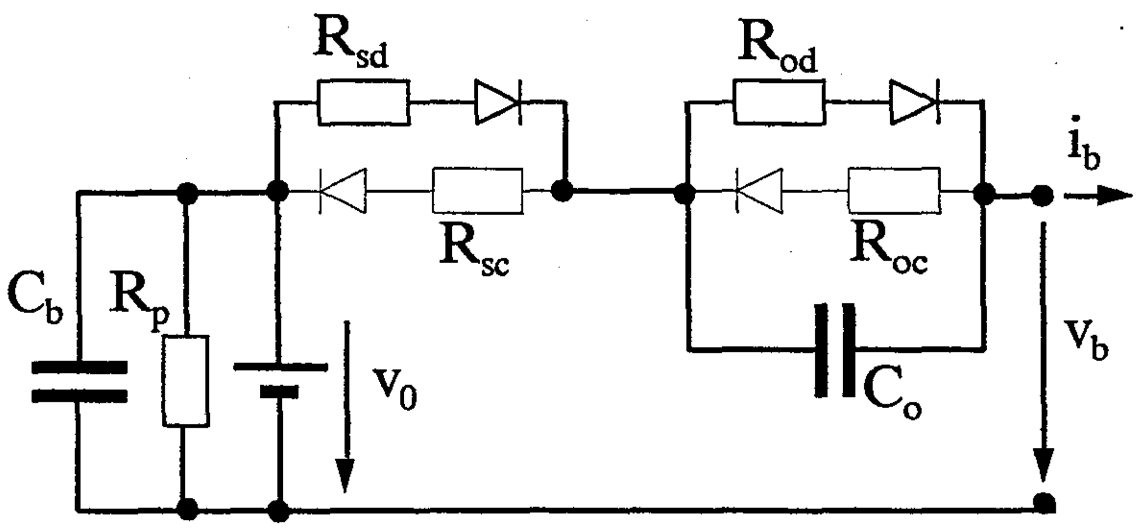

Fig. 2.14 Battery Equivalent Circuit: Salameh Model

Again, the diodes are strictly for directional purposes and in this sense ideal. The battery capacity is C_b, the self discharge resistance R_p. Devices with index 'o' stem from the overvoltage circuit, whereas 'd' and 'c' denote 'discharge' and 'charge'. Although it seems to be a linear circuit - apart from the diodes - it is not. All devices are non- linear. The state of charge can therefore only be worked out in an iterative way.

(iii) The Manwell Model

This model ([27]) places the emphasis on the electric charge. It assumes that the electric charge in a battery is either available or chemically bound. Charging and discharging causes a transfer of charge from one to the other 'container', though the sum of both may decrease with the time. According to the model the amount of available charge, $Q_1(t)$, and bound charge, $Q_2(t)$ at time t can be written as ([27], eq. 8,9)

$$
\begin{aligned}
Q_1(t) &= Q_{1,0} + \frac{(Q_0 kc - I)(1-e^{-kt})}{k} - \frac{Ic(kt-1+e^{-kt})}{k} \\
Q_2(t) &= Q_{2,0}e^{-kt} + Q_0(1-c)(1-e^{-kt}) - \frac{I(1-c)(kt-1+e^{-kt})}{k}
\end{aligned}
\tag{2.97}
$$

with $Q_{1,0}$ and $Q_{2,0}$ denoting the charges at the beginning of the calculations. The sum of both is denoted by $Q_0 = Q_{1,0} + Q_{2,0}$. The parameter k is a rate parameter. The width of the charge containers is described by c. Assuming a constant voltage the maximum discharge current is ([27], eq. 22)

$$I_{d\max} = \frac{k Q_{1,0} e^{-kt} + Q_0 kc(1 - e^{-kt})}{1 - e^{-kt} + c(kt - 1 + e^{-kt})} \tag{2.98}$$

The maximum charge current can be obtained from ([27], eq. 23)

$$I_{c\max} = \frac{-kcQ_{\max} + kQ_{1,0} e^{-kt} + Q_0 kc(1 - e^{-kt})}{1 - e^{-kt} + c(kt - 1 + e^{-kt})} \tag{2.99}$$

Here, Q_{max} is the maximum battery capacity.

The model in this form does not take into account any temperature effects. For moderate temperatures, however, it procures accurate results. There are two major advantages of this model: First, it requires only 3 parameters, Q_{max}, k and c. In comparison, the Shepherd model requires 10 parameters, the Salameh model draws data from curves in order to determine its underlying non- linear elements. Second, the Manwell model is based on the electric charge, a fact that simplifies the determination of the state of charge. In the electric models, the state of charge has to be calculated by solving a differential equation. Hence, for the generation of time series of the state of charge in the section on time series, the Manwell model is used.

2.3.2.4 Lifetime Considerations

Depending on theoretical assumptions different statements can be made about the lifetime of a battery, which is measured in the number of cycles, N. The simplest relationship is ([22])

$$N\,DOD \approx \text{constant} \tag{2.100}$$

as long as the battery is not overcharged or overdischarged. Other laws are similar and do in fact converge into above relationship under certain conditions. It is recommended ([11]) to operate the battery between 40% SOC and 80% - 90% SOC. In [39] we have found some typical values concerning the lifetime:

60% DOD 2000 cycles

30% DOD	4000 cycles
10% DOD	6000 cycles

Summarising, it can be said that the charger/ discharger of the battery should be aware of the fact that an increased lifetime is only possible with a shallow depth of discharge.

2.4 Diesel Generator

With regard to the objective of this study just two facets of the operation of the diesel are of significance: Fuel consumption and life time, both of whom are covered in the following two sections.

2.4.1 Fuel Consumption and Efficiency

Fig. 2.15 illustrates a typical course of the fuel consumption as a function of the output power P_{Diesel} ([28]) as well as the corresponding normalized efficiency η_{Diesel}. Here, the power axis is conveniently normalized to the rated power $P_{Diesel,r}$ and the fuel consumption $F(P_{Diesel})$ is normalized to the consumption at the rated power, $F(P_{Diesel,r})$. The graph gives rise to a linearization of the fuel consumption $F(P_{Diesel})$,

$$F(P_{Diesel}) = F(P_{Diesel,r}) \left(f_0 + f \frac{P_{Diesel}}{P_{Diesel,r}} \right) \tag{2.101}$$

with the dimension [volume/s]. Given the figures in [28] we have computed the linear regression coefficients to be $f_0 = 0.15$ and $f = 0.81$. This data may serve as long as no specific data are given. Summarizing we can say that the diesel should always be operated above a certain minimum load in order to maintain efficiency.

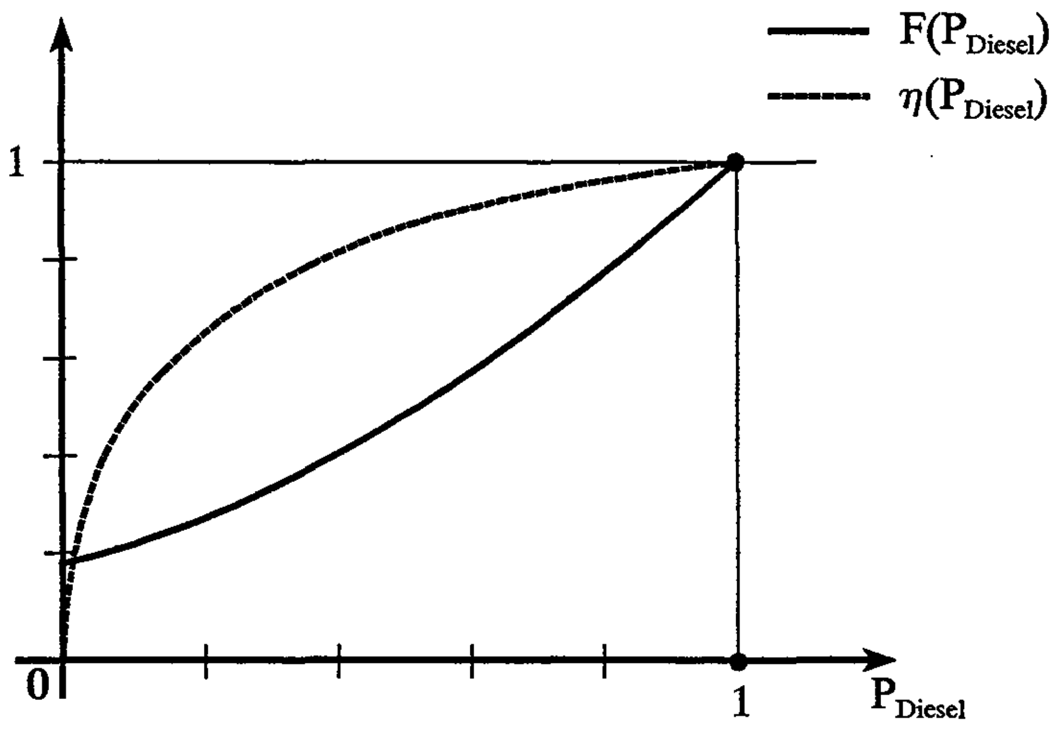

Fig. 2.15 Fuel Consumption and Efficiency

2.4.2 Lifetime Considerations

Operating the diesel under light load causes the engine oil to foul, thus leading to an increasing wear and consequently higher maintenance costs and shorter life span. A model of a diesel engine bearing wear has been proposed ([10]). At this stage we can, however, not envisage an efficient way of including these results into the theory presented here. For now we will therefore just bear in mind that the recommended load ranges between 50% and 80% for prolonged operation ([11]). This conclusion falls significantly short of the expectations aroused by the heading as we are still not able to quantize the influence of the load or the frequency of start/ stop- cycles on the lifetime or the maintenance factor of the diesel.

3. Power Supply

3.1 Wind Turbine

In the study presented here we assume that the operation of a wind turbine is described by its power- speed curve. In absence of a specific characteristic a model curve as shown in Fig. 3.1 will be used ([16]) :

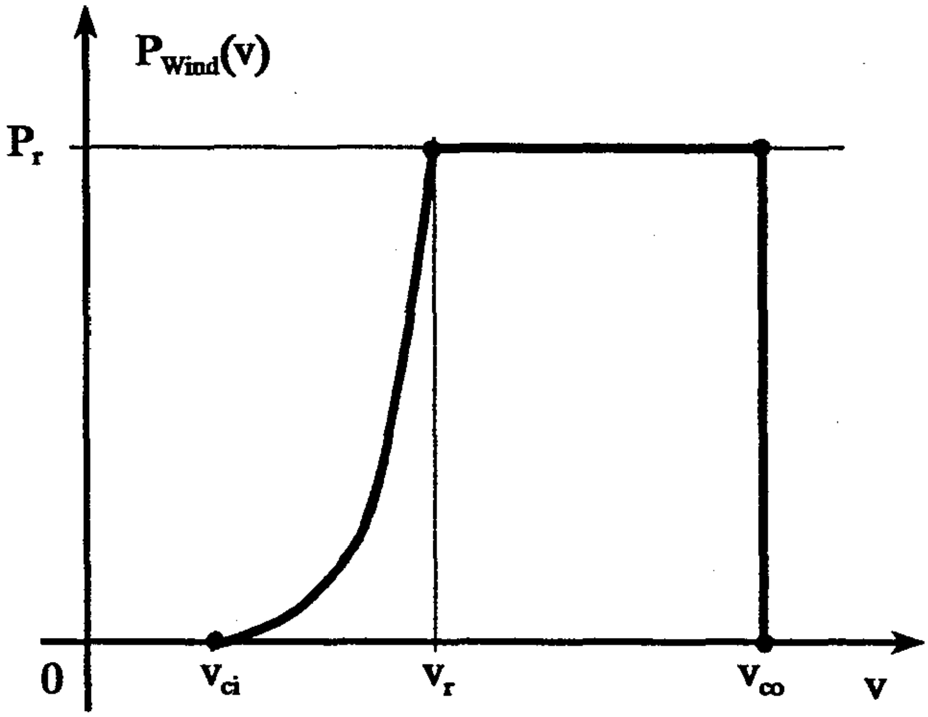

Fig. 3.1 P-v- Characteristic of a Wind Turbine

$$P_{turb}(v) = \begin{cases} 0 & v \leq v_{ci} \\ P_r\left(\dfrac{v-v_{ci}}{v_r-v_{ci}}\right)^3 & v_{ci} \leq v \leq v_r \\ P_r & v_r \leq v \leq v_{co} \\ 0 & v \geq v_{co} \end{cases} \tag{3.1}$$

Here, P_r is the rated power of the wind turbine, which is the power supplied by the turbine at the rated wind speed v_r. The wind speeds v_{ci} and v_{co} are called cut-in and cut-out speed respectively. They define the interval in which the wind generator is operated. If the turbine was operating at a wind speed below v_{ci}, the engine wear would be too big to operate in an efficient way. On the other side, the turbine is stopped in case of a wind speed above v_{co}. This is merely for economic reasons as an operation above v_{co} would require a more expensive turbine. P_{turb} is the power supplied by the turbine. The power that is actually available is further reduced by an efficiency factor η_w:

$$P_{Wind} = \eta_w P_{Turb} \tag{3.2}$$

3.2 The Photovoltaic Array

3.2.1 The Equivalent Circuit

An equivalent circuit of a single diode model of a solar cell (index j) is drawn in Fig. 3.2. The current generated by the incoming light is $I_{ph,j}$ and will be discussed in chapter 3.2.4.

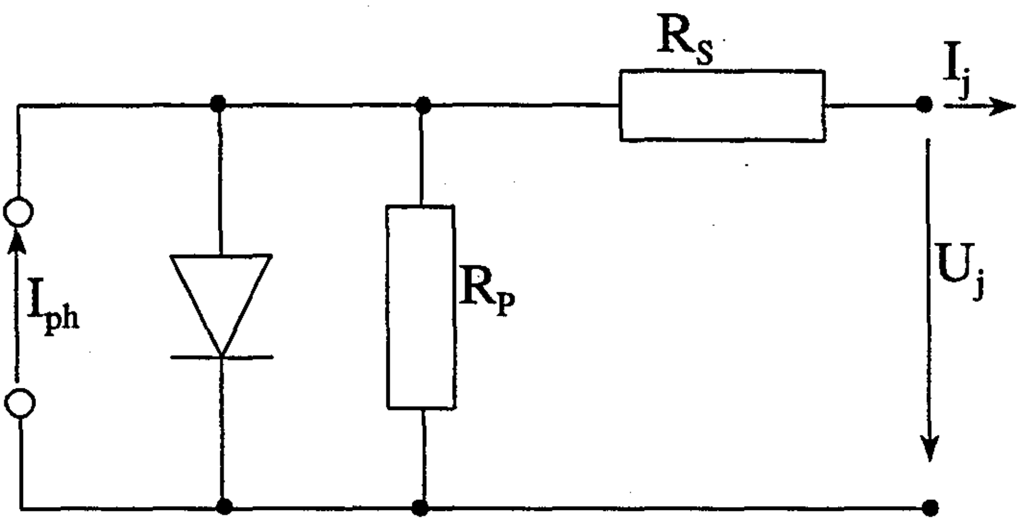

Fig. 3.2 Equivalent Circuit of a Solar Cell

R_p and R_s denote the parallel and the serial resistance. The diode is determined by its quality factor A (usually in the range of A $\in$ [1, 2]) and reverse saturation current I_{0j}. For an array of N_s serial and N_p parallel solar cells the I-U- characteristic is given by

$$I = I_{ph} - I_0\left[\exp\left(\frac{U+IR_s}{U_T}\right) - 1\right] - \frac{U+IR_s}{R_p} \tag{3.3}$$

where U_T symbolizes the thermal voltage

$$U_T = \frac{AkT_{cell}}{e} \tag{3.4}$$

with elementary charge e and cell temperature T_{cell}. The total series resistance R_s, photo current I_{ph} and reverse saturation current I_0 can be calculated from the values of the single cell via

$$
\begin{aligned}
I_{ph} &= I_{ph,j} N_p \\
I_0 &= I_{0,j} N_p \\
R_s &= R_{s,j} \frac{N_s}{N_p}
\end{aligned}
\tag{3.5}
$$

It is worth mentioning that R_p and R_s influence the characteristic in a significant way. Fig. 3.3 qualitatively sketches the impact of R_p and R_s. The continuous curve represents the ideal array with $R_s = 0$ and $R_p \to \infty$, whereas the dotted curves depict the effect of the impedances.

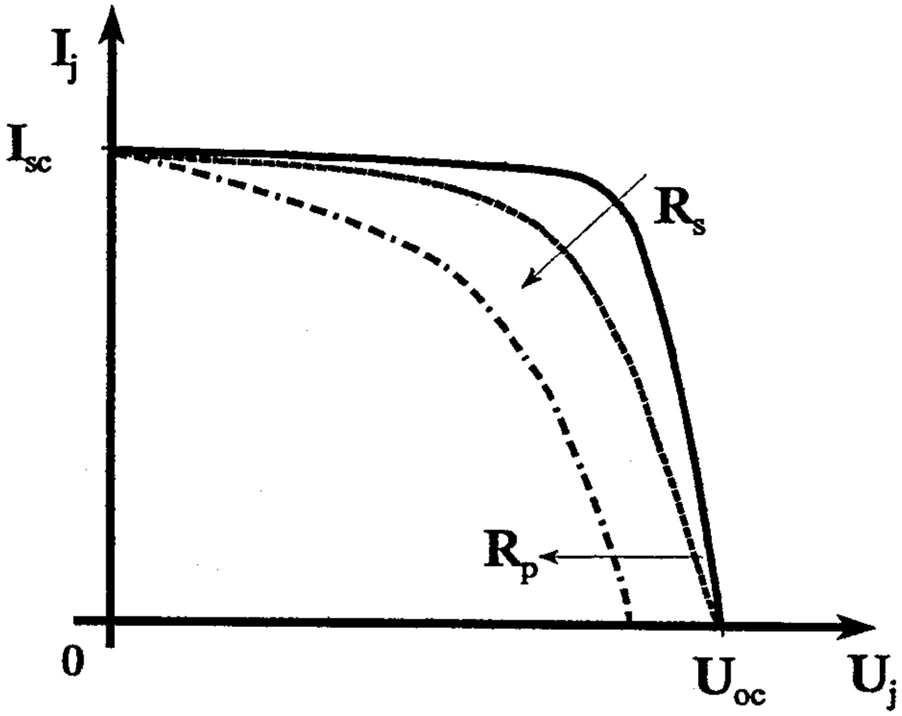

Fig. 3.3 I-U- Characteristic of a Solar Cell

3.2.2 PV Power Supply

The power supplied by the photovoltaic array, P_{sol}, is $P_{sol} = UI$, where I and U have to satisfy the characteristic (3.3). In order to find out the point (I_{mp}, U_{mp}) for which the maximum power $P_{mp} = I_{mp}U_{mp}$ is supplied by the array, we will simplify the equivalent circuit by omitting the parallel resistance R_p and we are then able to write the array voltage in the form

$$U = A\,U_T \ln\left[\frac{I_{ph} - I + I_0}{I_0}\right] - IR_s \tag{3.6}$$

The current at the maximum power point can be assessed by setting the current derivation of the power to zero and it is ([24], eq. 19)

$$I_{ph} = I_{mp} + I_0\left[\exp\left(\frac{2\,I_{mp}R_s}{A\,U_T} + \frac{I_{mp}}{I_{ph} - I_{mp} + I_0}\right) - 1\right] \tag{3.7}$$

Equation (3.7) has to be solved numerically for I_{mp}. U_{mp} can be determined by evaluating (3.6). The maximum power will then be the product of both.

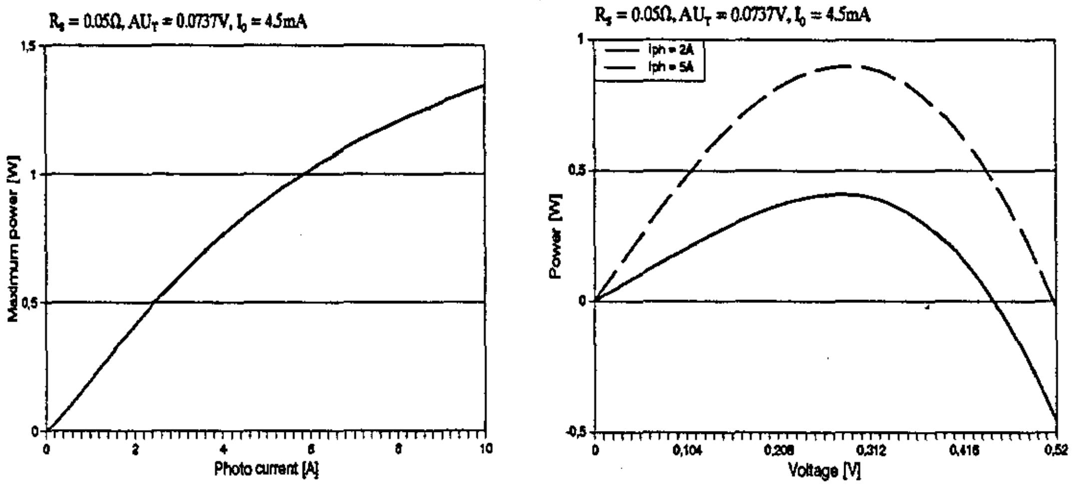

Fig. 3.4: Power characteristics of a solar cell

The diagrams in Fig. 3.4 demonstrate the dependency of the maximum power point as a function of the voltage (right hand side) and the photo current (left hand side). Having assumed typical values R_s = 0.05 Ω, AU_T = 0.0737 V and I_0 = 4.5 mA we have calculated the maximum power point for given photo currents using the method described above. Some values are presented in Tab. 3.1

I_{ph} [A]	0.0	1.25	1.875	2.5	3.125	3.75	5.0
P_{mp} [W]	0.0	0.25	0.375	0.5	0.625	0.7	0.9

Tab. 3.1: Photo current versus maximum power

The values in Tab. 3.1 give rise to the presumption of a linear relation between P_{mp} and I_{ph}. Not quite. The linear approximation is only legitimate for sufficiently small photo currents. Towards larger values of I_{ph} the power curve will significantly flatten out as outlined in Fig. 3.4.

In practice, a maximum power tracker may be inserted between the photovoltaic array and the load (i.e. the DC- bus) in order to ensure optimum operation. A maximum power tracking facility is an adjustable ratio DC to DC transformer which basically contains a parallel high frequency MOSFET switch. It provides a matching between the load and the photovoltaic array such that the solar cell is operated in the maximum power point. In general maximum power point trackers can be classified into step-down trackers ([36]) and step-up trackers ([35]). The first one drives a high voltage load from a low voltage PV array whereas the latter one operates vice versa.

It is, however, suggested ([23] p.434) that an MPP tracker does not pay off in case it requires additional hardware. Jantsch ([23]) reports a best fixed voltage system which yields an annual energy output of 98.4% of an MPP operated system.

For the purpose of this paper we assume that a reasonably good power tracker (with efficiency η_{mpt}) is in charge. The power delivered by the solar cell will then be reduced to

$$P_{sol} = \eta_{mpt} P_{mp} \tag{3.8}$$

In case no MPP tracker was used, the factor η_{mpt} would summarily cover the expected losses, caused by the lack of an MPP tracker.

3.2.3 Temperature Dependency

Unlike the wind turbine the solar cell characteristics vary sensitively with the temperature. In general, the cell efficiency will decrease upon increasing temperature. The influence of the temperature can be included in equations (3.3) and (3.7) by applying ([24] eq. 16-18)

$$\begin{aligned} I_0(T) &= I_0(T_r)\left(\frac{T}{T_r}\right)^3 \exp\left(-b\left(\frac{1}{T}-\frac{1}{T_r}\right)\right) && b = 4400 \\ I_{ph}(T) &= I_{ph}(T_r)\left(1 + a(t-T_r)\right) && a = 5.7E\text{-}4 \\ U_T(T) &= U_T(T_r)\,\frac{T}{T_r} \end{aligned} \tag{3.9}$$

where T_r is a reference temperature (usually 25°C). If hourly mean temperature values throughout the year are given, we will employ (3.9). Otherwise, the values at reference point are used. However, calculations in this paper have been carried out without taking the temperature dependency into account.

3.2.4 Photo Current and Efficiency

Only a fraction of the energy of the incoming light can be converted into electric energy for several reasons:

- Photons with an energy $h\nu < E_g$ (E_g stands for the minimum band gap of the semi conductor) will not be absorbed.
- The surplus energy of absorbed photons will be thermalized, thus causing even a further reduction of the efficiency as temperature rises.
- Not every generated electron contributes to a voltage eE_g.
- Already absorbed electrons are likely to be recombined, especially if they are close to surfaces.
- Even if the light beam and the array surface were perpendicular a reflexion would be caused due to the different refraction indices of the air and the semi conductor.

For the purpose of this paper, however, we are content to introduce an efficiency factor ζ_{sol}

that summarizes all the mentioned processes and assume a linear relationship

$$I_{ph} = A\ \zeta_{sol}\ \Phi_{\perp} \tag{3.10}$$

between the photo current and the product of the intensitiy of the perpendicular light $\phi_{\perp}$ and the active array area A (not to be confused with the diode factor A introduced previously). For a silicon solar cell, for example, it is $\zeta_{sol} \approx 0.28\ AW^{-1}$ ([9] p.73).

3.3 Combined Renewable Power

The renewable power supply consists of both the wind power (3.2) and the solar power (3.8). As far as the photovoltaic array is concerned, we assume a linear relationship between the maximum output power and the photo current (chapter 3.2.2). Taking (3.10) into account, a linear relationship between the solar power P_{sol} and the clearness index k (see chapter 2.2) ,

$$P_{sol} = \xi_{sol}\ k \tag{3.11}$$

can be concluded. The maximum power will be supplied by the photvoltaic array if the clearness index reaches its maximum. Suppose the maximum clearness index is K_0. This coefficient can be used to normalize the solar power,

$$p_s = \min\left\{\frac{P_{sol}}{\xi_{sol}\ K_0}\ ,\ 1\right\} = \min\left\{\frac{k}{K_0}\ ,\ 1\right\} \tag{3.12}$$

for simplification of further calculations. The *min-* operator is used to ensure that the normalized power is within the range $p_s \in [0,1]$. For a clearness index $k > K_0$ the power output will not increase as the system is in saturation. In the same manner, the wind turbine power (3.1), (3.2) is normalized to the rated power,

$$p_w = \frac{P_{wind}}{\eta_w\ P_r} = \frac{P_{turb}}{P_r} \tag{3.13}$$

The total renewable power, P_{ren}, is $P_{ren} = P_{wind} + P_{sol}$. Its maximum $P_{ren,max}$ is reached when the wind turbine is operated in its rated power and the clearness index is $k = K_0$. Hence, the maximum is $P_{ren,max} = \xi_{sol}K_0 + \eta_w P_r$ .Introducing the dimensionless parameter

$$\zeta = \frac{1}{1 + \frac{\eta_w P_r}{\zeta_{sol} K_0}} \tag{3.14}$$

an elegant normalized expression for the total renewable power is given by

$$p_{ren} = \frac{P_{ren}}{P_{ren,max}} = \zeta \, p_s + (1 - \zeta) \, p_w \tag{3.15}$$

The normalized parameters p_s, p_w and p_{ren} are dimensionless numbers in the interval [0,1]. In the next chapter we will resume the discussion from chapter 2 by extending the statistical models to the normalized renewable power.

4. Statistical System Modelling

The previous chapter was concerned with the modelling of the electric power supplied by the various components of the system. Assume for the moment that all components are linked together in one system. The output of the system, which is the total power, is obviously depependant on a huge variety of parameters, that can be categorised:

(i) **Fixed Parameters**

Fixed parameters do not change their value during operation of the system. For example, the choice of a wind turbine determines cut-in, cut-out and rated wind speed. Once the wind turbine is chosen, they can not be altered.

(ii) **Random Input Parameters**

Random input parameters are the wind speed, v, the clearness index, k, and the external power demand, due to their very nature.

(iii) **Derived Random Parameters**

Derived random parameters are parameters that depend on the random input parameters. For instance, the mean wind speed.

(iv) **Controller Dependant Parameters**

These are parameters whose values are influenced by the controller. For instance, the state of charge of the battery falls into this category as the controller determines whether to charge or discharge the battery.

Please note that the parameter categories listed here are not mutually exclusive. The state of charge, for example, is both a derived random and a controller dependant parameter. The intention of the categorisation is much more to focus on the fact that, although concise models for the power supply have been developed, the behaviour of many a parameter is all but fixed. Due to the statistical nature of wind speed and clearness index, the whole system is a non- deterministic system, which can only be described employing statistical methods. There are several reasons for doing this.

First, it leads to a better appreciation of the influence of both the random input parameters and fixed parameters on the system.

Second, synthetic time series of the power output can be used for an off-line optimization of some of the fixed parameters. For instance, the fractional power factor (i.e. the ratio

between rated wind and rated solar power) could be optimized off-line for given (typical) wind and clearness index data taken at the site in question.
Third, statistical methods can be used to predict the power supplied by the renewable energy sources or the state of charge for given observations of the random input parameters and the state of charge. Again, this might be interesting for a better understanding of what is going on in the system. Though, there is another reason. As mentioned in the introduction (section 1), the main purpose of the controller in this hybrid system is to be in charge of the battery (charging, discharging or disconnecting) and the diesel (switch on and off). Statistical methods could be used to design the controller, which is not covered in this paper. For instance, various control policies could be compared off-line by generating time series. Later in this chapter, a very crude battery control policy is applied to generate time series of the state of charge. In this instance, the battery is being discharged (if possible) as soon as the renewable energy sources can not meet the power demand and it is always charged at times when there is a surplus. Other, more sophisticated policies can be easily implemented (or incorporated in the programme) as the important tools are developed here. The controller could, however, as well use statistical methods (e.g. first passage times) on-line and decide depending on those values. Hence, the methods developed here can be used at design stage as well as during operation.

This chapter is divided into three sections, of which the first is concerned with distribution functions. The second section covers the generation of synthetic time series of the power supplied by the renewable energy sources and the state of charge of the battery. The last section takes a deeper look at first passage time problems.

4.1 Distributions

The purpose of this section is to introduce the probability distribution functions of some stochastic processes that occur in the system. The first part is devoted to the wind speed. It is only included because the mathematical functions involved are simple, thus helping to appreciate the formalism and methods. The main emphasis however, is placed on the wind turbine power and the photovoltaic array power. The discussion on distributions closes with the distribution of the joint renewable power, which is the sum of the power supplied by the wind turbine and the photovoltaic array.

4.1.1 Wind Speed Distribution

Let us first recall the conditional probability density function $f_v(v \mid v(0) = v_0)$ of the wind speed v from equation (2.10), here in unnormalized form,

$$f_v(v \mid v(0) = v_0) = \frac{1}{\sqrt{2\pi\sigma_v^2(1-r_v^2)}} \exp\left[-\frac{1}{2}\left(\frac{v-(\bar{v}+(v_0-\bar{v})r_v)}{\sigma_v\sqrt{(1-r_v^2)}}\right)^2\right] \tag{4.1}$$

with the corresponding distribution function

$$F_v(v \mid v(0) = v_0) = \Phi\left(\frac{v-(\bar{v}+(v_0-\bar{v})\,r_v)}{\sigma_v\sqrt{1-r_v^2}}\right) \tag{4.2}$$

Fig. 4.1 and Fig. 4.2 depict the probability density and the corresponding distribution function of the wind speed fluctuations for a mean wind speed of 16 m/s and three values for the standard deviation σ_v, where stationarity is assumed (I.e. $r_v = 0.0$). For each graph 50 values have been calculated. Both pictures clearly display the influence of the standard variation. Increasing the standard deviation has the effect of increasing the probability for wind speed values that are further away from the mean.

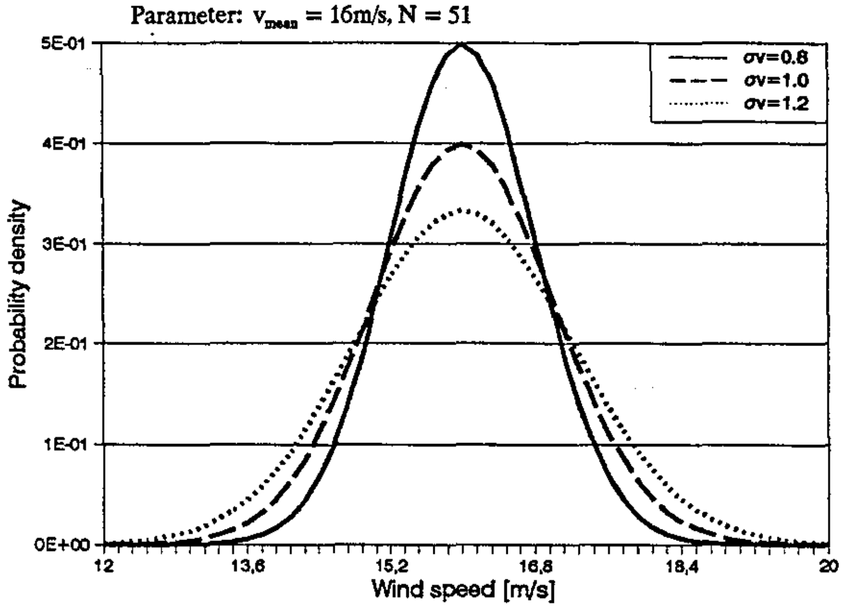

Fig. 4.1 Wind Speed Probability Density Function

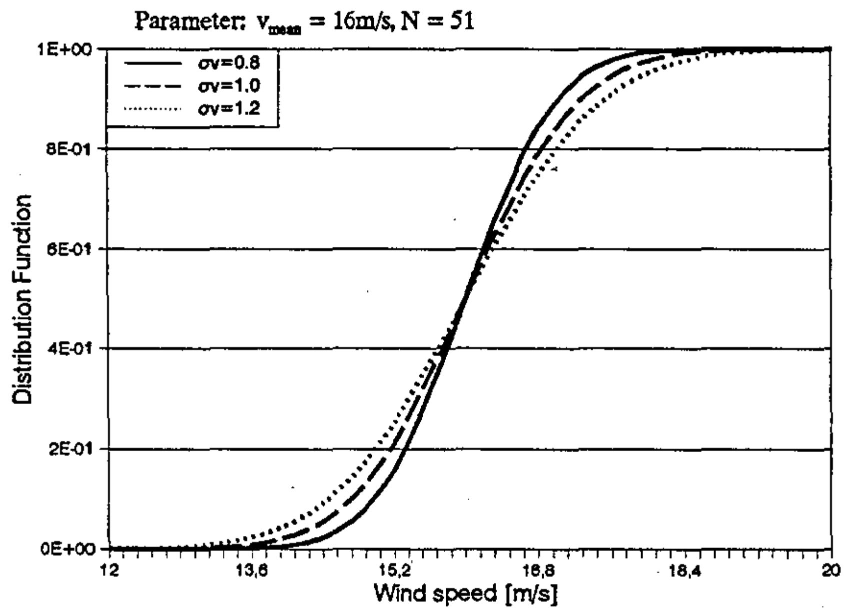

Fig. 4.2 Wind Speed Distribution Function

4.1.2 Wind Turbine Power Distribution

If we consider the random variable to be the input of the wind turbine characteristic (3.1) the distribution function of the normalized wind power p_w (eq. 3.13) can be expressed in terms of $F_v(v \mid v_0)$[7] (eq. (4.2)),

$$F_{pw}(p_w \mid v_0) = \begin{cases} 0 & p_w < 0 \\ F_v\left(v_{ci} + \sqrt[3]{p_w}(v_r - v_{ci}) \mid v_0\right) - F_v(v_{co} \mid v_0) + 1 & 0 \le p_w \le 1 \\ 1 & p_w > 1 \end{cases} \tag{4.3}$$

Here, the short hand $F_{pw}(p_w \mid V(0) = v_0) = F_{pw}(p_w \mid v_0)$ is used. Most conditional distribution functions in this paper are referred to by this notation. The wind power probability density function is attained by derivation:

$$f_{pw}(p_w \mid v_0) = \begin{cases} 0 & p_w < 0, p_w > 1 \\ F_v(v_{ci} \mid v_0)\ \delta(p_w) & \\ \quad + \left(F_v(v_{co} \mid v_0) - F_v(v_r \mid v_0)\right)\ \delta(p_w - 1) & 0 \le p_w \le 1 \\ \quad + \dfrac{v_r - v_{ci}}{3\sqrt[3]{p_w^2}}\ f_v\left(v_{ci} + \sqrt[3]{p_w} \mid v_0\right) & \end{cases} \tag{4.4}$$

Since the wind turbine P(v) characteristic (eq. 3.1) is not differentiable at $v = v_r$ and $v = v_{co}$, the distribution function $F_{pw}(v)$ reveals discontinuities at $p_w = 0$ and $p_w = 1$. This explains the emergence of the Dirac- function in the probability density function. In order to avoid these computational problems connected with the Dirac function, the power scale will be discretized,

$$p_{w,n} = \frac{n-1}{N-1} \quad , n = 1 \ldots N \tag{4.5}$$

where N power levels are allowed. As the power is now a discrete random variable, its distribution function will be a stair function with the distinct values

[7] Refer to chapter 6.2, for a discussion of functions of random variables.

$$G_{pw}(n \mid v_0) = F_{pw}\left(\frac{n-1}{N-1} \mid v_0\right) \tag{4.6}$$

The probability density function will now be replaced by a discrete probabilty function with values

$$g_{pw}(n \mid v_0) = \begin{cases} G_{pw}(1 \mid v_0) & n=1 \\ G_{pw}(n \mid v_0) - G_{pw}(n-1 \mid v_0) & n=2...N \end{cases} \tag{4.7}$$

The value $g_{pw}(n \mid v_0)$ is the probability that - at time t - a power output $p_w \in [p_{w,n-1}, p_{w,n}]$ may be observed under the condition that the wind speed was $v(0) = v_0$ at time $t = 0$. Summing up all $g_{wp}(n \mid v_0)$ over n yields 1.
The discussion of the functions $g_{pw}(n \mid v_0)$ and $G_{pw}(n \mid v_0)$ is conducted in two parts. First, we restrict ourselves to the stationary case. This is when the correlation coefficient r marches towards 1. Hence, the initial value has no influence on the stationary distribution.

4.1.2.1 Stationary Distribution

Fig. 4.3 shows probability functions $g_{pw}(n \mid v_0)$ for four different mean wind speeds as functions of the normalized power with N = 51 (4.5). For the rated wind speed, cut-in speed, cut-out speed and the standard variation σ_v typical values have been assumed. These constant parameters are displayed above each diagram. In Fig. 4.3 the values for $p_s = 1$ are omitted because of their magnitude. The curve with $\bar{v}$ = 18m/s for instance has a high probability for maximum power 1 eventhough it is not explicitly displayed. A better representation is therefore Fig. 4.4 where the corresponding distribution functions $G_{pw}(n \mid v_0)$ are depicted. For a mean wind speed that is well below the rated wind speed ($\bar{v}$ = 12m/s in comparison to v_r = 16m/s) the shape of the probability function of the wind turbine power is almost the same as the one for the wind speed itself as the maximum power (p = 1) is very unlikely. Increasing the wind speed increases the probability for maximum power which causes the distribution function to jump to 1 at p = 1. Mathematically, this is due to the fact that the probability function is not zero at p = 1. Physically, the reason for this is that a whole continuum of wind speed values do cause the same power, the maximum power (eq. 3.1).

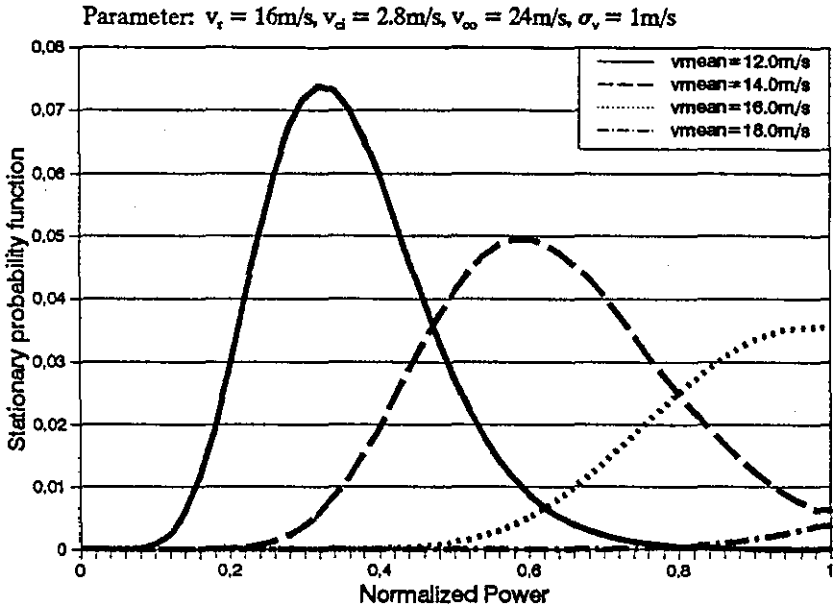

Fig. 4.3 Wind Turbine Power: Stationary Distribution

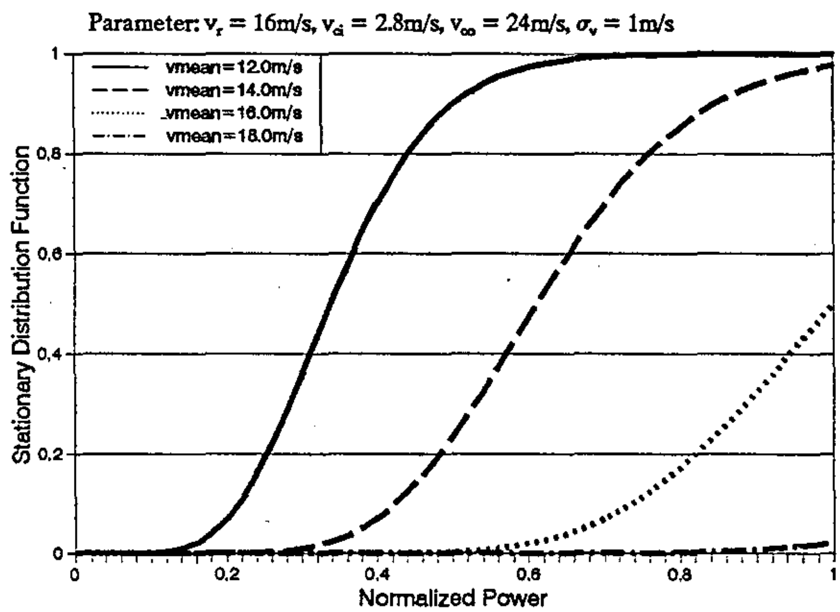

Fig. 4.4 Wind Turbine Power: Stationary Distribution

In Fig. 4.5 the probability function is shown for different rated wind speeds. It makes clear that the variation of the rated wind speed is on a par with the variation of the mean wind speed. The set of curves is almost identical to the set in Fig. 4.3.

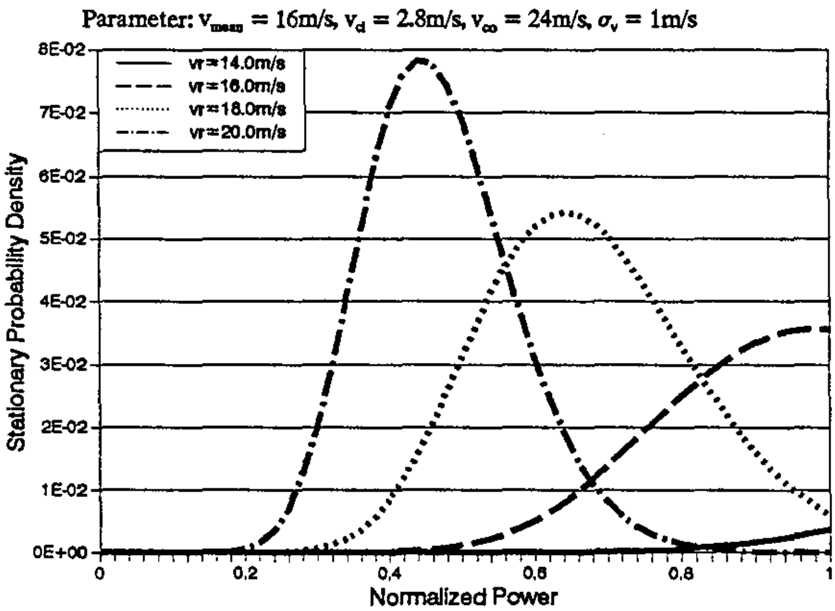

Fig. 4.5 Wind Turbine Power: Stationary Distribution

Eventually, Fig. 4.6 shows the influence of the standard variation σ_v. For comparison, the curve with σ_v = 1m/s is included in Fig. 4.5. As expected the probability curve becomes flatter while increasing the standard variation. A significant aspect is the increased probability at zero power in the σ_v = 4m/s curve. This is forced by the cut-out wind speed below which the turbine power is zero, eventhough the wind speed is not. Again, the values at p = 1 are omitted for the sake of a reasonable scale.

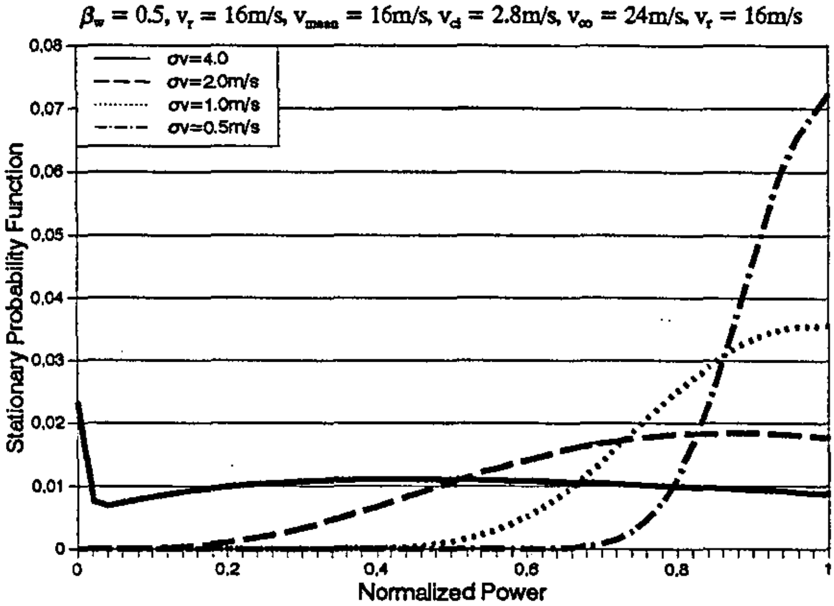

Fig. 4.6 Wind Turbine Power: Stationary Distribution

Fig. 4.7 shows the corresponding distribution functions including the jumps at p = 1. Note that the height of the jump at p = 1 is equal to the probability that the system delivers maximum power.

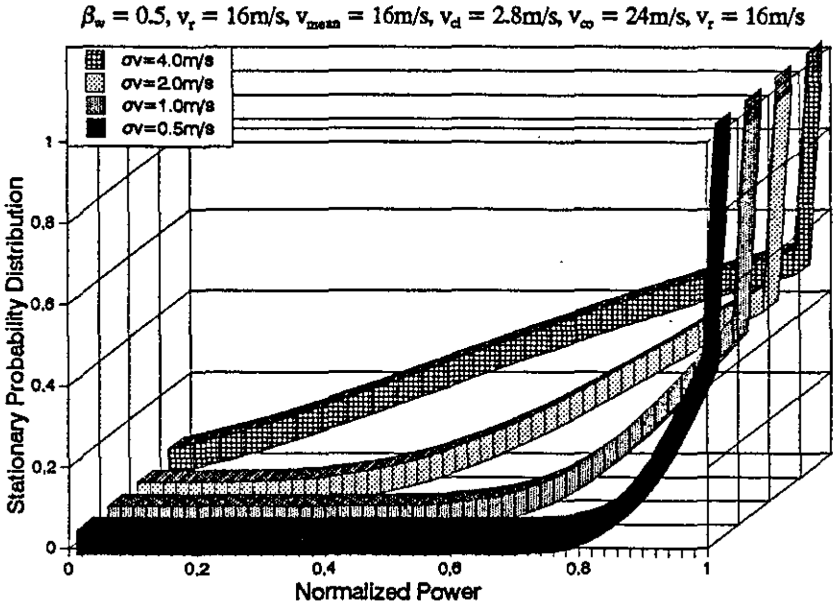

Fig. 4.7 Wind Turbine Power: Stationary Distribution

4.1.2.2 Conditional Distribution

Having examined the stationary case light is now shed on the conditional probability distribution. Fig. 4.8 illustrates the impact of the time on the probability function. The initial wind speed was chosen to be 12m/s, well below the mean wind speed. At time t = 0 the wind speed is known. Hence, the probability for one particular power value is one. As time goes by the range broadens and its peak moves towards the peak which corresponds to the mean wind speed. In fact, the stationary solution for this particular setting is included in Fig. 4.3. The graphical representation of $g_{pw}(n|v_0)$ in Fig. 4.8 emphasizes the fact that the power scale is discretized. It is worth pointing out that the time scale is not necessarily a typical one. Throughout the paper the time always appears in the product βt in the autocorrelation function. By chosing a different β the time scale will vary accordingly. For the calculations of the probability distributions an arbitrary value $\beta_w = 0.5s^{-1}$ is assumed. Different values, however, do not affect the results.

$\beta_w = 0.5$, $v_r = 16m/s$, $v_{mean} = 16m/s$, $v_{ci} = 2.8m/s$, $v_{co} = 24m/s$, $\sigma_v = 1m/s$, $v_0 = 12m/s$

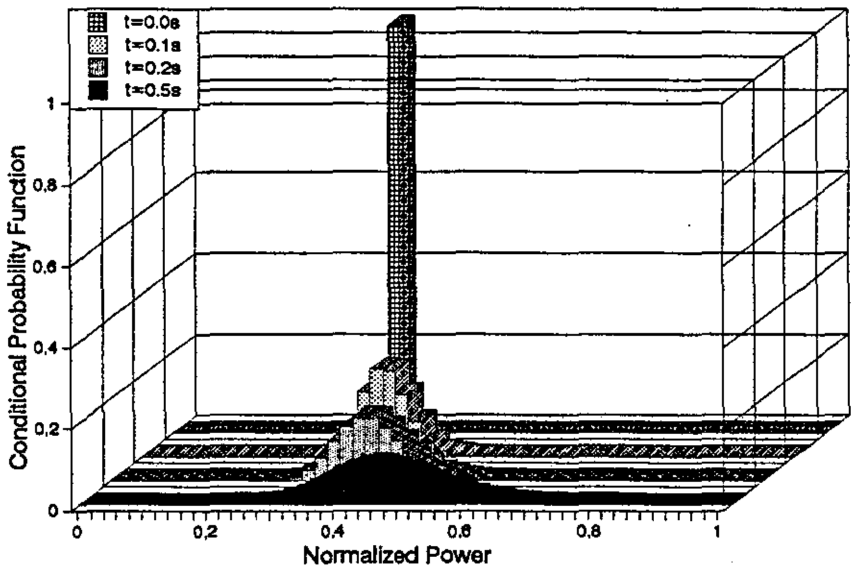

Fig. 4.8 Wind Turbine Power: Conditional Distribution

In Fig. 4.9 the probability function is shown for a set of initial wind speeds at one particular time t = 0.2s. For cross reference, the curve with initial wind speed v(0) = 12m/s is also included in Fig. 4.8. Bearing in mind that both the rated wind speed and the mean wind speed are 16m/s it is clear why the curve with initial wind speed v(0) = 18m/s is virtually zero anywhere except at maximum power p = 1.

$\beta_w = 0.5, v_r = 16m/s, v_{mean} = 16m/s, v_{ci} = 2.8m/s, v_{co} = 24m/s, \sigma_v = 1m/s, t = 0.2s$

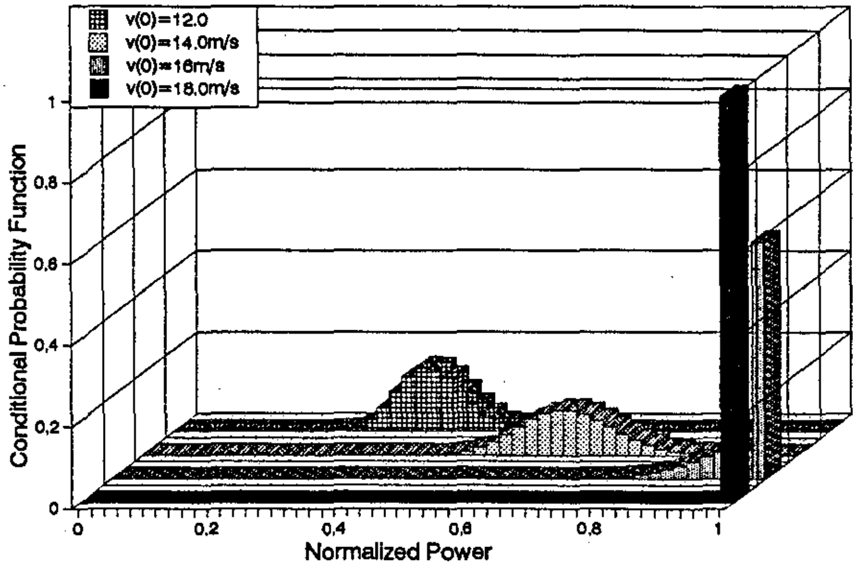

Fig. 4.9 Wind Turbine Power: Conditional Distribution

The corresponding distribution functions $G_{pw}(n \mid v_0)$ are depicted in Fig. 4.10.

$\beta_w = 0.5, v_r = 16m/s, v_{mean} = 16m/s, v_{ci} = 2.8m/s, v_{co} = 24m/s, \sigma_v = 1m/s, t = 0.2s$

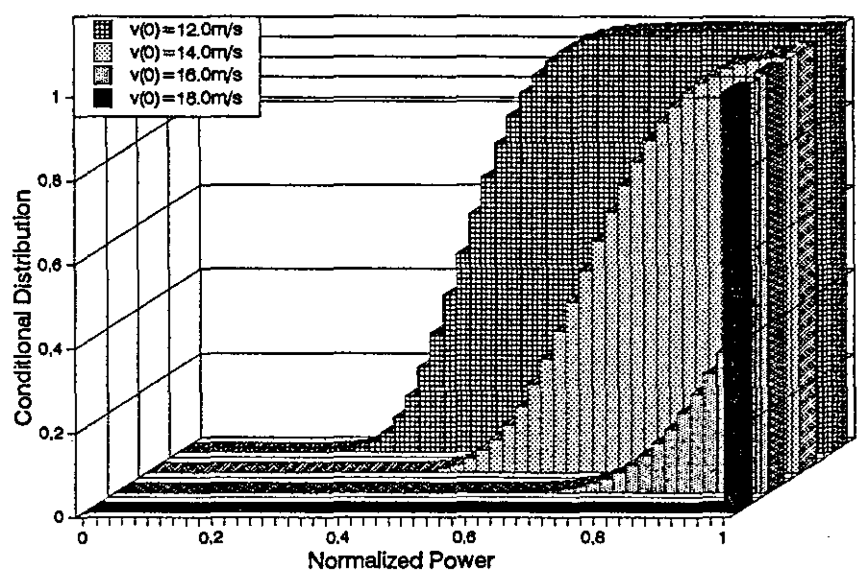

Fig. 4.10 Wind Turbine Power: Conditional Distribution

Fig. 4.11 displays curves with different mean wind speeds for an initial wind speed v_0 = 12m/s at time t = 0.5s. This is the same setting as in Fig. 4.3. That means that the curves in Fig. 4.11 move to Fig. 4.3 for $t \to \infty$. The interpretation is simple. Under higher mean wind speeds the system moves more quickly to higher power values than under lower mean wind speeds.

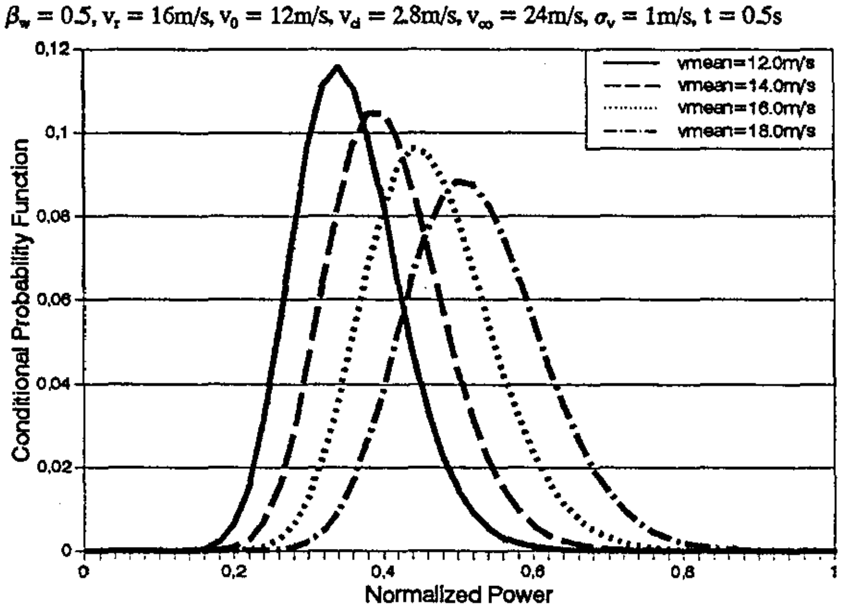

Fig. 4.11 Wind Turbine Power: Conditional Distribution

4.1.3 PV Array Power Distribution

4.1.3.1 Stationary Distribution

In analogy to chapter 4.1.2 the normalized solar power scale (3.12) will be discretized by

$$p_{s,n} = \frac{n-1}{N-1} \quad , n=1 \ldots N \tag{4.8}$$

With the normalizations (4.8) and (2.70) and assuming the linear relationship (3.12), the stationary distribution function $F_{ps}(n)$ can be phrased in terms of the distribution function $F_x(x)$ (eq. 2.79),

$$G_{ps}(n) = F_x\left(\frac{K_0\frac{n-1}{N-1}-k_{min}}{k_{max}-k_{min}}\right) \tag{4.9}$$

Similar to (4.7), a stationary probability function can be obtained from

$$g_{ps}(n) = \begin{cases} G_{ps}(1) & n=1 \\ G_{ps}(n) - G_{ps}(n-1) & n>1 \end{cases} \tag{4.10}$$

The stationary probability function $g_{ps}(n)$ is shown in Fig. 4.12 for 3 different clearness indexes and a constant standard variation σ_k. Again, the power is divided up into N = 51 values. The maximum (normalized) clearness index K_0 = 1.3 is assumed. Fig. 4.12 reveals that the probability function has in general two maxima due to the superposition of 2 beta-distributions. A higher clearness index moves both maxima towards maximum power. At the same time the peak of the low power maximum decreases in favor of the high power maximum.

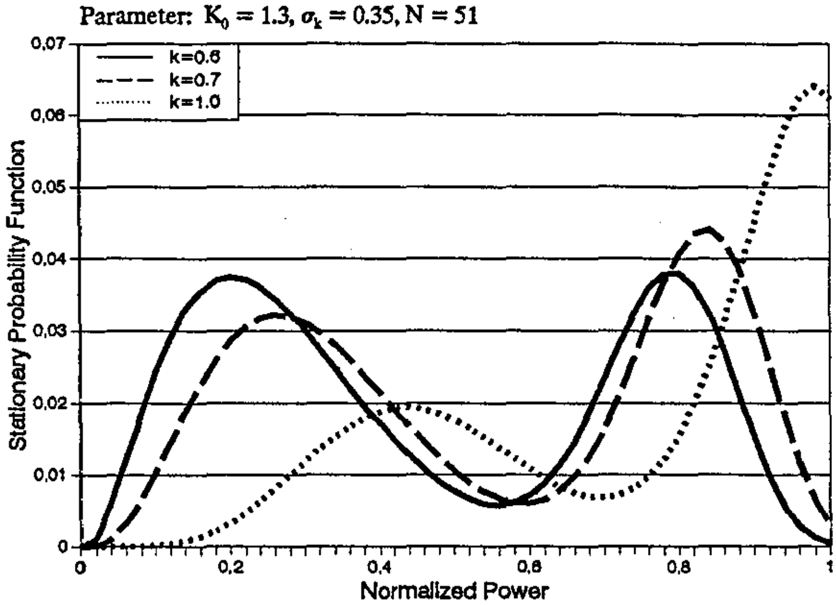

Fig. 4.12 PV Array Power: Stationary Distribution

This becomes even clearer in Fig. 4.13, where the corresponding distribution functions $G_{ps}(n)$ are drawn. This is an interesting result. Obviously, the system has two preferential points, neither of whom is the average. Hence, it is expected that the solar power sometimes may change rather abruptly by jumping from one to the other peak. In fact, this can be seen in the discussion of time series in chapter 4.2.

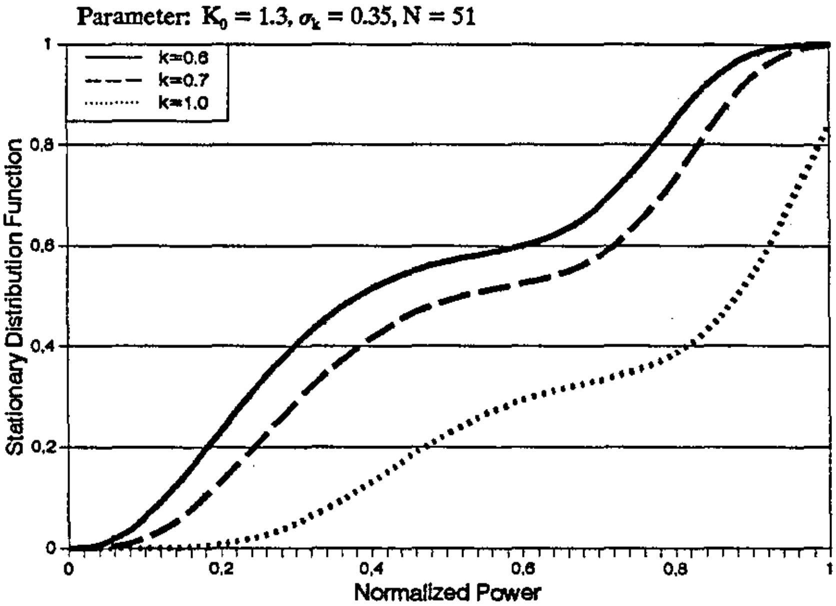

Fig. 4.13 PV Array Power: Stationary Distribution

In the graph depicting the distribution functions, the curves with higher clearness index are below the ones with a smaller k. Remember that any distribution function F(x) returns the probability that the system is in a state less than or equal x.

The impact of the standard variation σ_k is illustrated in Fig. 4.14 and Fig. 4.15. In the event of a small standard deviation σ_k the probability function is very much centered having one peak only. Larger values cause the two peaks that are mentioned earlier to seperate and drift apart towards minimum and maximum power respectively.

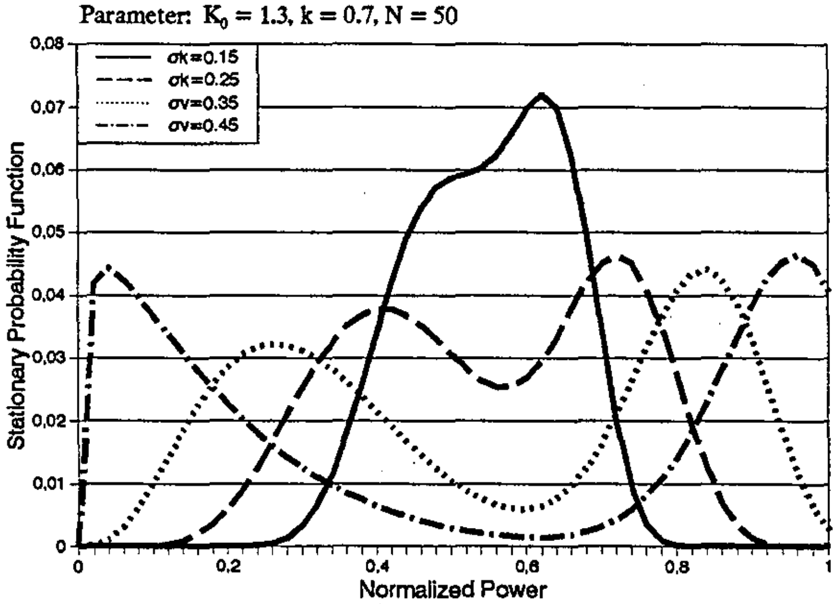

Fig. 4.14 PV Array Power: Stationary Distribution

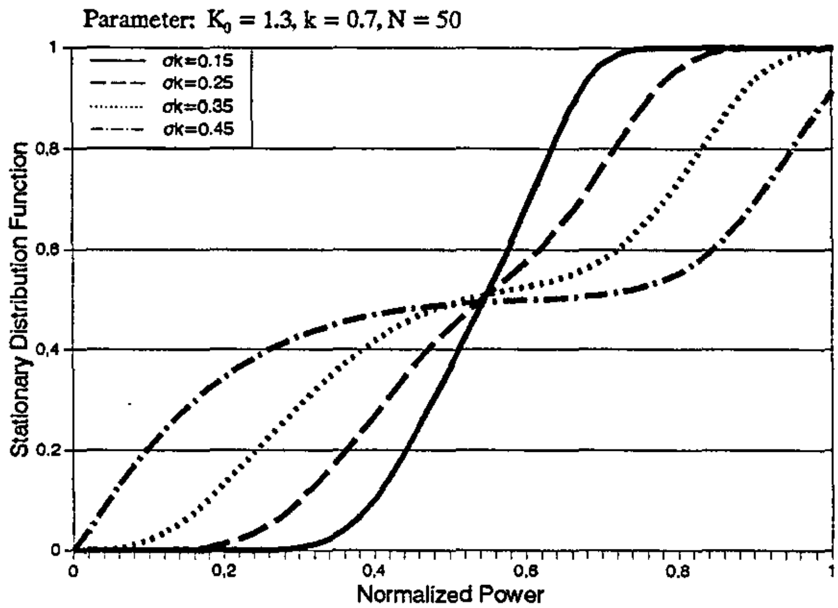

Fig. 4.15 PV Array Power: Stationary Distribution

The matching distribution functions in Fig. 4.15 disclose yet another pecularity. The power at which the distribution function is 0.5 is independent of the standard deviation. This power point is only a function of k and K_0. Hence, σ_k has an impact on the weighting and not on the average.

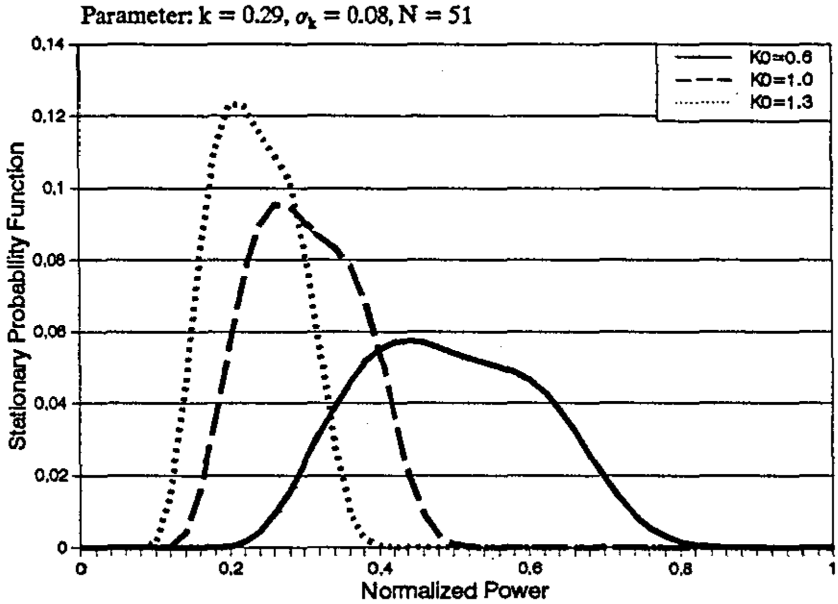

Fig. 4.16 PV Array Power: Stationary Distribution

In Fig. 4.16 curves are drawn with the maximum clearness index K_0 as parameter. This parameter was introduced in (3.12) to normalize the power scale. It specifies the clearness index above which the maximum power is attained. In general, a larger K_0 permits higher power values and broadens the shape of the probability function. It has, however, no effect on the qualitative course of the curve.

4.1.3.2 Conditional Distribution

In analogy to the previous section the discrete, conditional distribution function $\hat{G}_{ps}(n \mid k_0)$ of the solar power can be phrased in terms of the approximating conditional distribution function $F_x(x)$ (eq. 2.91):

$$\hat{G}_{ps}(n\,|\,k_0) = \hat{F}_x\left(\frac{K_0 \frac{n-1}{N-1} - k_{min}}{k_{max} - k_{min}} \;\middle|\; \frac{k - k_{min}}{k_{max} - k_{min}} \right) \tag{4.11}$$

Similar to (4.10) the discrete, conditional probability function can be obtained from

$$\hat{g}_{ps}(n\,|\,k_0) = \begin{cases} \hat{G}_{ps}(1\,|\,k_0) & n=1 \\ \hat{G}_{ps}(n\,|\,k_0) - \hat{G}_{ps}(n-1\,|\,k_0) & n>1 \end{cases} \tag{4.12}$$

Please note that the hat on $\hat{g}_{ps,n}$ signals that the normal distribution expansion (2.82) is applied. For large time values $t \to \infty$, $\hat{G}_{ps,n}$ represents the unconditional distribution function as the autocorrelation coefficient r_x touches zero.

Parameter: $K_0 = 1.3$, $k = 0.7$, $\sigma_k = 0.35$, $k(0) = 0.1$, $\beta_s = 0.5$, $N = 51$

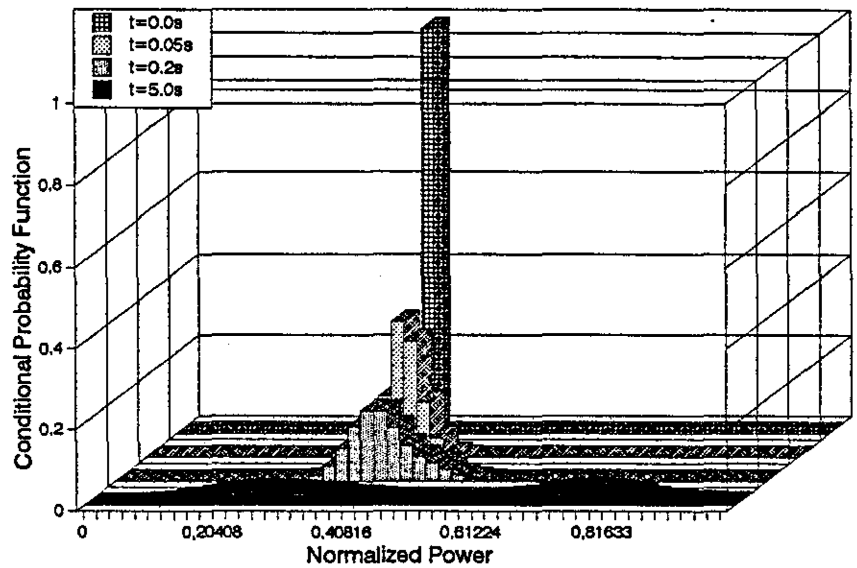

Fig. 4.17 PV Array Power: Conditional Distribution

In Fig. 4.17 an initial clearness index $k(0) = 0.1$ is assumed. The diagram shows the conditional probability function at 4 different times. Again, at time $t = 0$ the probability to observe the power value that corresponds to k(0) is 1. Later, the main bulk of the probability function moves on to higher power values.

Another interesting feature is the variation of the initial value k(0) as displayed in Fig. 4.18. It is no accident that the three curves have the same shape. It can be concluded from the conditional distribution function (2.91) that

$$\hat{F}_X(x|x_1) = \hat{F}_X((x_1 - x_0)\, r + x \,|\, x_1) \tag{4.13}$$

Hence, any variation of the initial clearness index can be translated into a shift along the clearness index axis. It finds its manifestation in Fig. 4.18.

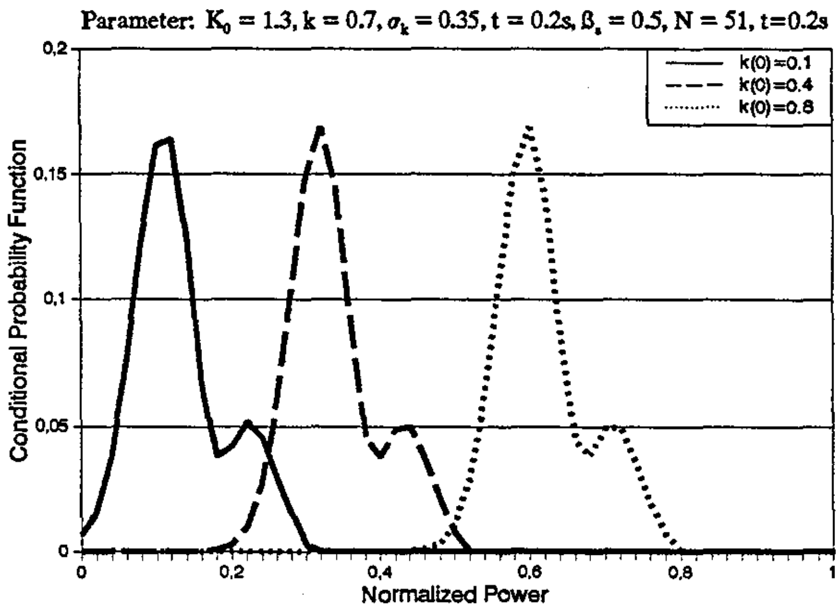

Fig. 4.18 PV Array Power: Conditional Distribution

4.1.4 Combined Power Distribution

Given the conditional probability functions for the wind turbine power (eq. (4.7)) and the solar power (eq. (4.12)), the total renewable power p_{ren} (eq. 3.15) can be obtained via convolution[8] if the stochastic processes of the wind speed and the clearness index are thought to be independent. Precedent to that let us denote the probability functions of (ζp_s) and ($1 - \zeta p_s$),

$$h_{ps}(n) = \hat{g}_{ps}\left(\frac{n}{\zeta} \mid k_0\right) \quad , \quad h_{pw}(n) = g_{pw}\left(\frac{n}{1-\zeta} \mid v_0\right) \quad , \quad n=1 \ldots N \tag{4.14}$$

before we can write the discrete probability function $g_{pren}(n \mid v_0,k_0)$ subject to the initial conditions $v(0) = v_0$ for the wind speed and $k(0) = k_0$ for the clearness index,

$$g_{Pren}(n \mid v_0,k_0) = \sum_{j=1}^{N} h_{pw}(j)\ hps(n-j) \quad , \quad n=1 \ \ldots \ N \tag{4.15}$$

The calculation of the convolution can be considerably speeded up by using the distribution functions rather than the probability functions. Hence, we define

$$H_{pw}(i) = \begin{cases} 0 & i<1 \\ G_{pw}\left(\dfrac{i}{1-\zeta} \mid v_0\right) & 1 \leq i \leq N \\ 1 & i>N \end{cases} \tag{4.16}$$

and

$$H_{ps}(i) = \begin{cases} 0 & i<1 \\ \hat{G}_{ps}\left(\dfrac{i}{\zeta} \mid k_0\right) & 1 \leq i \leq N \\ 1 & i>N \end{cases} \tag{4.17}$$

leading to

$$\begin{aligned} h_{ps}(j) &= H_{ps}(j) - H_{ps}(j-1) \\ h_{pw}(j) &= H_{pw}(j) - H_{pw}(j-1) \end{aligned} \tag{4.18}$$

[8]Refer to chapter 6.2 for more details

Now, the H_{ps} and H_{pw} values can be stored in vectors prior to the calculation of the convolution sum (4.15).

The stationary probability function is depicted in Fig. 4.19 with the fractional power factor ζ as paramter. In case of ζ = 0.0 only the wind turbine is used and the corresponding curve coincides with the $\bar{v}$ = 16m/s - curve in Fig. 4.3. On the other hand ζ = 1.0 signifies that the wind turbine is switched off with the resulting curve being the one in Fig. 4.19. The remnant two curves clearly mark the transition from one extreme to the other.

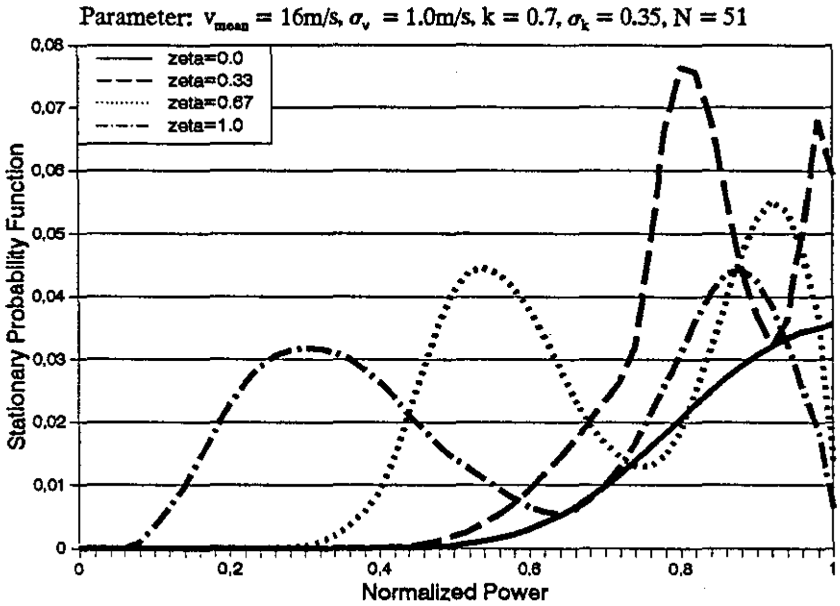

Fig. 4.19 PV Array Power: Conditional Distribution

As far as the conditional distribution is concerned, two scenarios are displayed for one specific time with ζ as parameter. First, in Fig. 4.20 a sudden wind speed slump (initial wind speed v_0 = 8m/s in relation to a mean wind speed $\bar{v}$ = 16m/s) is assumed. It is no surprise that a higher proportion of solar energy (greater ζ) causes the probability function at time t = 0.1s to have its peak at higher power values than in the wind turbine - only case.

The second scenario, as shown in Fig. 4.21, assumes a clearness index slump (initial clearness index k_0 = 0.1 and mean value k = 0.7).

Both scenarios demonstrate that a hybrid energy system is able to offset or at least restrain the effect of fluctuations, thus stabilizing the system. This discussion is continued in the chapter on time series where the same parameter settings will be encountered.

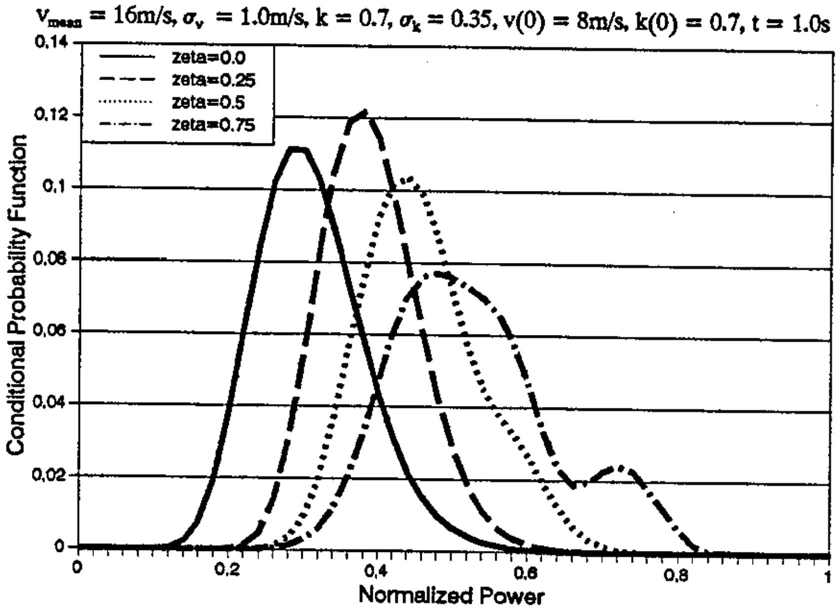

Fig. 4.20 Joint Renewable Power: Wind Speed Slump

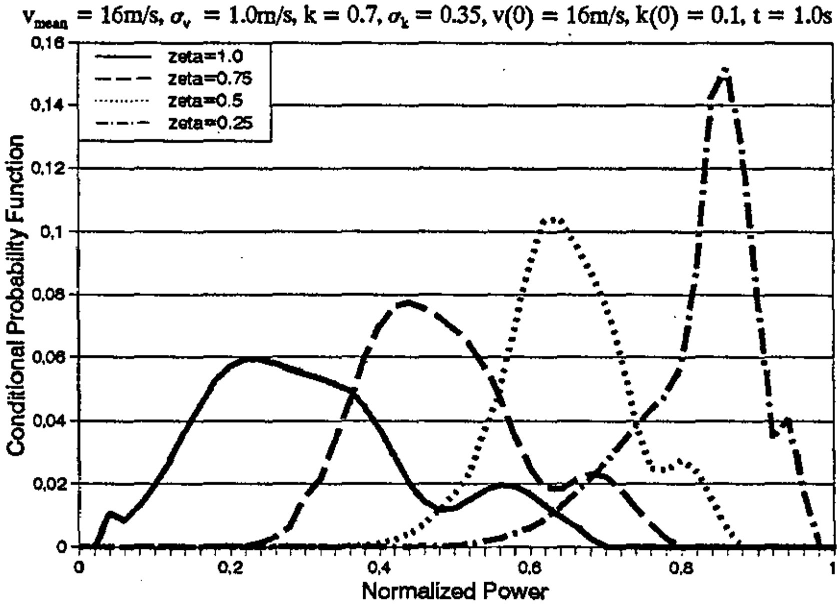

Fig. 4.21 Joint Renewable Power: Clearness Index Slump

4.2 Time Series

4.2.1 A General Time Series Algorithm

The purpose of this section is to present an algorithm to calculate synthetic time series of any stochastic process. It is applied to the processes discussed above in the following part.

Before defining the algorithm the framework has to be set out. First, let $F_\xi(\xi,\ \Delta t \mid \omega_0)$ denote the conditional distribution function with respect to the random variable ξ at time Δt subject to the initial value ω_0. Here, ξ and ω are vectors. In the framework of this paper they usually have one component, which corresponds either to the wind speed or the clearness index. Only in the case of the joint renewable power both components are needed. A function $\Xi(\xi) = \omega$ translates a given ξ into an initial vector. It is assumed that the inverse function $\Xi^{-1}(\omega)$ exists. Often, it is not the random variable ξ that is the desired magnitude. Therefore, a function $\psi = \Psi(\xi)$ is assumed that maps the vector ξ to a scalar variable ψ. Finally, a random number generator[9] is assumed that produces the random realizations, ξ. This random number generator is a functional of the underlying conditional distribution function $F(\xi,\ \Delta t \mid \omega_0)$, where Δt is the desired time step and ω_0 the set of initial values. Hence, it can be written as

$$\xi = \varrho\left[F_\xi(\xi, \Delta t \mid \omega_0)\right] \tag{4.19}$$

Given this preliminary, the algorithm to generate time series with values ψ_j and a time step Δt between any two values can now be formulated.

(1) Denote the set of initial values as ω_0. Calculate the first value of the time series from $\psi_0 = \Psi[\ \Xi^{-1}(\omega_0)]$.

(2) Set $j = 1$

(3) Initialize the random number generator with the current time. Link it to the underlying conditional distribution function. Set all initial values and the time step.

(4) Determine the next random vector $\xi_j = \varrho[F_\xi(\xi,\ \Delta t \mid \omega_{j-1})]$

[9]Refer to chapter 6.6 for a discussion of random number generators.

(5) Calculate set of initial values for next call: $\omega_j = \Xi(\xi_j)$

(6) Calculate next output value $\psi_j = \Psi(\xi_j)$

(7) Update $j = j + 1$

(8) If enough values have been calculated go back to step (3) to generate next value. Otherwise continue at (9).

(9) End of algorithm.

Each value is generated successively in step (4) by taking the last set of realizations, ξ, as initial values of the conditional distribution that governs the random number generator in the following call. Hence, each time the generator is being called the underlying distribution function might be different. At first glance, this algorithm might appear to be a bit nebulous. It will, however, gain substance in the following section. The reason for the general approach is that it allows an elegant implementation, independent of a specific distribution function[10] or requirements.

4.2.2 Case Study

4.2.2.1 Wind Speed Time Series

In case of wind speed time series the vectors have only one component, the wind speed, $\xi = \omega = v$ which coincides with the desired output magnitude, $\Psi(\xi) = \xi$. The underlying, conditional distribution function (4.2) is the well- known normal distribution[11]. Time series have been calculated for two parameter settings, the same as for the distribution functions in Fig. 4.1. In Fig. 4.22 three series are shown that have been generated using the same parameters. Fig. 4.23 shows three series based on the same parameters as in Fig. 4.22, except the standard variation being twice the previous value. The graphs clearly speak for themselves.

[10]Refer to chapter 7 for more details on the implementation of random generators and time series calculators (class TimeSeries).

[11]Algorithms to retrieve normal deviates are described in chapter 6.6.2.

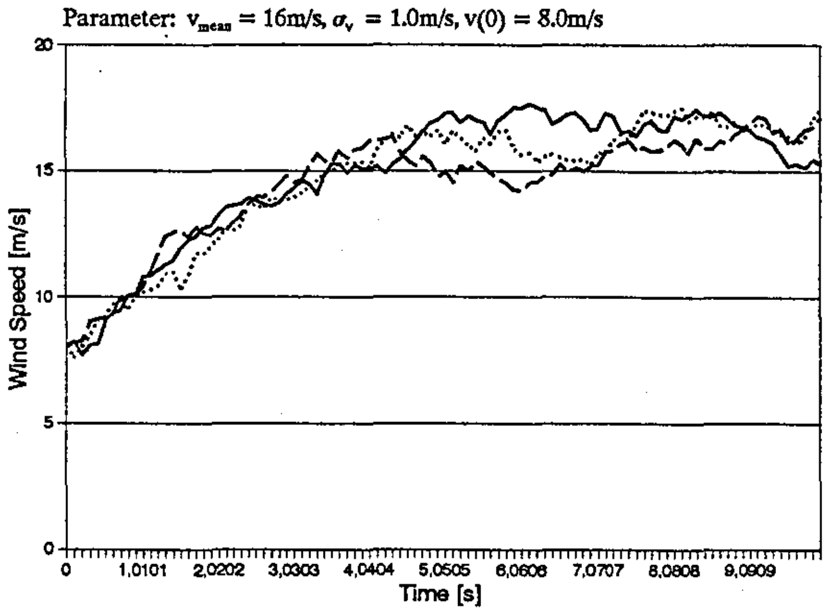

Fig. 4.22 Wind Speed Time Series

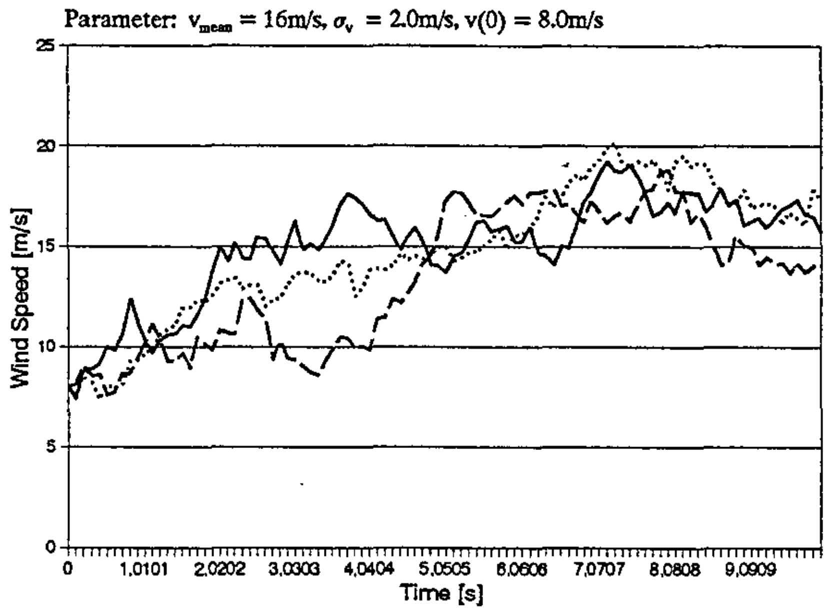

Fig. 4.23 Wind Speed Time Series

4.2.2.2 Wind Turbine Power Time Series

Again, the underlying stochastic process is the wind speed. Therefore the same random generator can be used as before in the case of wind speed time series. The difference is the output function $\Psi(\xi)$ which is now the wind turbine P-v- characteristic (3.1), normalized by (3.13). The diagrams in Fig. 4.25, Fig. 4.24 and Fig. 4.26 show normalized power time series for different mean wind speeds.

In Fig. 4.24 the mean wind speed ($\bar{v}$ = 14m/s) is below the rated wind speed (v_r = 16m/s) and the power slowly picks up. Concluding from the diagram it takes around 8s to pass the power level p = 0.6 for the first time.

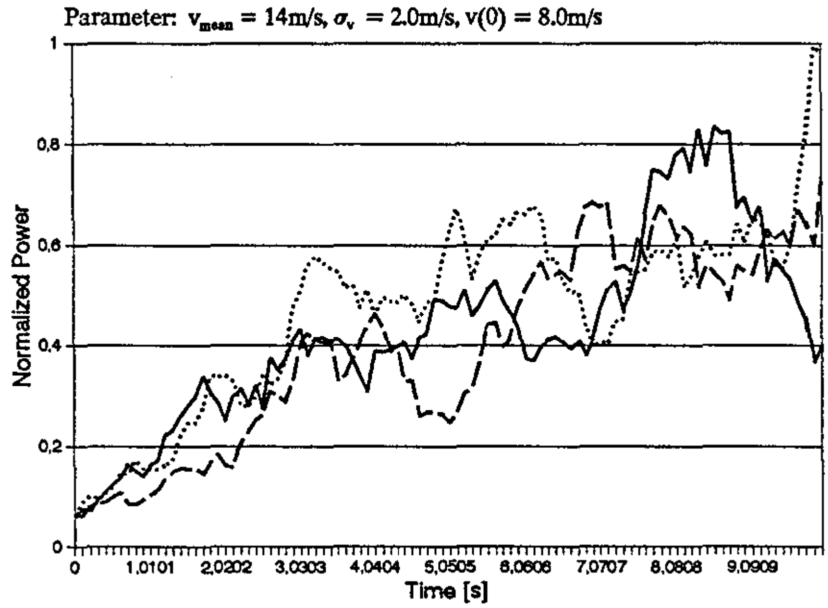

Fig. 4.24 Wind Turbine Power Time Series

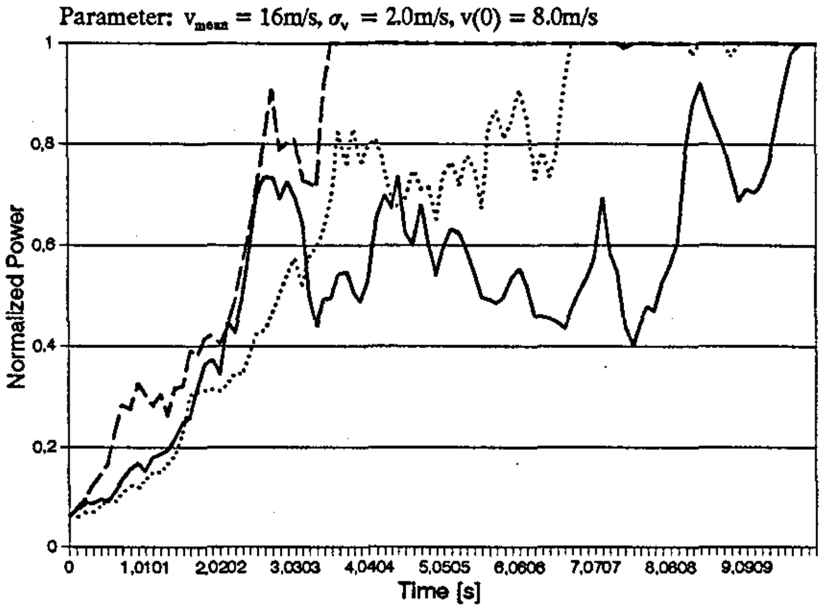

Fig. 4.25 Wind Turbine Power Time Series

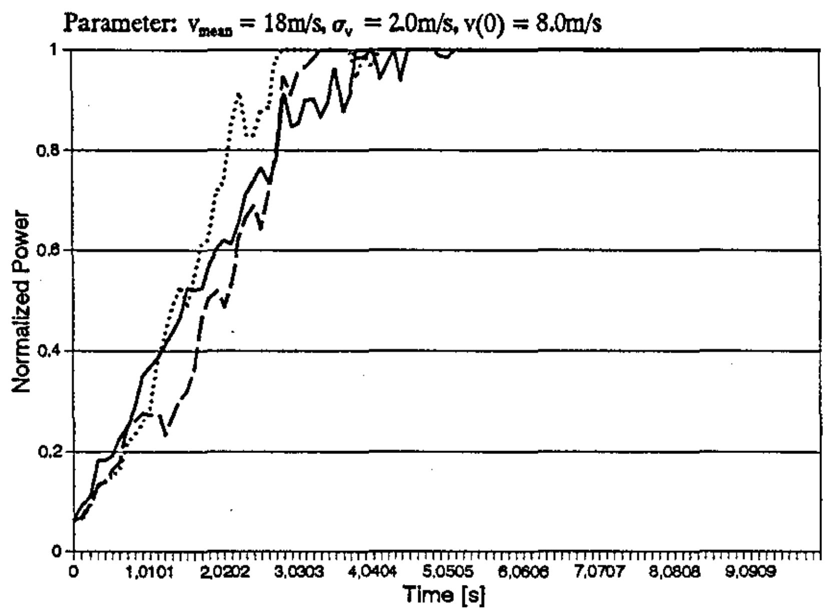

Fig. 4.26 Wind Turbine Power Time Series

In Fig. 4.25 and Fig. 4.26 the power will pick up a lot faster due to higher mean wind speeds of 16m/s and 18m/s respectively. A guess for the first passage time based on the graphs is 2s and 3s. Obviously, these are only very crude estimations of the first passage time and methods to calculate it are actually the center of discussion in the next chapter. We will, however, get back to these graphs in order to relate the results to single time series.

4.2.2.3 PV Array Power Time Series

The conditional distribution function to be applied to photovoltaic power time series is $\hat{G}_{ps}(n|k_0)$ from equation (4.11). The discrete power level $n \in [1, N]$ can be identified with $n = \xi$, whereas the initial condition is $k_0 = \omega_0$. As a result of this the functions Ξ and Ψ are set to be

$$\begin{aligned} \Xi(n) &= K_0 \frac{n-1}{N-1} \\ \Psi(n) &= \frac{n-1}{N-1} \end{aligned} \tag{4.20}$$

taking into account the normalization of the solar power (3.12) and the discretization (4.8). The time series values, produced from $\Psi(n)$, represent the normalized, discrete power. The random number generator used is described in chapter 6.6.3.

In Fig. 4.27 three time series have been recorded for a clearness index $k = 0.29$ and a standard deviation $\sigma_k = 0.08$. Once it has picked up the power stays within the range of the peak of the stationary probability function (as depicted in Fig. 4.16) which has only one peak for this particular parameter setting.

In contrast, Fig. 4.28 displays 3 time series with clearness index $k = 0.7$, standard deviation $\sigma_k = 0.35$ and otherwise identical parameters. The data in Fig. 4.12 is consistent with two peaks in the distribution.

The three time series in Fig. 4.29 correspond to the probability functions in Fig. 4.16 whose peaks match closely to the values of the time series.

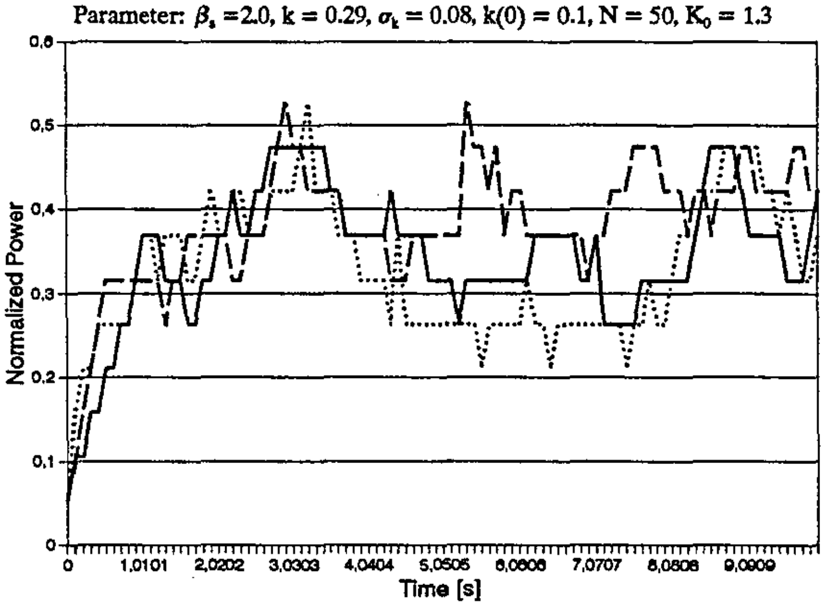

Fig. 4.27 PV Array Power Time Series

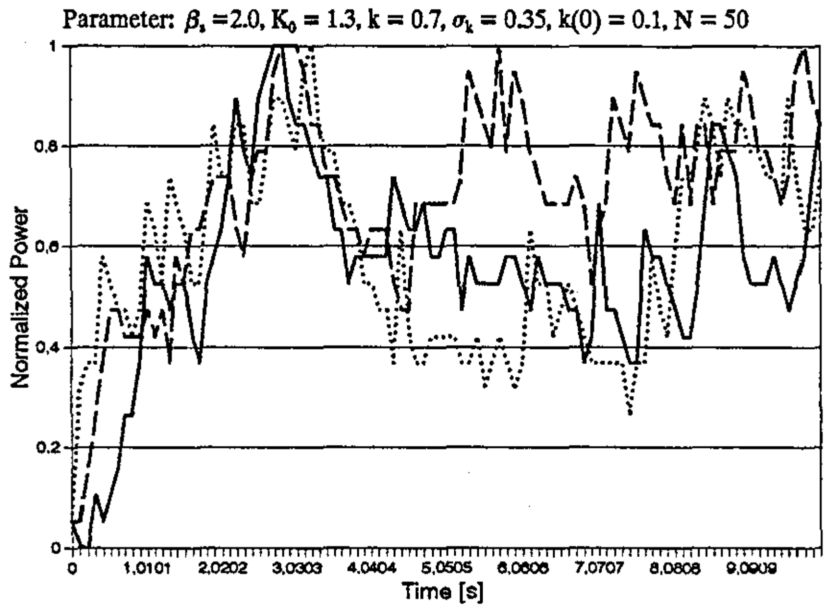

Fig. 4.28 PV Array Power Time Series

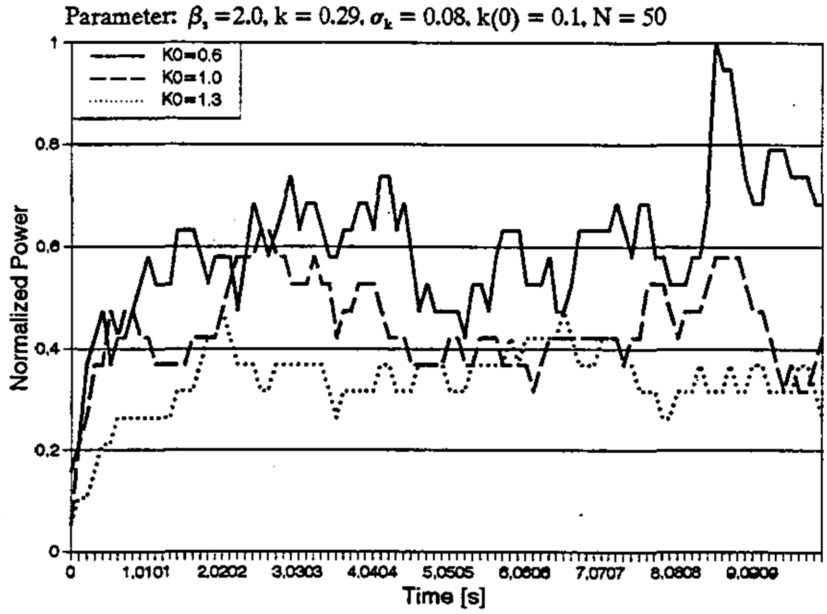

Fig. 4.29 PV Array Power Time Series

4.2.2.4 Joint Renewable Power Time Series

In case of joint renewable power time series the vectors ξ and ω hold two components. The first is identical to the wind power case, the second to the solar power case. The two underlying stochastic processes are treated completely separate throughout, including two random number generators. They are only brought together in the output function $\Psi(\xi)$ which coincides with the normalized expression for the total renewable power (3.15). The following diagrams, Fig. 4.30 and Fig. 4.31, take up the scenarios from last chapter, namely Fig. 4.20 and Fig. 4.21. They illustrate - what was already predicted then - that a combination of two renewable energy sources stabilizes the system and smoothens the output.

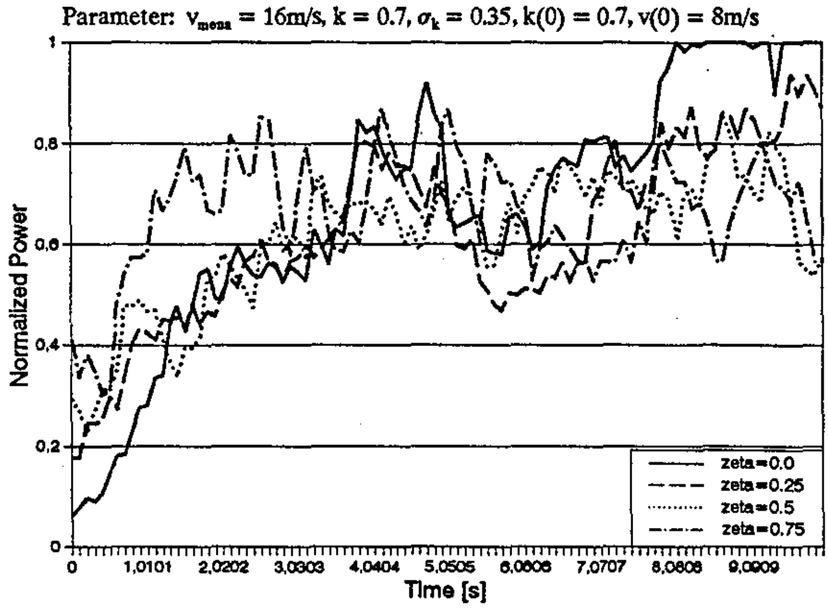

Fig. 4.30 Wind Speed Slump

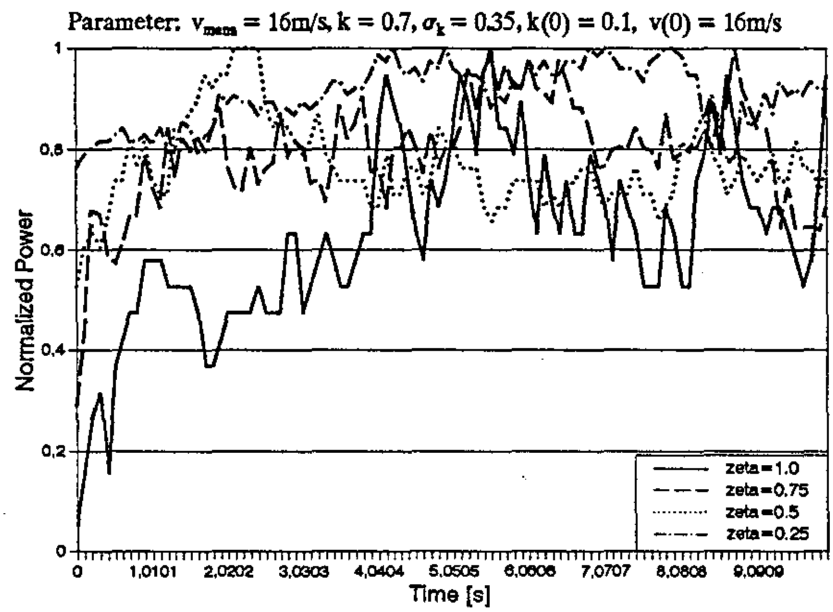

Fig. 4.31 Clearness Index Slump

4.2.2.5 State of Charge Time Series

In case of time series tracking the state of charge of the battery, the joint renewable time series generator is being used. Given the joint renewable power at each time step, the state of charge can be calculated. Using the Manwell battery model, the state of charge can be determined as follow:

(i) Prior to the initialisation of the time series generator the amount of available charge at the beginning, Q_{10} , and the amount of bound charge at the beginning, Q_{20} , have to be specified.

(ii) In order to simplify calculations it has been assumed that the power demand, P_{ex} (the power to be delivered), is constant throughout the time series generation.

(iii) Assume the time series algorithm generates a value that represents the joint renewable power, P_{ren} . Compare P_{ren} with the power demand P_{ex}.
If ($P_{ren} > P_{ex}$) go to step (iv). Charging the battery.
If ($P_{ren} = P_{ex}$) continue with next time step.
If ($P_{ren} < P_{ex}$) go to step (v). Discharging the battery.

(iv) Charging the battery:
First, calculate the maximum (negative) charge current, $I_{c,max}$, according to equation (2.99). Second, calculate the actual charge current, I_c , from

$$I_c = \frac{P_{ren} - P_{ex}}{V} \tag{4.21}$$

Here, V is the constant voltage with which the battery is charge. Now set $I_c = I_{c,max}$ if $I_c < I_{c,max}$. In this case a surplus energy of $\Delta P = P_{ren} - P_d - VI_{c,max}$ cannot be used to charge the battery and has to be dumped. With the given value of $I_c = I$ calculated Q_1 and Q_2 with the help of equation (2.97).

(v) Discharging the battery:
First calculate the (positive) maximum discharge current using equation (2.98). The demanded current is

$$I_d = \frac{P_{ren} - P_{ex}}{V} \tag{4.22}$$

Set $I_d = I_{d,max}$ if $I_d > I_{d,max}$. In this case the power delivered by both the renewable energy sources and the battery is not enough to meet the power demand P_{ex} . The power deficit $\Delta P = P_{ren} - P_{ex} - V\, I_{d,max}$ has to be covered by the diesel engine. As in (iv) calculate Q_1 and Q_2 from equation (2.97), the state of charge from equation (2.92) and continue by fetching the next time series value.

Fig. 4.32, Fig. 4.33 and Fig. 4.34 illustrate the course of the state of charge for various scenarios. For all calculations the following values for the battery parameters have been assumed: $k = 0.5s^{-1}$, $c = 1.0$, $Q_{max} = 193.6Ah$, $V = 11.5V$. The rated (maximum) joint renewable power has been assumed to be $P_{ren,max} = 7kW$ (compare discussion in section 3.3. In Fig. 4.32 and Fig. 4.33 the assumed power demand is $P_{ex} = 5kW$. Please note that both scenarios, wind speed slump and clearness index slump, correspond to the already examined cases in section 4.1.4 (on distributions) and in section 4.2.2.4 (on joint renewable power time series). The wind speed slump causes the battery to be discharged in order to meet the power demand. With increasing wind speed, however, the battery can be re-charged again after some time. For $\zeta = 0$ (wind turbine only) the battery is going to be discharged deeper than for $\zeta > 0$ (joint wind turbine and photovoltaic array).

The underlying scenario in Fig. 4.34 is identical to Fig. 4.33 except that the power demand is only $P_{ex} = 3.5kW$. Here, the depth of discharge caused by the wind speed slump is only marginal and the battery can be charge after a very short period.

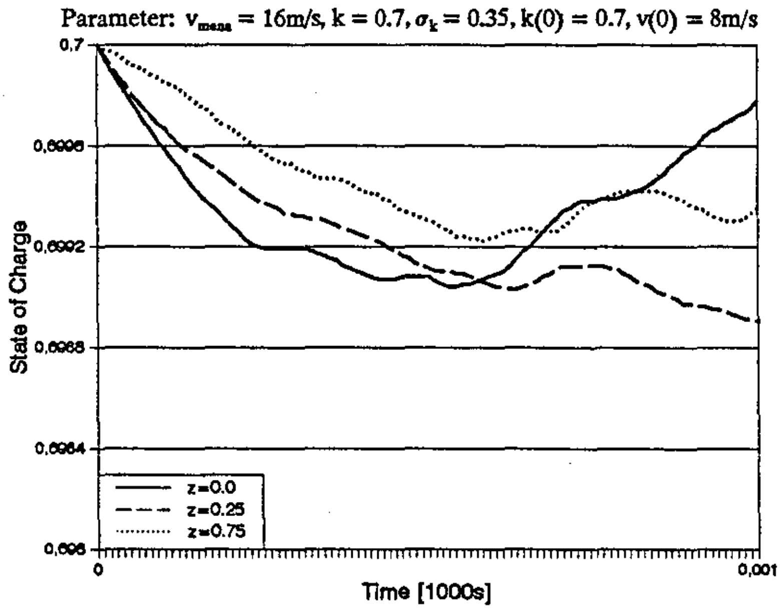

Fig. 4.32 State of Charge: Wind Speed Slump

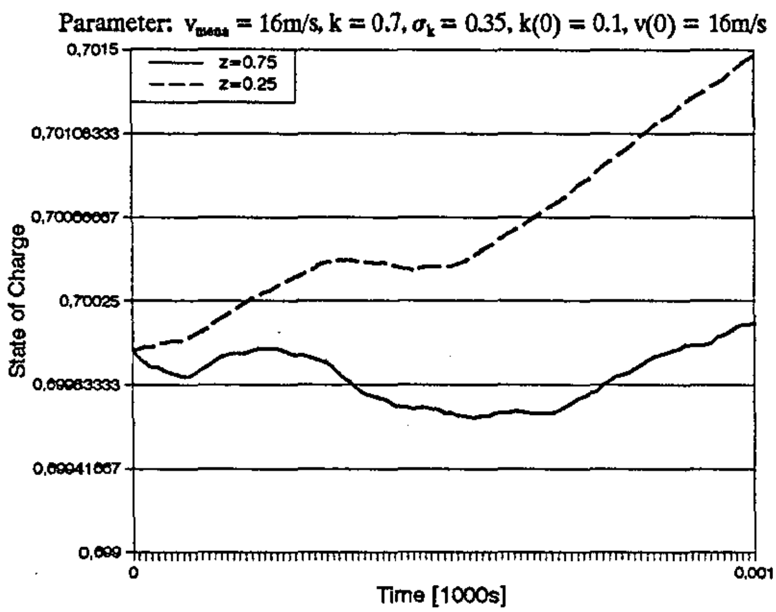

Fig. 4.33 State of Charge: Clearness Index Slump

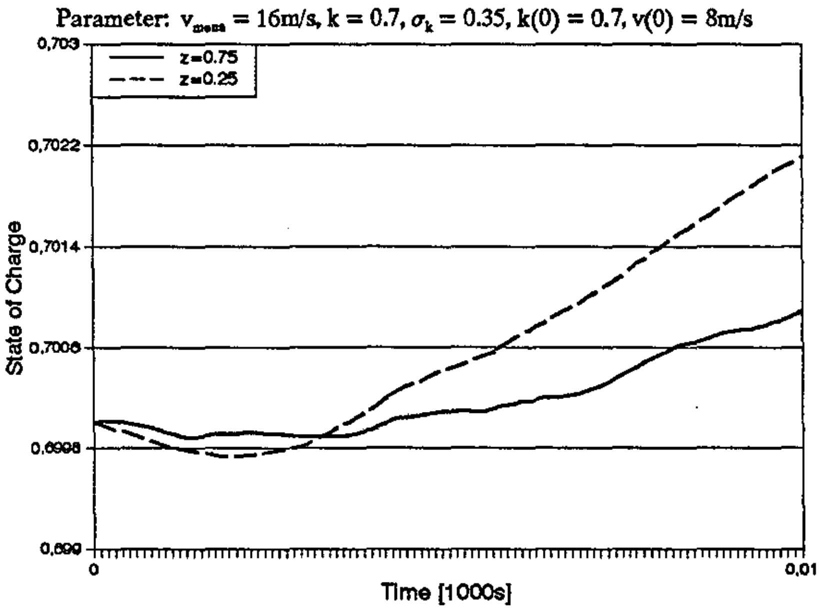

Fig. 4.34 State of Charge: Wind Speed Slump

4.3 First Passage Time

The first passage time problem was already solved for the wind speed in chapter 2.1.2.2. This was an analytical solution and it was pointed out that the same way is not viable for more difficult stochastic processes. The coverage of probability distributions and time series gives way to two further algorithms which are the focus of this chapter. Their differences and similarities are highlighted in section 4.3.3.

4.3.1 Time Series Approach

As mentioned above the first passage time is the expected time T_{fp} that elapses until a stochastic process reaches a passage level for the first time subject to an initial observation. In general, the first passage time is a function of the passage level x_p , the initial value x_0 and the underlying conditional distribution function $F(x,t|x_0)$. The idea behind a time series approach to the first passage time problem is to follow up a time series and record the time when the passage level is hit for the first time. For the simplicity of the calculations involved it is assumed that the initial value is always less than or equal to the passage level. The algorithm to calculate the first passage time is as follows:

(1) Specify the initial value x_0, the passage level x_p and the time step Δt that is inherent in the time series.

(2) Initialize the random number generator with the appropriate probability distribution.

(3) Set $n = 0$ (n being the counter of time series taken into account)

(4) Set $T = 0$ (T being the sum of first passage times from the individual time series.)

(5) Set $t = 0$ (t being the time scale in one time series) and reset the time series calculator.

(6) Set $j = 0$ (j being the counter of the number of generated time series values)

(7) Generate next time series value x. Set $j = j+1$.

(8) If $(x > x_p)$ go to (12)

(9) The process has not yet passed the specified passage level: Update time $t = t+\Delta t$.

(10) If $(j > 1000)$ exit the procedure with error message. This is just a safety measure in order to prevent a possible deadlock. The number 1000 is merely a suggestion which seems to be realistic. In the program this limit can be interactively specified by the

user.

(11) Repeat steps from (7).

(12) The process has passed the specified passage level: Add $T = T + t$ and update $n = n + 1$.

(13) If ($n < N_T$) start with new time series from step (5). N_T is the number of time series taken into account. Obviously, a large N_T stabilizes the result but causes the calculation time to increase. Numerical results (section 4.3.1.1) suggest that numbers between 10 and 20 already procure reasonably good results.

(14) The first passge time is the average, $T_{fp} = T / N_T$.

This algorithm is illustrated and discussed in several examples in the following sub- sections.

4.3.1.1 Time Series Approach: Wind Speed

Applying the algorithm described above the first passage time has been calculated for the same parameter setting as in the time series in Fig. 4.22 and Fig. 4.23. It is displayed for an initial value of $v(0) = 8$m/s as a function of the wind speed passage level v_p in Fig. 4.32. Hence, it shows the expected time it takes to encounter a wind speed v_p or greater for the first time subject to an initial observation of v(0). Not surprisingly, the first passage time is shorter if the standard variation is smaller. In Fig. 4.35 the first passage time is plotted as a function of the initial wind speed assuming a passage level $v_p = \bar{v} = 16$m/s. In both diagrams the number of time series taken into account, N_t, was set to 20.

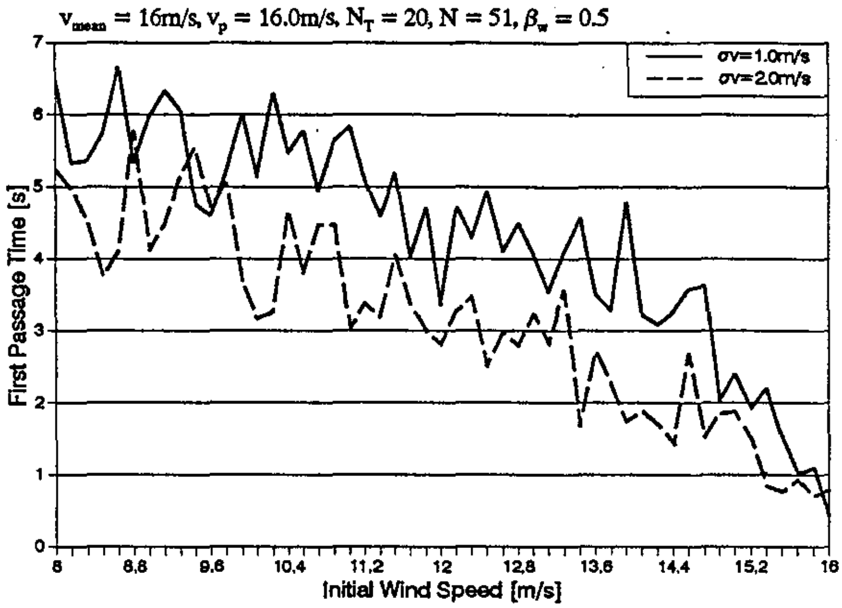

Fig. 4.35 Time Series Method - Wind Speed

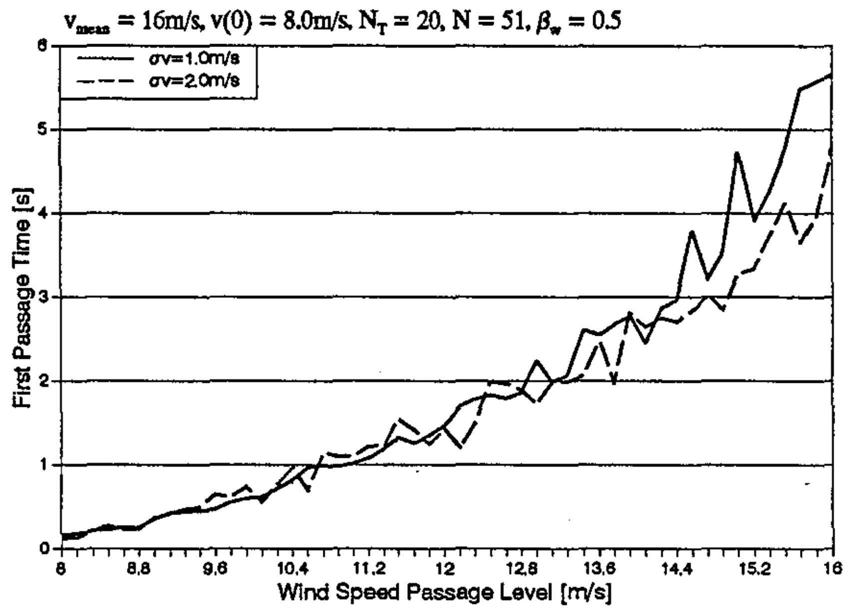

Fig. 4.36 Time Series Method - Wind Speed

Fig. 4.37 depicts first passage times over the wind speed passage level for different values of N_t. For N_t = 5 the variations are fairly significant, though even there the trend is distinct. The curves get smoother for greater values of N_t. The improvement stemming from an increase in N_t = 10 to 20, however, seems not to be worth twice the computing time.

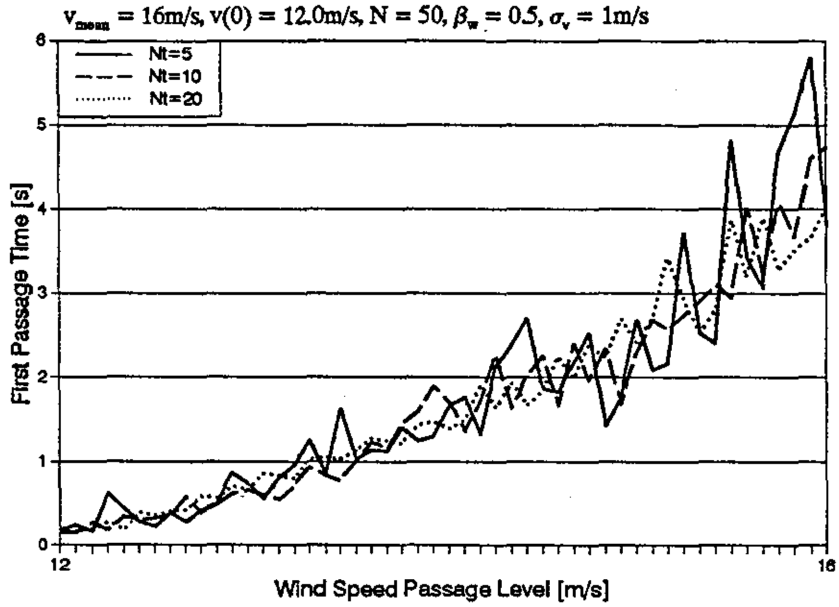

Fig. 4.37 Influence of Number of Time Series

4.3.1.2 Time Series Approach: Wind Turbine Power

Results for the wind turbine power are illustrated in Fig. 4.38 and Fig. 4.39. They correspond to the time series displayed in Fig. 4.24, Fig. 4.25 and Fig. 4.26. Fig. 4.38 depicts the first passage time as a function of the specified passage level of the normalized wind turbine power, whereas Fig. 4.39 captures the first passage time as a function of the initial wind speed, assuming a constant power passage level p_p = 0.8. Both diagrams clearly demonstrate that the first passage time rises immensely in the event of low mean wind speeds.

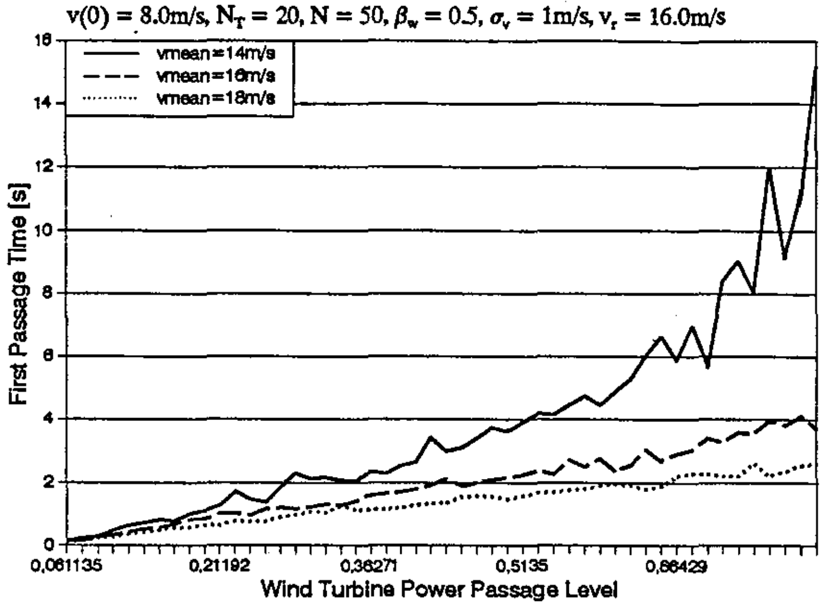

Fig. 4.38 Time Series Method - Wind Turbine Power

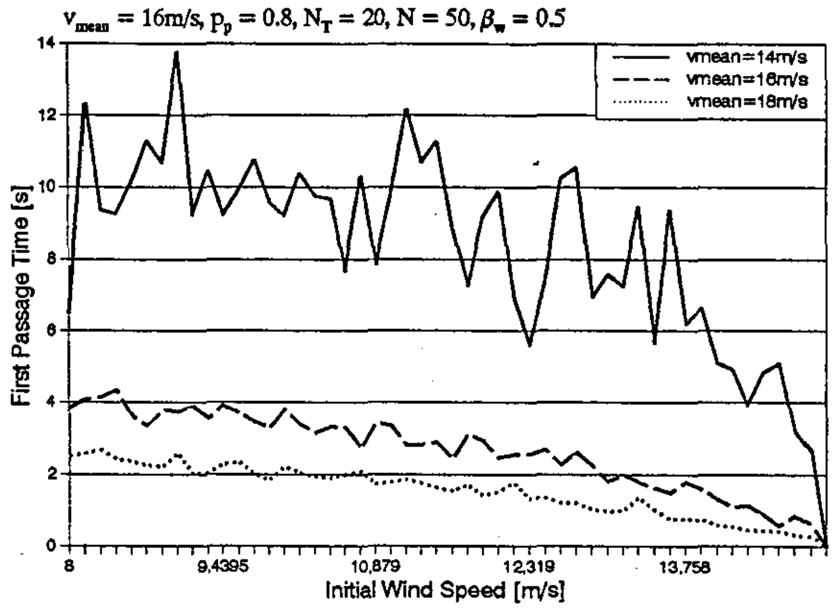

Fig. 4.39 Time Series Method - Wind Turbine Power

4.3.1.3 Time Series Approach: PV Array Power

The first passage time as a function of the passage level of the photovoltaic array power is illustrated in Fig. 4.40 and Fig. 4.41. Here, Fig. 4.40 corresponds to time series diagram Fig. 4.28, while Fig. 4.41 corresponds to Fig. 4.27. Note that the first passage time is the expected *average* time. It does not give any clue towards the variance. For instance, looking at the time series realizations Fig. 4.27 a large variance of the first passage time is expected which is due to the two peaks in the underlying distribution function. The first passage time algorithm, however, only yields the average time.

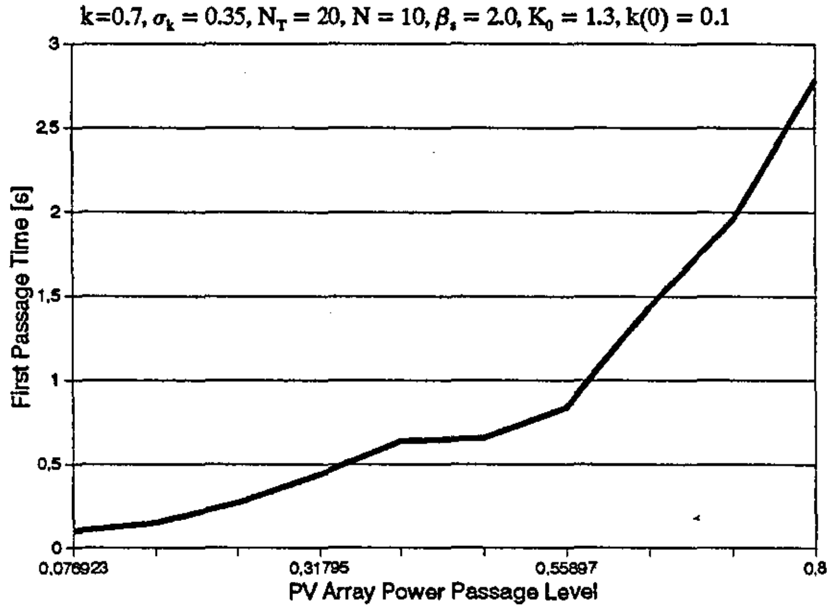

Fig. 4.40 Time Series Method - PV Array Power

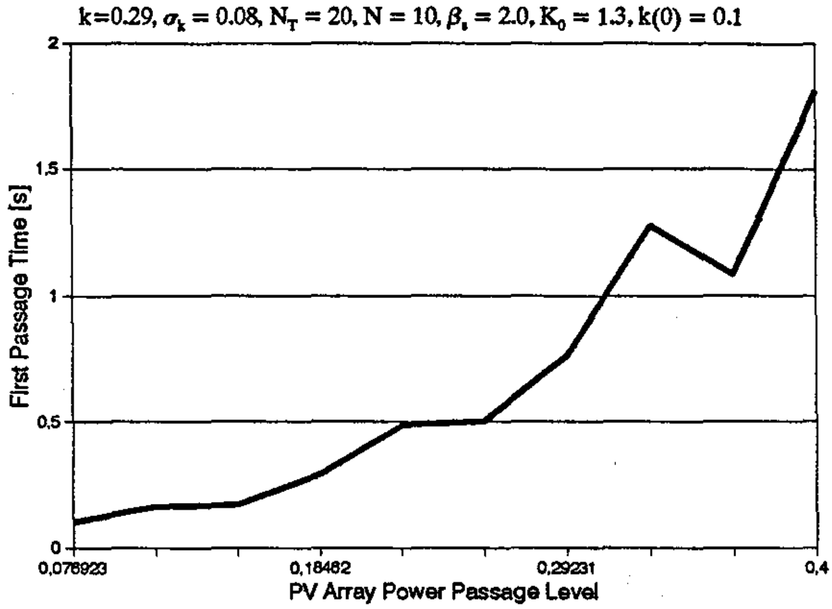

Fig. 4.41 Time Series Method - PV Array Power

4.3.1.4 Time Series Approach: Joint Renewable Power

The first passage time as a function of the passage level of the joint renewable power is depicted in Fig. 4.42 and Fig. 4.43. Fig. 4.42 simulates a slump in the wind speed with an initial wind speed of v(0) = 8m/s. This scenario is identical to 4.30. Greater ζ - values, signifying a higher proportion of solar energy, reduce the first passage time considerably. For ζ = 0.75 the impact of the wind speed slump is almost insignificant. Fig. 4.43 on the other hand simulates a solar energy slump, corresponding to 4.31. In relation to Fig. 4.42 solar energy and wind energy are just swapped. The qualitative results are the same.

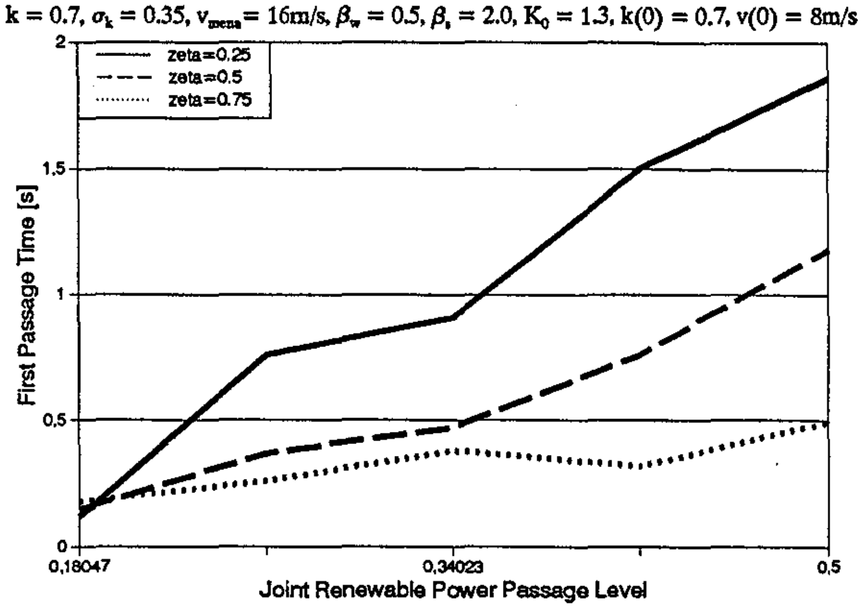

Fig. 4.42 First Passage Time: Wind Speed Slump

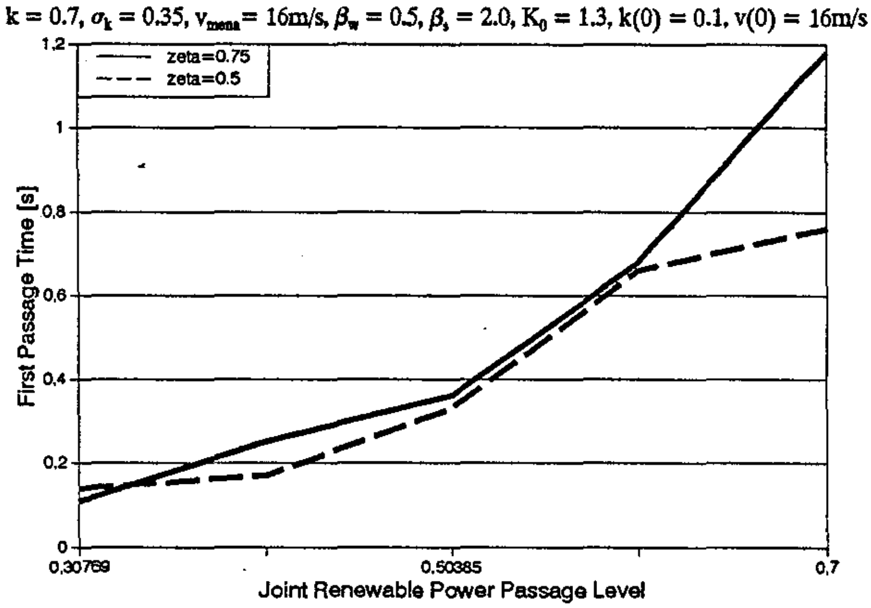

Fig. 4.43 First Passage Time - Clearness Index Slump

4.3.2 Markov Chain Approach

In this section a technique is presented to work out the expected first passage time of a stochastic process using Markov chains, as mentioned in the first discussion of the first passage time problem in chapter 2.1.2.2. A Markov chain ([20]) is a discrete-value, discrete-time Markov process. A Markov process on the other hand is a stochastic process for which the conditional probability density function at any time and for any given number k of previous observations, depends only on the most recent observation:

$$f_x(x \mid X(t_1)=x_1, X(t_2)=x_2, \ldots, X(t_k)=x_k) = f_x(x \mid X(t_1)=x_1)\ ,\ t_1 > t_2 > \ldots > t_k \tag{4.23}$$

Hence, the evolution of the process can be phrased in terms of the so-called transition probability

$$g_{nm}(j) = p(X_j = n \mid X_{j-1} = m) \tag{4.24}$$

This is the probability that the process X changes from value m to n within the time interval [j-1,j]. If $p_j(k)$ denotes the probability $p(X_j = n)$ all probabilities can be put into a vector

$$P(j) = [p_1(j)\ \ldots\ p_N(j)]^T \tag{4.25}$$

with N components (for N possible values of X). The progress of the process can then be expressed in matrix representation

$$P(j) = G(j)\ P(j-1) \tag{4.26}$$

where G(j) is the transition matrix with elements $g_{mn}(j)$ as defined above. The algorithm whose description follows has been inspired by an algorithm proposed by Paynter ([32]), which has been further developed in the frame of this paper.

The algorithm exploits the same idea that stood behind the analytical approach in 2.1.2.2. Assume the output of the stochastic process to be representable by a whole number in the closed interval [1, N]. Hence, there are only N different states to observe. Assume further that q is the passage level in question, where q is too a whole number, $q \in [1, N]$. Back in chapter 2.1.2.2 a system was thought of being filled with particles. Particles that reach level q were taken out of the ensemble. In this context, the same can be achieved by introducing an (N+1 x N+1) - matrix G with the elements ($n,m \in [1,N+1]$)

$$g_{nm} = \begin{cases} 0 & \begin{cases} m>q\ ,\ n \neq N+1 \\ m \leq q\ ,\ n=N+1 \end{cases} \\ 1 & m>q\ ,\ n=N+1 \\ p_{nm} & \text{otherwise} \end{cases} \tag{4.27}$$

where p_{nm} is the corresponding transition probability. Hence, the transition matrix looks like

$$G = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1q} & 0 & \cdots & 0 \\ \vdots & & & \vdots & \vdots & & \vdots \\ p_{N1} & p_{N2} & \cdots & p_{Nq} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 & \cdots & 1 \end{bmatrix} \tag{4.28}$$

Below the passage level, G of (4.28) is identical to the transition matrix of the stochastic process in question. Only difference: Once a particle has passed q, the transition probability for returning is zero and it will end up in state (N+1). After applying (4.26) over and over all particles will eventually be in state (N+1), P(N+1) = 1.

Assume now an initial state u, u<q. The initial probability vector P(0) has therefore the components $p_j = \delta_{ju}$, where δ is the Kronecker symbol,

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i=j \end{cases} \tag{4.29}$$

The probability that at time k the system is in one of the states above the passage level is simply

$$C(k) = \sum_{j=q+1}^{N+1} p_j(k) \tag{4.30}$$

C(0) is zero as it is assumed that u < q. The next value, C(1) is the probability that the passage level has been passed after the first time step. As a result, the associated first passage time - after one time step - is $T_{fp}(1) = 1 * \Delta t * C(1)$. After the second time step the volume above q will have increased by $\Delta C = [C(2) - C(1)]$, which can be interpreted as the probability for passing q during the second time step. The resulting passage time is

$$T_{fp}(2) = \Delta t\,[2(1-C(1))\Delta C] + T_{fp}(1) \tag{4.31}$$

The term (1 - C(1)) in (4.31) is the probability that the system has not passed q within the first time step. This makes both events ('passing q in time step 1' and 'passing q in time step 2') exclusive so that the probabilities can be added up, leading to (4.31). This can be continued until C(k) is 1 or very close to 1. This technique can be put into a more general algorithm:

(1) Specify N, the number of discrete levels of the underlying stochastic process.

(2) Specify q, the passage level, $q \in [1,N]$. Calculate the transition matrix G of the enlarged system (4.28) given a time step Δt.

(3) Specify u, the initial value, $u < q$.

(4) Specify N_i, the maximum number of iterations permitted and δ, the stop criterion, $\delta < 1.0$

(5) Set counter j = 1

(6) Set initial probability vector P(0) with components $p_j = \delta_{ju}$.

(7) Initialize coefficients C(0) = 0.0, ET(0) = 0.0, $\gamma = 1.0$

(8) Matrix multiplication P(j) = G * P(j-1)

(9) Calculate C(j) from (4.30).

(10) Calculate ΔC = C(j) - C(j-1)

(11) Calculate ET(j) = j * γ * ΔC + ET(j-1)

ET(j) is the normalized first passage time that accumulates the results of the preceding time steps. Multiplied by the time step Δt is the real first passage time. It is denoted ET to make clear this is the formula for the expected time T, the first passage time.

(12) Increment j = j + 1

(13) If (1.0 - C(j) < δ) go to step (16). Otherwise, stop criterion not met. Continue with step (14).

(14) If ($j > N_i$) return with an error message. The maximum number of iterations has been reached. This is just to make sure that a deadlock can not occur.

(15) If ($j \le N_i$) repeat iteration from step (8).

(15) If ($j \leq N_i$) repeat iteration from step (8).

(16) The first passage time T_{fp} is T_{fp} = ET(j) * Δt.

This algorithm can be seen as a template for any stochastic process. What is left to specify from case to case is the initial value, the passage level and the underlying distribution. And this is actually the main difficulty associated with this algorithm as it requires to calculate the transition matrix. This is discussed in detail in the following sections on the particular stochastic processes, i.e. wind speed, wind power and solar power.

4.3.2.1 Markov Chain Approach: Wind Speed

In order to apply the above algorithm to the wind speed, the wind speed scale has to be discretized. Assume that M classes C_i (i = 1...M) along the wind speed axis are defined by the wind speed intervals $C_i \in [v_{i-1}, v_i]$. As the normal distribution is used to describe the wind speed fluctuations, the extreme values v_0 and v_M are $\pm\infty$. For the values in between the relationship

$$v_n = \sigma_v \, u \left[2\frac{n-1}{M-2} - 1 \right] + \bar{v} \quad , n=1...M-1 \quad , u = 4.753 \tag{4.32}$$

is proposed. Here, σ_v is the standard deviation and $\bar{v}$ the average wind speed. The factor u = 4.753 was chosen so that $\Phi(v_1) = 10^{-6}$. The choice is however, an arbitrary one. For the reverse direction, calculating a discrete level n from a given speed v, the formula

$$n = \min_{i=1...M} \{ i \mid v_i \geq v \} \tag{4.33}$$

can be applied. It says that n is the minimum index for which $v_i \geq v$. Recalling the wind speed distribution function (4.2) allows to calculate the probability that the wind speed is - at time t - within class number i subject to the condition $v(0) = v_0$. It is

$$p_n(v_0) = F_v(v_n \mid v_0) - F_v(v_{n-1} \mid v_0) \tag{4.34}$$

class m to class n, where both classes are a whole range of wind speeds rather than just one value as the initial value in (4.34). Therefore, $p_n(v_0)$ has to be integrated over all v_0 values in class m and divided by the probability that it is in class m in the first place, that is

$$g_{nm} = \frac{\int_{v_{m-1}}^{v_m} p_n(v_0)\, \mathrm{d}v_0}{\Phi\left(\frac{v_m - \bar{v}}{\sigma_v}\right) - \Phi\left(\frac{v_{m-1} - \bar{v}}{\sigma_v}\right)} \tag{4.35}$$

(4.35) can not be analytically integrated, thus requiring a large amount of computing time. Instead, the following transition probability is suggested:

$$g_{nm} = \beta_m \exp\left[- \frac{u^2 \left(\left(\frac{2(n-1)}{M-1} - 1 \right) - \left(\frac{2(m-1)}{M-1} - 1 \right) \right)^2}{2\,(1 - r^2)} \right] \tag{4.36}$$

The coefficients β_m can be obtained from the normalization condition

$$\sum_n g_{nm} = 1 \tag{4.37}$$

The transition probability g_{nm} as in (4.36) has the same characteristic as the probability density function (4.1), namely the $\exp(-x^2)$ functionality. In fact, (4.36) can be obtained from (4.1) by substituting

$$v = \sigma_v\, u \left(\frac{2(n-1)}{M-1} - 1 \right) + \bar{v} \tag{4.38}$$

for v and v_0 and replacing the factor in front of the *exp* by β_m. The process is stationary when the correlation coefficient is zero and the transition probability simply becomes a probability for class n irrespective of m.

Given the transition probability g_{nm} (4.36) and the conversions from wind speed to discrete numbers and vice versa, (4.32) and (4.33), the first passage time of wind speed fluctuations can be calculated by following the above Markov chain algorithm. Results for a mean wind can be calculated by following the above Markov chain algorithm. Results for a mean wind speed of 16m/s are shown in Fig. 4.44 and Fig. 4.45, where M = 20 classes were taken into account. In Fig. 4.44, where an initial wind speed of 12m/s was assumed, two curves for different standard variations are drawn as functions of the passage level of the wind speed. Fig. 4.45 depicts the first passage time as a function of the initial wind speed assuming a passage level of 16.0 m/s.

In Fig. 4.46 the Markov Chain and the Time Series approach are compared by applying them to the same parameter setting. Although the methods are very different the results are not inconsistent.

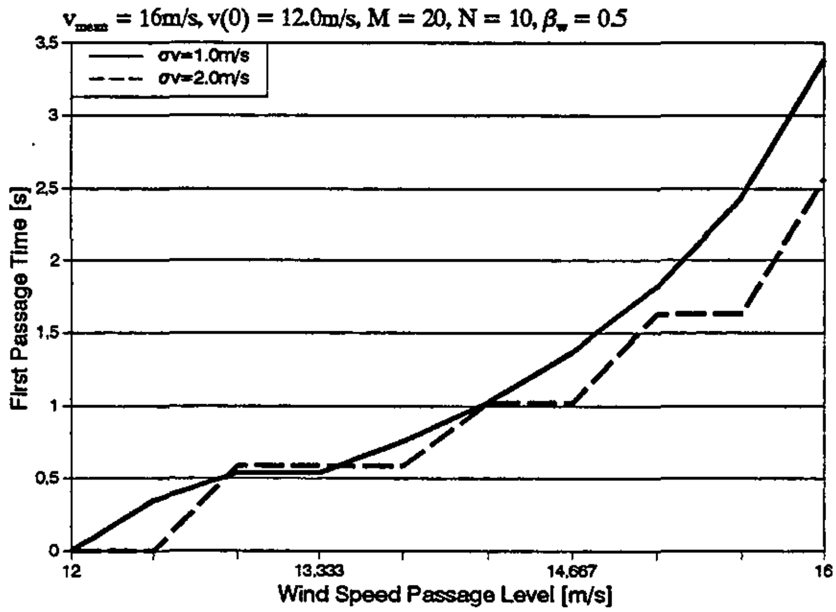

Fig. 4.44 Markov Chain Method - Wind Speed

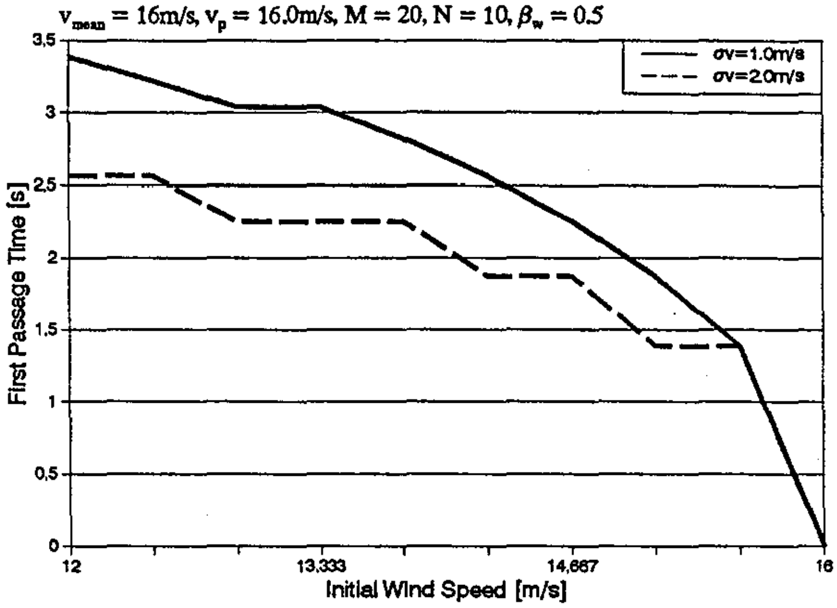

Fig. 4.45 Markov Chain Method: Wind Speed

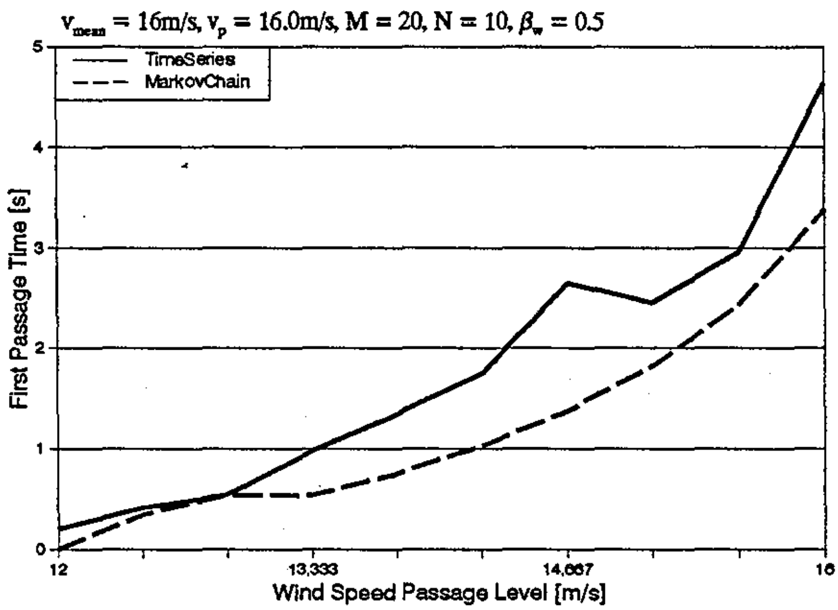

Fig. 4.46 Time Series versus Markov Chain Approach

The discussion of comparison is continued at the end of this chapter. But before that the stochastic processes of the wind power and the solar power are subjected to the Markov Chain approach. Unlike the wind speed these processes have already been discretized in chapter 4.1, thus making life a lot easier.

4.3.2.2 Markov Chain Approach: Wind Turbine Power

The power scale in the conditional distribution of the wind turbine power is already discretized in (4.5). The initial value, v_0, in (4.6) however is not. In order to use it for the Markov chain algorithm, v_0 in (4.6) has to be derived from a given initial power level m. As the power - wind- characteristic (3.1) is not a strictly monotonic function the wind speed can not always be concluded from a power value. If the power is zero valid wind speed values are $v < v_{ci}$ and $v > v_{co}$; if it is 1 valid wind speed values are between v_r and v_{co}. In order to circumvent this problem the following mapping between wind speed values v and discrete power levels m is assumed:

$$v(m) = \begin{cases} \min\{v_{ci}, \bar{v}\} & m=1 \\ v_{ci} + (v_r - v_{ci}) \sqrt[3]{\dfrac{m-1}{M-1}} & m=2...M-1 \\ \max\{v_r, \min\{\bar{v}, v_{co}\}\} & m=M \end{cases} \tag{4.39}$$

That means, if m = 1 (power is zero) the wind speed is assumed to be v_{ci} unless the mean wind speed $\bar{v}$ is less. In case of m = M, which corresponds to maximum power p = 1, the formula returns a wind speed equal to the mean wind speed, though not below the rated wind speed v_r or above cut- out speed v_{co}. The result can directly be inserted in (4.7), thus leading to the desired transition probability g_{nm}. Results are illustrated in Fig. 4.47 and Fig. 4.48 for a variety of mean wind speed values. Qualitatively, the results match Fig. 4.38 and Fig. 4.39 where the first passage time is calculated using the time series algorithm.

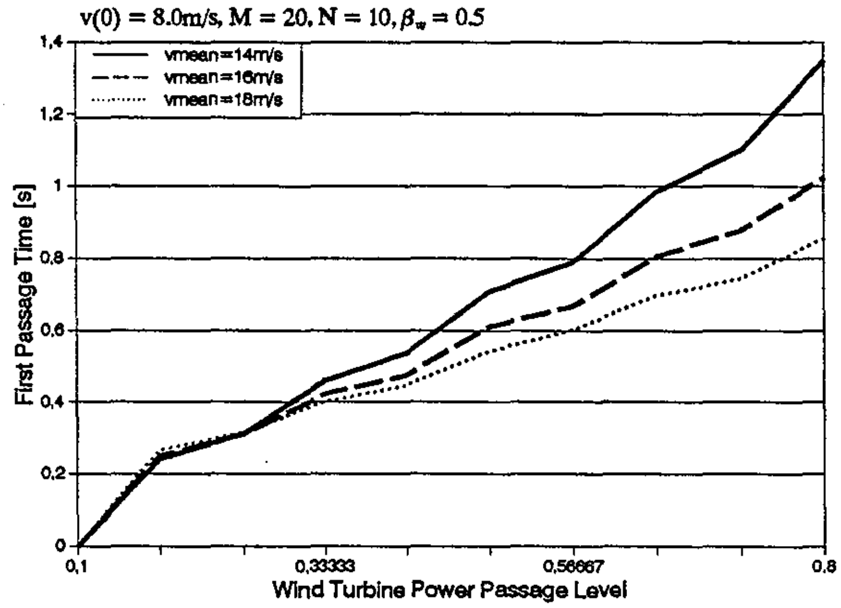

Fig. 4.47 Markov Chain Method - Wind Turbine Power

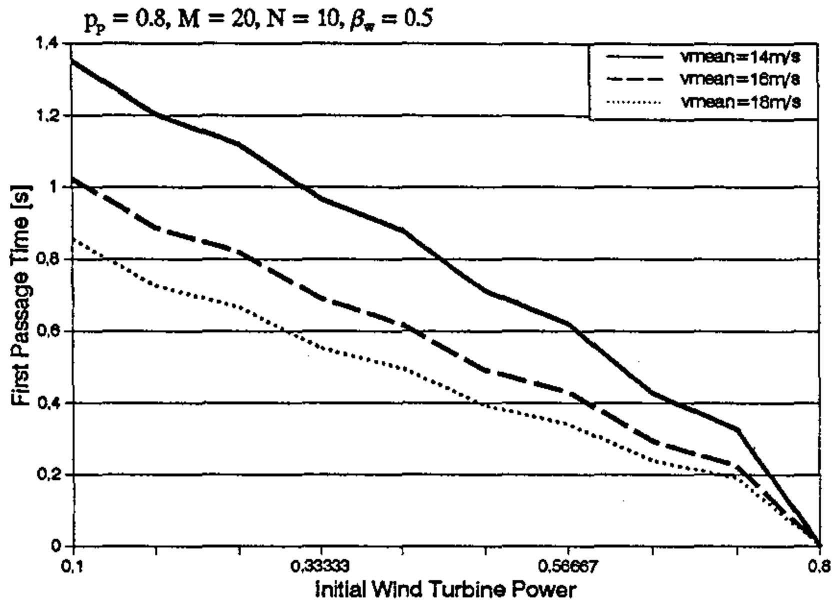

Fig. 4.48 Markov Chain Method - Wind Turbine Power

First passage times calculated via the Markov chain algorithm are, however, significantly shorter. This is illustrated in a direct comparison in Fig. 4.49. Here, identical initial conditions apply to both curves. Obviously, the transition matrix G allows the process to advance quicker than expected. Why is this discrepency? First, the time series approach tracks the wind speed, not the wind turbine power. As mentioned above, wind speed values can be uniquely translated into power values, but not the other way round. Second, the Markov chain method uses a discrete wind turbine power distribution, whereas the time series approach applies the continuous wind speed distribution - two different distribution types and two different underlying stochastic processes. The comparison of both algorithms is continued in section 4.3.3.

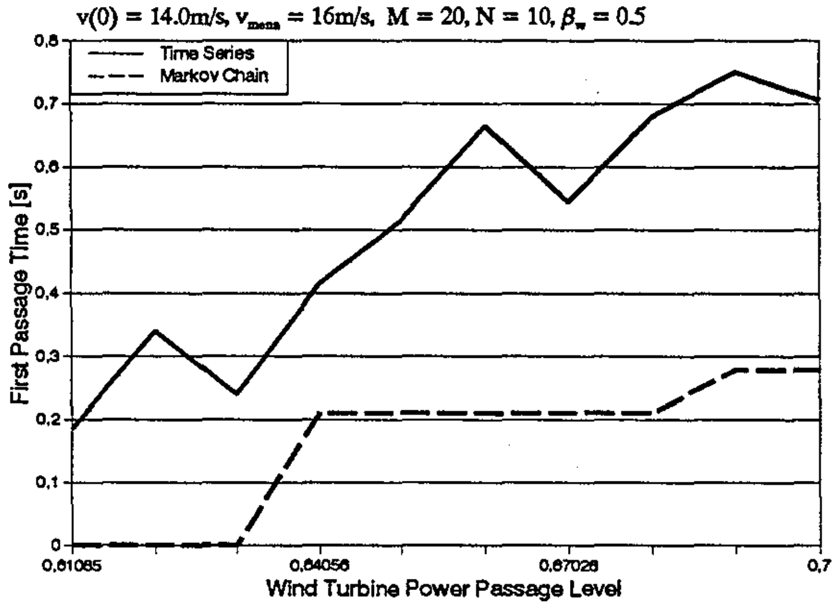

Fig. 4.49 First Passage Time - Wind Turbine Power

4.3.2.3 Markov Chain Approach: PV Array Power

The fluctuations of the photovoltaic power is governed by the conditional distribution (4.12), which can be used in the Markov chain algorithm without further alterations as the mapping between the clearness index k and the normalized power is linear. Fig. 4.50 illustrates a comparison between time series approach and Markov chain approach by using identical initial conditions. For the distribution of the PV power M = 20 discretization were taken into account. Fig. 4.50 shows a good agreement between both algorithms. Unlike in the case of the wind power both algorithms do employ the same distribution formula.

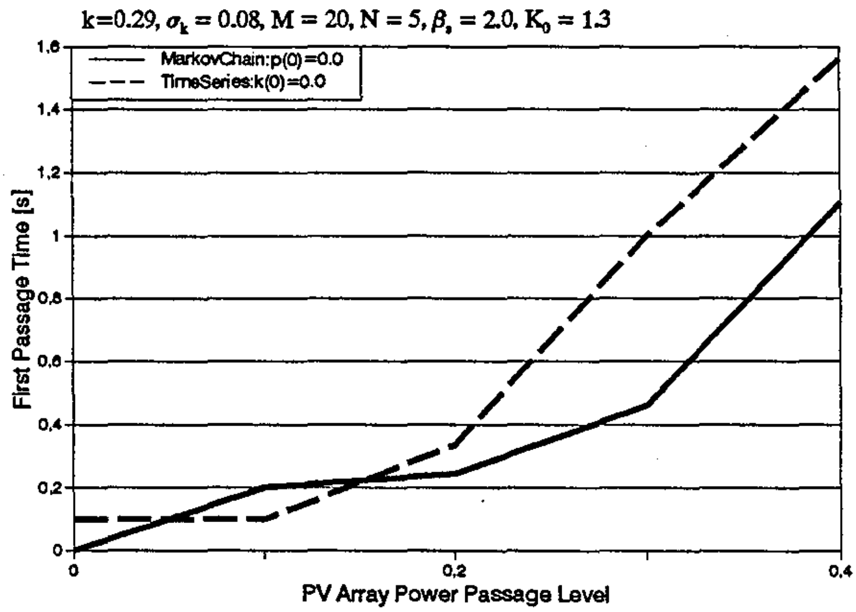

Fig. 4.50 Time Series versus Markov Chain Approach

4.3.3 Time Series versus Markov Chains - A Comparison

The time series algorithm monitors the meteorological data, wind speed and clearness index, as it goes along and translates them into power values. To use this algorithm these parameters need to be given. The Markov Chain algorithm on the other hand, does not need meteorological data as it is tracking the power. Hence, if wind speed or clearness index are not monitored only the Markov chain algorithm can be used to estimate the first passage time. However, in the case of the wind turbine ambiguities occur as both minimum and maximum power could be caused by a wide range of wind speed values, causing the Markov chain algorithm to be less accurate than the time series approach. For the stochastic processes 'wind speed' and 'PV array power' both algorithms procure similar results.

For the values used in the examples the time series algorithm proved, in general, to be faster than the Markov chain algorithm - with the exception of PV array calculations. The Markov chain algorithm initially calculates the whole transition matrix G. It is not being recalculated throughout the algorithm. Only matrix multiplications on G are carried out once G is established. The time series algorithm has to return to the conditional distribution each time a random number is generated. As a result of this the Markov chain algorithm is advantegous whenever the evaluation of the conditional distribution function is time consuming, as it is in the case of the PV array power.

Finally, both algorithms calculate the first passage time successively by moving along the time axis. In contrast, the analytical method requires the evaluation of an integral or differential equation. It follows from this observation that the time series method is also based on the assumption of a Markov process. Hence, both methods assume the same physical processes. The difference is a mathematical one. Whereas the "Markov chain" method uses theoretical transition probabilities, the time series method uses a random number generator.

5. Summary

This paper centers on an autonomous energy supply plant that consists of a wind turbine, a photovoltaic array, a battery unit and a fossil fuel engine. The purpose was to develop and examine statistical models that describe the system and the influence of various parameters, such as the wind speed and the light intensity, on it.

This has been achieved in three steps. First, the energy sources involved have been discussed in chapter 2. It has been shown that the short term wind speed turbulence can be described by the Ornstein- Uhlenbeck process. Likewise, the short term fluctuations of the solar clearness index can be expressed in terms of mathematical functions. The third energy source is the battery unit, which may be charged in the event of a surplus energy or discharged if necessary. Three models for a lead- acid battery have been discussed: Two electric models and one based on the electric charges. For the purpose of this paper the latter one has been selected. Finally, a brief section has been devoted to the fossil fuel engine.

In the second step the power supply by this system has been modelled. For the wind turbine a simple power- wind speed characteristic has been used. As far as the photovoltaic array is concerned it has been shown that it is reasonable to assume a linear relationship between the clearness index and the power supplied by the array.

Eventually, in the third step the results of the first two steps have been used to extract distribution functions which describe the stochastic processes *"Wind Turbine Power"*, *"Photovoltaic Array Power"*, *"Combined Renewable Power"* and the *"State of Charge"* of the battery.

The distribution functions have been used to generate synthetic time series and calculate first passage time values. Having written a programme it has been possible to calculate and illustrate the distribution functions, time series and first passage time values for a variety of parameters and scenarios. By this way it has been demonstrated that the usage of both wind turbine and photovoltaic array do stabilize the power supply function if there is either a wind speed slump or a clearness index slump. Moreover, the programme has permitted the comparison of two different algorithms to calculate the first passage time. The graphical presentation of distribution functions, time series and first passage time functions has helped to gain a deeper understanding of the stochastic processes involved in the system. In the introduction to the statistical system modelling it has been pointed out that the algorithms developed here can be used to design a controller that operates the system more efficiently. It has been stated that the time series algorithms can be used for both off-line optimization of some of the fixed parameters (such as the ratio between rated wind and photovoltaic array power) and on- line operation.

Finally, it is the author's pleasure to thank Dr. David Infield for many discussions, ideas, references and fruitful suggestions and Jonathan Cauldwell for his support.

6. Appendix I: Statistics

This appendix introduces the terminology and outlines some of the statistical methods used in this paper. These are in particular the concepts of the distribution functions and the autocorrelation function of a stochastic process.

6.1 Probability Distribution Functions

6.1.1 Continuous Distribution

A random variable is a transformation that maps the outcome of a random experiment to a real number. This real number is often referred to as a realization of X. The distribution function F(x) of a random variable X is the (theoretical) probability that the actual realization of the experiment will be less or equal the value x. Hence it can be written as

$$F(x) = p(X \leq x) \tag{6.1}$$

From (6.1) it can be concluded that F(x) is monotonic and it is $F(-\infty) = 0$ and $F(\infty) = 1$. Its first derivative,

$$f(x) = \frac{\partial F(x)}{\partial x} \tag{6.2}$$

is called the probability density function. In case the probability density function is known, the corresponding distribution function can be evaluated via the integral

$$F(x) = \int_{-\infty}^{x} f(\xi)\, d\xi \tag{6.3}$$

The same principles apply to two- dimensional distributions: Two random variables X and Y constitute the *joint distribution function*

$$F(x,y) = p(X \leq x, Y \leq y) = \int_{-\infty}^{x} \int_{-\infty}^{y} f(\xi,\eta)\, d\eta\, d\xi \tag{6.4}$$

with the joint probability density function f(x,y). In case the two random variables X and Y are statistically independent, the joint distribution function will just be the product of the two one- dimensional distribution functions $F_X(x)$ and $F_Y(y)$, $F(x,y) = F_X(x)F_Y(y)$.

6.1.2 Discrete Distribution

Often, the number of possible realizations of a random experiment is finite, as for example in the case of a dice. In this case the theoretical probability for one particular realization x_i with index i will be written as p_i. In this instance the distribution function has the shape of a stair function,

$$F(x) = \sum_{i=-\infty}^{\infty} p_i \, s(x-x_i) \tag{6.5}$$

where $s(x - x_i)$ stands for the unit step function

$$s(x-x_0) = \begin{cases} 0 & , x < x_0 \\ 1 & , x \geq x_0 \end{cases} \tag{6.6}$$

The corresponding probability density function will then be a series of weighted delta functions:

$$f(x) = \sum_{i=-\infty}^{\infty} p_i \, \delta(x-x_i) \tag{6.7}$$

For both numerical and graphical reasons the occurence of the delta function is often inconvenient. In this paper we have mostly calculated the probabilities p_i , depicted them in various graphics over the i - axis and called the p(i) relationship *probability function* in contrast to the proper probability density function. From a given distribution function F(x) the single event probabilities p_i can be calculated via the relation $p_i = F(x_i) - F(x_{i-1})$, which makes it very easy to switch from distribution to probability function and vice versa. As a result the distribution function F(x) too has only a finite number of values and can therefore be written as

$$F_i = \sum_{j=1}^{i} p_j \quad , \sum_{j=1}^{\Lambda} p_j = 1 \qquad , i=1 \ldots \Lambda \tag{6.8}$$

where Λ denotes the number of discrete levels.

6.2 Functions of Random Variables

Assume a random variable X with distribution function F(x) and corresponding density function f(x), whose realizations are channelled through a system with an input- output characteristic function H(x). The output can be described by a random variable Y with distribution function G(y). For the sake of simplicity we will only mention two special cases. First, it is assumed that H(x) is strictly monotonic in the interval $x \in [a,b)$. H(x) is constant in the interval [b,c] and zero below a and above b, continuous at both a and b. At first glance, these restrictions seem to be purely arbitrary. They reflect, however, exactly the course of the characteristic of the wind turbine (3.1). The distribution function of the output will then be

$$G(y) = \begin{cases} 0 & , y < H(a) \\ F(x(y)) + F(c) - F(b) & , H(a) \leq y \leq H(b) \\ 1 & , y > H(b) \end{cases} \tag{6.9}$$

where x(y) denotes the inverse function of H(x) in the interval [a,b). In the second special case we assume a linear transform $H(x) = \alpha x + \beta$. Here, the distribution function is simply

$$G(y) = F\left(\frac{y-\beta}{\alpha}\right) \tag{6.10}$$

with the corresponding probability density function

$$g(y) = \frac{1}{|\alpha|} f\left(\frac{y-\beta}{\alpha}\right) \tag{6.11}$$

Such a linear transform of a random variable is the input- output characteristic of the photovoltaic array (see chapter 2.2.4).

Now consider a function $Z = g(X,Y)$ of two random variables X and Y. The random variables can be described by the joint probability density function f(x,y). Here, we will be noting the density function $f_Z(z)$ of the new random variable Z for three special cases, all of which occur in this paper.

$$\textbf{Sum:} \quad Z = X+Y \quad F(z) = \int_{-\infty}^{\infty} f(x, z-x)\,\mathrm{d}x$$

$$\textbf{Product:} \quad Z = XY \quad F(z) = \int_{-\infty}^{\infty} f\left(x, \frac{z}{x}\right)\frac{1}{|x|}\mathrm{d}x \tag{6.12}$$

$$\textbf{Quotient:} \quad Z = \frac{X}{Y} \quad F(z) = \int_{-\infty}^{\infty} x f(zx, x)\,\mathrm{d}x$$

The expression for the sum can be considerably simplified if statistical independence of X and Y is presumed. By this way the density function of Z can be concluded without knowledge of the joint probability density, just by evaluating the convolution integral

$$f_Z(z) = \int_{-\infty}^{\infty} f_x(x)\, f_y(z-x)\,\mathrm{d}x \tag{6.13}$$

where $f_x(x)$ and $f_y(y)$ are the density functions corresponding to X and Y. With the help of this relationship we were able to formulate a distribution of the sum of both wind and solar power in chapter 4.1.4.

6.3 Conditional Distributions

A conditional distribution in the context of this paper is a distribution of a random variable subject to a specific condition. Often this condition is an observation of the underlying stochastic process at another time. A conditional distribution function is written in the form $F(y \mid X=x)$, which signifies the distribution of the random variable Y under the condition that another random variable X maps onto its realization x. Given the joint probability distribution function $f_{xy}(x,y)$ of two random variables X and Y and the probability density function of Y, $f_y(y)$ the conditional probability density function $f_x(x \mid Y=y)$ can be calculated from

$$f_x(x \mid Y=y) = \frac{f_{xy}(x,y)}{f_y(y)} \tag{6.14}$$

6.4 The Autocorrelation Function

A stochastic process is a time dependant process which can be described by a probability distribution function F(x) and the autocorrelation function $R_{xx}(\tau)$. The latter is a measure for the correlation between the realizations of the random variable at time zero and time τ. An autocorrelation function value of zero signifies that the realization at time τ is not in any way dependant on the value of the realization at time zero. Assuming the stochastic process to be stationary (the statistical characteristics such as mean value and variance are time independent) and ergodic[12] the autocorrelation function can be worked out from

$$R_{xx}(\tau) = \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} x(t)\ x(t+\tau)\mathrm{d}t \tag{6.15}$$

with x(t) being a realization of the process over the time t. If x(t) represents an energy variable the autocorrelation function $R_{xx}(0)$ at $\tau = 0$ can be interpreted as the average process power. This characteristic brings about the Wiener- Chintchin transform from the autocorrelation function $R_{xx}(\tau)$ to the corresponding power spectrum $S_{xx}(\omega)$, which is formally on a par with the Fourier transform,

[12]Assume a stochastic process as an output of an experiment. The output is $s_i(t)$ as a function of time. The experiment is repeated N times (i=1...N). Now, the values of $s_i(t_k)$ (N values) can be put together in a sample k. A stochastic process is called ergodic if the statistical values of any sample coincide with the ones of any time function. It is worth noting that it can not be proved that a stochastic process is ergodic or not. It is more a conceptual idea. Ergodicity is, however, usually assumed as it enables to evaluate the autocorrelation in the time domain without knowing the joint probability distribution.

$$S_{xx}(\omega) = \int_{-\infty}^{\infty} R_{xx}(\tau)\,e^{-i\omega\tau}\,d\tau$$
$$R_{xx}(\tau) = \frac{1}{2\pi}\int_{-\infty}^{\infty} S_{xx}(\omega)\,e^{i\omega\tau}\,d\tau \tag{6.16}$$

The double index *xx* is there to remind one of the random variable X that stands behind the stochastic process.

For the description of time discrete processes the same concepts apply. Only the results have to be adjusted accordingly. Given a series of observations x_i ($i \in \mathbb{N}$) taken at in constant time intervals T, the autocorrelation coefficients

$$R_j = \lim_{N\to\infty} \frac{1}{N}\sum_{l=0}^{N-1} x_{l-j}x_l \tag{6.17}$$

converges towards the proper autocorrelation function $R_{xx}(jT)$, presumed stationarity and ergodicity. In full analogy to the Fourier transform (6.16) in the time continuous case, here the discrete Fourier transform will yield the power spectrum:

$$S_{xx}(\omega) = \sum_{k=-\infty}^{\infty} R_k e^{-ik\omega T}$$
$$R_k = \frac{T}{2\pi}\int_0^{\frac{2\pi}{T}} S_{xx}(\omega)\,e^{ik\omega T}\,d\omega \tag{6.18}$$

The inverse transform, however, is not part of the discrete Fourier transform as the power spectrum has not been discreteized.

6.5 Normal Distribution and Normal Process

6.5.1 Normal Distribution

The so called *standard normal distribution* or *Gaussian distribution* is a distribution defined by the probability density function

$$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{1}{2}\frac{(x-a)^2}{\sigma^2}\right) \tag{6.19}$$

Its mean value is a, its standard variation σ. For the special case of $a = 0$, $\sigma = 1$ the distribution is called *standard normal* or *Gaussian distribution* and the corresponding distribution function is defined by ([1], def. 26.2.2)

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{1}{2}\xi^2\right) d\xi \tag{6.20}$$

The distribution function of a normal distribution is then

$$F(x) = \Phi\left(\frac{x-a}{\sigma}\right) \tag{6.21}$$

The probability density function of two- dimensional or bivariate normal distribution for two identical distributed random variables X and Y with zero mean, standard variation σ and correlation coefficient r is given by

$$f_{xy}(x,y) = \frac{1}{2\pi\sigma^2\sqrt{1-r^2}} \exp\left[-\frac{1}{2(1-r^2)}\left(\frac{x^2+y^2-2rxy}{\sigma^2}\right)\right] \tag{6.22}$$

where the correlation coefficient is defined via the covariance v_{xy} , $r = v_{xy} / \sigma^2$.

6.5.2 Normal Process

A stochastic process X(t) is called normal if the random variables $X(t_1)$, $X(t_2)$ ... belong to a multi- dimensional normal distribution. The probability density of X(t) under the condition of a given observation x_0 at time $t = 0$ can be calculated via (6.14) and (6.22) and it is

$$f(x|x_0) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{1}{2}\frac{(x-x_0 r)^2}{1-r^2}\right) \tag{6.23}$$

The corresponding distribution function can be expressed in terms of the Gaussian distribution (6.12),

$$F(x|x_0) = \Phi\left(\frac{x - x_0 r}{\sqrt{1-\sigma^2}}\right) \tag{6.24}$$

Hence, its mean value is the product rx_0 and time dependant if r is a function of t.

6.6 Random Numbers

This section discusses random number generators that are able to retrieve numbers drawn from a given distribution. In fact, they are algorithms that return a number each time they are called upon. As they all have a period after which they will repeat the same sequence of numbers, the numbers are called pseudo random. Thanks to long periods the numbers appear however to be random. The Kolmogorov- Smirnov- test ([33], p.623) may be applied to check whether the empirical distribution of a stochastic process matches a theoretical distribution function. Its measure is the maximum value of the absolute difference between the theoretical distribution function and the empirical distribution function of a given sample of numbers. The Kolmogorov- Smirnov test is, however strictly not applicable to check the performance of a random number generator. The following sections discuss random generators for several distribution functions. For more details on their implementation and typical results of the corresponding Kolmogorov- Smirnov- tests refer to section 7.1.2.2.

6.6.1 Uniform Deviates

A uniform deviate is a random number drawn from a uniform distribution. It is assumed to return numbers that are evenly distributed over the open interval (0,1). Throughout this chapter we will denote a uniform deviate with $\breve{u} \in (0,1)$. In the following chapters it will be discussed how a uniform deviate can be used in order to generate random numbers drawn from a normal distribution (chapter 6.2) or any discrete distribution (chapter 6.6.3). They are necessary to produce synthetic time series of the wind speed and clearness index fluctuations.

6.6.2 Transformation Method and Normal Deviates

Assume random numbers are to be generated, drawn from a distribution that can be described by its probability density function f(x) or the corresponding distribution function F(x). Given a uniform deviate u (uniformly distributed in (0,1) a random number y of some arbitrary distribution F(x) can be generated via the inverse function of F(x),

$$y(u) = F^{-1}(u) \tag{6.25}$$

This method is, however, not always feasible and depends on whether $F^{-1}(x)$ can be evaluated or not.

A normal deviate is a random number y, drawn from a normal distribution with mean x_{mean} and standard deviation σ^2. If $x_{mean} = 0$ and $\sigma = 1$, the numbers may be called *standard normal deviates*. They will be denoted with y_s. The corresponding distribution function is the standard normal distribution (defined in equation (6.20). There are many methods to generate standard normal deviates using uniform deviates u ($0 < u < 1$), two of which will be discussed briefly. The first method applies (6.25) directly. For the inverse of F(x) an approximation has been used ([1], eq. 26.2.22). Thanks to the symmetry of the normal distribution,

$$\Phi(x) = 1 - \Phi(-x) \tag{6.26}$$

the random number y_s may be worked out from the relationship

$$y_s = F^{-1}(u) \approx \begin{cases} \dfrac{a_0 + a_1 t}{1 + b_1 t + b_2 t^2} - t \ , \ t = \sqrt{\ln\left(\dfrac{1}{u^2}\right)} & , \ 0 < u \leq \dfrac{1}{2} \\ -F^{-1}(1 - u) & , \ \dfrac{1}{2} < u < 1 \end{cases} \tag{6.27}$$

with the coefficients $a_0 = 2.30753$, $a_1 = 0.27061$, $b_1 = 0.99229$ and $b_2 = 0.04481$. The second method is the *Box-Muller* ([33], p.289f) method. Given two uniform deviates u_1, $u_2 \in (0,1)$ and applying the transfer methods for two variables, it can be shown the the two parameters y_1 and y_2,

$$y_1 = \sqrt{-2\ln x}\cos(2\pi x_2)$$
$$y_2 = \sqrt{-2\ln x}\sin(2\pi x_2) \tag{6.28}$$

are both independently distributed according to the standard normal distribution $\Phi(x)$. Both methods require one uniform deviate for each normal deviate. The Box-Muller method, however, requires less computing time. It was therefore the one that has been implemented in the project. Having determined a standard normal deviate y_s, it can be easily transferred to a normal deviate y by computing

$$y = \sigma y_s + x_{mean} \tag{6.29}$$

6.6.3 Deviates of Discrete Distributions

As shown above, discrete distributions can be described by the distribution coefficents F_i (6.8). Given a uniform deviate $u \in (0,1)$ a random number y of the discrete distribution can be obtained via

$$y = \{i \mid (F_i \geq u \,,\, F_{i+1} \leq u)\} \tag{6.30}$$

This means that y returns the i for which $F_i \geq u$ and $F_{i+1} \leq u$ is. Hence, this is in fact the transformation method for discrete distributions.

7. Appendix II: Programme Documentation

7.1 Functional Specification

7.1.1 Getting Started

A programme has been written that carries out all the calculations described in this paper. It runs on a Windows 3.1 environment. The executable file is called **owrenw.exe.** In order to run it successfully the dynamically linked library **bwcc.dll** has to be accessible during run-time. To make sure that Windows is able to find it, it has to be in one of the following directories:

- In the same directory as **owrenew.exe,**
- In the Windows system directory
- In a directory that is included in the environment variable PATH.

The file **owrenew.dlg** contains user preferences and chosen parameters of the last session. It should reside in the same directory as **owrenew.exe.** It is not necessary to run the programme, but will be automatically created upon exiting the programme to Windows. After starting the programme a new window will appear on the screen, which is the main window of the application. Its main features are a menu bar to select further actions and a white board for graphical display. It is best to click with the mouse on the top right hand corner button to maximise the main window. The programme can be exited via Alt-D-X or by double clicking the top left hand corner. The programme has a Windows icon associated with it that can be included by using the Windows Setup utility.

7.1.2 Programme Description

In this section all menu options are described along with the dialog windows they will cause to open. There are 5 main items on the menu bar:

- **Distributions:** For all calculations of probability distribution functions.
- **Applications:** For random number generators, time series and the first passage time problems.
- **Options:** Setting up usere preferences and parameters.

- **Export:** Exporting data to Word Perfect Presentation.
- **Help:** The on-line help feature is not implemented.

7.1.2.1 Distributions

(i) Wind Speed Distribution

The dialog window *"Wind Speed Distribution"* prepares for the calculations of the stationary probability density function of short term wind speed fluctuations as described in section 4.1.1. It permits to select either the calculation of the probability density function or the corresponding distribution function. Moreover, it asks for four parameters to be specified:

- **Mean wind speed:** This is the mean wind speed $\bar{v}$ (equation 4.1) in m/s.
- **Minimum wind speed:** This is only for display purposes. The first value to be calculated will be v = *minimum wind speed.*
- **Maximum wind speed:** This is the last value to be calculated.
- **Number of evaluations:** Number of points to be calculated within the open interval [*minimum wind speed, maximum wind speed*].

Other parameters such as the wind speed standard deviation should be specified in the *Settings* dialog window (see below). Once all parameters are set, press the OK button of the *"Wind Speed Distribution"* window. The dialog window disappears and a new *Calculations* dialog window appears on the screen. Press OK to start calculations. The progress of the calculations can be monitored by looking at the *Calculations* window where the elapsed time and some other bits of information are depicted. Press OK (or ENTER) once the calculations are finished in order to continue. The calculated points are now shown in a graph in the main window.

(ii) Wind Power Distribution

The dialog window *"Wind Power Distribution"* prepares for the calculation of distribution functions of the wind turbine power (section 4.1.2). It allows to choose between probability density function and distribution function as well as between stationary and conditional distribution. Parameters to be specified prior to continuation are:

- **Mean wind speed**: Same as in (i)
- **Steps on power axis**: This is the number of discrete levels along the power scale. See equation (4.5) in section 4.1.2.
- **Time tau [s]**: The time τ for which the distribution function is to be calculated. It appears in the autocorrelation coefficient r_v in equations (4.1) and (4.2). It is only to be specified if the conditional function is chosen.
- **Initial wind speed**: The initial wind speed v_0 in equations (4.3) and (4.4) in the case of a conditional distribution. This field is grey and cannot be selectec if the stationary distribution is selected.

Again, other parameters may be specified in the *Settings* window. Once having pressed the OK button the procedure is identical to (i).

(iii) Solar Power Distribution

This is the dialog window for the calculations described in section 4.1.3. Again, it gives the option to choose between probability density function and distribution function. Furthermore, the user has to select one of the following options:

- **Analytical Distribution:** This denotes the distribution function (4.9) using the Beta-function and not the approximation via Gaussian functions. It is for stationary distributions only.
- **Approximation:** This is now the distribution function (4.11) employing the approximation, though only for stationar distributions.
- **Conditional Distribution:** This is the conditional distribution (4.11), (4.12) for which an initial clearness index k has to be specified.
- **Quality of Approximation:** Having selected this option the difference between the analytical solution and its approximation is calcualted (equation 2.90).

Parameters can be entered too:

- **Average hourly clearness index k:** See discussion in section 2.2.4.1.
- **Standard deviation σ_k:** See discussion in section 2.2.4.1.
- **Steps on power axis:** See above (ii).
- **Time tau [s]:** See above (ii).
- **Initial clearness index k:** This field can only be entered if the conditional

distribution is to be calculated.

- **Number of trial points:** This is variable M in equation (2.85), an optimization variable - not necessary if stationary distribution is to be calculated. For reasonable values refer to discussion in section 2.2.4.2.
- **Number of coefficients:** This is variable Q in equation (2.85) and is not necessary for stationary distributions. Again, for reasonable values refer to section 2.2.4.2.

Furthermore, the user can tick the **Bypass** option. If a distribution is to be calculated that is based on the approximating formula, various optimisation parameters have to be determined prior to evaluating the distribution formula (2.82). The calculated optimisation parameters are stored in a file <solar.dat>. In case the same input parameters hold true the next time the approximation is used, the optimisation parameters are read from the file rather than repeating the same calculation - though only the **Bypass** - option is selected. In order to save time make sure the option is always selected. In case the input parameters don not match with the parameters on the file the optimisation calculation will be carried out anyway.

(iv) Joint Renewable Distribution

Here is the dialog box for the calculation of combined power distributions as outlined in section 4.1.4. The layout of the window is very similar to the other distribution dialog windows giving the user the option to select between the joint density function (stationary) and the joint conditional distribution as defined by equation (4.15). The only additional parameter is the fractional power factor ζ (equation (3.15)).

7.1.2.2 Applications

(i) Random Numbers

This dialog box and the corresponding calculations have been implemented in order to check the quality of random numbers generating algorithms as discussed in section 6.6. The user can choose one distribution type and enter relevant distribution parameters. Upon pressing the OK button, the programme will generate N (as specified in the input field **Number of trials**) random numbers and calculate the sample's mean value and variance. Moreover, it will carry out a Kolmogorov- Smirnov test and print out the test result. The number of classes necessary for the test can be inserted in the input field **Number of classes.** For more details on the implementation of the Kolmogorov- Smirnov test and the significance of the test result see [33], page 623ff. Tests can be repeated by pressing the **Retry** button.

- **Uniform distribution:** In order to generate uniform deviates the random number generator of the C- standard library is used, whose period length is guaranteed to be 2^{32} ([5], rand()). The expected theoretical mean value of a distribution which is uniformly distributed in [0,1] is 0.5, its variance is 1/12 = 0.08333. A typical result is mean 0.5163 and variance 0.08501 with N = 100 trials. As mentioned in section 6 the uniform deviates are used to generate other random numbers, such as normal deviates.
- **Normal distribution:** The generator of random numbers taken from a standard normal distribution with zero mean and variance 1 is implemented using the Box-Muller method (see section 6.6.2). A typical result (for N = 100) is mean 0.04730 and variance 1.04078. Normal deviates are used in all time series calculations that include the wind speed distribution.
- **Beta distribution:** This random number generator is implemented by employing the rejection method for continuous distributions (compare [33], p.290). It is, however, never used for time series calculations. It is here more for development purposes and is now obsolete.
- **Binomial distribution:** Binomial deviates are generated using the rejection method as introduced in section 6.6.3. Although the binomial distribution is not required in the time series calculations of this paper it has been implemented here to confirm the rejection method using a well known discrete distribution. The binomial distribution depends on two parameters, n and p. Here, n is the number of trials and p the probability that an event occurs. The theoretical mean is np, its variance np(1-p). As the binomial distribution is a discrete distribution, the Komogorov- Smirnov test is not applicable. Though, test results of the mean value and the variance suggest that the implemented method is reliable. It is used for all time series calculations involving discrete distributions.

(ii) Time Series

The dialog window *"Time Series"* prepares for the generation of time series as discussed in section 4.2. The window is divided into three parts. First, the user can select one of the following time series:

- **Wind Speed:** Wind speed time series as outlined in section 4.2.2.1.
- **Wind Power:** Wind turbine power time series as outlined in section 4.2.2.2.
- **Solar Power:** Photovoltaic array power time series as outlined in section 4.2.2.3.
- **Combined Renewable:** Joint renewable power time series as outlined in section 4.2.2.4.
- **Battery: State of Charge:** State of charge time series as outlined in section 4.2.2.5.
- **Power Deficit:** Here, the programme generates a time series of the joint renewable power and tracks the state of charge of the battery. It then compares the power supplied by the renewable energies and the battery with the power demand. If the power demand is greater, hence if there is a power deficit it will go into the power deficit time series. If there is no deficit, the time series value will be zero. A power surplus is not recorded.

Second, the user has to enter initial values (dependend on the chosen type of time series):

- **Initial wind speed [m/s]:** Field only visible if selected time series use the wind.
- **Initial clearness index k(0):** Field only visible for calculations including the PV array.
- **Available charge Q10:** Field only visible for calculations which need the battery.
- **Bound charge Q20:** Field only visible for calculations which need the battery.

Third, there are two input fields that are applicable to all time series calculations:

- **Time step [s]:** This is the implied time interval between two time series values and corresponds to Δt in section 4.2.1.
- **Number of points:** Number of time series values to be generated in one calculation.

(iii) First Passage Time Problems

The dialog window *"First Passage Time Problems"* refers to the calculations in section 4.3. First, the user selects the underlying, physical process: Wind speed, wind turbine power, solar power or joint renewable power. Second, he selects the method to be used, which is either **Time Series Approach** (see section 4.3.1) or **Markov Chain Approach** (see section 4.3.2). Third, he can select a calculation technique:

- **Calculate one passage time value only**: For a given initial value and a chosen passage level the programme computes the first passage time.
- **Passage time as function of initial value**: For a given, fixed passage level the programme computes a series of first passage times. The first value to be calculated assumes the value entered into one of the initial value fields as initial value. The last value to be calculated assumes the initial value to be identical to the passage level. The total number of values to be calculated is specified in the input field **Number of values.**
- **Passage time as function of passage level**: For a given, fixed initial value (or a set of initial values in the case of joint renewable power) the programme computes a series of first passage times. The first value to be calculated assumes the passage level to be identical to the initial value. The last value to be calculated assumes the passage level to be the value entered into one of the passage level input fields. Again, the total number of values to be calculated is specified in the input field **Number of values.**

Fourth, there are some additional input fields, which may not be visible, depending on the selection of the process, the method and the calculation technique.

- **Underlying time step**: Only applicable if time series approach is selected. It has the same significance as in the *Time series* dialog window above.
- **Initial wind speed**: Initial wind speed in [m/s].
- **Initial clearness index**: Initial clearness index k(0).
- **Initial power**: Initial, normalised power $\in [0,1]$.
- **Wind speed level**: Passage level for the wind speed in [m/s].
- **Clearness index level**: Passage level for the clearness index k.
- **Power level**: Passage level of the normalised power $\in [0, 1]$.

7.1.2.3 Options

(i) Settings

In the *"Settings"* dialog window the user can enter parameters of physical relevance. Values entered here are used by the calculations unless altered in another dialog window. However, if a parameter of the *Settings* window is altered in another dialog box, it will be updated in the *Settins* window as well, so that there is never an ambiguity which value might be used in calculations as it is always the value last seen be the user.

- **Cut-in wind speed:** See section 3.1.
- **Cut-out wind speed:** See section 3.1.
- **Rated wind speed:** See section 3.1.
- **Mean wind speed:** See section 2.1.
- **Wind standard deviation:** See equation (2.3).
- **Auto correlation coefficient** β**w:** Wind speed autocorrelation coefficient β_v (see equation (2.9) and discussion below it).
- **Max clearness index K0:** This is parameter K_0 in equation (3.14).
- **Hourly clearness index k:** This is the hourly average clearness index k as introduced in section 2.2.4.1.
- **Standard variation** σ**k:** Standard variation of the hourly clearness index k, as defined in equation (2.65).
- **Auto correlation coefficient** β**s:** Autocorrelation coefficient β_x of the normalised clearness index x, as defined in equation (2.81).
- **Fractional power factor zeta:** Definition in equation (3.14).
- **Battery: Factor k:** All battery parameters refer to the Manwell model in section 2.3.2.3 part (iii).
- **Battery: Factor c:** see factor k above.
- **Battery: Qmax [Ah]:** This is the battery capacity Q_b as discussed in section 2.3.2.2. Please note that the value to be entered should be in Ampère hours.
- **Battery: Voltage [V]:** This is the (constant) battery voltage. See discussion of Manwell model in section 2.3.2.3 part (iii).
- **Nominal Renewable Power [W]:** The combined (non normalised), maximum renewable power in Watt, as defined in equation (3.15). Hence, this is the total installed power. This parameter is only used for state of charge time series.
- **Power Demand [W]:** This is the power demand P_{ex} as in section 4.2.2.5.

(ii) Maths

In the *"Maths"* dialog box the user can specify some mathematical parameters:

Solar Power: Approximation of Distribution

- **Number of coefficients:** See discussion of *Solar Power Distribution* window.
- **Number of trial points:** See discussion of *Solar Power Distribution* window.

First Passage Time Problem

- **Number of time series:** Number of time series taken into account while calculating the first passage time using the time series approach. Refer to discussion in section 4.3.1.
- **Max number of iterations:** (Time series approach) See discussion of time series approach algorithm in section 4.3.1, point (10).
- **Max number of iterations:** (Markov chain approach) See discussion of Markov chain approach algorithm in section 4.3.2, point (14).
- **Stopping criterion:** Stopping criterion in Markov chain approach to first passage times. See discussion of algorithm in section 4.3.2, point (13).
- **Number of grid points:** This parameter is a software development parameter and is now without any significance.

Process Discretization

- **Number of classes:** For discrete distributions that are discretised along the power axis. Refer to equations (4.5) or (4.8).

(iii) Directories

In the *"Directories"* window the user can specify the location of dialog or user files.

- **Solar Data:** The optimisation parameters for the approximation of the PV array power distribution are stored in the file with the name specified here. Please refer to the discussion on the bypass option in the *Solar Power Distribution* window.
- **Dialog Data:** This is a software development field which is now not used at all.

(iv) Display

In the *"Display"* dialog window the user is given a variety of options for display purposes.

- **Auto display of graphics:** If this option is ticked, the graph of the last calculation will be automatically rebuilt after the display of other dialog windows. If the option is switched off, the graph is shown right after the calculation but is not being shown once another dialog box has been opened.

- **Accumulate data series:** If this option is switched on, up to 4 data series are accumulated and shown in the graph at the same time (in different colours). If the option is switched off only one data series is shown in the graph.
- **Ask for legend text:** If this option is switched on, the programme asks the user for a legend text to be associated with a curve. The legend text does not appear on the screen. It is, however, exported to Word Perfect Presentation. See discussion on the dialog window *Export Data.*

(v) Export Data

The *"Export Data"* dialog window prepares for the export of the data of the most recently calculated data series to a file. If the option *"accumulate data series"* is switched on the data of all curves in the latest accumulation are exported. The format of this export file is data compatible with import requirements for Word Perfect Presentation diagrams. Hence, data calculated here can be exported to diagrams in Word Perfect Presentation. All diagrams in this paper have been produced using this technique.

- **New file:** Save data to a new file. If file already exists, its content will be overwritten.
- **Attach data to file:** Append data of last curve to the end of the specified file.
- **File name:** Name of the file the data should be sent to. If no pathname is specified, the current working directory is assumed.

7.1.2.4 Help

The on-line help is not implemented.

7.1.3 Bugs and Errors

The programme is designed in way so that it is unlikely to crash. Every input field (i.e. fields into which the user can type) are thoroughly checked. Messages do appear if the format is wrong. For instance if the user types a word where a number is expected. Moreover, messages do warn the user if the programme thinks some input parameters are out of range. For instance, if the user enters 1.2 into a normalized parameter field that expects only numbers between 0 and 1, or if the cut-in wind speed is greater than the cut-out wind speed. In these instances the user can choose to abort the intended action or to ignore it. It is strongly recommended that the user never ignores the warning as this may result in severe errors. Remember that warnings are given for a reason. The option to ignore is implemented for software development purposes only.

Most internal errors should be captured before a crash and an error message is printed out on the screen while the programme is suspended. Although these errors are not damaging, they are not intended to occur. As at print time no situation is known of where such an error occurred.

It may happen that after some time that the headline in the graphs is displayed in a small font rather than a big font. This is due to the limited number of font resources in Windows. The problem has been recognised but not fixed. It has, however, no impact on anything else. If a user cannot live without the big font, he is advised to quit Windows and start Windows again. Other bugs are not known.

7.2 Technical Design

In this section the design of the programme is discussed. It is written in C++, using the Borland C++ 3.1 compiler for Windows. It uses the standard C/C++ library, Borland Class library and the Object Windows C++ library. Readers who are not familiar with C++, object oriented programming and Object Windows C++ may find this section difficult to understand. Object Windows C++ ([3], [4]) is a class library that is used for all windows in the programme. The next paragraph gives an overview of the files that make up the source code. It is followed by a discussion of the main programme and an outline of the implementation philosophy. Although the number of classes and functions may seem at first glance hard to swallow, the concept is simple and the structure logical. After the introduction into the programme idea section 7.3 gives a complete class reference, discussing all classes and their public and protected members. Section 7.4 describes all global functions. From there it should be no problem to undeerstand the source code.

7.2.1 The File Structure

7.2.1.1 Header Files

Header files in C/C++ (extension .h) are there to define classes and constants, declare global functions and data types and define macros. Every class, structure, function or data type is defined in a header file. A listing of all header files is printed in section 7.5 of this paper. The header files can be grouped as follows:

(i) General Purpose C- Functions

These header files define constants and functions that can be considered as an extension of the standard C- library.

<boolwin.h>	Definition of Boolean constants TRUE, FALSE, YES, NO, OK and some mathematical constants.
<cstring.h>	Definition of functions on C- strings.
<error.h>	Definition of an error handler.

(ii) Mathematical functions and classes

These header files define mathematical functions and objects. They are not project specific. Among the classes are an implementation of a vector class, matrix class and a class that represents functions of one variable.

<diffcalc.h>	Definition of the class *objfunc* which is the implementation of a function of one variable.
<mathfunc.h>	Declaration of mathematical functions.
<vectors.h>	Definition of the classes VECTOR and MATRIX.

(iii) Windows

These header files define all objects that are inherited from Object Windows C++ classes. Hence, the prefix 'ow'. These objects are usually windows or dialog boxes used in the project.

<owcalc.h>	Definition of all window objects on which calculations are carried out.
<owdialg.h>	Definition of all dialog windows.
<owlappl.h>	Definition of general purpose dialog windows or input fields in dialog boxes.
<owparam.h>	Definition of the structure Param. This structure acts as in interface between dialog windows and calculation related classes. Definition of class Graph which acts as an interface between calculations and the graphic window TGraph.
<owplot.h>	Definition of graphic related classes.
<owrenew.h>	Definition of the graphic window, TRenewPlot, the main window, TMainWindow, and the main application, TRenewApp.
<owres.h>	Definition of all constants used for the windows resources.
<owstat.h>	Definition of abstract calculation windows classes.

(iv) Project Objects

These header files define all mathematical objects that are directly project related.

<distrib.h>	Definition of classes in the context of distribution functions: E.g. the implementation of a discrete distribution or a continuous distribution.
<joint.h>	Definition of the class *ProbJointPower*, the implementation of the joint renewable power distribution.
<passage.h>	Definition of first passage time problem related classes.
<random.h>	Definition of random number generator related classes.
<series.h>	Definition of time series related classes.
<solar.h>	Definition of classes that deal with the photovoltaic array and the distribution of the PV array power.
<wind.h>	Definition of wind and wind power related classes.

7.2.1.2 Source Files

Source files (extension *.cpp) contain the code for the functions (or class member functions) defined in the header files. There is usually a mapping between header files and source files. E.g. the code for functions defined in *wind.h* can be found in *wind.cpp*. There are just two exceptions to this rule. First, there is no source file *boolwin.cpp* as the header *boolwin.h* does not define any functions. Second, the functions contained in the source file *linalg.cpp* are defined in the header file *mathfunc.h*. A listing of the source file *owrenew.cpp* is included in section 7.5.2. The listing of other source files is not included in this paper in order to avoid overloading. The complete source code, though, is shipped together with the executable file. Readers interested in the complete source code are referred to the disk.

7.2.1.3 Resource File

Another important file is the resource file *owres.c* which contains data for the layout of the dialog windows, such as coordinates and other attributes. The resource file *owres.rc* has been created using Borland Resource Workshop ([6]). Some of the resources, such as input fields or dialog windows are given uinque identity numbers. These constants are defined in the header file *owres.h* which is included by the resource file and other source files.

7.2.1.4 Other Files

The file *owrenew.def* is to be included in the project file. It contains text that serves as information but is otherwise not important. The library file *bwcc.lib* is included in the project file *owrenew.prj* as well. This is the library that renders the dialog windows the 'Borland' look rather than the 'Microsoft' look. As mentioned earlier the file *bwcc.dll* should be accessible at runtime for the same reason. The programme does not work without. Finally, the project file *owrenew.prj* contains all files to be compiled and linked. It is a software development tool.

7.2.2 The Programme Structure

The main routine of the programme is located in *owrenew.cpp* right at the end (see listing in section 7.5.2). It is a typical Object Windows C++ routine. Readers who are not familiar with Object Windows C++ should first read the programming handbook ([3]).

In the main routine two classes are initialised, *param* and *GraphData*. Their significance is mentioned later. Then, an instance of the class *TRenewApp* is created, which is inherited from the Object Windows C++ class *TApplication*. The application is run. Upon exit of the application the objects *param* and *GraphData* are deleted. Now what exaclty happens in *TRenewApp*?

Basically, it initialises the main window, *class TMainWindow* (inherited from Object Windows C++ class *TWindow*), which is the window that is visible on the screen and contains the menu bar. Now, the programme works in the main window and waits for commands, such as a selection of one of the menus. Generally, every window is actually represented by a class. All events that happen in a window (such as the selection of a menu item or if the user presses a button) are handled in the corresponding class. Hence, actions in the main window are handled in *TMainWindow*. Have a look at the definition of *TMainWindow* in the header file *owrenew.h*. For instance, there is a function *CMWindSpeed ( )* = [CM_FIRST + cmWindSpeed]. This function is carried out as soon as the event 'cmWindSpeed' occurs. This particular event occurs as soon as the menu item 'Wind Speed Distribution' in 'Distributions' is selected. The function *CMWindSpeed* (see listing of *owrenew.cpp* in section 7.5.2) opens the dialog window 'Wind speed dialog', which is represented by the class *TSpeedDialog*, which is inherited from the Object Windows C++ class *TDialog*. It is defined in the header file *owdialg.h*. Now execution is transferred to the instantiation of *TSpeedDialog*. Here, the user can enter some parameters. If he presses the Cancel button the programme goes back to the main window. Otherwise it transfers execution to the next window, *TWindSpeedObject*, defined in header file *owcalc.h*. This is the calculation window. If the user presses the OK button the calculations are carried out by calling the member function *workOutValues()*. If the user selects OK after the termination of the calculations execution goes back one window to *TSpeedDialog* and from there to the main window *TMainWindow*. All the other menu items are handled in a similar way.

On top of the main window lays a graphic window, *TRenewPlot*, which is inherited from *TPlot* and the Object Windows C++ class *TWindow*. Every time the execution returns from the calculation window to the main window, the graphical window checks whether it has to draw a graph. *TRenewPlot* is defined in *owrenew.h* as well. It receives the data for the curves (i.e. the data of the last calculations) via the variable *GraphData* (definition in header *owparam.h*). The data calculated in *TWindSpeedObject* for instance are stored in *GraphData* and can be picked up by the graphic window *TRenewPlot* when it has to draw itself.

There is another interface variable worth mentioning. It is *param*, which is of type *Param* as defined in *owparam.h*. Every time a dialog window is initiated the default data for its input fields or radio buttons are taken from *param*. In fact, in the case of the dialog class *TSpeedDialog*, the appropriate data from *param* are loaded into an instance of a class *TTransSpeedDlg* (defined in *owdialg.h)* via its member function *setParameter()*. Then data are transferred to the dialog *TSpeedDialog* and appear on the screen. The user is now given the opportunity to overwrite the parameters in the input fields. If he chooses 'OK' at the end, the buffer *TTransSpeedDlg* is updated with the new data. So, if he opens the same dialog again, the input fields are now filled with the new data. Otherwise, if he chooses 'Cancel' the buffer is not being updated, which is indeed the functionality of a cancellation.

All actions are implemented in a similar way. Look at Fig. 7.1. Every dialog window that appears upon selection of a menu item in the main window is directly inherited from the base class *TDialog*. E.g. *TSettingsDialog* is the class corresponding to the settings dialog window. Every dialog class is given a parameter buffer class as described above. E.g. the buffer that corresponds to *TSettingsDialog* is *TTransSettingsDlg*. All calculations are carried out on the calculation window which is itself a dialog window. If a calculation is to be carried out that produces only one value, hence a graphical display is not possible, the class to be used is directly inherited from *TStatusWindow*. E.g. the class *TPassageTime*, when only one first passage time value at a time is to be calculated. If a whole curve is to be computed, the class to be used is inherited from *TMultiValObject*. E.g. *TWindSpeedObject*. In all classes with postfix 'Object' calculations are carried out. That means that their member functions initialise the mathematical objects. There are no mathematics involved in classes with postfixes 'Dialog', 'Dlg' or 'Window'.

This paragraph was intended to give an overview of the principles of the programme. All classes and their member functions as well as all global functions are listed and discussed in the following sections ordered by header files. Especially the class reference is - together with the source code - a very thorough documentation of the programme.

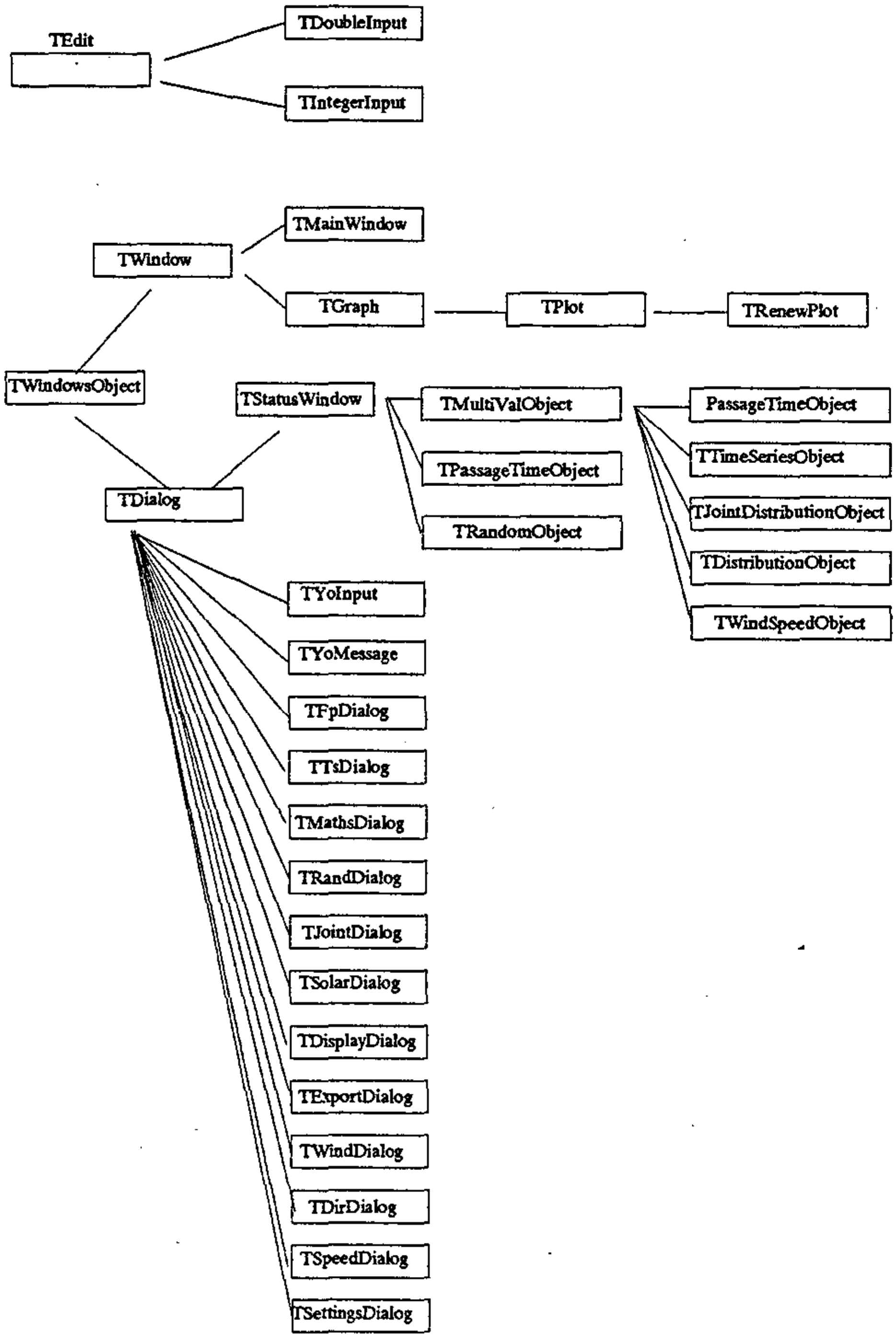

Fig. 7.1: Class Structure of Windows Objects

7.3 Class Reference

In this section a complete class reference is given. The first part consists of a list of all classes together with a short description and the header file it is defined in. In the second part the classes are discussed in more detail discussing all constructors, protected and public data elements, member functions and operators.

CLASSES - OVERVIEW

axis	Implementation of a coordinate axis	<owplot.h>
BetaKgSTest	Kolmogorov- Smirnov test for Beta-distribution	<random.h>
betaRand	Random number generator for beta-distribution	<random.h>
ContCondSolApprox	Conditional distribution of the PV array power	<solar.h>
ContCondWindPower	Conditional distribution of the wind turbine power	<wind.h>
ContinuousDistribution	Continuous distribution	<distrib.h>
ContSolAppQual	Quality of approximation	<solar.h>
ContSolApprox	Distribution of the PV array power using approximation	<solar.h>
ContSolApproxX	Conditional distribution of the normalised clearness index x	<solar.h>
ContSolExact	Analytical solution of the PV array power distribution	<solar.h>
ContSolExactX	Analytical solution of the distribution of the normalised clearness index x.	<solar.h>
ContWindPower	Distribution of the wind turbine power	<wind.h>
DiscretDistribution	Implementation of a discrete distribution	<distrib.h>
discretRand	Generation of random numbers of any discrete distribution	<random.h>
DiscretRandomizer	Random number generator for discrete distributions	<distrib.h>
DiscretWindSpeed	Discrete distribution of wind speed fluctuations	<wind.h>
DiscSolApprox	PV array power as a discrete distribution	<solar.h>
DiscretWindPower	Discrete distribution of wind turbine power fluctuations	<wind.h>

Graph	Interface between graphic window and calculations	<owparam.h>
JointPassageTimes	Object function for first passage times of joint renewable power fluctuations	<passage.h>
JointPowerTimeSeries	Joint renewable power time series	<series.h>
KgSTest	Abstract class of a Kolmogorov-Smirnov test	<random.h>
MATRIX_	Implementation of a matrix with real elements	<vectors.h>
MCPassageTime	First passage time using the Markov chain apporach	<passage.h>
MCWindSpeedPassageTime	First passage time of wind speed fluctuations using the Markov chain approach	<passage.h>
MCWindPowerPassageTime	First passage time of wind turbine power fluctuations using the Markov chain approach	<passage.h>
MCSolarPowerPassageTime	First passage time of PV array power fluctuations using the Markov chain approach	<passage.h>
MCJointPowerPassageTime	First passage time of joint renewable power fluctuations using the Markov chain approach	<passage.h>
MeritSol	Object to optimise the approximation used for the distribution of the PV array power.	<solar.h>
msgObjfunc	Function of one variable	<distrib.h>
NormKgSTest	Kolmogorov- Smirnov test for normal distribution	<random.h>
normRand	Generation of normal deviates	<random.h>
objfunc	Function of one variable	<diffcalc.h>
owObjfunc	Implementation of a function of one variable	<diffcalc.h>
pairvec	Double vector that stores x- and y-values	<diffcalc.h>
Param	Structure that holds parameters for dialog windows	<owparam.h>
PassageTime	First Passage Time Object	<passage.h>
PassageTimes	First passage time problems in case more than one value is to be calculated.	<passage.h>
PassageTimesObject	Calculation of first passage times	<owcalc.h>

PowerDeficitTimeSeries	Time series of the power deficit	<series.h>
ProbCondSolApprox	Conditional distribution of the PV array power as *statfunc* object	<solar.h>
ProbCondWindPower	Conditional distribution - representing the wind turbine power - as *statfunc* object	<wind.h>
ProbJointPower	Joint renewable power probability function	<joint.h>
ProbSolAppQual	Quality of approximation as *statfunc* object	<solar.h>
ProbSolApprox	Solar distribution (using the approximation) as *statfunc* object	<solar.h>
ProbSolExact	Analytical solar distribution as *statfunc* object	<solar.h>
ProbWindPower	Stationary distribution - representing the wind turbine power - as *statfunc* object	<wind.h>
rejectRand	Generation of random numbers of any distribution	<random.h>
SolarPowerPassageTimes	Object function for first passage times of PV array power fluctuations	<passage.h>
SolarPowerTimeSeries	Solar power time series	<series.h>
SolarRandomizer	Random number generator for the distribution of the PV array power	<solar.h>
SolConstants	Store for clearness index distribution parameters.	<solar.h>
Speed	Wind speed fluctuations	<wind.h>
SpeedDens	Probability density function of wind speed fluctuations	<wind.h>
SpeedDist	Distribution function of wind speed fluctuations	<wind.h>
StateOfChargeTimeSeries	State of charge time series	<series.h>
statfunc	Implementation of a statistical function	<distrib.h>
TDirDialog	Dialog window 'Directories'	<owdialg.h>
TDisplayDialog	Dialog window 'Display Options'	<owdialg.h>
TDistributionObject	Calculation of wind power and PV array distributions	<owcalc.h>
TDoubleInput	Input field for a real number	<owlappl.h>
TDoubleInputI	Input field for a real number	<owlappl.h>
TExportDialog	Dialog window 'Export'	<owdialg.h>
TFpDialog	Dialog window 'First Passage Time Problems'	<owdialg.h>
TGraph	General purpose graphic window	<owplot.h>
TimeSeries	Time Series	<series.h>
TimeSeriesOne	Time series with only one initial value	<series.h>

TIntegerInput	Input field for an integer number	<owlappl.h>
TIntegerInputI	Input field for an integer number	<owlappl.h>
TJointDialog	Dialog window 'Joint Renewable Distribution'	<owdialg.h>
TJointDistributionObject	Calculation of the joint renewable power distribution	<owcalc.h>
TMainWindow	Implementation of the main window	<owrenew.h>
TMathsDialog	Dialog window 'Mathematical Options'	<owdialg.h>
TMultiValObject	Calculation window for the computation of more than one value	<owstat.h>
TPassageTimeObject	Calculation of first passage time	<owcalc.h>
TPlot	Graphical representation of functions	<owplot.h>
TRenewApp	Main application	<owrenew.h>
TRenewPlot	Graphic window of project	<owrenw.h>
TSJointPassageTime	First passage time of joint renewable power fluctuations using the time series approach	<passage.h>
TSPassageTime	First passage time by time series approach	<passage.h>
TSSolarPowerPassageTime	First passage time of PV array power fluctuations using the time series approach	<passage.h>
TStatusWindow	Calculation window	<owstat.h>
TSWindSpeedPassageTime	First passage time for wind speed fluctuations using the time series approach	<passage.h>
TSWindPowerPassageTime	First passage time of wind turbine power fluctuations using the time series approach	<passage.h>
TRandDialog	Dialog window 'Random Numbers'	<owdialg.h>
TRandomObject	Random number generator calculations	<owcalc.h>
TSettingsDialog	Dialog window 'Settings'	<owdialg.h>
TSolarDialog	Dialog window 'Solar Power Distribution'	<owdialg.h>
TSpeedDialog	Dialog window 'Wind Speed Distribution'	<owdialg.h>
TTimeSeriesObject	Calculation of time series	<owcalc.h>
TTransDirDlg	Parameter transfer buffer for *TDirDialog*	<owdialg.h>
TTransDisplayDlg	Parameter transfer buffer for *TDisplayDialog*	<owdialg.h>
TTransExportDlg	Parameter transfer buffer for *TExportDialog*	<owdialg.h>

TTransFpDlg	Parameter transfer buffer for *TFpDialog*	<owdialg.h>
TTransJointDlg	Parameter transfer buffer for *TJointDialog*	<owdialg.h>
TTransMathsDlg	Parameter transfer buffer for *TMathsDialog*	<owdialg.h>
TTransRandDlg	Parameter transfer buffer for *TRandDialog*	<owdialg.h>
TTransSettingsDlg	Parameter transfer buffer for *TSettingsDialog*	<owdialg.h>
TTransSolarDlg	Parameter transfer buffer for *TSolarDialog*	<owdialg.h>
TTransSpeedDlg	Parameter transfer buffer for *TTransSpeedDlg*	<owdialg.h>
TTransTsDlg	Parameter transfer buffer for *TTsDialog*	<owdialg.h>
TTransWindDlg	Parameter transfer buffer for *TWindDialog*	<owdialg.h>
TTsDialog	Dialog window 'Time Series'	<owdialg.h>
TYoMessage	Message window	<owlappl.h>
TYoInput	Dialog window with one input field	<owlappl.h>
TWindDialog	Dialog window 'Wind Power Distribution'	<owdialg.h>
TWindSpeedObject	Calculation of the wind speed distribution	<owcalc.h>
UniKgSTest	Kolmogorov- Smirnov test for uniform distribution	<random.h>
uniRand	Generation of uniform deviates	<random.h>
uniRejectRand	Random number generator	<random.h>
VECTOR_	Vector with real elements	<vectors.h>
WindPowerPassageTimes	Object function for first passage times of wind turbine power fluctuations	<passage.h>
WindSpeedPassageTimes	Object function for first passage times of wind speed fluctuations	<passage.h>
WindSpeedTimeSeries	Wind speed time series	<series.h>
WindPowerTimeSeries	Wind power time series	<series.h>

CLASSES - REFERENCE

axis <owplot.h>

Implementation of a coordinate axis within the diagram in the class *TPlot.*

Constructors:

axis (HDC aDC, RECT* aCurRect); Initialising with window context *aDC* (see Object Windows C++ manual) and implied rectangular that represents the diagram.

Data elements:

curRect	RECT* curRect;	Rectangular that represents the diagram

Member functions:

setAxis

void setAxis (int dir, int just, int coord, double mini, double maxi, const char* alpha, double ax, int n, int axlog, int axgrid, double dist, int mode);

Determination of the attributes of an axis:

dir	Direction: HORIZ_DIR (horizontal), VERT_DIR (vertical)
just	Text justification: LEFT_TEXT (left justification), RIGHT_TEXT (right jusstification), BOTTOM_TEXT (text below axis), TOP_TEXT (text above axis).
coord	axis coordinate (relative to the rectangular)
mini	start value of the axis
maxi	end value
text	axis text
axle	Distance between to marks (only for linea axis)
num	For linear axis: Numbering only every num- th mark. For logarithmic axis: num = 1: Numbering of the 10-marks. num = 2: Numbering at 2 and 10; num = 3: at 2,5,10; num = 4: at 2,3,5,10.
axlog	LIN (linear), LOG (logarithmic)
axgrid	Draw a grid (YES or NO)
grid	Grid distance (for linear axis only)
mode	Presentation mode for the marks: IN_AXLE (axle points inwards), OUT_AXLE (axle points outwards), CENTER_AXLE (axle sit on the middle of the axis).

drawAxis	void drawAxis ( );	draw axis with specified attributes

BetaKgSTest <random.h>

Kolmogorov- Smirnov test for Beta- distribution, derived from *KgSTest.*

Constructors:

BetaKgSTest (int n, int r, double a, double b);
Construct test object for n classes, r trial points and distribution parameters a and b,

Member functions:

theoretProb	double theoretProb (double x);	see *KgSTest::theoretProb.*
initialize	void intiialize ( ); initialise *randomizer* with *betaRand* object.	

betaRand <random.h>

Implementation of a random number generator for beta- distributed numbers. It is derived from *uniRejectRand.*

Constructors:

betaRand (double alpha, double beta); Constructor with distribution parameters alpha and beta.

ContCondSolApprox <solar.h>

Conditional distribution of the PV array power, derived from *ContSolApprox*. Only difference to the base class is *setUp*, where *ContSolApprox::setCorrelation* is called automatically.

Constructors:

ContCondSolApprox ( ); Default constructor

Member functions:

setUp int setUp (TStatusWindow*, Param*);
see discussion above.

ContCondWindPower <wind.h>

Conditional distribution of the wind turbine power. This class is derived from *ContWindPower*. Only difference is that *ContWindPower::setCorrelation* is called within

ContCondWindPower::setUp so that *ContWindPower::F* always returns the conditional distribution function if called from *ContCondWindPower.*

Constructors:
ContCondWindPower ( ); call constructor of base class

Member functions:

setUp	int setUp (TStatusWindow*, Param*); see discussion above.

ContinuousDistribution <distrib.h>

Abstract class that represents a continuous distribution. Again, this is a conditional distribution, subjected to the initial value *initVal.*

Constructors:
ContinuousDistribution ( ); Default Constructor

Data elements:

initVal	protected: double initVal; implied initial value.

Member functions:

setUp	virtual int setUp (TStatusWindow*, Param*) = 0; Parameter setting function. Abstract function that must be overwritten in derived functions.
setInitVal	virtual void setInitVal (double x); set initial value *initVal.*
F	virtual double F (double x) = 0; Probability distribution function F(x). Abstract function that must be overwritten in derived functions.

ContSolAppQual <solar.h>

Quality of approximation, derived from class *ContinuousDistribution.*

Constructors:
ContSolAppQual ( ); Default Constructor

Member functions:

F	virtual double F (double x);	Equation (2.90)
setUp	int setUp (TStatusWindow*, Param*);	

ContSolApprox <solar.h>

Distribution of the PV array power (using the approximation), derived from class *ContinuousDistribution.*

Constructors:

ContSolApprox ( );	Default constructor

Data elements:
protected:

sol	MeritSol* sol;	pointer to the optimisation class
sc	SolConstants sc;	store of the distribution parameters

Member functions:

F	double F (double p);	Equation (4.11), though with normalised p instead of integer n.
setUp	int setUp (TStatusWindow*, Param*); Setting up the parameters. It is here that the optimisation is carried out by searching for the minimum of the merit function provided by *sol*. A golden search is carried out using *objfunc::goldenSection.* The calculations are implemented as described in 2.2.4.2.	
setCorrelation	void setCorrelation (double time, double beta); Unless *setCorrelation* is called the stationary distribution is being calculated.	
setInitVal	void setInitVal (double initK); Initialising the distribution with an average hourly clearness index k(0).	

ContSolApproxX <solar.h>

Conditional distribution of the normalised clearness index x, derived from *ContSolApprox.*

Constructors:

ContSolApproxX ( );	Default Constructor

Member functions:

F	double F (double x);	Equation (2.91)

ContSolExact <solar.h>

Analytical solution of the PV array power distribution, derived from class *ContinuousDistribution.*

Constructors:

ContSolExact ( );	Default constructor

Data elements:
protected:

solC	SolConstants colC;	Store of distribution parameters

Member functions:
protected:

Fx	double Fx (double x);	Equation (2.79)

public:

F	double F (double p);	Equation (4.9)
setUp	int setUp (TStatusWindow*, Param*);	

ContSolExactX <solar.h>

Analytical solution of the distribution of the normalised clearness index x, derived from *ContSolExact.*

Constructors:

ContSolExact ( );	Default Constructor

Member functions:

F	double F (double x);	Equation (2.79)

ContWindPower <wind.h>

Distribution of the wind turbine power. This class is derived from *ContinuousDistribution.*

Constructors:

ContWindPower ( );	Default constructor

Data elements:
protected:

r	double r;	autocorrelation function $r = \exp(-\beta t)$

Member functions:

F	double F (double p);	Equation (4.3)
setUp	int setUp (TStatusWindow*, Param*);	

	Parameter setting. Return OK if no error occurred.
setCorrelation	void setCorrelation (double time, double beta); Define autocorrelation function r = exp (- *beta* * *time*)

DiscretDistribution <distrib.h>

Abstract class of a discrete distribution. It actually is a conditional distribution with initial value (or call it conditional value) m.

Constructors:

DiscretDistribution (int n);	Initialisation for n classes.

Member functions:

setUp	virtual int setUp (TStatusWindow*, Param*) = 0; Initialisation with parameters. Abstract function has to be overwritten in derived classes. Returns OK if no error occurred. Otherwise ERROR.
gnm	virtual double gnm (int n, int m) = 0; returns the transition probability g_{nm} (probability for system to change from state m to state n in one step). Abstract class that has to be overwritten in derived classes.
Gn	virtual double Gn (int n); returns the probability that the system is in a state n or smaller provided the initial value is m. (m can be set by function setM) I.e. the distribution function. The default return value is 1. If another value is desired, Gn has to be overwritten.
setM	virtual void setM (int m); Set the initial value m
getN	virtual void getN (double p) = 0; returns the class if the probability distribution value p is given, provided m is the initial value. In a way this is the inverse function to Gn. It is an abstract function and has to be overwritten in derived classes.
getClasses	int getClasses ( ); returns the number of classes

discretRand <random.h>

discretRand is immediately derived from *uniRand*. This class is designed for the case where the probability distribution is of a discrete type and the probabilities p_j (j=1...N) for the N possible events j are given in a vector px.

Constructor:

discretRand (VECTOR* x); Initialization with vector x as described above.

Member functions:

update	void update (void* xx); Change distribution parameters (i.e. the probability vector px) even after initialisation. It is: px = (VECTOR*) xx;

DiscretRandomizer <distrib.h>

Abstract class of a random number generator for discrete distributions, derived from class uniRand.

Constructors:

DiscretRandomizer ( ); Default Constructor

Data elements:

distribution	protected: DiscretDistribution* distribution; Derived classes do have to install the desired distribution here. This is the distribution that governs the random number generator.

Member functions:

setUp	virtual int setUp (TStatusWindow*, Param*) = 0; Setting up parameters. Return OK if ok, otherwise ERROR.
setM	void setM (int m); set initial value m in distribution. See class *DiscretDistribution.*
getRandomNumber	double getRandomNumber ( ); generates and returns next random number.

DiscretWindSpeed <wind.h>

Discrete distribution of wind speed fluctuations as used in first passage time problems using the Markov chain approach. The class is derived from *DiscretDistribution.*

Constructors:

DiscretWindSpeed (int n); calls constructor of base class

Member functions:

gnm	double gnm (int n, int m);

	transition probability. See *DiscretDistribution::gnm*	
getN	int getN (double v);	see *DiscretDistribution::getN*
setUp	int setUp (TStatusWindow*, Param*);	
	Parameter setting. Return OK if no error occurred.	

DiscSolApprox <solar.h>

Implementation of adiscrete distribution that represents the PV array power. It is a class derived from *DiscretDistribution.*

Constructors:

DiscSolApprox (int n); Construct the class with n discretisation levels.

Member functions:

setUp	int setUp (TStatusWindow*, Param*);	
gnm	double gnm (int n, int m);	see *DiscretDistribution:gnm*
Gn	double GN (int n);	see *DiscretDistribution::Gn*
setM	void setM (int m);	overwrites *DiscretDistribution::setM*
getN	int getN (double x);	see *DiscretDistribution::getN*

DiscretWindPower <wind.h>

Discrete distribution of wind turbine power fluctuations as used in first passage time problems using the Markov chain approach. The class is derived from *DiscretDistribution.*

Constructors:

DiscretWindPower (int n); calls constructor of base class

Member functions:

gnm	double gnm (int n, int m);	
	transition probability. See *DiscretDistribution::gnm*	
Gn	double Gn (int m);	see *DiscretDistribution::Gn*
getN	int getN (double v);	see *DiscretDistribution::getN*
setUp	int setUp (TStatusWindow*, Param*);	
	Parameter setting. Return OK if no error occurred.	

Graph <owparam.h>

Interface between graphic window and calculations. Calculation objects store values here. They can be picked up by the graphic window, which is an instance of class *TRenewPlot.* It can store the function values of up to four curves.

Constructors:

Graph ( );	Default constructor for 4 curves

Data elements:

x	VECTOR x;	x - values
y	VECTOR y[4];	y - values (up to 4 curves)
legend	char legend [4][20];	Legend text for the export to Word Perfect Presentation
scale	double scale;	Scaling factor for display purposes.
curveNo	int curveNo;	Number of sets of curve data currently stored. curveNo < 4.
min	double min;	Minimum value on x - axis
max	double max;	Maximum value on x - axis
headline	char headline[40];	Headline of graph
subline	char subline[50];	Text below headline
axtext	char axtext[40];	Text below x- axis

Member functions:

setHeadline	void setHeadline (char* text);	define headline
setSubline	void setSubline (char* text);	define line below headline
setAxtext	void setAxtext (char* text);	define text belowe x- axis

JointPassageTimes <passage.h>

Object function for first passage times of joint renewable power fluctuations, derived from *PassageTimes.*

Constructors:

WindSpeedPassageTimes (int select);

Constructor: If *select* = 0 the data element *passageTime* is initialised with an instance of *TSJointPowerPassageTime.* Otherwise with *MCJointPowerPassageTime.*

Member functions:

SetUp — int SetUp (TStatusWindow*, Param*);
individual set-up of initial values and passage levels.

JointPowerTimeSeries <series.h>

Implementation of joint renewable power time series, derived from *TimeSeries.*

Constructors:

JointPowerTimeSeries ( ); — Default constructor. Initialises a *SolarPowerTimeSeries* and a *WindPowerTimeSeries* object for the two underlying processes.

Member functions:
protected:

getRandomNumber	double getRandomNumber ( ); returns next random number from the implied random number generator.	

public:

update	void update ( );	see *TimeSeries::update*
getOutput	double getOutput ( );	see *TimeSeries::getOutput*
setUserInit	void setUserInit (void*);	see *TimeSeries::setUserInit*
getInitRandomVal	double getInitRandomVal ( ); overwrites *TimeSeriesOne::getInitRandomVal.*	
setUp	int setUp (TStatusWindow*, Param*); Parameter setting	
eval	double eval (double); return next time series value. the argument is not used.	

KgSTest <random.h>

Abstract class of a Kolmogorov- Smirnov test.

Constructors:

KgSTest (int n);	Construct a test with n trial points.

Data elements:
protected:

size	double size;	number of trials. This is of type 'double' for data conversion reasons.
k	int k;	number of classes.
mean	double mean;	mean value of sample
var	double var;	variance ov sample
x,y,r	VECTOR x,y,r;	Vectors holding the results.(r holding the generated numbers. x and y holding the theoretical distribution.)
randomizer	uniRand* randomizer;	random number generator to be used in the test.

Member functions:
protected:

initialize	virtual void initialize ( ); Per default this function does nothing. In derived classes, however, this is the place to initialise the random number generator *randomizer.*
theoretProb	virtual double theoretProb (double x) = 0; This function has to be overwritten by derived classes. It has to return the theoretical probability for values smaller than or equals

	x.
maxDistance	double maxDistance ( ); This function calculates the maximum distance between a generated point and the theoretical distribution function.
doValues	void doValues ( ); generate the random numbers and pack them into vector r.
calcCumDist	void calcCumDist ( ); internal function for the Kolmogorov- Smirnov test.
public:	
doTest	double doTest ( ); Carries out the Kolmogorov- Smirnov test and returns the test result. (See [33]).
getMean	double getMean ( ); Return the mean value of the sample
getVar	double getVar ( ); Return the variance of the sample

MATRIX_ <vectors.h>

typedef MATRIX_<int>	IMATRIX;
typedef MATRIX_<double>	MATRIX;

Constructors:

MATRIX_ (int n);	initialises an n x n - matrix.
MATRIX_ (MATRIX_ A);	initialises a copie of matrix A.
MATRIX_ (int m, int n);	initialises an m x n - matrix.

Data members:

col	int col;	Number of columns
row	int row;	Number of rows

Member functions:

col_to_vec	void col_to_vec (int i, VECTOR_<T>& v); move values of the i-th column to vector v.
create	void create (int m, int n); Allocation of memory on the heap for an m x n- matrix.
diag_to_vec	void diag_to_vec (VECTOR& v); move diagonal elements to vector v.
maxval	T maxval (int& i, int& j); returns the maximum value of the matrix. Indices see minval().
minval	T minval (int& i, int& j); returns the minimum value of the matrix. Its indices are updated and passed by reference.

vec_to_col	void vec_to_col (int i, VECTOR_<T>& v); moves i-th column vector to vector v.
print	void print (ostream& op); Standard output to screen.
build	void build (istream& ip); Standard input via istream.

Operators:

( )	A(int i) A(int i, int j)	Access to element A_{ii} Access to element A_{ij}.
+, +=, -, -=	Matrix addition:	**A + B , A - B (A, B** Matrices)
*	Multiply with number: Matrix multiplication: Multiply by vector:	$\mathbf{B} = \mathbf{A} * \alpha$, $\mathbf{B} = \alpha * \mathbf{A}$, $\mathbf{A} *= \alpha$ $\mathbf{C} = \mathbf{A} * \mathbf{B}$ $\mathbf{v} = \mathbf{A} * \mathbf{u}$, $\mathbf{v} = \mathbf{u}^T * \mathbf{A}$
/	Division by number α:	$\mathbf{A} = \mathbf{B} / \alpha$; $\mathbf{A} /= \alpha$;
=	**A = B;**	
<< >>	operator (ostream& op, MATRIX& A); operator (istream& ip, MATRIX& A);	

MCPassageTime <passage.h>

Abstract class that calculates the first passage time using the Markov chain approach. It is derived from *PassageTime.*

Constructors:

MCPassageTime ( );	Default constructor

Data elements:
protected:

classes	int classes; Number of discretisation levels
distribution	DiscretDistribution* distribution; Underlying discrete distribution that is used int the calculations.

Member functions:
protected:

discretize	int discretize (double x); Given an initial level x (depending on the selection this could be a wind speed, clearness index or normalised power value) this function returns the class number the argument is in. It calls *distribution->getN (x).*
public:	
Eval	double Eval (double x); returns the first passage time (with non discretised passage level x) using the Markov chain approach.
SetUp	virtual int SetUp (TStatusWindow*, Param*); Parameter setting for Markov chain approach
setInitLevel	void setInitLevel (void*); Assumes the argument to be double* and copies it into *PassageTime::initLevel.*

MCWindSpeedPassageTime <passage.h>

Object that calculates the first passage time of wind speed fluctuations using the Markov chain approach. It is derived from *FPPassageTime.*

Constructors:

MCWindSpeedPassageTime ( ); Default constructor

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting the parameters and initialising *distribution* with an instance of *DiscretWindSpeed.*

MCWindPowerPassageTime <passage.h>

Object that calculates the first passage time of wind turbine pwoer fluctuations using the Markov chain approach. It is derived from *FPPassageTime.*

Constructors:

MCWindPowerPassageTime ( ); Default constructor

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting the parameters and initialising *distribution* with an instance of *DiscretWindPower.*

MCSolarPowerPassageTime <passage.h>

Object that calculates the first passage time of PV arra power fluctuations using the Markov chain approach. It is derived from *FPPassageTime.*

Constructors:

MCSolarPowerPassageTime ( );	Default constructor

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting the parameters and initialising *distribution* with an instance of *DiscSolApprox.*

MCJointPowerPassageTime <passage.h>

Object that calculates the first passage time of joint renewable pwoer fluctuations using the Markov chain approach. It is derived from *FPPassageTime.*

Constructors:

MCJointPowerPassageTime ( );	Default constructor

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting the parameters.

MeritSol <solar.h>

Object to optimise the approximation used for the distribution of the PV array power. It is derived form *msgObjfunc.*

Constructors:

MeritSol (SolConstants*, Param*);

Data elements:

psc	SolConstants* psc;	pointer to the distribution parameter store
initialx	double initialx;	initial normalised clearness index x_0.
u	VECTOR u;	Coefficient vector. See equation (2.82).
sigma	VECTOR sigma;	See equation (2.84).
lambda	VECTOR lambda;	This is sigma / epsilon (see (2.84))
Fxm	VECTOR Fxm;	Vector with distribution function values. Right hand side of (2.87).
QPlusOne	double QPlusOne;	Number of generating functions used + 1 (see equation 2.87)
MPlusOne	double MPlusOne;	Number of trial points + 1

Member functions:

Eval	double Eval (double x); Calculates the merit function, equation (2.85).	
fx	double fx (double x);	Equation (2.75)
Fx	double Fx (double x);	Equation (2.79)
Fp	double Fp (double p);	Distribution function in power values p. Compare equation (4.9)
FxApprox	double FxApprox (double x);	Equation (2.82)
FpApprox	double FpApprox (double p); as *F Approx* but with power value p as argument. It is internally converted into a normalised clearness index x before calling *FxApprox.*	
setUp	int setUp ( );	Parameter initialisation

Operators:

The stream operators are used to save optimisation data to a file and retrieve it next time in order to save computing time.

friend ostream& operator << (ostream& outstr, MeritSol* v);

friend istream& operator >> (istream& instr, MeritSol* v);

msgObjfunc <distrib.h>

Abstract class, derived from *objfunc*. It is an extension in that it can monitor the elapsed calculation time and then present messages.

Constructors:

msgObjfunc ( ); Default constructor

Member functions:

enableTimeMsg	void enableTimeMsg ( ); permit time messages being sent to the message queue, specified by the handle set in *setHandle*.
enableValueMsg	void enableValueMsg ( ); permit messages of the value of the calculation sent to the message queue.
setHandle	void setHandle ( ); set Windows handle. I.e. Handle of appropriate dialog window.
eval	double eval (double); Function from base class *objfunc*, here overwritten.
Eval	double Eval (double) = 0; Evaluation of object function. This abstract function has to be

overwritten in derived classes.

NormKgSTest <random.h>

Kolmogorov- Smirnov test for normal distribution, derived from *KgSTest.*

Constructors:

NormKgSTest (n);	Construct test object for n trial points.

Member functions:

theoretProb	double theoretProb (double x);	see *KgSTest::theoretProb.*
initialize	void intiialize ( ); initialise *randomizer* with *normRand* object.	

normRand <random.h>

normRand is derived from *uniRand.* It implements a random number generator, producing a series of numbers that are normal distributed with mean *mean* and standard deviation *sigma.* It implements the Box- Muller method (C.Press: Numerical recipes, 1992, p.289) drawing the uniform deviates from *uniRand.*

Constructors:

normRand ( ); Initialization for standard normal deviates (i.e zero mean and unit standard variation.)

normRand (double mean, double sigma); Initialization with *mean* and *sigma.*

Member functions:

getRandomNumber	virtual double getRandomNumber ( ); returns the next random number. It overwrites the *getRandomNumber* function of *uniRand.*
update	void update (void* x); the first double value in x is interpreted as the mean value, the second as the variance. This gives the opportunity to change the parameters even after initialisation.

objfunc <diffcalc.h>

Abstract class, which provides operations on functions of one variable.

Data members:

x, y	VECTOR x, y;	x- and y- values (y- values are the function values)

Member functions:

eval

virtual double eval (double x) = 0;

Evaluation of the object function at x. This function has to be provided by derived classes as this is an abstract function.

bracketRoot

BOOL bracketRoot (double x0, double step, double &a, double &b, int maxit, int mode);

Starting in x0 with a step width a, the algorithm searches for a bracket {a,b} in which a root of the object function is contained.
For mode:
mode = DETECT_EQUI: The algorithm determines the search direction. The step width does not change.

mode = DETECT_DYNA: The step width will be increased dynamically from step to step.

mode = DOWN_EQUI: Algorithm searches only towards smaller values than x0. Equidistant step width.

mode = DOWN_DYNA: Dynamic step width
mode = UP_EQUI: Search towards greater values than x0.
mode = UP_DYNA: Dynamic step width.
The function returns ERROR if maximum number of function evaluations, maxit, is reached. Otherwise OK.

goldenSection

double goldenSection (double ax, double bx, double cx, double fb, double tol, double& xmin);
For the bracket of the minimum { ax, bx, cx } the function determines the minimum, xmin, and returns the value at xmin. The tolerance is tol. fb is the function value at bx. The algorithm uses the golden section search.

compEquiVal

void compEquiVal (double xmin, double xmax, int n);
Function computes n equidistant function values in the open interval [xmin, xmax]. The results are stored in x and y respectively.

owObjfunc <diffcalc.h>

This class is derived from objfunc and extended by an info facility. This is useful if the underlying object function is evaluated N times and N is known before.

Member functions:

getPercentage	double getPercentage ( ); returns the percentage of the number of evaluations carried out in relation to the total number N.
prepForEquiVal	void prepForEquiVal (double xmin, double xmax, int N); Preparation of the series of N evaluations on the interval [xmin, xmax].
compEquiVal	void compEquiVal ( ); Evaluation of the object function. Subsequent calls cause the function to be evaluated at different x- values - as stated in prepForEquiVal ( ). The y - values are stored in vector y in objfunc.

pairvec <diffcalc.h>

Constructors:

pairvec (int n);	initialises the class with n (x,y) - pairs
pairvec ( );	initialises the class with size = 0.

Data members:

size	int size;	Dimension of x and y
x, y	VECTOR x, y;	x- und y- values as vectors

Member functions:

create	void create (int n); Allocation of memory on the heap	
move	void move (int i, int j);	moves i-th element to j-th place
move_down	void move_down ( );	moves all components one place down
swap	void swap (int i, int j);	Swap i-th and j-th elements.

Operators:

<<	operator << (ostream& op, pairvec& v);
>>	operator >> (istream& ip, parivec& v);

Param <owparam.h>

Structure that holds parameters for all dialog windows. It serves as an interface between dialog windows and calculation objects as both access it.

```
struct Param {
  double tau;           // time
  int    eval;          // number of function evaluations
  int    type;          // = 0 (distribution) , = 1 (density)
  int    distSelect;    // chosen distribution selection:
                        // = 0 : Wind turbine power
                        //   1 : Conditional wind turbine power
                        //   2 : Exact Solar
                        //   3 : Approximated solar
                        //   4 : Approximated solar, conditional
                        //   5 : Quality of approximation
  int    filter;        // filter of inspection windows
    int    classes;      // number of discretisation levels in a discrete
                        // distribution

  // Wind parameters:
  double wiVci;         // cut- in speed
  double wiVco;         // cout- out speed
  double wiVr;          // rated wind speed
  double wiVmean;       // mean wind speed
  double wiVmin;        // minimum wind speed for wind speed distribution
  double wiVmax;        // maximum wind speed for wind speed distribution
  double wiSigma;       // variance of wind speed fluctuations
  double wiBeta;        // wind autocorrelation coefficient
  double wiInitV;       // initial wind speed

  // Solar parameters:
  double solK;          // average hourly clearness index k
  double solSigmaK;     // standard deviation of solar irradiation
  double solK0;         // absolute maximum possible clearness index
  double solInitK;      // initial average hourly clearness index
  double solBeta;       // solar autocorrelation coefficient bsol
  int    solTrial;      // number of trial points in normal approximation
  int    solCoeff;      // number of coefficients in normal approximation
  int    solBypass;     // bypass of major calculations by retrieving
                        // old data

  // Combined renewables parameters:
  double comZeta;       // fractional power factor zeta
  double comInitP;      // Initial p value (normalised power)

  // Random numbers dialog:
  double ranA;        // Parameter alpha for beta- distribution
  double ranB;        // Parameter beta for beta- distribution
  double ranP;        // Parameter p for binomial distribution
  double ranU;        // Parameter u for normal distribution (not used!!)
  int    ranClass;    // Number of classes for Kolmogorov- Smirnov  test
  int    ranTrial;    // Number of trials in Kolmogorov- Smirnoc test
  int    ranSelect;   // Last selection (i.e. distribution type)

  // Time series parameters:
  double tsTimeStep;  // Duration of a single time step
  int    tsPoints;    // Length of a time series
  int    tsSelect;    // Last selection (type of time series)

  // First passage time parameters:
  int    fpTsTrial;   // Number of time series taken into account
  int    fpTsMaxIt;   // Max iterations in Time series mode
  double fpMcStopCrit;// Stopping criterion in Markov chain mode
  int    fpMcMaxIt;   // Max iterations in Markov chain mode
  int    fpMcGrid;    // Markov chain mode: Grid Number Q
  double fpPassV;     // Passage level: Wind speed v
  double fpPassK;     //                Clearness index k
  double fpPassP;     //                Power level p
  int    fpNoVal;     // Number of values to be calculated in
```

```
                          // function-as-mode
  int    fpSelectProcess;    // Flags
  int    fpSelectMethod;     // Markov chain - or time series approach
  int    fpSelectCalc;       // Calculation technique selected.
  // Battery parameters
  double batK;            // Battery parameter k
  double batC;            // Battery parameter c
  double batQMax;         // Battery capacity
  double batV;            // Voltage
  double batQ10;          // Initial available charge Q10
  double batQ20;          // Initial bound charge, Q10 + Q20 <= 1.0
  // Denormalized system
  double sysPDemand;      // Power demand
  double sysPRen;         // Installed maximum renewable power
  // Display options
  int    disAuto;         // automatic re-drawing of graphics
  int    disAccu;         // accumulate data series when possible
  int    disOldEval;      // last eval
  int    disOldType;      // last window type
  double disOldVmin;      // last minimum speed
  double disOldVmax;      // last maximum speed
  int    disFirstCurve;   // = 1 if first curve, otherwise 0
  int    disLegend;       // = 1 if legend desired, otherwise 0
};
```

PassageTime <passage.h>

This is an abstract class that represents a first passage time calculator. It is derived from *msgObjfunc*. For a given passage level and initial value the first passage time is calculated in the function *Eval*, which has to be provided in derived classes.

Constructors:

PassageTime ( );	Default Constructor

Data elements:
protected:

passLevel	double passLevel;	passage level (speed, clearness index or power)
initLevel	double initLevel;	initial value (speed, clearness index or power, depending on selection)
timeStep	double timeStep;	time step (for time series approach only)

Member functions:
protected:

SetUp	virtual int SetUp (TStatusWindow*, Param*) = 0; Derived classes have to provide their own SetUp functions.

public:

setUp	int setUp (TStatusWindow*, Param*); Setup function that calls *SetUp*.
setPassLevel	void setPassLevel (double newLevel); Sets the passage level to *newLevel*.
setInitLevel	virtual void setInitLevel (void* initSet) = 0;

Sets initial level. As there could be not only one but two values that define the initial state (wind speed and clearness index in the case of joint renewable power) the new initial state, *initSet* is a void*. It has to be defined in derived classes.

PassageTimes <passage.h>

Abstract class that is able to calculate more than one first passage time value in one set. Hence, it is derived from *owObjfunc* and has a *PassageTime** object as data element.

Constructors:

PassageTimes ( );	Default constructor	

Data elements:

protected:

selectCalc	int selectCalc;	see *setUp.*
noVal	int noVal;	see *setUp.*
passageTime	PassageTime* passageTime	Implied passage time object

public:

minVal	double minVal;	minimum value / start value (either initial value or passage level depending on the selection)
maxVal	double maxVal;	maximum value / end value (either initial value or passage leve depending on the selection)

Member functions:

protected:

SetUp

virtual int SetUp (TStatusWindow*, Param*) = 0;
has to be overwritten by derived classes

public:

setUp

int setUp (TStatusWindow*, Param* param);
Parameter setup. *selectCalc* is initialised with *param->fpSelectCalc* (see *Param::fpSelectCalc*) and noVal with *param->fpNoVal.*

eval

double eval (double);
returns the first passage time as a function of either the initial value or the passage level depending on the selection, *selectCalc.*

PassageTimesObject <owcalc.h>

Calculation window on which calculations of first passage times are carried out, derived from *TMultiValObject.* This class is to be used if the first passage time is to be calculated as a function of the initial value or the passage level and more than one value has to be determined. All necessary functions are privately overwritten. See *TMultiValObject.*

Constructors:
PassageTimesObject (PTWindowsObject AParent, LPSTR ATitle);

PowerDeficitTimeSeries <series.h>

Implementation of time series of the power deficit that may occur if the joint renewable power and the power delivered by the battery is not sufficient to meet the power demand. The class is immediately derived from *StateOfChargeTimeSeries*. The power difference can be picked up in the field *StateOfChargeTimeSeries::deltaP*.

Constructors:
PowerDeficitTimeSeries ( ); Default constructor calls base class constructor

Member functions:
eval double eval (double);
returns next time series value. Argument is not used.

ProbCondSolApprox <solar.h>

Conditional distribution of the PV array power (using the approximation) embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:
ProbCondSolApprox ( ); Constructor initialises *statfunc::distribution* with a *ContCondSolApprox* object.

ProbCondWindPower <wind.h>

Conditional distribution - representing the wind turbine power - embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:
ProbCondWindPower ( ); Constructor initialises *statfunc::distribution* with a *ContCondWindPower* object.

ProbJointPower <joint.h>

Implementation of the probability function of the joint renewable power, derived from *owObjfunc.*

Constructors:

ProbJointPower (int n); Construction for n different power levels.

Member functions:

eval double eval (double p);
return probability for normalised power level p

setUp int setUp (TStatusWindow*, Param*);
Setting up the parameters.

ProbSolAppQual <solar.h>

Quality of approximation embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:

ProbSolAppQual ( ); Constructor initialises *statfunc::distribution* with both a *ContSolApprox* and a *ContSolExact* object.

ProbSolApprox <solar.h>

Distribution of the PV array power (using the approximation) embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:

ProbSolApprox ( ); Constructor initialises *statfunc::distribution* with a *ContSolApprox* object.

ProbSolExact <solar.h>

Analytical solution of the distribution of the PV array power embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:

ProbSolExact ( );	Constructor initialises *statfunc::distribution* with a *ContSolExact* object.

ProbWindPower <wind.h>

Stationary distribution - representing the wind turbine power - embedded in a *statfunc* object. This is necessary to ensure that it can be easily used by dialog window classes. Moreover, the function *statfunc::eval* can calculate both the distribution function and the probability function.

Constructors:

ProbWindPower ( );	Constructor initialises *statfunc::distribution* with a *ContWindPower* object.

rejectRand <random.h>

rejectRand is immediately derived from *uniRand.* It is a virtual base class for a random number generator applying the 'rejection method' (W. Press: Numerical recipes, 1992, p.290). Derived classes have to specify the comparison function, the original density function and the inverse distribution function.

Constructor:

rejectRand ( );	Default constructor

Member functions:

compFunc	virtual double compfunc (double) = 0; Comparison function. Has to be defined in derived classes.
origFunc	virtual double origFunc (double) = 0; Original underlying probability density function. Has to be defined in derived classes. It is assumed that it takes only arguments in the interval [0,1].
invInteg	virtual double invInteg (double) = 0; Inverse function of the normalized integral of the comparison function, returning only numbers in the interval [0,1]. Has to be defined in derived classes.
getRandomNumber	virtual double getRandomNumber ( ); returns the next random number.

SolarPowerPassageTimes <passage.h>

Object function for first passage times of PV array power fluctuations, derived from *PassageTimes.*

Constructors:

WindSpeedPassageTimes (int select);

Constructor: If *select* = 0 the data element *passageTime* is initialised with an instance of *TSSolarPowerPassageTime.* Otherwise with *MCSolarPowerPassageTime.*

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); individual set-up of initial values and passage levels.

SolarPowerTimeSeries <series.h>

Implementation of PV array power time series, derived from *TimeSeriesOne.*

Constructors:

SolarPowerTimeSeries ( );	Default constructor. Initialises a *SolarRandomizer* object as internal random number generator.

Member functions:

protected:

getRandomNumber	double getRandomNumber ( ); returns next random number from the implied random number generator.	

public:

getOutput	double getOutput ( );	see *TimeSeries::getOutput*
update	void update ( );	see *TimeSeries::update*
getInitRandomVal	double getInitRandomVal ( ); overwrites *TimeSeriesOne::getInitRandomVal.*	
setUp	int setUp (TStatusWindow*, Param*); Parameter setting	

SolarRandomizer <solar.h>

Random number generator for the distribution of the PV array power, derived from *DiscretRandomizer.*

Constructors:

SolarRandomizer ( );	Default Constructor

Member functions:

setUp	int setUp (TStatusWindow*, Param*);

SolConstants <solar.h>

Store for clearness index distribution parameters. Compare section 2.2.4.1

Constructors:

SolConstants ( );	Default constructor

Data elements:

w	double w;	equation (2.77)
deltaKK0	double deltaKK0;	$(k_{max} - k_{min}) / K_0$ (see section 2.2.4.1)
kminK0	double kminK0;	k_{min} / K_0 (see section 2.2.4.1)
deltaK	double deltaK;	$(k_{max} - k_{min})$
kmin	double kmin;	k_{min}
correl	double correl;	correlation coefficient β_x.
a,b	VECTOR a,b;	equation (2.76)

Member functions:

setUp	int setUp (Param*); The function takes the relevant parameters off the *Param* structure and calculates the values of the data elements above.
xTok	void xTok (double x, double* k); Inverse functionality to equation (2.70).
kTox	void kTox (double k, double* x); See equation (2.70).

Speed <wind.h>

Abstract class that represents the distribution of wind speed fluctuations. The class is derived from *owObjfunc*.

Constructors:

Speed ( );	Default constructor

Data elements:
protected:

vmean	double vmean;	mean wind speed
vsigma	double vsigma;	wind speed standard variation

Member functions:

eval	double eval (double v) = 0;	see *objfunc::eval*
setUp	int setUp (Param*);	Parameter setting

SpeedDens <wind.h>

Probability density function of wind speed fluctuations. It is derived from *Speed.*

Constructors:

SpeedDens ( );	Default constructor

Member functions:

eval	double eval (double v); Equation (4.2), but stationary only

SpeedDist <wind.h>

Distribution function of wind speed fluctuations. It is derived from *Speed.*

Constructors:

SpeedDist ( );	Default constructor

Member functions:

eval	double eval (double v); Equation (4.1), but stationary only

StateOfChargeTimeSeries <series.h>

Implementation of time series of the state of charge of the battery, derived from *TimeSeries.*

Constructors:

StateOfChargeTimeSeries ( );	Default constructor. Initialises a *JointPowerTimeSeries* object for the underlying process.

Data elements:
protected:

deltaP	double deltaP;	difference between delivered and demanded power.

Member functions:

protected:		
update	void update ( );	see *TimeSeries::update*
getOutput	double getOutput ( );	see *TimeSeries::getOutput*
public:		
setUserInit	void setUserInit (void*);	see *TimeSeries::setUserInit*
setUp	int setUp (TStatusWindow*, Param*); Parameter setting	
eval	double eval (double); return next time series value. the argument is not used.	

statfunc <distrib.h>

Abstract class of a statistical function, derived from *owObjfunc*. It can be either a distribution or a probability density function.

Constructors:

statfunc ( );	Default Constructor

Data elements:

type	protected: int type; *type* is either 1 (distribution function) or 0 (probability density function).
distribution	protected: ContinuousDistribution* distribution; Pointer to the implied distribution. Has to be set up in derived classes.

Member functions:

eval	double eval (double); returns either the distribution or the probability density.
setUp	virtual int setUp (TStatusWindow*, Param*); Parameter setting
setType	void setType (int aType); specify function type. See data elemetn *type* for more details.

TDirDialog <owdialg.h>

Implementation of the 'Directories' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TDirDialog (PTWindowsObject AParent, LPSTR ATitle);

TDisplayDialog <owdialg.h>

Implementation of the 'Display Options' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TDisplayDialog (PTWindowsObject AParent, LPSTR ATitle);

TDistributionObject <owcalc.h>

Calculation window on which calculations of both wind power and PV array power distributions are carried out, derived from *TMultiValObject*. All necessary functions are privately overwritten. See *TMultiValObject*.

Constructors:
TDistributionObject (PTWindowsObject AParent, LPSTR ATitle);

TDoubleInput <owlappl.h>

Implementation of an input field in a dialog window that expects a real number. If the input is not valid a message window pops up and the dialog window cannot be closed. *TDoubleInput* is derived from the Object Windows C++ class *TEdit*.

Constructors:
TDoubleInput (PTWindowsObject AParent, int ResourceId);

Data elements:

x	double x;	Input value as a number and not text.

Member functions:

Transfer	virtual WORD Transfer (void* DataPtr, WORD TransferFlag); Transfer and conversion from data element x to the string in the input field.
CanClose	virtual BOOL CanClose ( ); tries to convert string from input field to a double. If successful it returns OK. Otherwise ERROR.

TDoubleInputI <owlappl.h>

This class is derived from *TDoubleInput*. In addition it checks whether the value x lies in an interval [minVal, maxVal]. If not a message *aMessage* pops up.

Constructors:
TDoubleInputI (PTWindowsObject AParent, int ResourceId, const double aMinVal, const double aMaxVal, const char* aMessage);

Member functions:

CanClose	virtual BOOL CanClose ( ); see *TDoubleInput*

TExportDialog <owdialg.h>

Implementation of the 'Export' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TExportDialog (PTWindowsObject AParent, LPSTR ATitle);

TFpDialog <owdialg.h>

Implementation of the 'First Passage Time Problems' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TFpDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:

virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
: Function is carried out upon initialisation of the window.

virtual void HandleOp0Msg (RTMessage) = [WM_FIRST + idFpOp0];
: Function is called upon selection of 'Wind Speed' option in the dialog window. If this option is selected input fields are made visible or invisible as appropriate. The id- constant is defined in *owres.h*.

virtual void HandleOp1Msg (RTMessage) = [WM_FIRST + idFpOp1];
: Function is called upon selection of 'Wind Power' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp2Msg (RTMessage) = [WM_FIRST + idFpOp2];
: Function is called upon selection of 'Solar Power' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp3Msg (RTMessage) = [WM_FIRST + idFpOp3];
: Function is called upon selection of 'Combined Renewable' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp4Msg (RTMessage) = [WM_FIRST + idFpOp4];
: Function is called upon selection of Time Series Approach' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp5Msg (RTMessage) = [WM_FIRST + idFpOp5];
: Function is called upon selection of 'Markov Chain Approach' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp6Msg (RTMessage) = [WM_FIRST + idFpOp6];
: Function is called upon selection of 'Calculate one value only' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp7Msg (RTMessage) = [WM_FIRST + idFpOp7];
: Function is called upon selection of 'as function of initial value' option in the dialog window. See *HandleOp0Msg* above.

virtual void HandleOp8Msg (RTMessage) = [WM_FIRST + idFpOp8];
Function is called upon selection of 'as function of passage level' option in the dialog window. See *HandleOp0Msg* above.

TGraph <owplot.h>

General purpose graphic window, derived from the Object Windows C++ class *TWindow*. It provides graphic resources such as a font, a pen and a brush. It offers functions to draw lines, write text or numbers.

Constructors:
TGraph (PWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);

Data elements: (protected)

logFont	LOGFONT logFont;	Font: Attributes
TheFont	HFONT TheFont;	Font: Resource (handle)
oldFont	HFONT oldFont;	Font: old resource (in order to go back to old font)
logPen	LOGPEN logPen;	Pen: Attributes
ThePen	HPEN ThePen;	Pen: Resource handle
oldPen	HPEN oldPen;	Pen: old resource handle
logBrush	LOGBRUSH logBrush;	Brush: Attributes
TheBrush	HBRUSH TheBrush;	Brush: Resource handle
oldBrush	HBRUSH oldBrush;	Brush: old resource handle
backGround	COLORREF backGround;	Background color
DC	HDC DC;	Screen context. See [3] and [4] for further details.

Member functions:

clearScreen	void clearScreen ( );	clear the screen
setTextHeight	void setTextHeight (int n);	set text height
setPenSize	void setPenSize (int n);	set pen width
setPenStyle	void setPenStyle (int n);	set style of pen
setPenColor	void setPenColor (COLORREF c);	set color of pen
setBrushStyle	void setBrushStyle (int n);	set style of brush
setBrushColor	void setBrushColor (COLORREF c);	set brush color
setBrushHatch	void setBrushHatch (int n);	set pattern of brush
setColor	void setColor (COLORREF c);	set color of current resource
open	virtual void open ( );	open and initialise window
close	virtual void close ( );	close window and delete all resources
Line	void Line (int x1, int y1, int x2, int y2);	

draw line from (x1,y1) to (x2,y2)

DoubleOut — void DoubleOut (double number, int dec, int x, int y);
print out *number* starting at coordinate (x,y) with *dec* decimal points.

IntegerOut — void IntegerOut (int number, int x, int y);
print out *number* starting at coordinate (x,y).

TextOut — void TextOut (char* text, int x, int y);
print out text string *text*, starting at point (x,y).

TimeSeries <series.h>

Abstract class of a time series object, derived from *owObjfunc*.

Constructors:

TimeSeries ( ); — Default Constructor

Member functions:

protected:

update — virtual void update ( ) = 0;
Has to be defined in derived classes. It takes the output of the time series generator and channels it back to the initial values. This is the function $\Xi(\xi)$ in the time series algorithm point (5), section 4.2.1.

getOutput — virtual double getOutput ( ) = 0;
Has to be defined in derived classes. It returns the desired output variable. This is the function $\Psi(\xi)$ in the time series algorithm point (6) in section 4.2.1.

public:

setUp — virtual int setUp (TStatusWindow*, Param*) = 0;
Has to be defined in derived classes.

setUserInit — virtual void setUserInit (void* v) = 0;
Has to be defined in derived classes. It sets initial value(s) as specified in v. It could be an initial wind speed, initial clearness index or both.

TimeSeriesOne <series.h>

Time series object, derived from *TimeSeries*. Though, it allows only one initial value, either wind speed or clearness index, but not both.

Constructors:

TimeSeriesOne ( );	Default constructor	

Data elements:
protected:

initUserVal	double initUserVal;	initial value as specified by the user
randomVal	double randomVal;	current value of the underlying stochastic process
outVal	double outVal;	output value

Member functions:
protected:

getInitRandomVal	virtual double getInitRandomVal ( ); returns initial value of underlying stochastic process
getRandomNumber	virtual double getRandomNumber ( ) = 0; Has to be defined in derived classes. It has to return the next random number.

public:

eval	double eval (double); returns the next time series value. The argument is not used, though necessary as this object is derived from *owObjfunc*.
setUserInit	void setUserInit (void*); see *TimeSeries::setUserInit*.

TIntegerInput <owlappl.h>

Implementation of an input field in a dialog window that expects an integer number. If the input is not valid a message window pops up and the dialog window cannot be closed. *TIntegerInput* is derived from the Object Windows C++ class *TEdit*.

Constructors:
TIntegerInput (PTWindowsObject AParent, int ResourceId);

Data elements:

n	int n;	Input value as a number and not text.

Member functions:

Transfer	virtual WORD Transfer (void* DataPtr, WORD TransferFlag); Transfer and conversion from data element n to the string in the input field.
CanClose	virtual BOOL CanClose ( ); tries to convert string from input field to an integer. If successful it returns OK. Otherwise ERROR.

TIntegerInputI <owlappl.h>

This class is derived from *TIntegerInput*. In addition it checks whether the value x lies in an interval [minVal, maxVal]. If not a message *aMessage* pops up.

Constructors:
TIntegerInputI (PTWindowsObject AParent, int ResourceId, const int aMinVal, const int aMaxVal, const char* aMessage);

Member functions:

CanClose	virtual BOOL CanClose ( ); see *TDoubleInput*

TJointDialog <owdialg.h>

Implementation of the 'Joint Renewable Distribution' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TJointDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Function is carried out upon initialisation of the window.
virtual void HandleCondMsg (RTMessage) = [WM_FIRST + idOpCond];
Function is called upon selection of 'Joint Conditional Function' option in the dialog window. If this option is selected input fields are made visible or invisible as appropriate. The id- constant is defined in *owres.h*.
virtual void HandleProbMsg (RTMessage) = [WM_FIRST + idOpProbDens];
Function is called upon selection of 'Joint Density Function' option in the dialog window. See *HandleCondMsg* above.

TJointDistributionObject <owcalc.h>

Calculation window on which calculations of joint renewable power distributions are carried out, derived from *TMultiValObject*. All necessary functions are privately overwritten. See *TMultiValObject*.

Constructors:
TJointDistributionObject (PTWindowsObject AParent, LPSTR ATitle);

TMainWindow <owrenew.h>

Implementation of the main window with the menu bar. It is derived from the Object Windows C++ class *TWindow*.

Constructors:
TMainWindow (PTWindowsObject AParent, LPSTR ATitle);

Data elements:

TTransSettingsDlg	TransSettingsDlg;	Buffer for "Settings" window
TTransDirDlg	TransDirDlg;	Buffer for "Directories" window
TTransExportDlg	TransExportDlg;	Buffer for "Export" window
TTransDisplayDlg	TransDisplayDlg;	Buffer for "Display" window
TTransSpeedDlg	TransSpeedDlg;	Buffer for "Wind Speed Distributions" window
TTransWindDlg	TransWindDlg;	Buffer for "Wind Power Distribution" window
TTransSolarDlg	TransSolarDlg;	Buffer for "Solar Power Distribution" window
TTransJointDlg	TransJointDlg;	Buffer for "Joint Renewable Power Distribution" window
TTransRandDlg	TransRandDlg;	Buffer for "Random number" dialog window
TTransMathsDlg	TransMathsDlg;	Buffer for "Maths" window
TTransTsDlg	TransTsDlg;	Buffer for "Time Series" window
TTransFpDlg	TransFpDlg;	Buffer for "First Passage Time Problems" window
PTRenewPlot	testplot;	Graphic window that sits on top of the main window.

Member functions:

CanClose
virtual BOOL CanClose ( );
Pops up a message window and asks whether the user really wants to quit. If 'Yes' the function returns YES. Otherwise NO.

virtual void CMWindSpeed (RTMessage) = [CM_FIRST + cmWindSpeed];
Function is called upon selection of "Wind speed distribution" menu item. It opens the appropriate dialog by initialising an instance of class *TSpeedDialog*.

virtual void CMSettings (RTMessage) = [CM_FIRST + cmSettings];
Function is called upon selection of "Settings" menu item. It opens the appropriate dialog by initialising an instance of class *TSettingsDialog*.

virtual void CMMaths (RTMessage) = [CM_FIRST + cmMaths];
Function is called upon selection of "Maths" menu item. It opens the appropriate dialog by initialising an instance of class

TMathsDialog.

virtual void CMWindPower (RTMessage) = [CM_FIRST + cmWindPower];
Function is called upon selection of "Wind Power Distribution" menu item. It opens the appropriate dialog by initialising an instance of class *TWindDialog.*

virtual void CMSolar (RTMessage) = [CM_FIRST + cmSolar];
Function is called upon selection of "Solar Power Distribution" menu item. It opens the appropriate dialog by initialising an instance of class *TSolarDialog.*

virtual void CMRenewable (RTMessage) = [CM_FIRST + cmRenewable];
Function is called upon selection of "Joint Renewable Distribution" menu item. It opens the appropriate dialog by initialising an instance of class *TJointDialog.*

virtual void CMExport (RTMessage) = [CM_FIRST + cmExport];
Function is called upon selection of "Export" menu item. It opens the appropriate dialog by initialising an instance of class *TExportDialog.*

virtual void CMDisplay (RTMessage) = [CM_FIRST + cmDisplay];
Function is called upon selection of "Display Options" menu item. It opens the appropriate dialog by initialising an instance of class *TDisplayDialog.*

virtual void CMHelp (RTMessage) = [CM_FIRST + cmHelp];
Function is called upon selection of "Help" menu item. It pops up a message that this feature is not implemented.

virtual void CMDir (RTMessage) = [CM_FIRST + cmDirectories];
Function is called upon selection of "Directories" menu item. It opens the appropriate dialog by initialising an instance of class *TDirDialog.*

virtual void CMRandom (RTMessage) = [CM_FIRST + cmRandom];
Function is called upon selection of "Random Numbers" menu item. It opens the appropriate dialog by initialising an instance of class *TRandomDialog.*

virtual void CMTimeSeries (RTMessage) = [CM_FIRST + cmTimeSeries];
Function is called upon selection of "Time Series" menu item. It opens the appropriate dialog by initialising an instance of class *TTsDialog.*

virtual void CMFpt (RTMessage) = [CM_FIRST + cmFirstPassage];
Function is called upon selection of "First Passage Time Problems" menu item. It opens the appropriate dialog by initialising an instance of class *TFpDialog.*

Operators:
Save dialog window data to a file and retrieving them in the next session by using the stream operators. They affect all data stored in the buffers with prefix 'Trans'.

friend ostream& operator << (ostream&, RTMainWindow);

friend istream& operator >> (istream&, RTMainWindow);

TMathsDialog <owdialg.h>

Implementation of the 'Mathematical Options' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TMathsDialog (PTWindowsObject AParent, LPSTR ATitle);

TMultiValObject <owstat.h>

This class is derived from *TStatusWindow*. It is designed for the case that more than one value is to be calculated.

Constructors:
TMultiValObject (PTWindowsObject AParent, LPSTR ATitle, int eval);
initialise the class with *eval* being the number of function evaluations to be carried out.

Member functions:
protected:

workOutBasic	virtual int workOutBasic ( ) = 0; The function *TStatusWindow::workOut* has been split up here into two parts: First, calculations that have to be carried out prior to the evaluation of the first function value. This goes in here.
workOutValues	virtual int workOutValues ( ) = 0; This is the second part, where all values are calculated. The split is necessary as after *workOutBasic* the parameters are checked. In case they are pointless (return value of *workOutBasic* not OK) a message window will inform the user. Otherwise the programme continues with the calculation of the function values in *workOutValues.*
areParametersOK	virtual int areParametersOK ( ) = 0; This function is only called if the accumulation of curves in the diagram is desired. Here is the function to check that the current curve is compatible with the last calculations.
setOldParameter	virtual void setOldParameter ( ) = 0; This function is to be called after *areParameterOK* and is used by the next calculations for the same reason as stated in *areParametersOK.*
workOut	int workOut ( ); overwrites *TStatusWindow::workOut* by splitting up into *workOutBasic* and *workOutValues.*

calcValues	void calcValues (owObjfunc* func, double xmin, double xmax); The function carries out *eval* function evaluations on the object function *func* in the x- interval [*xmin, xmax*]. The number of evaluations is already specified in the constructor.
public:	
calc	static void calc (owObjfunc* func, double xmin, double xmax, int N, TStatusWindow* window); Static member function that carries out N function evaluations on *func* by using the status window *window*.

TPassageTimeObject <owcalc.h>

Calculation window on which the calculation of the first passage time is carried out provided only one value is required at the time, derived from *TStatusWindow*. If a whole curve of first passage time values (e.g. as a function of hte initial value) is required use class *PassageTimesObject*. All necessary functions are privately overwritten. See *TStatusWindow*.

Constructors:
TPassageTimeObject (PTWindowsObject AParent, LPSTR ATitle);

Member functions:

workOut	protected: int workOut ( ); carry out the random number generator test.
writeRep1	protected: void writeRep1 ( ); write reply to parent StatusWindow into textline.

TPlot <owplot.h>

Graphical representation of functions. *TPlot* draws a complete coordinate system, with axes, grid lines, text and curves. It is derived from *TGraph*.

Constructors:
TPlot (PTWindowsObject AParent, LPSTR ATtitle, PTModule AModule = NULL);

Member functions:
public:

plot	virtual void plot ( );	do nothing! This function has to be overwritten by derived classes.
draw	virtual void draw ( );	draw the whole diagram by calling *plot*.
Paint	virtual void Paint (HDC PaintDC, PAINTSTRUCT _FAR& P); overwrites *Paint* from *TWindow*. See Object Windows C++ guide for more details.	
setHeadLine	void setHeadLine (const char*);	set headline
setSubLine	void setHeadLine (const char*);	set line below headline
plotFactor	void plotFactor (double x);	

	write scaling factor on top of y- axis. This is basically the data element *scale* in *Graph.*	
protected:		
Text functions:		
plotHeadLine	void plotHeadLine( );	plot headline
plotSubLine	void plotSubLine ( );	plot line below headline
Coordinates and Positioning		
drawMargin	void drawMargin ( );	draw rectangular (circumference of the diagram)
xcoord	int xcoord (double x);	return coordinate on the screen for x-value
ycoord	int ycoord (double y);	return coordinate on the screen fro y-value
setCoordinates	void setCoordinates (double xmin, double xmax, double ymin, double ymax); specifies the valid diagram coordinates.	
setAutoCoord	double setAutoCoord (double xmin, double xmax, VECTOR* yval, int n=0); automatic determination of the coordinates dependent on the given start and finish value on the x- axis and the vector with the corresponding y- values, *yval.* If y- values of more than one curves are to be taken into account n is to be set > 0. Function returns the calculated scaling factor for the y- axis. This factor should be printed out using *plotFactor* function.	
setAutoAxAttr	void setAutoAxAttr (double& xaxle, double& yaxle, int& xnum, int& ynum, double& xgrind, double& ygrid); automatic determination of attributes of an axis given the input parameters. See class *axis* for significance of the parameters.	
setViewport	void setViewport (int xmin, int xmax, int ymin, int ymax); Specifiaction of location of diagram in the window	
Axes, curves and the coordinate system:		
drawUpperX	void drawUpperX (double mini, double maxi, double axle, int num, int log, const char* text, int axle_mode); draw upper x- axis with parameters as in *axis::setAxis.*	
drawLowerX	void drawLowerX (double mini, double maxi, double axle, int num, int log, int dist, const char* text, int axle_mode); draw lower x- axis with parameters as in *axis::setAxis.*	
drawRightY	void drawRightY (double mini, double maxi, double axle, int num, int log, const char* text, int axle_mode); draw right hand y- axis with parameters as in *axis::setAxis.*	
drawLeftY	void drawLeftY (double mini, double maxi, double axle, int num, int log, int dist, const char* text, int axle_mode); draw left y- axis with parameters as in *axis::setAxis.*	
drawLinCoord	void drawLinCoord (double xaxle, int xnum, int xaxgrid, double xgrid, const char* xtext, double yaxle, int ynum, int yaxgrid,	

	double ygrid, const char* ytext); Draw a linear coordinate system. Parameters as in *axis::setAxis*. Please note that *setCoordinates* has to be called prior to this function.
drawAutoLinCoord	void drawAutoLinCoord (double xmin, double xmax, VECTOR* yval, const char* xtext, const char* ytext, int xaxgrid, int yaxgrid, double scale int n = 0); Draw a linear coordinate system using *drawLinCoord*. Though, before call *setAutoCoord*.
drawCurve	void drawCurve (VECTOR& x, VECTOR& y, DRA_MODE draw_mode); Draw a curve with its x- and y- values in the diagram. *draw_mode* is one of the following options: PIXEL Do not connect two points POLYGON Do connect subsequent points by a line STEP Draw function as a staircase function DIRAC Draw function as a Dirac function

TRenewApp <owrenew.h>

Main application, derived from Object Windows C++ class *TApplication*.

Constructors:
TRenewApp (LPSTR AName, HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow);

Member functions:

InitMainWindow	virtual void InitMainWindow ( ); overwrites *TApplication::InitMainWindow* and intialises an instance of *TMainWindow*.

TRenewPlot <owrenw.h>

Implementation of the graphic window that draws the diagrams. It is directly derived from *TPlot*. It is extended by the *clear* - flag. See data element below.

Constructors:
TRenewPlot (PTWindowObject AParent, LPSTR ATitle, PTModule AModule = NULL);

Data elements:

clear	int clear;	If *clear* is set to NO, the window draws the implied diagram. Otherwise the next call to *Paint* causes the window to be cleared.

Member functions:

Paint	virtual void Paint (HDC PaintDC, PAINTSTRUC _FAR& PaintInfo); calls *TPlot::Paint* if *clear* is YES. Otherwise it calls *TPlot::draw.*
plot	void plot ( ); overwrites *TPlot::plot.* It draws the whole diagram given the curve data in *GraphData* which is an instance of class *Graph.*

TSJointPassageTime <passage.h>

Object that calculates the first passage time of joint renewable power fluctuations using the time series approach. It is derived from *TSPassageTime.*

Constructors:

TSPassageTime ( );	Constructor that initialises a *JointPowerTimeSeries* object in place of *timeSeries* data element.

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting up the appropriate parameters.

TSPassageTime <passage.h>

Object that calculates the first passage time using the time series approach. It is directly derived from *PassageTime.*

Constructors:

TSPassageTime ( );	Default Constructor

Data elements:
protected:

timeSeries	TimeSeries* timeSeries; Time series object to be used in the first passage time calculations.

Member functions:
protected:

SetUp	virtual int SetUp (TStatusWindow*, Param*); see *PassageTime::setUp.*

public:

Eval	double Eval (double x); returns the first passage time where x is the passage level. It uses the time series *timeSeries.* Hence, derived classes need to initialise the time series they require.
setInitLevel	void setInitLevel (void*);

overwrites *PassageTime::setInitLevel* for time series approach objects. It calls *TimeSeries::setUserInit*

TSSolarPowerPassageTime <passage.h>

Object that calculates the first passage time of PV array power fluctuations using the time series approach. It is derived from *TSPassageTime.*

Constructors:

TSPassageTime ( );	Constructor that initialises a *SolarPowerTimeSeries* object in place of *timeSeries* data element.

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting up the appropriate parameters.

TStatusWindow <owstat.h>

Window that pops up just before starting a calculation. Upon pressing the OK button the calculations are carried out. The status of the calculations can be observed by looking at the status lines in the window. It is derived from *TDialog.*

Constructors:
TStatusWindow (PTWindowsObject AParent, LPSTR ATitle);

Data elements:

temp	static double temp; This is a static data element. Calculation objects can write values in it that can be picked up by *TStatusWindow.*

Member functions:
protected:

giveWarning	int giveWarning (char* message); opens a window issuing a warning with text *message*. The user is given three options: OK, Ignore or Abort. Depending on his selection the return value is IDOK, IDIGNORE or IDABORT.	
writeRep1	virtual void writeRep1 ( );	print out the first status line.
writeRep2	virtual void writeRep2 ( );	print out the second status line.
workOut	virtual int workOut ( ) = 0; Abstract function that must be overwritten in derived classes. It carries out all the calculations. It returns OK if no error occurred.	

virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Initialisation of the dialog window

virtual void Ok (RTMessage) = [ID_FIRST + IDOK];
Function that is called upon the selection of the OK button. It calls *workOut* to carry out the calculations.

virtual void Retry (RTMessage) = [ID_FIRST + IDRETRY];
Function that is called upon the selection of the Retry button. It is almost identical with *Ok*. Only that the Retry button can not always be selected.

virtual void TimeMsg (RTMessage) = [WM_USER+WM_MSGOBJFUNC];
Function called upon a time message that is invoked in an instance of the class *msgObjfunc*. Calculations should be carried out in this class, as it enables them to send time messages. *TStatusWindow* receives the time message and write then the elapsed time (since starting the calcualtions) to the status line.

public:

writeTime — void writeTime ( );
write the time elapsed to the satus line

isEnoughTime — int isEnoughTime ( );
in order to avoid writing to the screen too often this function can be asked prior to writing to the screen whether enough time has been elapsed since last writing. If so, it returns YES. Otherwise NO.

writeStatus1 — void writeStatus1 (char* text);
write *text* to first status line.

writeStatus2 — void writeStatus2 (char* text);
write *text* to second status line in the dialog window.

TSWindSpeedPassageTime <passage.h>

Object that calculates the first passage time of wind speed fluctuations using the time series approach. It is derived from *TSPassageTime*.

Constructors:

TSWindSpeedPassageTime ( ); — Constructor that initialises a *WindSpeedTimeSeries* object in place of *timeSeries* data element.

Member functions:

SetUp — int SetUp (TStatusWindow*, Param*);
Setting up the appropriate parameters.

TSWindPowerPassageTime <passage.h>

Object that calculates the first passage time of wind turbine power fluctuations using the time series approach. It is derived from *TSPassageTime*.

Constructors:

TSPassageTime ( );	Constructor that initialises a *WindPowerTimeSeries* object in place of *timeSeries* data element.

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); Setting up the appropriate parameters.

TRandDialog <owdialg.h>

Implementation of the 'Random Numbers' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:

TRandDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:

virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Function is carried out upon initialisation of the window.

virtual void HandleUniMsg (RTMessage) = [WM_FIRST + idOpRandOp0];
Function is called upon selection of 'Uniform distribution' option in the dialog window. If this option is selected input fields are made visible or invisible as appropriate. The id- constant is defined in *owres.h*.

virtual void HandleNormMsg (RTMessage) = [WM_FIRST + idOpRandOp1];
Function is called upon selection of 'Normal distribution' option in the dialog window. See *HandleUniMsg* above.

virtual void HandleBetaMsg (RTMessage) = [WM_FIRST + idOpRandOp2];
Function is called upon selection of 'Beta- distribution' option in the dialog window. See *HandleUniMsg* above.

virtual void HandleBiMsg (RTMessage) = [WM_FIRST + idOpRandOp2];
Function is called upon selection of 'Binomial distribution' option in the dialog window. See *HandleUniMsg* above.

TRandomObject <owcalc.h>

Calculation window on which the random number generators are tested, derived from *TStatusWindow*. All necessary functions are privately overwritten. See *TStatusWindow*.

Constructors:

TRandomObject (PTWindowsObject AParent, LPSTR ATitle);

Member functions:

workOut	protected: int workOut ( ); carry out the random number generator test.
writeRep1	protected: void writeRep1 ( ); write reply to parent StatusWindow into textline.

TSettingsDialog <owdialg.h>

Implementation of the 'Settings' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TSettingsDialog (PTWindowsObject AParent, LPSTR ATitle);

TSolarDialog <owdialg.h>

Implementation of the 'Solar Power Distribution' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TSolarDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Function is carried out upon initialisation of the window.
virtual void HandleAnalytMsg (RTMessage) = [WM_FIRST + idOpAnalyt];
Function is called upon selection of 'Analytical Function' option in the dialog window. If this option is selected input fields are made visible or invisible as appropriate. The id- constant is defined in *owres.h*.
virtual void HandleApproxMsg (RTMessage) = [WM_FIRST + idOpApprox];
Function is called upon selection of 'Approximation' option in the dialog window. See *HandleAnalytMsg* above.
virtual void HandleCondMsg (RTMessage) = [WM_FIRST + idOpCond];
Function is called upon selection of 'Conditional Distribution' option in the dialog window. See *HandleAnalytMsg* above.
virtual void HandleQualMsg (RTMessage) = [WM_FIRST + idOpQual];
Function is called upon selection of 'Quality of Approximation' option in the dialog window. See *HandleAnalytMsg* above.

TSpeedDialog <owdialg.h>

Implementation of the 'Wind Speed Distribution' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TSpeedDialog (PTWindowsObject AParent, LPSTR ATitle);

TTimeSeriesObject <owcalc.h>

Calculation window on which calculations of time series are carried out, derived from *TMultiValObject*. All necessary functions are privately overwritten. See *TMultiValObject*.

Constructors:
TTimeSeriesObject (PTWindowsObject AParent, LPSTR ATitle);

TTransDirDlg <owdialg.h>

Parameter transfer buffer for *TDirDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransDirDlg ( );	Default Constructor

Data elements:

solFile	char solFile[50];	file name of file for solar data
dlgFile	char dlgFile[50];	file name of dialog file (not used in the programme!)

Operators:
friend ostream& operator << (ostream&, TTransDirDlg&);
friend istream& operator >> (istream&, TTransDirDlg&);

TTransDisplayDlg <owdialg.h>

Parameter transfer buffer for *TDisplayDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransDisplayDlg ( );	Default Constructor

Data elements:

opAuto	WORD opAuto;	Flag: Automatic display of graphs
opAccu	WORD opAccu;	Flag: Accumulating data series
opLegend	WORD opLegend;	Flag: Ask for legend text

Member functions:

setParameter	void setParameter ( );

	Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:
friend ostream& operator << (ostream&, TTransDisplayDlg&);
friend istream& operator >> (istream&, TTransDisplayDlg&);

TTransExportDlg <owdialg.h>

Parameter transfer buffer for *TExportDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransExportDlg ( );	Default Constructor

Data elements:

opNew	WORD opNew;	Flag: Export data to new file
opAttach	WORD opAttach;	Flag: Attach data to existing file
expFile	char expFile[50];	Name of export file

Member functions:

setParameter	void setParameter ( ); Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:
friend ostream& operator << (ostream&, TTransExportDlg&);
friend istream& operator >> (istream&, TTransExportDlg&);

TTransFpDlg <owdialg.h>

Parameter transfer buffer for *TFpDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransFpDlg ( );	Default Constructor

Data elements:

fpOp0	WORD fpOp0;	Flag: Wind Speed
fpOp1	WORD fpOp1;	Flag: Wind turbine power time series
fpOp2	WORD fpOp2;	Flag: PV array power time series
fpOp3	WORD fpOp3;	Flag: Joint renewable power time series
fpOp4	WORD fpOp4;	Flag: Time Series Approach
fpOp5	WORD fpOp5;	Flag: Markov Chain Approach

fpOp6	WORD fpOp6;	Flag: Calculate one value only
fpOp7	WORD fpOp7;	Flag: Calculate first passage time as a function of the initial value
fpOp8	WORD fpOp8;	Flag: Calculate first passage time as a function of the passage level
initV	double initV;	Initial wind speed
initK	double initK;	Initial average hourly clearness index k(0)
initP	double initP;	Initial, normalised power
passV	double passV;	wind speed passage level
passK	double passK;	clearness index passage level
passP	double passP;	power passage level
timeStep	double timeStep;	time step for time series approach only
noVal	int noVal;	number of values to be calculated if more than one value is required

Member functions:

setParameter — void setParameter ( );
Ttssfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransFpDlg&);
friend istream& operator >> (istream&, TTransFpDlg&);

TTransJointDlg <owdialg.h>

Parameter transfer buffer for *TJointDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransJointDlg ( ); — Default Constructor

Data elements:

opJointDens	WORD opJointDens;	Flag: Joint density function (stationary)
opJointCond	WORD opJointCond;	Flag: Joint conditional density function
vmean	double vmean;	Average wind speed
initialv	double initialv;	Initial wind speed
clearness	double clearness;	Average hourly clearness index k
initialK	double initialK;	Initial average hourly clearness index k(0)
tau	double tau;	Time tau (for conditional distribution)
eval	int eval;	Number of evaluations
zeta	double zeta;	Fractional power factor ζ

Member functions:

setParameter	void setParameter ( ); Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransJointDlg&);
friend istream& operator >> (istream&, TTransJointDlg&);

TTransMathsDlg <owdialg.h>

Parameter transfer buffer for *TMathsDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransMathsDlg ( );	Default Constructor

Data elements:

solTrial	int solTrial;	number of trial points in optimisation of approximated distribution function of PV array power.
solCoeff	int solCoeff;	number of coefficients in approximated distribution function of PV array power.
fpTsTrial	int fpTsTrial;	Number of time series taken into account in first passage time calculations using the time series approach.
fpTsMaxIt	int fpTsMaxIt;	Maximum number of iterations in the time series approach algorithm for first passage times
fpMcStopCrit	double fpMcStopCrit;	Stopping criterion in the Markov chain approach algorithm.
fpMcMaxIt	int fpMcMaxIt;	Maximum number of iterations in the Markov chain approach algorithm.
fpMcGrid	int fpMcGrid;	Not in use.
classes	int classes;	Number of discrete levels in a discrete distribution.

Member functions:

setParameter	void setParameter ( ); Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransMathsDlg&);
friend istream& operator >> (istream&, TTransMathsDlg&);

TTransRandDlg <owdialg.h>

Parameter transfer buffer for *TRandDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransRandDlg ( ); Default Constructor

Data elements:

ranOp0	WORD ranOp0;	Flag: Uniform distribution
ranOp1	WORD ranOp1;	Flag: Normal distribution
ranOp2	WORD ranOp2;	Flag: Beta- distribution
ranOp3	WORD ranOp3;	Flag: Binomial distribution
ranA	double ranA;	Parameter α for Beta- distribution
ranB	double ranB;	Parameter β fro Beta- distribution
ranClass	int ranClass;	Number of classes for Kolmogorov-Smirnov test
ranTrial	int ranTrial;	Number of random numbers to be generated per set.

Member functions:

setParameter — void setParameter ( );
Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransRandDlg&);
friend istream& operator >> (istream&, TTransRandDlg&);

TTransSettingsDlg <owdialg.h>

Parameter transfer buffer for *TSettingsDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransSettingsDlg ( ); Default Constructor

Data elements:

wiVci	double wiVci;	cut-in wind speed
wiVco	double wiVco;	cut-out wind speed
wiVr	double wiVr;	rated wind speed
wiVmean	double wiVmean;	mean wind speed
wiSigma	double wiSigma;	wind standard deviation σ_k
wiBeta	double wiBeta;	autocorrelation coefficient of wind

		turbulence β_v
solK0	double solK0;	max hourly clearness index K_0.
solSigmaK	double solSigmaK;	standard deviation of the average hourly clearness index, σ_k.
solBeta	double solBeta;	autocorrelation coefficient of the hourly clearness index, β_x
comZeta	double comZeta;	Fractional power factor ζ
batK	double batK;	Battery factor k
batC	double batC;	Battery factor c
batQMax	double batQMax;	Battery Capacity
batV	double batV;	Battery voltage
sysPRen	double sysPRen;	Installed renewable power
sysPDemand	double sysPDemand;	Power demand

Member functions:

setParameter — void setParameter ( );
Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransSettingsDlg&);
friend istream& operator >> (istream&, TTransSettingsDlg&);

TTransSolarDlg <owdialg.h>

Parameter transfer buffer for *TSolarDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransSolarDlg ( ); — Default Constructor

Data elements:

opProb	WORD opProb;	Flag: probability density function
opDist	WORD opDist;	Flag: distribution function
opAnalyt	WORD opAnalyt;	Flag: Analytical function
opApprox	WORD opApprox;	Flag: Approximation
opCond	WORD opCond;	Flag: conditional process
opQual	WORD opQual;	Flag: Quality of approximation
opBypass	WORD opBypass;	Flag: Bypass selected.
clearness	double clearness;	Average hourly clearness index
initialK	double initialK;	Initial average hourly clearness index k(0)
trial	int trial;	Number of trial points (for approximation only)
coeff	int coeff;	Number of coefficients in approximation

of distribution function.

Member functions:

setParameter	void setParameter ( ); Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransSolarDlg&);
friend istream& operator >> (istream&, TTransSolarDlg&);

TTransSpeedDlg <owdialg.h>

Parameter transfer buffer for *TSpeedDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransSpeedDlg ( );	Default Constructor

Data elements:

opProb	WORD opProb;	Flag: Probability density function
opDist	WORD opDist;	Flag: Distribution function
vmean	double vmean;	mean wind speed
vmin	double vmin;	Speed at which to start calculations
vmax	double vmax;	Speed at which to finish calculations
eval	int eval;	Number of evaluations required

Member functions:

setParameter	void setParameter ( ); Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransSpeedDlg&);
friend istream& operator >> (istream&, TTransSpeedDlg&);

TTransTsDlg <owdialg.h>

Parameter transfer buffer for *TTsDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransTsDlg ( );	Default Constructor

Data elements:

tsOp0	WORD tsOp0;	Flag: Wind speed time series
tsOp1	WORD tsOp1;	Flag: wind turbine power time series
tsOp2	WORD tsOp2;	Flag: PV array power time series
tsOp3	WORD tsOp3;	Flag: Joint renewable power time series
tsOp4	WORD tsOp4;	Flag: State of charge time series
tsOp5	WORD tsOp5;	Flag: Power deficit time series
tsTimeStep	double tsTimeStep;	Time step Δt that is implicit in the time series
tsPoints	int tsPoints;	Number of points to be calculated
initV	double initV;	Initial wind speed
initK	double initK;	Initial average hourly clearness index k(0)
initQ10	double initQ10;	Initial available charge Q_{10}
initQ20	double initQ20;	Intiial bound charge Q_{20}

Member functions:

setParameter — void setParameter ( );

Ttssfer above data elements to the corresponding fields in the global variable *param*.

Operators:

friend ostream& operator << (ostream&, TTransTsDlg&);
friend istream& operator >> (istream&, TTransTsDlg&);

TTransWindDlg <owdialg.h>

Parameter transfer buffer for *TWindDialog*. All input paramters in the dialog window appear here as data elements.

Constructors:

TTransWindDlg ( ); — Default Constructor

Data elements:

opProb	WORD opProb;	Flag: probability density function
opDist	WORD opDist;	Flag: distribution function
opStationary	WORD opStationary;	Flag: stationary process
opCond	WORD opCond;	Flag: conditional process
vmean	double vmean;	mean wind speed
eval	int eval;	number of evaluations required
tau	double tau;	time tau
initialv	double initialv;	initial wind speed

Member functions:

setParameter — void setParameter ( );

Transfer above data elements to the corresponding fields in the global variable *param*.

Operators:
friend ostream& operator << (ostream&, TTransWindDlg&);
friend istream& operator >> (istream&, TTransWindDlg&);

TTsDialog <owdialg.h>

Implementation of the 'Time Series' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TTsDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Function is carried out upon initialisation of the window.
virtual void HandleOp0Msg (RTMessage) = [WM_FIRST + idTsOp0];
Function is called upon selection of 'Wind Speed' option in the dialog window. If this option is selected input fields are made visible or invisible as appropriate. The id- constant is defined in *owres.h*.
virtual void HandleOp1Msg (RTMessage) = [WM_FIRST + idTsOp1];
Function is called upon selection of 'Wind Power' option in the dialog window. See *HandleOp0Msg* above.
virtual void HandleOp2Msg (RTMessage) = [WM_FIRST + idTsOp2];
Function is called upon selection of 'Solar Power' option in the dialog window. See *HandleOp0Msg* above.
virtual void HandleOp3Msg (RTMessage) = [WM_FIRST + idTsOp3];
Function is called upon selection of 'Combined Renewable' option in the dialog window. See *HandleOp0Msg* above.
virtual void HandleOp4Msg (RTMessage) = [WM_FIRST + idTsOp4];
Function is called upon selection of 'Battery: State of Charge' option in the dialog window. See *HandleOp0Msg* above.
virtual void HandleOp5Msg (RTMessage) = [WM_FIRST + idTsOp5];
Function is called upon selection of 'Power Deficit' option in the dialog window. See *HandleOp0Msg* above.

TYoMessage <owlappl.h>

Implementation of a message window with title *ATitle*, and four different actions that can be taken. See member functions. *TYoMessage* is derived from *TDialog*. Which event functions may be called depends on the resource ID the class was constructed with. E.g. it might be a window with only a Yes and No button. In this instance the function CMIgnore could not be called as there is no Ignore button.

Constructors:
TYoMessage (PTWindowsObject AParent, LPSTR ATitle, char* a Message);

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
virtual void CMYes (RTMessage) = [ID_FIRST + IDYES];
Function called upon event 'CMYes'
virtual void CMNo (RTMessage) = [ID_FIRST + IDNO];
Function called upon event 'CMNo'
virtual void CMIgnore (RTMessage) = [ID_FIRST + IDIGNORE];
Function called upon event 'CMIgnore' (Ignore - button)
virtual void CMAbort (RTMessage) = [ID_FIRST + IDABORT];
Function calles upon event 'CMAbort' (Abort - button)

TYoInput <owlappl.h>

Implementation of a dialog window with one input field. It is derived from *TDialog.*

Constructors:
TYoInput (PTWindowsObject AParent, LPSTR ATitle, char* title, char* input);

Data elements:

textBuffer	char textBuffer[80];	Implied input string

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Initialisation of the window.

TWindDialog <owdialg.h>

Implementation of the 'Wind Power Distribution' dialog window, derived from *TDialog* of the Object Windows C++ library.

Constructors:
TWindDialog (PTWindowsObject AParent, LPSTR ATitle);

Member functions:
virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
Function is carried out upon initialisation of the window.
virtual void HandleCondMsg (RTMessage) = [WM_FIRST + idOpCond];
Function is called upon selection of 'Conditional distribution'

option in the dialog window. If this option is selected the input fields for time tau and initial wind speed have to be made visible. The constant idOpCond is defined in *owres.h.*

virtual void HandleStatMsg (RTMessage) = [WM_FIRST + idOpStationary];

Function is called upon selection of 'Stationary distribution' option in the dialog window. It makes the input fields for the time tau and the initial wind speed invisible. Compare with *HandleCondMsg*

TWindSpeedObject <owcalc.h>

Calculation window on which calculations of wind speed distributions are carried out, derived from *TMultiValObject.* All necessary functions are privately overwritten. See *TMultiValObject.*

Constructors:

TWindSpeedObject (PTWindowsObject AParent, LPSTR ATitle);

UniKgSTest <random.h>

Kolmogorov- Smirnov test for uniform distribution, derived from *KgSTest.*

Constructors:

UniKgSTest (n); Construct test object for n trial points.

Member functions:

theoretProb double theoretProb (double x); see *KgSTest::theoretProb.*

initialize void intiialize ( );
initialise *randomizer* with *uniRand* object.

uniRand <random.h>

Random number generator. The numbers are drawn from a uniform distribution using the standard C - library function *rand().* Before generating number for the first time the member function *initialize ( )* should be called.

Constructor:

uniRand ( ) Default constructor

Member functions:

initialize void initialize ( );
initializes the generator with the current time.

getRandomNumber	virtual double getRandomNumber ( ); returns the next random number.

uniRejectRand <random.h>

Implementation of a random number generator using the rejection mehtod with a uniform distribution as comparison function for distributions with non zero values in the interval [0,1]. It is immediately derived from *rejectRand.*

Constructors:

uniRejectRand ( );	Default constructor with unit ceiling.
uniRejectRand (double max);	Constructor with ceiling *max.*

VECTOR_ <vectors.h>

Implementation of a vector with real number as elements. The class implements a huge variety of functions on vectors and operations.

typedef VECTOR_<int> IVECTOR;
typedef VECTOR_<double> VECTOR;
typedef VECTOR_<float> FVECTOR;

Constructors:

VECTOR_ (VECTOR_& v);	Copy constructor.
VECTOR_ (int n);	Construction of an n- dimensional vector.

Data members:

dim	int dim;	Dimension of the vector

Member functions:

add	void add (T x); add an element, x, to the vector and increment its dimension by 1.
build	void build (istream& ip); Standard input vid input stream ip
copy	VECTOR_<T> copy (int n); returns a vector containing the first n components of **this.*
create	void create (int n); allocates memory for n components.
del	void del (int n); deletes element numbe n from the vector and decrements the dimension by 1.
move_down	T move_down ( );

	moves down all components by 1. Return element that is no longer in the vector.
move_up	T move_up ( ); moves up all components by 1. Return element thath is no longer in the vector.
mul	friend MATRIX_ mul (VECTOR_& u, VECTOR_& v); Vecor ultipliction $\mathbf{A} = \mathbf{u}\ \mathbf{v}^T$
print	void print (ostream& op); Standard output on the stream.
search	int search (T x); Search for element x in the vector. Return the index of the first element. If x is not element of the vector, return 0, otherwise its index.
set	void set (T x); Set all components on x.
swap	void swap (int i, int j); Swap the i-th and j-th element.

Operators:

( )	v(int i);	Access to elemnt i (indices from 1 ...dim)
+, +=, -, -=	**v** + **u**, **v** + a, **v** - **u**, **v** - a (**u**, **v**, Vectors; a real number) Note.: Addition or subtraction of a real number means all components are affected in the same way.	
*	multply by number: $\mathbf{u} = \alpha * \mathbf{v}$, $\mathbf{u} = \mathbf{v} * \alpha$, $\mathbf{v} *= \alpha$ multiply each component: $\mathbf{u} = \mathbf{v} * \mathbf{w}$	
/	Divide by number α: $\mathbf{u} = \mathbf{v} / \alpha$; $\mathbf{u} /= \alpha$;	
=	**v** = **u** Assignment. Works even if dimensions of both vectors befor assignment are not the same	
==	(**u** == **v**)	TRUE, if all components in **u** and **v** are identical.
!=	(**u** != **v**)	TRUE, if at least two components of **u** and **v** are different.
<,<=	(**u** <= **v**)	TRUE, if all components of **u** are less than the components of **v**.
>,>=	(**u** >= **v**)	TRUE, if all components of **u** are greater than the components of **v**.
<<	operator << (ostream& op, VECTOR_<T>&v);	
>>	operator >> (istream& ip, VECTOR_<T>& v);	

WindPowerPassageTimes <passage.h>

Object function for first passage times of wind turbine pwoer fluctuations, derived from *PassageTimes.*

Constructors:

WindSpeedPassageTimes (int select);
Constructor: If *select* = 0 the data element *passageTime* is initialised with an instance of *TSWindPowerPassageTime.* Otherwise with *MCWindPowerPassageTime.*

Member functions:

SetUp — int SetUp (TStatusWindow*, Param*);
individual set-up of initial values and passage levels.

WindPowerTimeSeries <series.h>

Implementation of wind turbine power time series, derived from class *WindSpeedTimeSeries.*

Constructors:

WindPowerTimeSeries ( ); — calls the constructors of the base class.

Member functions:

getWindPower — static double getWindPower (double v, double vci, double vco, double vr);
return the wind turbine power for a given wind speed, v, cut-in wind speed vci, cut-out wind speed, vco, and a rated wind speed, vr. It uses equation (3.1).

getV — static double getV (double p, double vci, double vr);
Inverse function to *getWindPower.* It returns the wind speed for a given power p, cut-in wind speed vci and rated wind speed, vr. It uses the invertible part of (3.1) only.

getOutput — double getOutput ( ); see *TimeSeries::getOutput*

setUp — int setUp (TStatusWindow*, Param*);

WindSpeedPassageTimes <passage.h>

Object function for first passage times of wind speed fluctuations, derived from *PassageTimes.*

Constructors:

WindSpeedPassageTimes (int select);
Constructor: If *ele t* = 0 the data element *pa ageTi e* is initialised with an instance of *TSWindSpeedPassageTime*. Otherwise with *MCWindSpeedPassageTime.*

Member functions:

SetUp	int SetUp (TStatusWindow*, Param*); individual set-up of initial values and passage levels.

WindSpeedTimeSeries <series.h>

Implementation of wind speed time series, derived from *TimeSeriesOne.*

Constructors:

WindSpeedTimeSeries ( );	Default constructor. Initialises *uniRand* object as internal random number generator.

Data elements:
protected:

r	double r;	autocorrelation coefficient
sigma	double sigma;	wind speed standard deviation

Member functions:
protected:

getRandomNumber	double getRandomNumber ( ); returns next random number from the implied random number generator.	
getOutput	double getOutput ( );	see *TimeSeries::getOutput*

public:

setUp	int setUp (TStatusWindow*, Param*); Parameter setting	
update	void update ( );	see *TimeSeries::update*
setCorrelation	void setCorrelation (double r);	set correlation coefficient

7.4 Global Functions

This section discusses all global functions. They are listed in alphabetical order within the source files they are in.

<cstring.cpp> String Functions

---

catDayName <cstring.h>

Function: Concatenate full day name
Syntax: void catDayName (char* buffer, int day);

Purpose: Upon *day* the function concatenate the full day name ("Monday", ...) .*day* = 0 points to "Sunday"

catDbl <cstring.h>

Function: Concatenate double number into string
Syntax: void catDbl (char* buffer, double x);
void catDbl (char* buffer, double x, int width);
Remark: see copyDbl ( );

catDMY <cstring.h>

Function: Concatenate day, month and year
Syntax: void catDMY (char* buffer, int dd, int mm, int yy);
Remark: see catDMY ( );

catEco <cstring.h>

Function: Concatenate double number in economics format
Syntax: void catEco (char* buffer, double x);
void catEco (char* buffer, double x, int width);
Remark: see copyEco ( );

catField <cstring.h>

Function: Concatenate a field to a string
Syntax: void catField (char* buffer, char* field, int width, int margin = RIGHT);
Remark: see copyField

catHMS <cstring.h>

Function: Concatenate hour, minute, second
Syntax: void catHMS (char* buffer, int hh, int mm, int ss);
Remark: see catHMS ( );

catInt <cstring.h>

Function: Concatenate integer number into string
Syntax: void catInt (char* buffer, int x);
void catInt (char* buffer, int x, int width);
Remark: see copyInt ( );

copyDbl <cstring.h>

Function: Convert a double number into a string
Syntax: void copyDbl (char* buffer, double x);
void copyDbl (char* buffer, double x, double width);

Purpose: x will be converted into a string. In the second version, *buffer* will have the length *width.*

copyDMY <cstring.h>

Function: Convert day, month and year into a string
Syntax: void copyDMY (char* buffer, int dd, int mm, int yy = -1);

Purpose: Format of *buffer* will be: 12.07.84 or 12.07.1984 (if yy > 0) or 12.07. (if yy < 0). *dd* is the day, *mm* the month (1 .. 12) and *yy* the year.

copyEco <cstring.h>

Function: Convert a double number into economics format
Syntax: void copyEco (char* buffer, double x);
void copyEco (char* buffer, double x, int width);

Purpose: x will be converted into an economics format like 2.356,75. In the second version, *buffer* will have the length *width.*

copyField <cstring.h>

Function: Copy a field into a string
Syntax: void copyField (char* buffer, char* field, int width, int margin = RIGHT);

Purpose: Copy *field* into *buffer* in a field of *width* bytes. The alignment will be either to the right margin (*margin* = RIGHT) or to the left (*margin* = LEFT)

copyHex <cstring.h>

Function: Copy hexadecimal umber into a string
Syntax: void copyHex (char* buffer, unsigned short x);

Purpose: x will be converted into a string of the form **0x0A1E**

copyHMS <cstring.h>

Function: Convert hour, minute and second into a string
Syntax: void copyHMS (char* buffer, int hh, int mm, int ss = 60);

Purpose: Format of *buffer* will be: 07:12:42 (if ss < 60) or 07:12 (if ss == 60). *hh* is the hour, *mm* the minute and *ss* the seconds.

copyInt <cstring.h>

Function: Convert an integer into a string
Syntax: void copyInt (char* buffer, int x);
void copyInt (char* buffer, int x, int width);

Purpose: x will be converted into a string. In the second version, *buffer* will have the length *width*.

decodeString <cstring.h>

Function: Decoding a string from a file
Syntax: void decodeString (char* aString);

Purpose: Removing special characters for 'New Line', 'Space' and 'NULL'.

getDbl <cstring.h>

Function: Convert a string into a double
Syntax: BOOL getDbl (char* buffer, double& x);

Purpose: The function returns the converted x as output.
Return: ERROR if a format error occurred, otherwise OK

getInt <cstring.h>

Function: Convert a string into an integer
Syntax: BOOL getInt (char* buffer, int& x);

Purpose: The function returns the converted x as output.
Return: ERROR if a format error occurred, otherwise OK

getMonthAndYear <cstring.h>

Function: Copy month and year into a string
Syntax: void getMonthAndYear (char* buffer, int month, int year);

Purpose: Copy int *buffer* "January 1994" depending on *month* and *year.*

getString <cstring.h>

Function: Get a string from a stream
Syntax: void getString (istream& instr, char* aString);

Purpose: Copy next string of instr into *aString* (until 'Space' of 'New Line') Special characters for 'New Line' and 'Space' will be removed in *aString*. So not the NULL- character (char NULLSTRING). If **aString* == NULLSTRING the actual string in the stream was NULL:

place <cstring.h>

Function: Insertion of a string into another
Syntax: void place (char* buffer, char* text, int row, int col);
void place (char* buffer, double x, int row, int col, int width);

Purpose: Insertion of *text* into *buffer* in row number *row*, starting at column number *col.* The routine will fill in '\n' and ' ' where necessary. The second version places a double number in a field of length *width.*

replace <cstring.h>

Function: replace a character in a string by another
Syntax: void replace (char* buffer, char a, char b);

Purpose: Bytes in *buffer* that are equal to *a* will be replaced by *b.*

splitDMY <cstring.h>

Function: Conversion of a string into day, month and year
Syntax: void splitDMY (char* buffer, int& day, int& month, int& year);

Purpose: Given *buffer* as input, the routine return *day*, *month* and *year* as output

strToLower <cstring.h>

Function: Convert string into lower case
Syntax: void strToLower (char* string);

strToUpper <cstring.h>

Function: Convert string into upper case
Syntax: void strToUpper (char* string);

<linalg.cpp> Linear Algebra

comp_inv <mathfunc.h>

Function: calculates the inverse matrix
Syntax: BOOL comp_inv (MATRIX& A);
Return: ERROR, if A singular; otherwise OK.

det <mathfunc.h>

Function: calculate the determinant of a matrix A
Syntax: double det (MATRIX& A);
Remark: Algorithm by [33], p. 49

lineqsol <mathfunc.h>

Function: Solve the linear matrix equation **Ax = b**
Syntax: BOOL lineqsol (MATRIX& A, VECTOR& x, VECTOR& b);
Return: ERROR, if equation cannot be solved. Otherwise OK.

luback <mathfunc.h>

Function: Back substitution
Syntax: BOOL ludecomp (MATRIX& A, IVECTOR& index, double* d);

Purpose: Successive calculation of the coefficients in the linear system. This function is used in *lineqsol.*
Return: ERROR, if matrix singular.
Remark: Algorithm by [33], p.47

ludecomp <mathfunc.h>

Function: L-U- decomposition of a matrix
Syntax: BOOL (MATRIX& A, IVECTOR& index, double* d);

Purpose: The given matrix A is replaced by its LU- decomposition. index is an output vector that record the row permutation effected by the partial pivoting. d is an output as ±1 depending on whether the number of row interchanges was even or

odd. The routine is used in combination with *luback* to solve linear equations.
Return: ERROR, if matrix singular.
Remark: Algorithm see [33], p.46

<mathfunc.cpp> Mathematical Functions

beta <mathfunc.h>

Function: evaluates the first derivative of the unnormalized, incomplete Beta- function
Syntax: double beta (double α, double β, double x);

Purpose: beta $(\alpha,\beta,x) = x^{\alpha-1}(1-x)^{\beta-1}$
Remark: Algorithm by [33], p. 226ff.

Beta <mathfunc.h>

Function: evaluates the Beta- function
Syntax: double Beta (double x, double y);
double Beta (double α, double β, double x);

Purpose: The first version calculates the Beta- function, $B(\alpha,\beta)$. The second calculates the normalized, incomplete Beta- function $I(\alpha,\beta,x)$
Remark: Algorithm by [33], p. 226ff.

bino <mathfunc.h>

Function: calculates the binomial coefficient $\binom{n}{k}$
Syntax: double bino (double n, double k);

Bnp <mathfunc.h>

Function: calculates the distribution function of the binomial distribution B(n,p) at point k
Syntax: double Bnp (double n, double p, double k);

Purpose: $Bnp(n,p,k) = \sum\binom{n}{j}\, p^j\, (1-p)^{n-j}$ (j= 0..k)
Remark: Algorithm by [33], p. 229

cot <mathfunc.h>

Function: berechnet cot(x)
Syntax: double cot(double x);

cube <mathfunc.h>

Function: cubic function x^3
Syntax: double cube (double x);

erf <mathfunc.h>

Function: calculates the error function erf(x)
Syntax: double erf (double x);
Remark: Algorithm by [33], p. 220

erfc <mathfunc.h>

Function: calculates the complementary error function erfc(x)
Syntax: double erfc (double x);
Remark: Algorithm by [33], p. 220

fact <mathfunc.h>

Function: calculate the faculty n!
Syntax: double fact (double n);
Remark: Algorithm by [33], p. 215

Gamma <mathfunc.h>

Function: calculate the Gamma function $\Gamma(x)$
Syntax: double Gamma (double x);
double Gamma (double a, double x);

Purpose: The first version calcualtes the Gamma function $\Gamma(x)$. The second calculates the normalised, incomplete Gamma function $\hat{\gamma}(\alpha,x) = \gamma(\alpha,x) / \Gamma(\alpha)$.
Remark: Algorithm by [33], p. 213ff

isinterval <mathfunc.h>

Function: Interval test
Syntax: int isinterval (double x, double a, double b);
int isinterval (int x, int a, int b);
Return: YES, if $x \in [a, b]$; else NO

lngamma <mathfunc.h>

Function: calculates the logarithm of the gamma function $\ln(\Gamma(x))$
Syntax: double lngamma (double x);

Purpose: This function is incorporated in the function *Gamma* to calculate the gamma

function.
Remark: Algorithm by [33], p.214

phi <mathfunc.h>

Function: calculates the first derivative of the normal distribution, $\partial_x[\phi((x-a)/\sigma^2)]$ with mean a and standard variation σ
Syntax: double phi (double x, double a, double var);
double phi (double x, double a, double σ^2, double x(0), double r);

Purpose: The first version calculates the function as stated above. The second version is the density function $f(X(t) \mid X(0) = x(0))$ of a conditional normal distribution with correlation coefficient r. (Equation 4.1)

PHI <mathfunc.h>

Function: Calculate the normal distribution
Syntax: double PHI (double x);
double PHI (double x, double a, double var);
double PHI (double x, double a, double σ^2, double x(0), double r);

Purpose: PHI (x) calculates the standard normal distribution. PHI (x, a, var) calculates the normal distribution with mean a and variance var. PHI $(x,a,\sigma^2,x(0),r)$ calculates the distribution function $F(x \mid X(0) = x(0))$ of a conditional distribution with correlation coefficient r. (Equation 4.2).
Remark: The function uses the functin *Gamma* (compare discussion of relationship between error function and Γ - function in [33], p.220)

SIGN <mathfunc.h>

Function: Signum- Function
Syntax: SIGN(x)
Return: -1 or 1

sqr <mathfunc.h>

Function: Square function x^2
Syntax: T sqr (T x);

SWAP <mathfunc.h>

Function: Swap two arguments
Syntax: void SWAP (double& a,double& b);
void SWAP (int& a, int& b);

exportData <plot.h>

Function: Data export to Word Perfect Presentation

Syntax: BOOL exportData (VECTOR& data, char* fileName, int mode);

Purpose: The components of vector *data* are written to file *fileName*. Depending on *mode* data are appended to the file (mode = ATTACH) or existing data in the file are overwritten with the new data (mode = NEW). If the file does not exist the mode NEW is assumed.

Return: Return value is ERROR if specified file could not be opened. Otherwise OK.

7.5 Listings

7.5.1 Header Files

7.5.1.1 <boolwin.h>

```
/*****************************************************************************/
/*** Module: BOOLWIN.H                                                    ***/
/***                                                                      ***/
/*** consists of basic type declarations and constants                    ***/
/*****************************************************************************/

#ifndef BOOLWIN_HEADER
#define BOOLWIN_HEADER

#include <windows.h>

/*** Definitions of constants ***********************************************/
#define  YES     1
#define  NO      0
#define  TRUE    1
#define  FALSE   0
#define  OK      1

#define LEFT     102
#define RIGHT    103

/*** Definitions of constants ***********************************************/
#define EPS       1.0e-5
#define EPS_G     1.0e-7
#define FPMIN     1.0e-30
#define FPMAX     1.0e+30
#define ITMAX     100
#define JMAX      20
#define TINY      1.0e-20

#endif

/*** End of BOOLWIN.H *******************************************************/
```

7.5.1.2 <cstring.h>

```
/*****************************************************************************/
/***                                                                      ***/
/*** Module: CSTRING.H                                                    ***/
/***                                                                      ***/
/*****************************************************************************/

#ifndef CSTRING_HEADER
#define CSTRING_HEADER

#ifndef BOOLWIN_HEADER
#include <boolwin.h>
#endif

#include <iostream.h>

/*** Global definitions *****************************************************/
#define MAXSTRING      90
#define MAXTEXT       200
#define normString     40
```

```
#define EMPTYSTRING   '▒' // Character No. 178
#define SPACE         '¥' // Character No. 157
#define NULLSTRING    '╣' // Character No. 185
#define NEW_LINE      '▄' // Character No. 220
#define NEWPAGE       '╫' // Character No. 215

/*** I/O functions    ****************************************************/
void printString      (ostream& outstr, char* aString              );
void printStringPlus  (ostream& outstr, char* aString              );
void decodeString     (char* aString                               );
void getString        (istream& instr, char* aString               );
/*** Lower case- Upper case routines *************************************/
void       strToUpper (char* buffer                                );
void       strToLower (char* buffer                                );
/*** Cat - routines ******************************************************/
void       catField   (char*, char*, int, int margin = RIGHT       );
void       catHex     (char* buffer, unsigned short                );
void       catInt     (char* buffer, int                           );
void       catInt     (char* buffer, int, int                      );
void       catDbl     (char* buffer, double                        );
void       catDbl     (char* buffer, double, int                   );
void       catEco     (char* buffer, double                        );
void       catEco     (char* buffer, double, int                   );
void       catDMY     (char* buffer, int dd, int mm, int yy = -1);
void       catHMS     (char* buffer, int hh, int mm, int ss = 60);
/*** Copy routines *******************************************************/
void       copyField  (char*, char*, int, int margin = RIGHT       );
void       copyHex    (char* buffer, unsigned short                );
void       copyInt    (char* buffer, int                           );
void       copyInt    (char* buffer, int, int                      );
void       copyDbl    (char* buffer, double                        );
void       copyDbl    (char* buffer, double, int                   );
void       copyEco    (char* buffer, double                        );
void       copyEco    (char* buffer, double, int                   );
void       copyDMY    (char* buffer, int dd, int mm, int yy = -1);
void       copyHMS    (char* buffer, int hh, int mm, int ss = 60);
/*** Place routines ******************************************************/
void       place      (char* buffer, char* text, int, int          );
void       place      (char* buffer, double,int,int,int            );
/*** Conversion routines *************************************************/
void       splitDMY   (char* buffer, int&, int&, int&              );
BOOL       getInt     (char* buffer, int&                          );
BOOL       getDbl     (char* buffer, double&                       );
/*** Replacement routine *************************************************/
void       replace    (char* buffer, char a, char b                );
/*** Calendar routines ***************************************************/
void catDayName      (char* buffer, int day                        );
void getMonthAndYear (char* buffer, int month, int year            );

#endif

/*** end of cstring.h ****************************************************/
```

7.5.1.3 <diffcalc.h>

```
/*************************************************************************/
/*** Module: DIFFCALC.H                                               ***/
/***                                                                  ***/
/*** consists of type and class definitions to differential and       ***/
/*** integral calculus of functions of one variable                   ***/
/*************************************************************************/

#ifndef DIFFCALC_HEADER
#define DIFFCALC_HEADER

#ifndef VECTORS_HEADER
#include <vectors.h>
```

```
#endif

#ifndef BOOLWIN_HEADER
#include <boolwin.h>
#endif

/*** Type declarations ******************************************************/
typedef enum {
              DETECT_EQUI,// determine search direction. Search at equidistant
points.
              DETECT_DYNA,// Search with dynamically increasing step width
              DOWN_EQUI,  // search along points smaller than x0
              DOWN_DYNA,
              UP_EQUI,    // search along points bigger than x0
              UP_DYNA,
} BRACKET_MODE;

typedef enum {
      POL_INT,         // polynomial approximation interpolation
      RAT_INT,         // rational function approximation interpolation
      SPLINE           // spline interpolation
} POL_MODE;

typedef enum {
      LIN,
      LOG
} REP_MODE;        // representation mode: linear/ log.

/*** structure to store function values *************************************/
class pairvec {
public :
      VECTOR         x;    // x- values
      VECTOR         y;    // y- values
      int            size; // number of values
      pairvec        (     ) {size=0;                          }
      pairvec        (int n) {size=n;x.create(n);y.create(n);}
      ~pairvec       (     ) { ;                               }
      void create    (int n) {size=n;x.create(n);y.create(n);}
      void move_down (void ) {x.move_down();y.move_down();    }
      void swap      (int i, int j) {x.swap(i,j);y.swap(i,j);}
      void move      (int i, int j) {x(i)=x(j) ; y(i)=y(j);   }
};

inline ostream& operator << (ostream& outstr, pairvec& v)
  { return outstr << v.size << ' ' << v.x << ' ' << v.y; }

inline istream& operator >> (istream& instr, pairvec& v)
  { return instr >> v.size >> v.x >> v.y; }

/****************************************************************************/
/*** Abstract class of an object function ***********************************/
/****************************************************************************/
class objfunc {
public :
      VECTOR   x;     // x- values
      VECTOR   y;     // y- values

      virtual double eval (double) = 0;  // abstract function
/*** Minimization and roots *************************************************/
      BOOL    bracketMin      (double&, double&, double&, double&,
double&, double&, int                  );
      double  goldenSection  (double, double, double, double ,
                               double ,double&);
/*** Determination of more than one function value *************************/
      void    compEquiVal (double, double, int                        );
};

/*** Object function with facilities for Turbo Vision Objects **************/
class owObjfunc : public objfunc {
  int      k;
  int      num;
  double   d;
```

```
  double  xmin;
public :
  double  getPercentage  (                      );
  void    prepForEquiVal (double, double, int);
  void    compEquiVal    (                      );
};

#endif
/*** End of diffcalc.h ****************************************************/
```

7.5.1.4 <distrib.h>

```
/**************************************************************************/
/***                                                                   ***/
/*** Module: DISTRIB.H                                                 ***/
/***                                                                   ***/
/*** Type Declarations for objects concerning distributions.          ***/
/**************************************************************************/

#ifndef DISTRIB_HEADER
#define DISTRIB_HEADER

#ifndef DIFFCALC_HEADER
#include <diffcalc.h>
#endif

#ifndef RANDOM_HEADER
#include <random.h>
#endif

#ifndef OWPARAM_HEADER
#include <owparam.h>
#endif

#define WM_MSGOBJFUNC   0x00
#define DENSITY_P       0x0000
#define DENSITY_X       0x0004
#define DISTRIBUTION    0x0001

class TStatusWindow;  // forward declaration

/**************************************************************************/
/*** Abstract class of a discrete distribution          ******************/
/**************************************************************************/

class DiscretDistribution {
protected :
  int   classes;
  int   initM;
public :
  DiscretDistribution             ( int n ) : classes (n) { ; }
  virtual ~DiscretDistribution ( ) { ; }
  virtual int     setUp ( TStatusWindow*, Param* ) = 0;
  virtual double gnm    ( int, int              ) = 0;
  virtual double Gn     ( int n ) { return 1;  }
  virtual void   setM   ( int m ) { initM = m; }
  virtual int    getN   ( double                ) = 0;
  int            getClasses ( ) { return classes; }
};

/**************************************************************************/
/*** Abstract class of a randomizer for discrete distributions *************/
/**************************************************************************/

class DiscretRandomizer : public uniRand {
protected :
  DiscretDistribution* distribution;
public :
```

```
  DiscretRandomizer ( ) : uniRand ( ) { ; }
  virtual ~DiscretRandomizer ( );
  virtual int setUp           ( TStatusWindow*, Param* ) = 0;
  void        setM            ( int                    );
  double      getRandomNumber (                        );
};

/***************************************************************************/
/*** Abstract class of a continuous distribution          ******************/
/***************************************************************************/

class ContinuousDistribution {
protected :
  double initVal;
public :
  ContinuousDistribution ( ) { ; }
  virtual ~ContinuousDistribution ( ) { ; }
  virtual int    setUp      ( TStatusWindow*, Param* ) = 0;
  virtual void   setInitVal ( double x               ) { initVal = x; }
  virtual double F          ( double                 ) = 0;
};

/***************************************************************************/
/*** class statfunc                             ***************************/
/***************************************************************************/

class statfunc : public owObjfunc {
  double  lastp;
  double  lastResult;
protected :
  int     type;   // = 1 : distribution , = 0 : density
  ContinuousDistribution* distribution;
public :
  statfunc                (                  );
  virtual        ~statfunc (                 );
  double         eval      (double           );
  virtual int    setUp     (TStatusWindow*, Param*);
  void           setType   (int aType        ) { type = aType; }
};

/***************************************************************************/
/*** class msgObjfunc                           ***************************/
/***************************************************************************/

class msgObjfunc : public objfunc {
  int  permitTime;
  int  permitValue;
  HWND handle;
public :
  msgObjfunc           (              ) : permitTime (0), permitValue (0) { ; }
  void enableTimeMsg   (              ) { permitTime = 1  ; }
  void enableValueMsg  (              ) { permitValue = 1 ; }
  void setHandle       (HWND aHandle) { handle = aHandle; }
  virtual double eval (double);
  virtual double Eval (double) = 0;
};

#endif

/*** End of distrib.h *****************************************************/
```

7.5.1.5 <error.h>

```
/***************************************************************************/
/*** MODUL : ERROR.H                                                    ***/
/***                                                                    ***/
/***************************************************************************/
```

```
#ifndef ERROR_HEADER
#define ERROR_HEADER

/*** Declarations of global functions ***************************************/
void error_message (const char far* message, const char far* modul);

#endif

/*** End of ERROR.H *********************************************************/
```

7.5.1.6 <joint.h>

```
/****************************************************************************/
/***                                                                     ***/
/*** Module: JOINT.H                                                     ***/
/***                                                                     ***/
/*** Header for joint power related objects                              ***/
/****************************************************************************/

#ifndef JOINT_HEADER
#define JOINT_HEADER

#ifndef DISTRIB_HEADER
#include <distrib.h>
#endif

#ifndef WIND_HEADER
#include <wind.h>
#endif

#ifndef SOLAR_HEADER
#include <solar.h>
#endif

/****************************************************************************/
/*** class ProbJointPower                                                ***/
/****************************************************************************/

class ProbJointPower : public owObjfunc {
  ContCondWindPower* windPower;
  ContCondSolApprox* solarPower;
  int num;
  VECTOR Gpw;
  VECTOR Gps;
  double gpw (int);
  double gps (int);
public :
  ProbJointPower           ( int n                 );
  virtual ~ProbJointPower  (                       );
  double  eval             ( double                );
  int     setUp            ( TStatusWindow*,Param* );
};

#endif

/*** End of joint.h *********************************************************/
```

7.5.1.7 <mathfunc.h>

```
/****************************************************************************/
/*** Module: MATHFUNC.H                                                  ***/
/***                                                                     ***/
/*** consists of definitions and prototypes for mathematical functions  ***/
/****************************************************************************/
```

```
#ifndef MATHFUNC_HEADER
#define MATHFUNC_HEADER

#ifndef VECTORS_HEADER
#include <vectors.h>
#endif

#ifndef BOOLWIN_HEADER
#include <boolwin.h>
#endif

/****************************************************************************/
/*** Utility functions                                                   ***/
/****************************************************************************/
void     SWAP     (double &a, double &b);
void     SWAP     (int    &a, int    &b);
BOOL     isinterval (double x, double a, double b);
BOOL     isinterval (int x   , int a   , int    b);

/****************************************************************************/
/*** Double precision library **********************************************/
/****************************************************************************/

/*** Mathematical functions ************************************************/
double    Beta     (double alpha, double x  );              // Beta function
double    Beta     (double, double, double  );  // Incomplete Beta function
double    beta     (double, double, double  );  // First derivative
double    bino     (double,double);                // Binominal coefficient
double    Bnp      (double n,double p,double k);
                                                                            //
Cumulative Bin. distribution
double    cube     (double x);
double    erf      (double);                                        // erf(x)
double    erfc     (double);                                       // erfc(x)
double    fact     (double);                                     // factorial
double    Gamma    (double);                                // gamma function
double    Gamma    (double, double        );    // Incomplete gamma function
double    max      (double, double);
double    min      (double, double);
double    phi      (double, double, double);
double    phi      (double, double, double, double, double); // cond. phi(x)
double    PHI      (double);                                        // PHI(x)
double    PHI      (double, double, double);                        // phi(x)
double    PHI      (double, double, double, double, double); // cond. Phi(x)
double    probks   (double);   // Kolmogorov- Smirnov probability function
double    SIGN     (double x);
double    sqr      (double x);

/*** Linear algebra ********************************************************/
BOOL     comp_inv (MATRIX&);                                      // Inverse
double   det      (MATRIX&);                                  //determinant
BOOL     lineqsol (MATRIX, VECTOR&, VECTOR );     //Linear equation solver
BOOL     ludecomp (MATRIX&, IVECTOR&, double*);        //LU- decomposition
void     luback   (MATRIX&, IVECTOR&, double*);          //Backsubstitution

#endif

/*** End of MATHFUNC.CPP ***************************************************/
```

7.5.1.8 <owcalc.h>

```
/****************************************************************************/
/*** Module: OWCALC.H                                                     ***/
/****************************************************************************/

/****************************************************************************/
/*** Object Windows C++: Calculations in the Windows inherited from      ***/
/*** either TStatusWindow or TMuliValObject                               ***/
/****************************************************************************/
```

```
#ifndef OWCALC_HEADER
#define OWCALC_HEADER

#ifndef OWSTAT_HEADER
#include <owstat.h>
#endif

#ifndef WIND_HEADER
#include <wind.h>
#endif

#ifndef SOLAR_HEADER
#include <solar.h>
#endif

#ifndef JOINT_HEADER
#include <joint.h>
#endif

#ifndef RANDOM_HEADER
#include <random.h>
#endif

#ifndef SERIES_HEADER
#include <series.h>
#endif

#ifndef PASSAGE_HEADER
#include <passage.h>
#endif

/*** TWindSpeedObject *****************************************************/
_CLASSDEF (TWindSpeedObject)
class TWindSpeedObject : public TMultiValObject {
private :
  SpeedDens* f;
  SpeedDist* F;
  int        workOutBasic    ( );
  int        workOutValues   ( );
  int        areParameterOK  ( );
  void       setOldParameter ( );
public :
  TWindSpeedObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TWindSpeedObject ( );
};

/*** TDistributionObject **************************************************/
_CLASSDEF (TDistributionObject)
class TDistributionObject : public TMultiValObject {
private :
  statfunc*   distribution;
  int         workOutBasic    ( );
  int         workOutValues   ( );
  int         areParameterOK  ( );
  void        setOldParameter ( );
public :
  TDistributionObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TDistributionObject ( );
};

/*** TJointDistributionObject *********************************************/
_CLASSDEF (TJointDistributionObject)
class TJointDistributionObject : public TMultiValObject {
private :
  ProbJointPower* jointPower;
  int         workOutBasic    ( );
  int         workOutValues   ( );
  int         areParameterOK  ( );
  void        setOldParameter ( );
public :
  TJointDistributionObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TJointDistributionObject ( );
```

```
};

/*** TRandomObject ********************************************************/

_CLASSDEF (TRandomObject)
class TRandomObject : public TStatusWindow {
  KgSTest* kgSTest;
  double test;
protected :
  int  workOut   ( );
  void writeRepl ( );
public :
  TRandomObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TRandomObject ( );
};

/*** TTimeSeriesObject ****************************************************/

_CLASSDEF (TTimeSeriesObject)
class TTimeSeriesObject : public TMultiValObject {
private :
  TimeSeries* timeSeries;
  int         workOutBasic    ( );
  int         workOutValues   ( );
  int         areParameterOK  ( );
  void        setOldParameter ( );
public :
  TTimeSeriesObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TTimeSeriesObject ( );
};

/*** TPassageTimeObject   *************************************************/

_CLASSDEF (TPassageTimeObject)
class TPassageTimeObject : public TStatusWindow {
  PassageTime* passageTime;
  double          time;
protected :
  int  workOut   ( );
  void writeRepl ( );
public :
  TPassageTimeObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TPassageTimeObject ( );
};

/*** PassageTimesObject ***************************************************/

_CLASSDEF (PassageTimesObject)
class PassageTimesObject : public TMultiValObject {
private :
  PassageTimes* passageTimes;
  int         workOutBasic    ( );
  int         workOutValues   ( );
  int         areParameterOK  ( );
  void        setOldParameter ( );
public :
  PassageTimesObject (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~PassageTimesObject ( );
};

#endif

/*** end of owcalc.h ******************************************************/
```

7.5.1.9 <owdialg.h>

```
/**************************************************************************/
/*** Module: OWDIALG.H                                                 ***/
/**************************************************************************/
```

```
/*************************************************************************/
/*** Header for <owdialg.cpp> defines the dialog windows objects for  ***/
/*** this programme. All dialog windows relate to Object Windows C++  ***/
/***                                                                  ***/
/***                                                                  ***/
/***   class TTransSettingsDlg       Settings Dialog:    Data         ***/
/***   class TSettingsDialog                             Window       ***/
/***                                                                  ***/
/***   class TTransSpeedDlg          Wind speed Dialog:  Data         ***/
/***   class TSpeedDialog                                Window       ***/
/***                                                                  ***/
/***   class TTransDirDlg            Directories Dialog: Data         ***/
/***   class TDirDialog                                  Window       ***/
/***                                                                  ***/
/***   class TTransWindDlg           Wind Power Dialog:  Data         ***/
/***   class TWindDialog                                 Window       ***/
/***                                                                  ***/
/***   class TTransExportDlg         Export Data Dialog: Data         ***/
/***   class TExportDialog                               Window       ***/
/***                                                                  ***/
/***   class TTransDisplayDlg        Display Options:    Data         ***/
/***   class TDisplayDialog                              Window       ***/
/***                                                                  ***/
/***   class TTransSolarDlg          Solar Power Dialog: Data         ***/
/***   class TSolarDialog                                Window       ***/
/***                                                                  ***/
/***   class TTransJointDlg          Joint Renewables:   Data         ***/
/***   class TJointDialog                                Window       ***/
/***                                                                  ***/
/***   class TTransRandDlg           Random Numbers:     Data         ***/
/***   class TRandDialog                                 Window       ***/
/***                                                                  ***/
/***   class TTransMathsDlg          Maths               Data         ***/
/***   class TMathsDialog                                Window       ***/
/***                                                                  ***/
/***   class TTransTsDlg             Time series:        Data         ***/
/***   class TTsDialog                                   Window       ***/
/***                                                                  ***/
/***   class TTransFpDlg             First passage time: Data         ***/
/***   class TFpDialog                                   Window       ***/
/***                                                                  ***/
/*************************************************************************/

#ifndef OWDIALG_HEADER
#define OWDIALG_HEADER

#ifndef OWRES_HEADER
#include "owres.h"
#endif

#ifndef OWLAPPL_HEADER
#include "owlappl.h"
#endif

#include <owl.h>
#include <dialog.h>
#include <iostream.h>
#include <edit.h>
#include <string.h>
#include <radiobut.h>

/*** Settings Window ***************************************************/
_CLASSDEF (TSettingsDialog)
class TSettingsDialog : public TDialog {
public :
  TSettingsDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransSettingsDlg {
public :
  TTransSettingsDlg ( );
  double wiVci;     // cut-in wind speed
```

```
  double wiVco;     // cut-out wind speed
  double wiVr;      // rated wind speed
  double wiVmean;   // mean wind speed
  double wiSigma;   // standard variation of wind turbulence
  double wiBeta;    // autocorrelation coefficient (wind)
  double solK0;     // maximum clearness index
  double solK;      // average hourly clearness index
  double solSigmaK;// standard deviation of clearness index
  double solBeta;   // autocorrealtion coefficient (solar)
  double comZeta;   // fractional power factor
  double batK;      // Battery parameters
  double batC;
  double batQMax;
  double batV;
  double sysPRen;      // Nominal renewable energy
  double sysPDemand;   // Power Demand

  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransSettingsDlg&);
  friend istream& operator >> (istream&, TTransSettingsDlg&);
};

/*** Wind Speed Dialog ***************************************************/
_CLASSDEF (TSpeedDialog)
class TSpeedDialog : public TDialog {
public :
  TSpeedDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransSpeedDlg {
public :
  TTransSpeedDlg ( );
  WORD   opProb;  // Flag: probability density function
  WORD   opDist;  // Flag: Distribution function
  double vmean;   // mean wind speed
  double vmin;    // minimum wind speed (for graph)
  double vmax;    // maximum wind speed (for graph)
  int    eval;    // number of function evaluations
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransSpeedDlg&);
  friend istream& operator >> (istream&, TTransSpeedDlg&);
};

/*** Directories Dialog **************************************************/
_CLASSDEF (TDirDialog)
class TDirDialog : public TDialog {
public :
  TDirDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransDirDlg {
public :
  TTransDirDlg ( );
  char  solFile[50];  // file name for solar data
  char  dlgFile[50];  // file name for dialog data
  friend ostream& operator << (ostream&, TTransDirDlg&);
  friend istream& operator >> (istream&, TTransDirDlg&);
};

/*** Wind Power Dialog ***************************************************/
_CLASSDEF (TWindDialog)
class TWindDialog : public TDialog {
  PTStatic        textTau;
  PTStatic        textInitialv;
  PTDoubleInputI  inTau;
  PTDoubleInputI  inInitialv;
  PTRadioButton   radioCond;
  char            bufTau[30];
  char            bufInitialv[30];
public :
  TWindDialog (PTWindowsObject AParent, LPSTR ATitle);
  virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
```

```
  virtual void HandleCondMsg (RTMessage) = [ID_FIRST + idOpCond];
  virtual void HandleStatMsg (RTMessage) = [ID_FIRST + idOpStationary];
};

class TTransWindDlg {
public :
  TTransWindDlg ( );
  WORD   opProb;          // Flag: probability density function
  WORD   opDist;          // Flag: distribution function
  WORD   opStationary;    // Flag: stationary process
  WORD   opCond;          // Flag: Conditional function
  double vmean;           // mean wind speed
  int    eval;            // number of evaluations
  double tau;             // time tau
  double initialv;        // initial wind speed
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransWindDlg&);
  friend istream& operator >> (istream&, TTransWindDlg&);
};

/*** Export Dialog ******************************************************/
_CLASSDEF (TExportDialog)
class TExportDialog : public TDialog {
public :
  TExportDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransExportDlg {
public :
  TTransExportDlg ( );
  WORD   opNew;           // Flag: new file
  WORD   opAttach;        // Flag: attach to existing file
  char   expFile[50];     // File name: Export file
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransExportDlg&);
  friend istream& operator >> (istream&, TTransExportDlg&);
};

/*** Display Dialog *****************************************************/
_CLASSDEF (TDisplayDialog)
class TDisplayDialog : public TDialog {
public :
  TDisplayDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransDisplayDlg {
public :
  TTransDisplayDlg ( );
  WORD   opAuto;          // Flag: Auto display of graphics
  WORD   opAccu;          // Flag: Accumulating data series
  WORD   opLegend;
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransDisplayDlg&);
  friend istream& operator >> (istream&, TTransDisplayDlg&);
};

/*** Solar Dialog *******************************************************/
_CLASSDEF (TSolarDialog)
class TSolarDialog : public TDialog {
  PTDoubleInputI  inTau;
  PTDoubleInputI  inInitialk;
  PTIntegerInputI inTrial;
  PTIntegerInputI inCoeff;
  PTCheckBox      checkBypass;
  PTStatic        textTau;
  PTStatic        textInitialk;
  PTStatic        textTrial;
  PTStatic        textCoeff;
  PTRadioButton   radioAnalyt;
  PTRadioButton   radioApprox;
  PTRadioButton   radioCond;
  PTRadioButton   radioQual;
```

```
  char           bufTau[30];
  char           bufInitialk[30];
  char           bufTrial[30];
  char           bufCoeff[30];
  void enableApprox  ( );
  void disableApprox ( );
  void enableCond    ( );
  void disableCond   ( );
protected :
  virtual void WMInitDialog    (RTMessage) = [WM_FIRST + WM_INITDIALOG];
  virtual void HandleAnalytMsg (RTMessage) = [ID_FIRST + idOpAnalyt    ];
  virtual void HandleApproxMsg (RTMessage) = [ID_FIRST + idOpApprox    ];
  virtual void HandleCondMsg   (RTMessage) = [ID_FIRST + idOpCond      ];
  virtual void HandleQualMsg   (RTMessage) = [ID_FIRST + idOpQual      ];
public :
  TSolarDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransSolarDlg {
public :
  TTransSolarDlg ( );
  WORD   opProb;         // Flag: probability density function
  WORD   opDist;         // Flag: Distribution function
  WORD   opAnalyt;       // Falg: Analytical solution
  WORD   opApprox;       // Flag: Approximation
  WORD   opCond;         // Flag: Conditional function
  WORD   opQual;         // Flag: Quality
  WORD   opBypass;       // Flag: Bypass
  double clearness;      // Clearness index
  double sigmaK;         // Standard variation of clearness index
  int    eval;           // number of function evaluations
  double tau;            // time tau
  double initialK;       // intitial clearness index
  int    trial;          // number of trial points
  int    coeff;          // number of coefficients in approximation
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransSolarDlg&);
  friend istream& operator >> (istream&, TTransSolarDlg&);
};

/*** Joint Renewable Dialog *********************************************/
_CLASSDEF (TJointDialog)
class TJointDialog : public TDialog {
  PTDoubleInputI inTau;
  PTDoubleInputI inInitialk;
  PTDoubleInputI inInitialv;
  PTStatic       textTau;
  PTStatic       textInitialk;
  PTStatic       textInitialv;
  char           bufTau[30];
  char           bufInitialv[30];
  char           bufInitialk[30];
  PTRadioButton  radioCond;
  void enableCond     ( );
  void disableCond    ( );
protected :
  virtual void WMInitDialog     (RTMessage) = [WM_FIRST + WM_INITDIALOG];
  virtual void HandleCondMsg    (RTMessage) = [ID_FIRST + idOpCond      ];
  virtual void HandleProbMsg    (RTMessage) = [ID_FIRST + idOpProbDens  ];
public :
  TJointDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransJointDlg {
public :
  TTransJointDlg ( );
  WORD   opJointDens;
  WORD   opJointCond;
  double vmean;
  double initialv;
  double clearness;
  double sigmaK;
```

```
  double initialK;
  double tau;
  int    eval;
  double zeta;
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransJointDlg&);
  friend istream& operator >> (istream&, TTransJointDlg&);
};

/*** Random Numbers Dialog ***********************************************/
_CLASSDEF (TRandDialog)
class TRandDialog : public TDialog {
  PTStatic          ranTextBeta;
  PTStatic           ranTextA;
  PTStatic           ranTextB;
  PTStatic           ranTextBi;
  PTStatic           ranTextP;
  PTStatic           ranTextClass;
  PTDoubleInputI     ranInputA;
  PTDoubleInputI     ranInputB;
  PTDoubleInputI     ranInputP;
  PTIntegerInputI    ranInputClass;
  PTRadioButton      ranRadio1;
  PTRadioButton      ranRadio2;
  PTRadioButton      ranRadio3;
  char               bufA[30];
  char               bufB[30];
  char               bufP[30];
  char               bufClass[30];
  void               HideBeta    ( );
  void               UnHideBeta  ( );
  void               HideBi      ( );
  void               UnHideBi    ( );
  void               HideClass   ( );
  void               UnHideClass( );
public :
  TRandDialog (PTWindowsObject AParent, LPSTR ATitle);
  virtual void WMInitDialog  (RTMessage) = [WM_FIRST + WM_INITDIALOG];
  virtual void HandleUniMsg  (RTMessage) = [ID_FIRST + idRanOp0];
  virtual void HandleNormMsg (RTMessage) = [ID_FIRST + idRanOp1];
  virtual void HandleBetaMsg (RTMessage) = [ID_FIRST + idRanOp2];
  virtual void HandleBiMsg   (RTMessage) = [ID_FIRST + idRanOp3];
};

class TTransRandDlg {
public :
  TTransRandDlg ( );
  WORD   ranOp0;          // Flag: Uniform distribution
  WORD   ranOp1;          // Flag: Normal distribution
  WORD   ranOp2;          // Flag: Beta distribution
  WORD   ranOp3;          // Flag: Binomial distribution
  double ranA;            // Input parameter: alpha
  double ranB;            // Input parameter: beta
  double ranP;            // Input parameter: p(binomial distr.)
  int    ranClass;        // Number of classes in chi▒ test
  int    ranTrial;        // Number of trials in chi▒ test
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransRandDlg&);
  friend istream& operator >> (istream&, TTransRandDlg&);
};

/*** Maths Dialog  ********************************************************/
_CLASSDEF (TMathsDialog)
class TMathsDialog : public TDialog {
public :
  TMathsDialog (PTWindowsObject AParent, LPSTR ATitle);
};

class TTransMathsDlg {
public :
  TTransMathsDlg ( );
  int    solTrial;
```

```
  int     solCoeff;
  int     fpTsTrial;
  int     fpTsMaxIt;
  double fpMcStopCrit;
  int     fpMcMaxIt;
  int     fpMcGrid;
  int     classes;
  void    setParameter ( );
  friend ostream& operator << (ostream&, TTransMathsDlg&);
  friend istream& operator >> (istream&, TTransMathsDlg&);
};

/*** Time Series Dialog    *********************************************/
_CLASSDEF (TTsDialog)
class TTsDialog : public TDialog {
  PTStatic         tsTextInitV;
  PTStatic         tsTextInitK;
  PTStatic         tsTextInitQ10;
  PTStatic         tsTextInitQ20;
  PTDoubleInputI tsInputInitV;
  PTDoubleInputI tsInputInitK;
  PTDoubleInputI tsInputInitQ10;
  PTDoubleInputI tsInputInitQ20;
  PTRadioButton   tsOp0;
  PTRadioButton   tsOp1;
  PTRadioButton   tsOp2;
  PTRadioButton   tsOp3;
  PTRadioButton   tsOp4;
  PTRadioButton   tsOp5;
  char             bufInitV[30];
  char             bufInitK[30];
  char             bufInitQ10[30];
  char             bufInitQ20[30];
  void             HideV     ( );
  void             UnHideV   ( );
  void             HideK     ( );
  void             UnHideK   ( );
  void             HideQ     ( );
  void             UnHideQ   ( );
public :
  TTsDialog (PTWindowsObject AParent, LPSTR ATitle);
  virtual void WMInitDialog  (RTMessage) = [WM_FIRST + WM_INITDIALOG];
  virtual void HandleOp0Msg  (RTMessage) = [ID_FIRST + idTsOp0];
  virtual void HandleOp1Msg  (RTMessage) = [ID_FIRST + idTsOp1];
  virtual void HandleOp2Msg  (RTMessage) = [ID_FIRST + idTsOp2];
  virtual void HandleOp3Msg  (RTMessage) = [ID_FIRST + idTsOp3];
  virtual void HandleOp4Msg  (RTMessage) = [ID_FIRST + idTsOp4];
  virtual void HandleOp5Msg  (RTMessage) = [ID_FIRST + idTsOp5];
};

class TTransTsDlg {
public :
  TTransTsDlg ( );
  WORD    tsOp0;        // Flag: Wind speed time series
  WORD    tsOp1;        // Flag: Wind power time series
  WORD    tsOp2;        // Flag: Solar power time series
  WORD    tsOp3;        // Flag: Combined power time series
  WORD    tsOp4;        // Flag: State of charge
  WORD    tsOp5;        // Flag: Power Deficit
  double tsTimeStep;
  int     tsPoints;
  double initV;
  double initK;
  double initQ10;
  double initQ20;
  void    setParameter ( );
  friend ostream& operator << (ostream&, TTransTsDlg&);
  friend istream& operator >> (istream&, TTransTsDlg&);
};

/*** First Passage Time Problems ******************************************/
_CLASSDEF (TFpDialog)
```

```
class TFpDialog : public TDialog {
  PTRadioButton    fpOp0;
  PTRadioButton    fpOp1;
  PTRadioButton    fpOp2;
  PTRadioButton    fpOp3;
  PTRadioButton    fpOp4;
  PTRadioButton    fpOp5;
  PTRadioButton    fpOp6;
  PTRadioButton    fpOp7;
  PTRadioButton    fpOp8;
  PTStatic         fpTextV0;
  PTStatic         fpTextK0;
  PTStatic         fpTextP0;
  PTStatic         fpTextPassV;
  PTStatic         fpTextPassK;
  PTStatic         fpTextPassP;
  PTStatic         fpTextNoVal;
  PTDoubleInputI   fpInputV0;
  PTDoubleInput    fpInputK0;
  PTDoubleInputI   fpInputP0;
  PTDoubleInputI   fpInputPassV;
  PTDoubleInput    fpInputPassK;
  PTDoubleInputI   fpInputPassP;
  PTIntegerInputI  fpInputNoVal;
  char             bufV0[30];
  char             bufK0[30];
  char             bufP0[30];
  char             bufPassV[30];
  char             bufPassK[30];
  char             bufPassP[30];
  char             bufNoVal[30];
  void             HideV0   ( );
  void             UnHideV0 ( );
  void             HideK0   ( );
  void             UnHideK0 ( );
  void             HideP0   ( );
  void             UnHideP0 ( );
  void             HidePassV   ( );
  void             UnHidePassV ( );
  void             HidePassK   ( );
  void             UnHidePassK ( );
  void             HidePassP   ( );
  void             UnHidePassP ( );
  void             HideNoVal   ( );
  void             UnHideNoVal ( );
public :
  TFpDialog (PTWindowsObject AParent, LPSTR ATitle);
  virtual void WMInitDialog  (RTMessage) = [WM_FIRST + WM_INITDIALOG];
  virtual void HandleOp0Msg  (RTMessage) = [ID_FIRST + idFpOp0];
  virtual void HandleOp1Msg  (RTMessage) = [ID_FIRST + idFpOp1];
  virtual void HandleOp2Msg  (RTMessage) = [ID_FIRST + idFpOp2];
  virtual void HandleOp3Msg  (RTMessage) = [ID_FIRST + idFpOp3];
  virtual void HandleOp4Msg  (RTMessage) = [ID_FIRST + idFpOp4];
  virtual void HandleOp5Msg  (RTMessage) = [ID_FIRST + idFpOp5];
  virtual void HandleOp6Msg  (RTMessage) = [ID_FIRST + idFpOp6];
  virtual void HandleOp7Msg  (RTMessage) = [ID_FIRST + idFpOp7];
  virtual void HandleOp8Msg  (RTMessage) = [ID_FIRST + idFpOp8];
};

class TTransFpDlg {
public :
  TTransFpDlg ( );
  WORD   fpOp0;         // Flag: Process: Wind speed
  WORD   fpOp1;         // Flag: Process: Wind Power
  WORD   fpOp2;         // Flag: Process: Solar Power
  WORD   fpOp3;         // Flag: Process: Combined Power
  WORD   fpOp4;         // Flag: Method: Time series approach
  WORD   fpOp5;         // Flag: Mehtod: Markov chain approach
  WORD   fpOp6;         // Flag: Calculation: only one value
  WORD   fpOp7;         // Flag:   as function of initial values
  WORD   fpOp8;         // Flag:   as function of passage levels
  double initV;         // Initial values
```

```
  double initK;
  double initP;
  double passV;         // Passage levels
  double passK;
  double passP;
  double timeStep;
  int    noVal;
  void   setParameter ( );
  friend ostream& operator << (ostream&, TTransFpDlg&);
  friend istream& operator >> (istream&, TTransFpDlg&);
};

#endif

/*** end of owdialg.h ****************************************************/
```

7.5.1.10 <owlappl.h>

```
/**************************************************************************/
/*** MODUL : OWLAPPL.H                                                 ***/
/***                                                                   ***/
/**************************************************************************/

#ifndef OWLAPPL_HEADER
#define OWLAPPL_HEADER

#include <windows.h>
#include <owl.h>
#include <edit.h>
#include <button.h>

/*** Constants ************************************************************/
#define idOwlApplText 100

/*** Double Input *********************************************************/
_CLASSDEF (TDoubleInput)
class TDoubleInput : public TEdit {
  BOOL validInput ( );
public :
  double x;
  TDoubleInput (PTWindowsObject AParent, int ResourceId);
  virtual WORD Transfer (void* DataPtr, WORD TransferFlag);
  virtual BOOL CanClose ( );
};

/*** Double Input in a specific interval **********************************/
_CLASSDEF (TDoubleInputI)
class TDoubleInputI : public TDoubleInput {
  double minVal;
  double maxVal;
  char   message[50];
public :
  TDoubleInputI (PTWindowsObject AParent, int ResourceId,
    const double aMinVal, const double aMaxVal, const char* aMessage);
  virtual BOOL CanClose ( );
};

/*** Integer Input ********************************************************/
_CLASSDEF (TIntegerInput)
class TIntegerInput : public TEdit {
  BOOL validInput ( );
public :
  int n;
  TIntegerInput (PTWindowsObject AParent, int ResourceId);
  virtual WORD Transfer (void* DataPtr, WORD TransferFlag);
  virtual BOOL CanClose ( );
};

/*** Integer Input in a specific interval *********************************/
```

```
_CLASSDEF (TIntegerInputI)
class TIntegerInputI : public TIntegerInput {
  int    minVal;
  int    maxVal;
  char   message[50];
public :
  TIntegerInputI (PTWindowsObject AParent, int ResourceId,
    const int aMinVal, const int aMaxVal, const char* aMessage);
  virtual BOOL CanClose ( );
};

/*** Message Dialog *******************************************************/
_CLASSDEF (TYoMessage)
class TYoMessage : public TDialog {
  PTStatic      statText;
  char          buffer[80];
public :
  TYoMessage (PTWindowsObject AParent, LPSTR ATitle,char*);
  virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
  virtual void CMYes        (RTMessage) = [ID_FIRST + IDYES];
  virtual void CMNo         (RTMessage) = [ID_FIRST + IDNO];
  virtual void CMIgnore     (RTMessage) = [ID_FIRST + IDIGNORE];
  virtual void CMAbort      (RTMessage) = [ID_FIRST + IDABORT ];
};

/*** InputDialog **********************************************************/
_CLASSDEF (TYoInput)
class TYoInput : public TDialog {
  PTEdit        inputLine;
  PTStatic      statText;
public :
  TYoInput (PTWindowsObject AParent, LPSTR ATitle,
       char* title,char* input);
  char textBuffer[80];
  virtual void WMInitDialog (RTMessage) = [WM_FIRST+WM_INITDIALOG];
};

#endif

/*** End of OWLAPPL.H ****************************************************/
```

7.5.1.11 <owparam.h>

```
/************************************************************************/
/*** Module: OWPARAM.H                                               ***/
/************************************************************************/

/************************************************************************/
/*** Header for <owparam.h> defines the parameter structures that     ***/
/*** serve as interfaces between windows objects and mathematical     ***/
/*** objects.                                                         ***/
/***                                                                  ***/
/***   struct Param               Parameter                           ***/
/***   class  Graph               Graphic related data                ***/
/************************************************************************/

#ifndef OWPARAM_HEADER
#define OWPARAM_HEADER

#ifndef DIFFCALC_HEADER
#include <diffcalc.h>
#endif

#include <string.h>

/*** Parameter ************************************************************/
```

```
struct Param {
  double tau;              // time
  int    eval;             // number of function evaluations
  int    type;             // = 0 (distribution) , = 1 (density)
  int    distSelect;       // chosen distribution selection:
                           // = 0 : Wind turbine power
                           //   1 : Conditional wind turbine power
                           //   2 : Exact Solar
                           //   3 : Approximated solar
                           //   4 : Approximated solar, conditional
                           //   5 : Quality of approximation
  int    filter;           // filter of inspection windows
  int    classes;

  // Wind parameters:
  double wiVci;            // cut in speed
  double wiVco;            // cout out speed
  double wiVr;             // rated speed
  double wiVmean;          // mean speed
  double wiVmin;           // minimum wind speed
  double wiVmax;           // maximum wind speed
  double wiSigma;          // variance of wind speed fluctuations
  double wiBeta;           // wind autocorrelation coefficient
  double wiInitV;          // initial v

  // Solar parameters:
  double solK;             // average hourly clearness index
  double solSigmaK;        // standard deviation of solar irradiation
  double solK0;            // absolute maximum possible clearness index
  double solInitK;         // initial k
  double solBeta;          // solar autocorrelation coefficient bsol
  int    solTrial;         // number of trial points in normal approximation
  int    solCoeff;         // number of coefficients in normal approximation
  int    solBypass;        // bypass of major calculations by retrieving old data

  // Combined renewables parameters:
  double comZeta;          // fractional power factor
  double comInitP;         // Initial p value

  // Random numbers dialog:
  double ranA;           // Parameter alpha for beta- distribution
  double ranB;           // Parameter beta for beta- distribution
  double ranP;           // Parameter p for binomial distribution
  double ranU;           // Parameter u for normal distribution
  int    ranClass;       // Number of classes for Chi▒ test
  int    ranTrial;       // Number of trials in Chi▒ test
  int    ranSelect;      // Last selection

  // Time series parameters:
  double tsTimeStep;     // Duration of a single time step
  int    tsPoints;       // Length of a time series
  int    tsSelect;

  // First passage time parameters:
  int    fpTsTrial;      // Number of time series taken into account
  int    fpTsMaxIt;      // Max iterations in Time series mode
  double fpMcStopCrit;// Stopping criterion in Markov chain mode
  int    fpMcMaxIt;      // Max iterations in Markov chain mode
  int    fpMcGrid;       // Markov chain mode: Grid Number Q
  double fpPassV;        // Passage level: Wind speed v
  double fpPassK;        //                Clearness index k
  double fpPassP;        //                Power level p
  int    fpNoVal;        // Number of values to be calculated in
                         // function-as-mode
  int    fpSelectProcess;   // Flags
  int    fpSelectMethod;
  int    fpSelectCalc;
  // Battery parameters
  double batK;
  double batC;
  double batQMax;
  double batV;           // Voltage
```

```
  double batQ10;
  double batQ20;       // Q10 + Q20 <= 1.0
  // Denormalized system
  double sysPDemand;
  double sysPRen;
  // Display options
  int    disAuto;      // automatic re-drawing of graphics
  int    disAccu;      // accumulate data series when possible
  int    disOldEval;   // last eval
  int    disOldType;   // last window type
  double disOldVmin;   // last minimum speed
  double disOldVmax;   // last maximum speed
  int    disFirstCurve; // = 1 if first curve, otherwise 0
  int    disLegend;    // = 1 if legend desired, otherwise 0
};

/******************************************************************************/
/*** class Graph                                                            ***/
/******************************************************************************/

/*** Graphic related data **************************************************/

class Graph {
public :
  VECTOR    x;           // x - values
  VECTOR    y[4];        // y - values
  char      legend[4][20];  // Legend for curves
  double    scale;       // scaling factor
  int       option;
  int       curveNo;     // number of curves in same graph
  double    min;            // minimum on x- axis
  double    max;            // maximum on x- axis
  char      headline[40];
  char      subline [50];
  char      axtext  [40];

  Graph  ( ) : curveNo(4) { ; }

  void      setHeadline (char* text) { strncpy (headline,text,39); }
  void      setSubline  (char* text) { strncpy (subline ,text,49); }
  void      setAxtext   (char* text) { strncpy (axtext  ,text,39); }
};

#endif

/*** ene of owparam.h ******************************************************/
```

7.5.1.12 <owplot.h>

```
/******************************************************************************/
/*** MODUL : OWPLOT.H                                                       ***/
/***                                                                        ***/
/******************************************************************************/

#ifndef OWPLOT_HEADER
#define OWPLOT_HEADER


#ifndef VECTORS_HEADER
#include <vectors.h>
#endif

#ifndef DIFFCALC_HEADER
#include <diffcalc.h>
#endif

#include <fstream.h>
#include <windows.h>
#include <owl.h>
```

```
/*** Constants **********************************************************/
#define TOP        100
#define BOTTOM     101
#define NEW        200
#define ATTACH     201

#define LIN        0
#define LOG        1

#define PIXEL      0
#define POLYGON    1
#define STEP       2
#define DIRAC      3

#define IN_AXLE      0
#define OUT_AXLE     1
#define CENTER_AXLE 2

#define TO_HORIZONTAL 0
#define TO_VERTICAL   1

typedef unsigned int DRA_MODE;
typedef unsigned int AXLE_MODE;

class axis;

/*** class TGraph ********************************************************/
_CLASSDEF (TGraph)
class TGraph : public TWindow {
protected :
  LOGFONT  logFont;     // Font
  HFONT    TheFont;
  HFONT    oldFont;
  LOGPEN   logPen;      // Pen
  HPEN     ThePen;
  HPEN     oldPen;
  LOGBRUSH logBrush;    // Brush
  HBRUSH   TheBrush;
  HBRUSH   oldBrush;
  COLORREF backGround; // Background Color
  HDC      DC;
public :
        TGraph  (PTWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);
  void          clearScreen   (              );
  void          setTextHeight (int n         );
  void          setPenSize    (int n         );
  void          setPenStyle   (int n         );
  void          setPenColor   (COLORREF color);
  void          setBrushStyle (int n         );
  void          setBrushColor (COLORREF color);
  void          setBrushHatch (int n         );
  void          setColor      (COLORREF color);
  virtual  void open          (              );
  virtual  void close         (              );
  void     Line          (int x1, int y1, int x2, int y2       );
  void     DoubleOut     (double number, int dec, int x, int y);
  void     IntegerOut    (int    number, int x, int y         );
  void     TextOut       (char* text, int x, int y            );
};

/*** class TPlot *********************************************************/
_CLASSDEF (TPlot)
class TPlot : public TGraph {
  char    headLine[50];
  char    subLine[60];
  int     curveNo;                          // curve number
  double  xquotlin, yquotlin, xquotlog, yquotlog;
  int     xlog, ylog;
  double  x_min,x_max,y_min,y_max;
  axis*   xBottom;
  axis*   xtop;
  axis*   yleft;
```

```
  axis*     yright;
  RECT      maxRect;
protected :
  RECT      curRect;
public :
  TPlot     (PTWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);
  ~TPlot    ( );
  virtual  void plot  ( ) { ; }
  virtual  void draw  ( );
  virtual  void Paint (HDC PaintDC, PAINTSTRUCT _FAR& PaintInfo);
  void     setHeadLine (const char*);
  void     setSubLine  (const char*);
  void     plotFactor  (double factor);
protected :
  void     plotHeadLine ( );
  void     plotSubLine  ( );
  void     drawMargin   ( );
  int      xcoord           (double x);
  int      ycoord           (double x);
  void     setCoordinates   (double xmini,double xmaxi,double ymini,
                                  double ymaxi);
  double   setAutoCoord     (double xmini, double xmaxi, VECTOR* yval, int n=0);
  void     setAutoAxAttr    (double&,double&,int&,int&,double&,double&);
  void     setViewport      (int,int,int,int);
  void     drawUpperX       (double mini,double maxi,double axle,int num,
                                  int log, const char* text,int axle_mode);
  void     drawRightY       (double mini,double maxi,double axle,int num,
                                  int log, const char* text,int axle_mode);
  void     drawLowerX       (double mini,double maxi,double axle,int num,
             int log, int grid, double dist,const char* text,int axle_mode);
  void     drawLeftY        (double mini,double maxi,double axle,int num,
                    int log, int grid, double dist,const char* text,int axle_mode);
  void     drawLinCoord  (double xaxle,int xnum,int xgrid,double xgriddist,
                    const char* xtext,double yaxle,int ynum,int ygrid,double
ygriddist, const char* ytext);
  void     drawLinCoord  (int, const char*, int, const char*);
  double   drawAutoLinCoord (double xmini, double xmaxi, VECTOR* yval,
       const char* xtext, const char* ytext, int xaxgrid, int yaxgrid,
       double scale, int n=0);
  void     drawCurve        (VECTOR&,VECTOR&, DRA_MODE draw_mode);
};

class axis { // Structure for description of an axis
       int    direction;// horizontal or vertical
       int    textjust; // text justification
       int    axle_mode;
       int    centercord;// central coordinate
       int    grid;       // Grid ?
       double min;        // Minimum
       double max;        // Maximum
       char   text[50];  // Axis text
       double axle;       // Distance between axles
       int    num;        // draw numbers every num-th axle
       double griddist;  // grid distance for linear representation
       int    linlog;     // linear or logarithmic representation
       void   logarith (int, int, const char*);
public:
  RECT*  curRect;
  HDC    DC;
  axis  (HDC aDC, RECT* aCurRect) { DC = aDC; curRect = aCurRect;}
       void   setAxis  (int dir, int just, int cord, double mini, double maxi,
                                                               const char*
alpha,double ax,int n,int axlog,int axgrid,
                                                               double dist, int
mode);
       void  drawAxis  (void);
};

/*** Function prototypes ******************************************************/
BOOL exportData    (VECTOR&, char*, int, char*, double);

#endif
```

```
/*** End of OWPLOT.H *******************************************************/
```

7.5.1.13 <owrenew.h>

```
/***************************************************************************/
/*** Module: OWRENEW.H                                                  ***/
/***************************************************************************/

/***************************************************************************/
/***                     Header of the main programme                   ***/
/***                                                                    ***/
/*** Definitions and declarations of:                                   ***/
/***      class TRenewPlot        (plot window)                         ***/
/***      class TMainWindow                                             ***/
/***      class TRenewApp         (application                          ***/
/***************************************************************************/

#ifndef OWRENEW_HEADER
#define OWRENEW_HEADER

#ifndef OWRES_HEADER
#include "owres.h"
#endif

#ifndef OWPLOT_HEADER
#include "owplot.h"
#endif

#ifndef OWLAPPL_HEADER
#include "owlappl.h"
#endif

#ifndef OWDIALG_HEADER
#include <owdialg.h>
#endif

#include <owl.h>
#include <dialog.h>
#include <iostream.h>
#include <edit.h>
#include <string.h>
#include <radiobut.h>

/*** Graphics Window ******************************************************/
_CLASSDEF (TRenewPlot)
class TRenewPlot : public TPlot {
  int delta;
  int start;
  int end;
  HBRUSH brushPen,oBrushPen;
public :
  int clear;
      TRenewPlot (PTWindowsObject AParent, LPSTR ATitle, PTModule AModule = NULL);
  virtual  void Paint (HDC PaintDC, PAINTSTRUCT _FAR& PaintInfo);
  void plot  (      );
};

/*** Main Window **********************************************************/
_CLASSDEF (TMainWindow)
class TMainWindow : public TWindow {
  void calc (owObjfunc*, double, double);
public :
  TTransSettingsDlg     TransSettingsDlg;
  TTransDirDlg          TransDirDlg;
  TTransExportDlg       TransExportDlg;
  TTransDisplayDlg      TransDisplayDlg;
  TTransSpeedDlg        TransSpeedDlg;
  TTransWindDlg         TransWindDlg;
  TTransSolarDlg        TransSolarDlg;
```

```
  TTransJointDlg         TransJointDlg;
  TTransRandDlg          TransRandDlg;
  TTransMathsDlg         TransMathsDlg;
  TTransTsDlg            TransTsDlg;
  TTransFpDlg            TransFpDlg;
  PTRenewPlot            testplot;
  TMainWindow            (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TMainWindow ( );
  virtual BOOL CanClose (                                 );
  virtual void CMWindSpeed (RTMessage) = [CM_FIRST + cmWindSpeed];
  virtual void CMSettings  (RTMessage) = [CM_FIRST + cmSettings ];
  virtual void CMMaths     (RTMessage) = [CM_FIRST + cmMaths];
  virtual void CMWindPower (RTMessage) = [CM_FIRST + cmWindPower];
  virtual void CMSolar     (RTMessage) = [CM_FIRST + cmSolar];
  virtual void CMRenewable (RTMessage) = [CM_FIRST + cmRenewable];
  virtual void CMExport    (RTMessage) = [CM_FIRST + cmExport];
  virtual void CMDisplay   (RTMessage) = [CM_FIRST + cmDisplay];
  virtual void CMHelp      (RTMessage) = [CM_FIRST + cmHelp];
  virtual void CMDir       (RTMessage) = [CM_FIRST + cmDirectories];
  virtual void CMRandom    (RTMessage) = [CM_FIRST + cmRandom];
  virtual void CMTimeSeries(RTMessage) = [CM_FIRST + cmTimeSeries];
  virtual void CMFpt       (RTMessage) = [CM_FIRST + cmFirstPassage];
  virtual void GetWindowClass (WNDCLASS& WndClass);
  friend ostream& operator << (ostream&, RTMainWindow);
  friend istream& operator >> (istream&, RTMainWindow);
};

/*** Application ********************************************************/
_CLASSDEF (TRenewApp)
class TRenewApp : public TApplication {
  int choice;
public:
  TRenewApp(LPSTR AName, HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
    : TApplication(AName, hInstance, hPrevInstance, lpCmdLine, nCmdShow),
      choice (0)
       { };
  virtual void InitMainWindow();
};

#endif

/*** ene of owrenew.h ***************************************************/
```

7.5.1.14 <owres.h>

```
/************************************************************************/
/*** Module: OWRES.H                                                 ***/
/************************************************************************/

#ifndef OWRES_HEADER

/*** Constants **********************************************************/
#define cmWindSpeed     500
#define cmSolar         501
#define cmRenewable     502
#define cmSettings      503
#define cmMaths         504
#define cmExport        505
#define cmWindPower     506
#define cmHelp          507
#define cmDirectories   510
#define cmRandom        512
#define cmTimeSeries    513
#define cmFirstPassage  514
#define cmDisplay       515

/*** ID- Constants ******************************************************/
```

```
// Parameter
#define idEval          100
#define idTau           101
#define idTextTau       102
#define idStatusText    103
#define idTimeText     104
#define idReportText    105
#define idClasses       106

// Dialogs
#define idOpProbDens    110
#define idOpDist        111
#define idOpAnalyt      112
#define idOpApprox      113
#define idOpCond        114
#define idOpQual        115
#define idOpStationary 116

// Wind parameter
#define idWiVci         120
#define idWiVco         121
#define idWiVr          122
#define idWiSigma       123
#define idWiVmean       124
#define idWiBeta        125
#define idWiVmin        126
#define idWiVmax        127
#define idWiTextInitV   128
#define idWiInitV       129

// Solar parameter
#define idSolK          140
#define idSolSigmaK     141
#define idSolK0         142
#define idSolBeta       144
#define idSolInitK      145
#define idSolTrial      146
#define idSolCoeff      147
#define idSolBypass     148
#define idSolTextInitK 149
#define idSolTextTrial 150
#define idSolTextCoeff 151

// Combined Renewables
#define idComZeta       160
#define idComP          161

// Export Dialog
#define idExpAttach     170
#define idExpNew        171
#define idExpFile       172

// Directories dialog
#define idDlgFile       180
#define idSolFile       181

// Random Numbers Dialog
#define idRanOp0        190
#define idRanOp1        191
#define idRanOp2        192
#define idRanOp3        193
#define idRanTextA      194
#define idRanTextB      195
#define idRanTextP      196
#define idRanTextBeta   197
#define idRanTextBi     198
#define idRanInputA     199
#define idRanInputB     200
#define idRanInputP     201
#define idRanTextClass 202
#define idRanTrial      203
#define idRanInputClass 205
```

```
// Time Series
#define idTsTimeStep   210
#define idTsPoints     211
#define idTsOp0        212
#define idTsOp1        213
#define idTsOp2        214
#define idTsOp3        215
#define idTsOp4        216
#define idTsOp5        217

// First passage times
#define idFpTsTrials   220
#define idFpTsMaxIt    221
#define idFpMcStopCrit 222
#define idFpMcMaxIt    223
#define idFpMcGrid     224
#define idFpNoVal      228
#define idFpOp0            230
#define idFpOp1            231
#define idFpOp2            232
#define idFpOp3            233
#define idFpOp4            234
#define idFpOp5            235
#define idFpOp6            236
#define idFpOp7            237
#define idFpOp8            238
#define idFpTextV0    240
#define idFpTextK0    241
#define idFpTextP0    242
#define idFpInputV0   243
#define idFpInputK0   244
#define idFpInputP0   245
#define idFpTextPassV      246
#define idFpTextPassK      247
#define idFpTextPassP      248
#define idFpInputPassV 249
#define idFpInputPassK 250
#define idFpInputPassP 251
#define idFpTextNoVal  252

// Display
#define idDisClear     260
#define idDisAccu      261
#define idDisLegend    262

// Battery
#define idBatK         270
#define idBatC         271
#define idBatQMax      272
#define idBatV         273
#define idBatQ10       274
#define idBatQ20       275

#define idBatTextQ10   276
#define idBatTextQ20   277

// System
#define idSysPDemand   280
#define idSysPRen      281

#endif

/*** end of owres.h ******************************************************/
```

7.5.1.15 <owstat.h>

```
/************************************************************************/
/*** Module: OWSTAT.H                                                ***/
```

```
/*************************************************************************/

/*************************************************************************/
/*** Object Windows C++: Calculations in the Status Window Environment ***/
/*************************************************************************/

#ifndef OWSTAT_HEADER
#define OWSTAT_HEADER

#ifndef DISTR_HEADER
#include <distrib.h>
#endif

#include <owl.h>
#include <dialog.h>
#include <edit.h>
#include <button.h>

/*** class TStatusWindow ************************************************/
_CLASSDEF (TStatusWindow)
class TStatusWindow : public TDialog {
private :
  PTStatic statusText1;
  PTStatic statusText2;
  PTStatic timeText;
  PTButton okButton;
  PTButton cancelButton;
  PTButton retryButton;
  double   lastTime;
  double   startTime;
  int      mode;
  void     startTimer ( );
  double   time       ( );
protected :
  int          giveWarning  (char*       );
  virtual void writeRep1    (            );
  virtual void writeRep2    (            ) { ; }
  virtual int  workOut      (            ) = 0;
  virtual void WMInitDialog (RTMessage   ) = [WM_FIRST+WM_INITDIALOG];
  virtual void Ok           (RTMessage   ) = [ID_FIRST+IDOK];
  virtual void Retry        (RTMessage   ) = [ID_FIRST+IDRETRY];
  virtual void TimeMsg      (RTMessage   ) = [WM_USER+WM_MSGOBJFUNC];
public :
  void          writeTime    (            );
  int           isEnoughTime (            );
  static double temp;
  void          writeStatus1     (char*               );
  void          writeStatus2     (char*               );

  TStatusWindow (PTWindowsObject AParent, LPSTR ATitle);
  virtual ~TStatusWindow ( );
};

/*** class TMultiValObject **********************************************/
_CLASSDEF (TMultiValObject)
class TMultiValObject : public TStatusWindow {
  int         eval;
  int         isAccuDesired (                         );
protected :
  virtual int  workOutBasic    (                         ) = 0;
  virtual int  workOutValues   (                         ) = 0;
  virtual int  areParameterOK  (                         ) = 0;
  virtual void setOldParameter (                         ) = 0;
  int          workOut         (                         );
  void         calcValues      (owObjfunc*,double,double);
public :
  TMultiValObject (PTWindowsObject AParent, LPSTR ATitle,int);
  virtual ~TMultiValObject ( ) { ; }
  static void  calc (owObjfunc*,double,double,int,TStatusWindow*);
};

#endif
```

```
/*** end of owstat.h ******************************************************/
```

7.5.1.16 <passage.h>

```
/***************************************************************************/
/***                                                                    ***/
/*** Module: PASSAGE.H                                                  ***/
/***                                                                    ***/
/*** Header for first passage time problems in the renewable energy    ***/
/*** project owrenew.prj                                                ***/
/***************************************************************************/

#ifndef PASSAGE_HEADER
#define PASSAGE_HEADER

#ifndef SERIES_HEADER
#include <series.h>
#endif

#ifndef VECTORS_HEADER
#include <vectors.h>
#endif

#ifndef OWPARAM_HEADER
#include <owparam.h>
#endif

class DiscretDistribution; // forward declaration

/***************************************************************************/
/*** Abstract class of a first passage time problem ***********************/
/***************************************************************************/

class PassageTime : public msgObjfunc {
protected :
  double passLevel;   // passage level (power / speed)
  double initLevel;   // initial level (power / speed)
  double timeStep;    // time step
  virtual int  SetUp        (TStatusWindow*, Param*) = 0;
public :
  PassageTime               (                       );
  virtual ~PassageTime      (                       ) { ; }
  int          setUp        (TStatusWindow*, Param*);
  void         setPassLevel (double newLevel        ) { passLevel=newLevel; }
  virtual void setInitLevel (void*                  ) = 0;
};

/***************************************************************************/
/*** Abstract class for first passage time - time series approach **********/
/***************************************************************************/

class TSPassageTime : public PassageTime {
private :
  int    repFactor;  // number of time series taken into account
  int    maxIt;      // maximum number of iterations
protected :
  TimeSeries* timeSeries;
  virtual int  SetUp        ( TStatusWindow*, Param* );
public :
  TSPassageTime             (                        );
  virtual ~TSPassageTime    (                        );
  double       Eval         ( double                 );
  void         setInitLevel ( void*                  );
};

/***************************************************************************/
/*** Wind speed passage time - time series approach                 ******/
/***************************************************************************/
```

```
class TSWindSpeedPassageTime : public TSPassageTime {
public :
  TSWindSpeedPassageTime (                          );
  int SetUp              ( TStatusWindow*, Param* );
};

/*****************************************************************************/
/*** Wind power passage time - time series approach                    ******/
/*****************************************************************************/

class TSWindPowerPassageTime : public TSPassageTime {
public :
  TSWindPowerPassageTime (                          );
  int SetUp              ( TStatusWindow*, Param* );
};

/*****************************************************************************/
/*** Solar power passage time - time series approach                   ******/
/*****************************************************************************/

class TSSolarPowerPassageTime : public TSPassageTime {
public :
  TSSolarPowerPassageTime (                          );
  int SetUp               ( TStatusWindow*, Param* );
};

/*****************************************************************************/
/*** Joint renewable passage time - time series approach               ******/
/*****************************************************************************/

class TSJointPowerPassageTime : public TSPassageTime {
public :
  TSJointPowerPassageTime (                          );
  int SetUp               ( TStatusWindow*, Param* );
};


/*****************************************************************************/
/*** Abstract class for Markov chain approach                          ******/
/*****************************************************************************/

class MCPassageTime : public PassageTime {
private :
  MATRIX G;          // Transition matrix
  VECTOR P;          // Probability vector
  double stopCrit;   // stopping criterion
  int    maxIt;      // maximum number of iterations
  int    discPassLevel; // discretized passage level
  int    discInitLevel; // discretized initial level
  void   updateG      (                             );
protected :
  int    classes;    // number of discretization levels
  DiscretDistribution* distribution;
  int    discretize          ( double                 );
public :
  MCPassageTime              (                        );
  double       Eval          ( double                 );
  virtual int  SetUp         ( TStatusWindow*, Param* );
  void         setInitLevel  ( void*                  );
};

/*****************************************************************************/
/*** class MCWindSpeedPassageTime                                      ******/
/*****************************************************************************/

class MCWindSpeedPassageTime : public MCPassageTime {
public :
  MCWindSpeedPassageTime ( ) : MCPassageTime ( ) { ; }
  int    SetUp           (TStatusWindow*, Param*);
};

/*****************************************************************************/
```

```
/*** class MCWindPowerPassageTime                                     ******/
/****************************************************************************/

class MCWindPowerPassageTime : public MCPassageTime {
public :
  MCWindPowerPassageTime (   ) : MCPassageTime ( ) { ; }
  int    SetUp            ( TStatusWindow*, Param* );
};

/****************************************************************************/
/*** class MCSolarPowerPassageTime                                    ******/
/****************************************************************************/

class MCSolarPowerPassageTime : public MCPassageTime {
public :
  MCSolarPowerPassageTime (   ) : MCPassageTime ( ) { ; }
  int    SetUp            ( TStatusWindow*, Param* );
};

/****************************************************************************/
/*** class MCJointPowerPassageTime                                    ******/
/****************************************************************************/

class MCJointPowerPassageTime : public MCPassageTime {
public :
  MCJointPowerPassageTime (   ) : MCPassageTime ( ) { ; }
  int    SetUp            ( TStatusWindow*, Param* );
};

/****************************************************************************/
/*** Abstract class of a first passage time problem ***********************/
/*** allowing to vary either the passage level or initial value.        ***/
/****************************************************************************/

class PassageTimes : public owObjfunc {
protected :
  int          selectCalc;
  int          noVal;     // number of values along the x- axis
  PassageTime* passageTime;
  virtual int  SetUp      ( TStatusWindow*, Param* ) = 0;
public :
  double minVal;
  double maxVal;
  PassageTimes          (                         );
  virtual ~PassageTimes (                         );
  int     setUp         ( TStatusWindow*, Param* );
  double  eval          ( double                  );
};

// Wind speed
class WindSpeedPassageTimes : public PassageTimes {
public :
  WindSpeedPassageTimes ( int                     );
  int SetUp             ( TStatusWindow*, Param* );
};

// Wind power
class WindPowerPassageTimes : public PassageTimes {
public :
  WindPowerPassageTimes (int                    );
  int SetUp             (TStatusWindow*, Param*);
};

// Solar power
class SolarPowerPassageTimes : public PassageTimes {
public :
  SolarPowerPassageTimes (int                    );
  int SetUp              (TStatusWindow*, Param*);
};

// Joint renewable power
class JointPowerPassageTimes : public PassageTimes {
```

```
public :
  JointPowerPassageTimes (int                    );
  int SetUp              (TStatusWindow*, Param*);
};

#endif

/*** End of passage.h ****************************************************/
```

7.5.1.17 <random.h>

```
/*************************************************************************/
/*** Module: RANDOM.H                                                 ***/
/***                                                                  ***/
/*** Definition of types and classes for random numbers               ***/
/*************************************************************************/

#ifndef RANDOM_HEADER
#define RANDOM_HEADER

#ifndef VECTORS_HEADER
#include <vectors.h>
#endif

#ifndef MATHFUNC_HEADER
#include <mathfunc.h>
#endif

/*** Constants **********************************************************/
#define NTAB      32

/*** Uniform deviates ***************************************************/
class uniRand {
public :
  uniRand                              (        ) { ; }
  virtual        ~uniRand              (        ) { ; }
  void           initialize            (        );
  virtual double getRandomNumber       (        );
  virtual void   update                ( void*  ) { ; }
};

/*** Gaussian deviates N(a, var) ****************************************/
class normRand : public uniRand {
  double  mean;
  double  sigma;
  int     iset;
  double  gset;
public :
  normRand                             (                    );
  normRand                             (double m, double s);
  virtual         ~normRand            (                    ) { ; }
  virtual double  getRandomNumber      (                    );
  void            update               ( void*              );
};

/*** Rejection method ***************************************************/
class rejectRand : public uniRand {
public :
  rejectRand                           (        );
  virtual double  getRandomNumber      (        );
protected :
  virtual double  compFunc             (double) = 0;
  virtual double  origFunc             (double) = 0;
  virtual double  invInteg             (double) = 0;
};

/*** Rejection method using a uniform distribution as comparison function */
/*** for distributions with non zero values in the interval [0,1]        */
class uniRejectRand : public rejectRand {
```

```
private :
  double ceiling;
public :
  uniRejectRand (           ) { ceiling = 1;   }
  uniRejectRand (double max) { ceiling = max; }
protected :
  virtual double  compFunc (double)   { return 1; }
  virtual double  invInteg (double y) { return y; }
};

/*** Beta distribution ****************************************************/
class betaRand : public uniRejectRand {
private :
  double  alpha,beta,fact;
protected :
  virtual double origFunc        (double          );
  virtual double compFunc        (double          );
public :
  betaRand                       (                );
  betaRand                       (double a, double b);
};

/*** Discrete distributions using the rejection method *******************/

class discretRand : public uniRand {
private :
  VECTOR* px;
  double  ceiling;
  double  getRandomNumber (       );
public :
  discretRand ( VECTOR* );
  void    update          ( void* );
};

/************************************************************************/
/*** Kolmogorov - Smirnov test ******************************************/
/************************************************************************/

/*** Abstract class for KGS test ****************************************/
class KgSTest {
protected :
  double   size;
  int      k;
  double   mean;
  double   var;
  VECTOR   x,y,r;
  uniRand* randomizer;
  virtual void   initialize   (          ) { ; }
  virtual double theoretProb  (double x) = 0;
  void           doValues     (          );
  double         maxDistance  (          );
  void           calcCumDist  (          );
public :
  double doTest       (            );
  double getMean      (            );
  double getVar       (            );
  KgSTest       (int n);
  ~KgSTest      (     );
};

/*** Kolmogorov- Smirnov Test for uniform distribution *******************/
class UniKgSTest : public KgSTest {
public :
  UniKgSTest         (int    n) : KgSTest (n) { ; }
  double theoretProb (double x) { return (x); }
  void   initialize  (          ) { randomizer = new uniRand ( ); }
};

/*** Kolmogorov- Smirnov Test for normal distribution ********************/
class NormKgSTest : public KgSTest {
public :
  NormKgSTest (int n) : KgSTest (n) { ; }
```

```
  double theoretProb (double x) { return (PHI (x) ); }
  void   initialize  (        )  { randomizer = new normRand ( ); }
};

/*** Kolmogorov- Smirnov Test for beta- distribution **********************/
class BetaKgSTest : public KgSTest {
  double alpha, beta;
  double classes;
public :
  BetaKgSTest (int n, int r, double a, double b);
  double theoretProb (double x);
  void   initialize  (      )  { randomizer = new betaRand (alpha,beta); }
};

#endif

/*** End of RANDOM.H ******************************************************/
```

7.5.1.18 <series.h>

```
/**************************************************************************/
/***                                                                   ***/
/*** Module: SERIES.H                                                  ***/
/***                                                                   ***/
/*** Header for time series objects within the renewable energy        ***/
/*** project owrenew.prj                                               ***/
/**************************************************************************/

#ifndef SERIES_HEADER
#define SERIES_HEADER

#ifndef VECTORS_HEADER
#include <vectors.h>
#endif

#ifndef DISTRIB_HEADER
#include <distrib.h>
#endif

#ifndef DIFFCALC_HEADER
#include <diffcalc.h>
#endif

#ifndef RANDOM_HEADER
#include <random.h>
#endif

#ifndef OWPARAM_HEADER
#include <owparam.h>
#endif

#ifndef SOLAR_HEADER
#include <solar.h>
#endif

/**************************************************************************/
/*** Abstract class of a time series *************************************/
/**************************************************************************/

class TimeSeries : public owObjfunc {
protected :
  virtual void   update       (                          ) = 0;
  virtual double getOutput    (                          ) = 0;
public :
  TimeSeries                  (                          ) { ; }
  virtual        ~TimeSeries  (                          ) { ; }
  virtual int    setUp        ( TStatusWindow*, Param*   ) = 0;
  virtual void   setUserInit  ( void*                    ) = 0;
};
```

```
class TimeSeriesOne : public TimeSeries {
protected :
  double         initUserVal;
  double         randomVal;
  double         outVal;
  virtual double getInitRandomVal ( ) { return initUserVal; }
  virtual double getRandomNumber  ( ) = 0;
public :
  TimeSeriesOne                 (                   );
  ~TimeSeriesOne                (                   );
  double        eval            ( double            );
  void       setUserInit        ( void*             );
};

/**************************************************************************/
/*** Class of wind speed time series ****************************************/
/**************************************************************************/

class WindSpeedTimeSeries : public TimeSeriesOne {
  double   vmean;      // mean wind speed
  double   effSigma;   // effective standard variation
  uniRand* randomizer;
protected :
  double   r;          // autocorrelation coefficient
  double   sigma;      // standard variation
  double getRandomNumber  ( ) { return (randomizer->getRandomNumber()); }
  double getOutput        (                          ) { return randomVal; }
public :
  int    setUp            ( TStatusWindow*, Param* );
  void   update           (                        );
  void   setCorrelation   ( double                 );
  WindSpeedTimeSeries     (                        );
  ~WindSpeedTimeSeries    (                        );
};

/**************************************************************************/
/*** Wind power time series                                           ****/
/**************************************************************************/

class WindPowerTimeSeries : public WindSpeedTimeSeries {
private :
  double vci,vco,vr;
public :
  WindPowerTimeSeries     (                        );
  static double getWindPower (double V,double Vci, double Vco, double Vr);
  static double getV         (double p,double Vci, double Vr);
  double getOutput        (                        );
  int    setUp            ( TStatusWindow*, Param* );
};

/**************************************************************************/
/*** Solar power time series                                          ****/
/**************************************************************************/

class SolarPowerTimeSeries : public TimeSeriesOne {
  double NMinusOne;
  double K0;
  SolarRandomizer* randomizer;
  double getRandomNumber  ( ) { return (randomizer->getRandomNumber()); }
public :
  SolarPowerTimeSeries    (                        );
  ~SolarPowerTimeSeries   (                        );
  int    setUp            ( TStatusWindow*, Param* );
  double getOutput        (                        );
  double getInitRandomVal (                        );
  void   update           (                        );
};

/**************************************************************************/
/*** Joint power time series                                          ****/
/**************************************************************************/
```

```
class JointPowerTimeSeries : public TimeSeries {
  SolarPowerTimeSeries* solarPowerTimeSeries;
  WindPowerTimeSeries*  windPowerTimeSeries;
  double                zeta;   // fractional power factor
public :
  JointPowerTimeSeries   (                         );
  ~JointPowerTimeSeries  (                         );
  void   update          (                         );
  double getOutput       (                         );
  double eval            ( double                  );
  int    setUp           ( TStatusWindow*, Param*  );
  void   setUserInit     ( void*                   );
};

/****************************************************************************/
/*** State of charge time series                                        ****/
/****************************************************************************/

class StateOfChargeTimeSeries : public TimeSeries {
  double batK,batC,batQMax,batV,batQ10,batQ20;
  double sysPRen,sysPDemand;
  double I,PNeed;
  double q1,q2;
  double kt;
  void   calcICharge     (                         );
  void   calcIDischarge  (                         );
protected :
  JointPowerTimeSeries* jointPowerTimeSeries;
  double deltaP;
  void   update          (                         );
  double getOutput       (                         );
public :
  StateOfChargeTimeSeries (                        );
  ~StateOfChargeTimeSeries(                        );
  double eval            ( double                  );
  int    setUp           ( TStatusWindow*, Param*  );
  void   setUserInit     ( void*                   );
};

/****************************************************************************/
/*** Power Deficit Time Series                                          ****/
/****************************************************************************/

class PowerDeficitTimeSeries : public StateOfChargeTimeSeries {
  double getOutput       (                         );
public :
  PowerDeficitTimeSeries (                         );
  double eval            ( double                  );
};

#endif

/*** End of series.h *******************************************************/
```

7.5.1.19 <solar.h>

```
/****************************************************************************/
/***                                                                     ***/
/*** Module: SOLAR.H                                                     ***/
/***                                                                     ***/
/*** Header for solar related objects                                    ***/
/****************************************************************************/

#ifndef SOLAR_HEADER
#define SOLAR_HEADER

#ifndef DISTRIB_HEADER
```

```
#include <distrib.h>
#endif

/*************************************************************************/
/*** Solar Power Constants                                            ***/
/*************************************************************************/

class SolConstants {
public :
  SolConstants ( );
  ~SolConstants( )  { ; }
  double w,deltaKK0,kminK0,deltaK,kmin,correl;
  VECTOR a,b;
  int  setUp (Param*          );
  void xTok  (double, double*);
  void kTox  (double, double*);
};

/*************************************************************************/
/*** The exact distribution (with beta functions)                     ***/
/*************************************************************************/

class ContSolExact : public ContinuousDistribution {
protected :
  SolConstants solC;
  double  Fx        ( double                   );
public :
  ContSolExact      (                          );
  ~ContSolExact     (                          ) { ; }
  double  F         ( double                   );
  int     setUp     ( TStatusWindow*, Param*   );
};

class ContSolExactX : public ContSolExact {
public :
  ContSolExactX     (                          );
  ~ContSolExactX    (                          ) { ; }
  double  F         ( double                   );
};

class ProbSolExact : public statfunc {
public :
  ProbSolExact ( );
};

/*************************************************************************/
/*** Object class for solar power with least square method ****************/
/*************************************************************************/

class MeritSol : public msgObjfunc {
  double alpha          ( int j, int k);
  double fp             ( double p     );  // density function in p = j /(N-1)
  double merit          (              );  // figure of merit
  int    solCoeff;
  int    solTrial;
  double solK;
  double solSigmaK;
  MATRIX AA;            // Coefficient matrix A
  MATRIX Alpha;         // Coefficient matrix
  VECTOR d;             // Coefficients of left side of normal equations
  VECTOR c;             // Coefficients of prob dens function
public :
  MeritSol (SolConstants*,Param*);
  ~MeritSol(              ) { ; }
  SolConstants* psc;
  double initialx;      // initial clearness index on x- scale
  VECTOR u;             // Coeff. vector of generating functions
  VECTOR sigma;         // Standard deviation vector
  VECTOR lambda;        // Standard deviation vector / epsilon
  VECTOR Fxm;           // vector with distribution function values
  double QPlusOne;      // number of generating functions used +1
  double MPlusOne;      // number of trial points + 1
```

```
  virtual double Eval          ( double x );
  double         fx            ( double x );  // density function in x
  double         Fx            ( double x );  // distribution function in x
  double         Fp            ( double p );  // distribution function in p
  double         FpApprox      ( double p );  // approx. dist function in p
  double         FxApprox      ( double x );  // approx. dist. function in x
  int            setUp         (          );

  friend ostream& operator << (ostream& outstr, MeritSol* v);
  friend istream& operator >> (istream& instr , MeritSol* v);
};

/****************************************************************************/
/*** Approximated Distribution                                           ***/
/****************************************************************************/

// Approximation of the solar distribution
class ContSolApprox : public ContinuousDistribution {
protected :
  MeritSol*    sol;
  SolConstants sc;
public :
  ContSolApprox          (                            );
  ~ContSolApprox         (                            );
  double F               ( double                     );
  int    setUp           ( TStatusWindow*, Param*     );
  void   setCorrelation  ( double time, double beta   );
  void   setInitVal      ( double initK               );
};

class ContSolApproxX : public ContSolApprox {
public :
  ContSolApproxX         (                            );
  ~ContSolApproxX        (                            ) { ; }
  double F               ( double                     );
};

class ProbSolApprox : public statfunc {
public :
  ProbSolApprox ( );
};

// Conditional distribution
class ContCondSolApprox : public ContSolApprox {
public :
  ContCondSolApprox ( ) : ContSolApprox ( ) { ; }
  int    setUp           ( TStatusWindow*, Param* );
};

class ProbCondSolApprox : public statfunc {
public :
  ProbCondSolApprox ( );
};

// Qualityfunction
class ContSolAppQual : public ContinuousDistribution {
  ContSolExact*   exact;
  ContSolApprox*  approx;
public :
  ContSolAppQual      (                        );
  ~ContSolAppQual     (                        );
  virtual double F    ( double                 );
  int    setUp        ( TStatusWindow*, Param* );
};

class ProbSolAppQual : public statfunc {
public :
  ProbSolAppQual ( );
};

/****************************************************************************/
/*** Discrete Distribution                                               ***/
```

```
/******************************************************************************/

class DiscSolApprox : public DiscretDistribution {
private :
  ContSolApprox* solApprox;
  double         K0;
  double         NMinusOne;
public :
  DiscSolApprox          ( int n                      );
  virtual ~DiscSolApprox (                            );
  int    setUp           ( TStatusWindow*, Param*     );
  double gnm             ( int, int                   );
  double Gn              ( int                        );
  void   setM            ( int                        );
  int    getN            ( double                     );
};

/******************************************************************************/
/*** Discrete Randomizer                                                    ***/
/******************************************************************************/

class SolarRandomizer : public DiscretRandomizer {
public :
  SolarRandomizer (                          );
  int  setUp      ( TStatusWindow*, Param* );
};

#endif

/*** End of solar.h ***********************************************************/
```

7.5.1.20 <vectors.h>

```
/******************************************************************************/
/*** Module: VECTORS.H                                                      ***/
/***                                                                        ***/
/*** consists of class definitions for vectors and arrays.                  ***/
/******************************************************************************/

#ifndef VECTORS_HEADER
#define VECTORS_HEADER

#include <iostream.h>
#include <complex.h>

/*** The general class of a linear chain *************************************/
template <class T> class CHAIN_ {
protected:
       T*  p;
       int size;
public:
       CHAIN_  (int n);
       CHAIN_  (void);
       CHAIN_  (CHAIN_& c);
       ~CHAIN_  () {if (size) delete p;}
       int minchainindex (void);
       int maxchainindex (void);
}

template <class T> class MATRIX_ ;

/*** The class of vectors *****************************************************/
template <class T> class VECTOR_ : public CHAIN_<T> {
public :
       int      dim;  // Dimension
       VECTOR_  (void)        : CHAIN_<T> ()               { dim = 0     ; }
       VECTOR_  (int n)       : CHAIN_<T> (n)              { dim = n     ; }
       VECTOR_  (VECTOR_& v) : CHAIN_<T> ((CHAIN_<T>&)v) { dim = v.size; }
       T&                 operator () (int i                           );
```

```
        VECTOR_&               operator =  (VECTOR_&                    );
        friend ostream&        operator << (ostream& outstr, VECTOR_& v );
        friend istream&        operator >> (istream& instr , VECTOR_& v );
        friend int             operator == (VECTOR_& u     , VECTOR_& v );
        friend int             operator != (VECTOR_& u     , VECTOR_& v );
        friend int             operator <  (VECTOR_& u     , VECTOR_& v );
        friend int             operator <= (VECTOR_& u     , VECTOR_& v );
        friend int             operator >= (VECTOR_& u     , VECTOR_& v );
        friend int             operator >  (VECTOR_& u     , VECTOR_& v );
        friend VECTOR_&        operator <<=(VECTOR_& u     , int      k );
        friend VECTOR_         operator << (VECTOR_& u     , int      k );
        friend MATRIX_<T>      mul         (VECTOR_& u     , VECTOR_& v );
        void                   create      (int dim );
        void                   add         (T   x   );
        void                   del         (int n   );
        void                   set         (T   x   );
        void                   print       (ostream&);
        void                   build       (istream&);
        int                    search      (T   x   );
        T                      move_down   (void    );
        T                      move_up     (void    );
        void                   swap        (int,int );
        VECTOR_<T>             copy        (int  n  );
        void                   heapSort    (        );
};

typedef VECTOR_<int>      IVECTOR;
typedef VECTOR_<double>   DVECTOR;

class VECTOR : public DVECTOR {
public :
        VECTOR  (            ) : VECTOR_<double> ( ) { ; }
        VECTOR  ( int n      ) : VECTOR_<double> (n) { ; }
        VECTOR  (VECTOR& v)    : VECTOR_<double> (v) { ; }
        VECTOR&               operator =  (VECTOR&              );
        friend  VECTOR        operator +  (VECTOR& u, VECTOR& v );
        friend  VECTOR        operator +  (VECTOR& u, double  v );
        friend  VECTOR&       operator += (VECTOR& u, VECTOR& v );
        friend  VECTOR&       operator += (VECTOR& u, double  v );
        friend  VECTOR        operator -  (VECTOR& u, VECTOR& v );
        friend  VECTOR        operator -  (VECTOR& u, double  v );
        friend  VECTOR&       operator -= (VECTOR& u, VECTOR& v );
        friend  VECTOR&       operator -= (VECTOR& u, double  v );
        friend  VECTOR        operator *  (VECTOR& u, double  v );
        friend  VECTOR&       operator *= (VECTOR& u, double  v );
        friend  VECTOR        operator *  (double  u, VECTOR& v );
        friend  VECTOR        operator *  (VECTOR& u, VECTOR& v );
        friend  VECTOR        operator /  (VECTOR& A, double  x );
        friend  VECTOR&       operator /= (VECTOR& A, double  x );
        friend  VECTOR        operator /  (VECTOR& u, VECTOR& v );
        friend  ostream&      operator << (ostream& , VECTOR& v );
        friend  istream&      operator >> (istream& , VECTOR& v );
        VECTOR          absval   (                    );
        double          abs      (                    );
        double          norm     (                    );
        double          mean     (                    );
        double          var      (double              );
        double          minval   (                    );
        int             minindex (                    );
        double          maxval   (                    );
        int             maxindex (                    );
        static VECTOR cross      (VECTOR &u, VECTOR &v);
        static double scalar     (VECTOR &u, VECTOR &v);
};

/*** The class of arrays ****************************************************/
template <class T> class MATRIX_ : public CHAIN_<T> {
public :
        int       row;    // Zeile
        int       col;    // Spalte
        MATRIX_   (void         )  : CHAIN_<T> (     ) { row = 0; col = 0;         }
        MATRIX_   (int m, int n)   : CHAIN_<T> (m * n) { row = m; col = n;         }
```

```
      MATRIX_  (int n)            : CHAIN_<T> (n * n) { row = col = n;           }
      MATRIX_  (MATRIX_& A)       : CHAIN_<T> ((CHAIN_<T>&)A)
                                    { row = A.row; col = A.col; }
      VECTOR_<T>          operator () (int       i                         );
      T&                  operator () (int       i      , int j            );
      MATRIX_&            operator =  (MATRIX_&                            );
      friend  ostream&    operator << (ostream& outstr, MATRIX_& A         );
      friend  istream&    operator >> (istream& instr , MATRIX_& A         );
      void                create      (int       m      , int       n      );
      void                vec_to_col  (int       i      , VECTOR_<T>& v    );
      void                col_to_vec  (int       i      , VECTOR_<T>& v    );
      void                diag_to_vec (VECTOR_<T>& v                       );
      T                   minval      (int&      i      , int&         j   );
      T                   maxval      (int&      i      , int&         j   );
      void                print       (ostream&                            );
      void                build       (istream&                            );
};

typedef MATRIX_<int>      IMATRIX;
typedef MATRIX_<double>   DMATRIX;

class MATRIX : public DMATRIX {
public :
      MATRIX  (              )  : MATRIX_<double> ( )     { ; }
      MATRIX  (int m, int n)    : MATRIX_<double> (m,n)   { ; }
      MATRIX  (int n         )  : MATRIX_<double> (n)     { ; }
      MATRIX  ( MATRIX& A   )   : MATRIX_<double> (A)     { ; }
      MATRIX&            operator =  (MATRIX&                 );
      friend  MATRIX     operator +  (MATRIX& A, MATRIX& B );
      friend  MATRIX&    operator += (MATRIX& A, MATRIX& B );
      friend  MATRIX     operator -  (MATRIX& A, MATRIX& B );
      friend  MATRIX&    operator -= (MATRIX& A, MATRIX& B );
      friend  MATRIX     operator *  (MATRIX& A, double   x );
      friend  MATRIX     operator *  (double   x, MATRIX& A );
      friend  MATRIX&    operator *= (MATRIX& A, double      );
      friend  MATRIX     operator *  (MATRIX& A, MATRIX& B );
      friend  VECTOR     operator *  (MATRIX& A, VECTOR& v );
      friend  VECTOR     operator *  (VECTOR& v, MATRIX& A );
      friend  MATRIX     operator /  (MATRIX& A, double   x );
      friend  MATRIX&    operator /= (MATRIX& A, double   x );
      friend  ostream&   operator << (ostream& , MATRIX& A );
      friend  istream&   operator >> (istream& , MATRIX& A );
      void               identity     (        );
      double             trace        (        );
      MATRIX             transp       (        );
};

#endif

/*** End of VECTORS.H *******************************************************/
```

7.5.1.21 <wind.h>

```
/****************************************************************************/
/***                                                                      ***/
/*** Module: WIND.H                                                       ***/
/***                                                                      ***/
/*** Header for wind related objects                                      ***/
/****************************************************************************/

#ifndef WIND_HEADER
#define WIND_HEADER

#ifndef DIFFCALC_HEADER
#include <diffcalc.h>
#endif

#ifndef DISTRIB_HEADER
#include <distrib.h>
```

```
#endif

#ifndef MATHFUNC_HEADER
#include <mathfunc.h>
#endif

class WindSpeedTimeSeries;   // Forward declaration

/******************************************************************************/
/*** Continuous Wind Speed Distribution ***************************************/
/******************************************************************************/

class Speed : public owObjfunc {
protected :
  double vmean;
  double vsigma;
public :
  Speed ( ) { ; }
  virtual double eval  (double) = 0;
  int            setUp (Param*);
};

class SpeedDist : public Speed {
public :
      SpeedDist (   ) { ; }
      double eval    (double v) { return (PHI(v,vmean,vsigma));}
};

class SpeedDens : public Speed {
public:
      SpeedDens ( )  { ; }
      double eval    (double v) { return (phi(v,vmean,vsigma));}
};

/******************************************************************************/
/*** Discrete Wind Speed Distribution *****************************************/
/******************************************************************************/

class DiscretWindSpeed : public DiscretDistribution {
private :
  double uAlpha;    // alpha quantile
  double vmean;     // mean wind speed
  double sigma;     // standard variation
  double r;         // correlation
  VECTOR points;
  VECTOR beta;
  double rawP  ( int , int              );
public :
  DiscretWindSpeed (int n) : DiscretDistribution (n) { ; }
  double gnm   ( int, int               );
  int    getN  ( double                 );
  int    setUp ( TStatusWindow*, Param* );
};

/******************************************************************************/
/*** Continuous Wind turbine power distribution *******************************/
/******************************************************************************/

class ContWindPower : public ContinuousDistribution {
private :
  double         fvvco,vci,vco,vr,vmean,sigmav;
  double         FW (double);
protected :
  double         r                   ; // autocorrelation coefficient
public :
  ContWindPower          (    );
  virtual ~ContWindPower (    ) { ; }
  double         F             (double                 );
  int            setUp         (TStatusWindow*, Param* );
  void           setCorrelation (double time, double beta);
};
```

```
/******************************************************************************/
/*** Continuous Conditional Wind turbine power distribution ****************/
/******************************************************************************/

class ContCondWindPower : public ContWindPower {
public :
  ContCondWindPower ( ) : ContWindPower ( ) { ; }
  int setUp (TStatusWindow*, Param*);
};

/******************************************************************************/
/*** Wind turbine power probability distributions *************************/
/******************************************************************************/

class ProbWindPower : public statfunc {
public :
  ProbWindPower ( );
};

class ProbCondWindPower : public statfunc {
public :
  ProbCondWindPower ( );
};

/******************************************************************************/
/*** Discrete Wind Power                                  ******************/
/******************************************************************************/

class DiscretWindPower : public DiscretDistribution {
private :
  double              vci,vco,vr,vmean;
  ContWindPower*      windPower;
  WindSpeedTimeSeries* timeSeries;
  double getPower    ( int n );
public :
  DiscretWindPower  ( int n );
  ~DiscretWindPower (       );
  double gnm    ( int, int              );
  double Gn     ( int                   );
  int    getN   ( double                );
  int    setUp  ( TStatusWindow*, Param* );
};

#endif

/*** End of wind.h ***********************************************************/
```

7.5.2 Source Files

7.5.2.1 <owrenew.cpp>

```
/****************************************************************************/
/*** Renewable Energy Resources for Windows                              ***/
/****************************************************************************/

#ifndef OWRENEW_HEADER
#include "owrenew.h"
#endif

#ifndef OWPLOT_HEADER
#include "owplot.h"
#endif

#ifndef OWCALC_HEADER
#include <owcalc.h>
#endif

#ifndef OWLAPPL_HEADER
#include <owlappl.h>
#endif

#ifndef OWPARAM_HEADER
#include <owparam.h>
#endif

#ifndef CSTRING_HEADER
#include <cstring.h>
#endif

#include <owl.h>
#include <button.h>
#include <edit.h>
#include <groupbox.h>
#include <radiobut.h>
#include <fstream.h>

#define dlgFile "owrenew.dlg"

/*** Module global prototypes ********************************************/
void NoFeatureMessage (HWND);

/*** Global variables ****************************************************/
Param*      param;    // Parameter
PTRenewApp App;
Graph*      GraphData;  // Graphic Data Interface

/****************************************************************************/
/*** class TRenewPlot                                                    ***/
/****************************************************************************/

TRenewPlot :: TRenewPlot (PTWindowsObject AParent, LPSTR ATitle, PTModule AModule)
    : TPlot (AParent, ATitle, AModule )
{
  delta = (curRect.right - curRect.left);
  start = curRect.left ;
  end   = start + delta;
  clear = YES;
}

void TRenewPlot :: Paint (HDC dc, PAINTSTRUCT _FAR& v) {
  if (clear == YES)
    TPlot :: Paint (dc, v);
  else {
    if (! param->disAuto)
      clear = YES;
```

```
    draw ( );
  }
}

void TRenewPlot :: plot ( ) {   // Display GraphData in a graph
  int       i;
      setHeadLine (GraphData->headline);
      setSubLine  (GraphData->subline );
  clearScreen ( );
  plotHeadLine ( );
  plotSubLine ( );
  drawMargin ( );
  GraphData->scale = drawAutoLinCoord (GraphData->min,GraphData->max,
     GraphData->y, GraphData->axtext," ",
     YES,YES,GraphData->scale,GraphData->curveNo);
  for (i=0;i<=GraphData->curveNo;i++) {
    drawCurve (GraphData->x,GraphData->y[i],GraphData->option);
    switch (i) {
      case 0 :
        setPenColor (RGB(255,0,0));
        break;
      case 1 :
        setPenColor (RGB(0,255,0));
        break;
      case 2 :
        setPenColor (RGB(0,0,255));
        break;
      case 3 :
      default :
        setPenColor (RGB(0,0,0));
        break;
    }
  }
  setPenColor (RGB(0,0,0));
}

/****************************************************************************/
/*** Main Platform                                                        ***/
/****************************************************************************/

/*** class TMainWindow *****************************************************/

TMainWindow :: TMainWindow (PTWindowsObject AParent, LPSTR ATitle)
    : TWindow(AParent, ATitle)
{
  Attr.Style |= WS_MAXIMIZE | WS_VISIBLE;
  AssignMenu ("COMMANDS");
  testplot = new TRenewPlot (this,NULL);
}

TMainWindow :: ~TMainWindow ( ) {
  delete testplot;
}

void TMainWindow :: GetWindowClass (WNDCLASS& WndClass) {
  TWindow :: GetWindowClass (WndClass);
  WndClass.hbrBackground = (HBRUSH) COLOR_APPWORKSPACE+1;
}

BOOL TMainWindow :: CanClose ( )
{
  BOOL retval;
  if (GetModule()->ExecDialog(new TYoMessage(this,"Question",
       "Do you want to quit to Windows?")) == IDYES){
    fstream op;
    op.open (dlgFile, ios :: out);
    if (op)
      op << *this;
    op.close ( );
    retval = True;
  }
  else
```

```
    retval = False;
  return retval;
}

void TMainWindow :: CMWindSpeed (RTMessage) {
  int retval=GetModule()->ExecDialog (new TSpeedDialog(this,"SpeedDialog"));
  if (retval == IDOK) {
      testplot->open ( );
      testplot->clearScreen ( );
      testplot->close ( );
      TransSpeedDlg.setParameter ( );
      TransSettingsDlg.wiVmean = TransSpeedDlg.vmean;
      if (GetModule()->ExecDialog(new TWindSpeedObject(this,"StatusWindow"))
         == IDOK)
        testplot->clear = NO;
  }
}

void TMainWindow :: CMSettings (RTMessage) {
  if (GetModule()->ExecDialog (new TSettingsDialog (this, "Settings"))== IDOK) {
    TransSettingsDlg.setParameter ( );
    TransSpeedDlg.vmean       = TransSettingsDlg.wiVmean;
    TransWindDlg.vmean        = TransSettingsDlg.wiVmean;
    TransSolarDlg.clearness = TransSettingsDlg.solK;
    TransSolarDlg.sigmaK      = TransSettingsDlg.solSigmaK;
    TransJointDlg.vmean       = TransSettingsDlg.wiVmean;
    TransJointDlg.sigmaK      = TransSettingsDlg.solSigmaK;
    TransJointDlg.clearness = TransSettingsDlg.solK;
  }
}

void TMainWindow :: CMMaths (RTMessage) {
  if (GetModule()->ExecDialog (new TMathsDialog (this, "Maths"))== IDOK) {
    TransMathsDlg.setParameter ( );
    TransSolarDlg.coeff = TransMathsDlg.solCoeff;
    TransSolarDlg.trial = TransMathsDlg.solTrial;
  }
}

void TMainWindow :: CMDir (RTMessage) {
  GetModule()->ExecDialog (new TDirDialog (this,"Directories"));
}

void TMainWindow :: CMDisplay (RTMessage) {
  if (GetModule()->ExecDialog (new TDisplayDialog (this,"Display"))== IDOK)
    TransDisplayDlg.setParameter ( );
}

void TMainWindow :: CMRandom (RTMessage) {
  if (GetModule()->ExecDialog (new TRandDialog (this,"RandomNumbers"))==IDOK){
    TransRandDlg.setParameter ( );
    GetModule()->ExecDialog(new TRandomObject(this,"StatusWindow"));
  }
}

void TMainWindow :: CMTimeSeries (RTMessage) {
  if (GetModule()->ExecDialog (new TTsDialog (this,"TimeSeries"))==IDOK) {
    TransTsDlg.setParameter ( );
    TransSettingsDlg.setParameter ( );
    TransMathsDlg.setParameter ( );
    param->solBypass = 1;
    testplot->open ( );
    testplot->clearScreen ( );
    testplot->close ( );
    if (GetModule()->ExecDialog(new TTimeSeriesObject(this,"StatusWindow"))
       == IDOK) {
      int        i,j;
      char       buffer[80];
      double     x = (GraphData->y[0])(1);
      for (i=0;i<=GraphData->curveNo;i++) {
        for (j=1;j<=GraphData->x.dim;j++) {
          if ((GraphData->y[i])(j) != x)
```

```
            break;
          }
        }
        if (i>GraphData->curveNo && j > GraphData->x.dim) {
          strcpy (buffer,"All data have same value: ");
          catDbl (buffer, x);
          GetModule()->ExecDialog(new TYoMessage(this,"Warning",
             buffer));
             return;
        }
        else
          testplot->clear = NO;
      }
    }
}

void TMainWindow :: CMFpt (RTMessage) {
  if (GetModule()->ExecDialog (new TFpDialog (this,"FirstPassageTime"))==IDOK) {
    TransSettingsDlg.setParameter ( );
    TransMathsDlg.setParameter ( );
    TransFpDlg.setParameter ( );
    param->solBypass = 1;
    if (param->fpSelectCalc==0)  // compute one value only
      GetModule()->ExecDialog(new TPassageTimeObject(this,"StatusWindow"));
    else {                       // compute more values
      testplot->open ( );
      testplot->clearScreen ( );
      testplot->close ( );
      if (GetModule()->ExecDialog(new PassageTimesObject(this,"StatusWindow"))
         == IDOK)
        testplot->clear = NO;
    }
  }
}

void TMainWindow :: CMWindPower (RTMessage) {
  if (GetModule()->ExecDialog (new TWindDialog (this, "WindPower")) == IDOK) {
    testplot->open ( );
    testplot->clearScreen ( );
    testplot->close ( );
    TransSettingsDlg.wiVmean = TransWindDlg.vmean;
    TransSettingsDlg.setParameter ( );
    TransWindDlg.setParameter ( );
    if (GetModule()->ExecDialog(new TDistributionObject(this,"StatusWindow"))
       == IDOK)
      testplot->clear = NO;
  }
}

void TMainWindow :: CMSolar (RTMessage) {
  if (GetModule()->ExecDialog (new TSolarDialog (this, "SolarPower")) == IDOK) {
    TransSettingsDlg.solK       = TransSolarDlg.clearness;
    TransSettingsDlg.solSigmaK  = TransSolarDlg.sigmaK;
    TransMathsDlg.solCoeff      = TransSolarDlg.coeff;
    TransMathsDlg.solTrial      = TransSolarDlg.trial;
    TransSettingsDlg.setParameter ( );
    TransSolarDlg.setParameter ( );
    testplot->open ( );
    testplot->clearScreen ( );
    testplot->close ( );
    if (GetModule()->ExecDialog(new TDistributionObject(this,"StatusWindow"))
       == IDOK)
      testplot->clear = NO;
  }
}

void TMainWindow :: CMRenewable (RTMessage) {
  if (GetModule()->ExecDialog (new TJointDialog (this, "RenewablePower")) == IDOK)
{
    TransSettingsDlg.solK       = TransJointDlg.clearness;
    TransSettingsDlg.solSigmaK  = TransJointDlg.sigmaK;
    TransSettingsDlg.wiVmean    = TransJointDlg.vmean;
```

```
void TRenewApp::InitMainWindow()
{
  TMainWindow* Main = new TMainWindow (NULL, Name);
  MainWindow = Main;
  fstream ip;
  ip.open (dlgFile, ios :: in);
  if (ip)
    ip >> *Main;
  ip.close ( );
  Main->TransSettingsDlg.setParameter ( );
  Main->TransSpeedDlg.setParameter    ( );
  Main->TransDisplayDlg.setParameter  ( );
}

/**************************************************************************/
/*** Main Programme *******************************************************/
/**************************************************************************/

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  TRenewApp RenewApp("Renewable Energy Short Term Prediction",
        hInstance, hPrevInstance, lpCmdLine, nCmdShow);
  param = new Param ( );
  param->disFirstCurve = YES;
  App = &RenewApp;
  GraphData = new Graph ( );
  RenewApp.Run();
  delete (param);
  delete (GraphData);
  return RenewApp.Status;
}

/*** end of file *********************************************************/
```

```
    TransSettingsDlg.comZeta    = TransJointDlg.zeta;
    TransMathsDlg.setParameter ( );
    TransSettingsDlg.setParameter ( );
    TransJointDlg.setParameter ( );
    param->solBypass = 1; // set solar bypass
    testplot->open ( );
    testplot->clearScreen ( );
    testplot->close ( );
    if (GetModule()->ExecDialog(new TJointDistributionObject(this,"StatusWindow"))
       == IDOK)
      testplot->clear = NO;
  }
}

void TMainWindow :: CMExport (RTMessage) {
  int errno = OK;
  if (GetModule()->ExecDialog (new TExportDialog (this, "Export")) == IDOK) {
    if (TransExportDlg.opNew == YES)
      errno = exportData (GraphData->x, TransExportDlg.expFile, NEW,"",
                          GraphData->scale);
    if (errno) {
      for (int i=0;i<=GraphData->curveNo;i++) {
        if ((errno = exportData (GraphData->y[i],TransExportDlg.expFile,
             ATTACH,GraphData->legend[i],GraphData->scale)) == ERROR)
          break;
      }
    }
    if (!errno)
      GetModule()->ExecDialog(new TYoMessage(this,"Message",
          "Could not open specified file"));
  }
}

void TMainWindow :: CMHelp (RTMessage) {
  GetModule()->ExecDialog(new TYoMessage(this,"Message",
     "Feature not implemented"));
}

ostream& operator << (ostream& outstr, RTMainWindow v) {
  outstr << v.TransSpeedDlg    << '\n';
  outstr << v.TransSettingsDlg << '\n';
  outstr << v.TransExportDlg   << '\n';
  outstr << v.TransDirDlg      << '\n';
  outstr << v.TransWindDlg     << '\n';
  outstr << v.TransSolarDlg    << '\n';
  outstr << v.TransJointDlg    << '\n';
  outstr << v.TransRandDlg     << '\n';
  outstr << v.TransMathsDlg    << '\n';
  outstr << v.TransTsDlg       << '\n';
  outstr << v.TransFpDlg       << '\n';
  outstr << v.TransDisplayDlg  << '\n';
  return outstr;
}

istream& operator >> (istream& instr, RTMainWindow v) {
  instr >> v.TransSpeedDlg
        >> v.TransSettingsDlg
        >> v.TransExportDlg
        >> v.TransDirDlg
        >> v.TransWindDlg
        >> v.TransSolarDlg
        >> v.TransJointDlg
        >> v.TransRandDlg
        >> v.TransMathsDlg
        >> v.TransTsDlg
        >> v.TransFpDlg
        >> v.TransDisplayDlg;
  return instr;
}

/*** Application **************************************************************/
```

8. References

[1] **Abramowitz, M.:** Handbook of mathematical functions, New York, 1965

[2] **Bădescu, V.:** Calculation of direct solar radiation on tilted surfaces, Solar Energy Vol. 48, No. 5, pp. 321 - 323, 1992

[3] **Borland:** Object Windows C++ Programming Handbook, Borland International, 1992

[4] **Borland:** Object Windows C++ Reference Handbook, Borland International, 1992

[5] **Borland:** Borland C++ 3.0 Reference Handbook, Borland International, 1992

[6] **Borland:** Resource Workshop User Handbook, Borland International, 1992

[7] **Bower, Ward I.:** Performance of battery charge controllers: An interim test report, 21st Photovoltaic specialists conference, 1990

[8] **Bronstein, Il'ja Nikolaevič:** Taschenbuch der Mathematik, Harri Deutsch, 1987

[9] **Buresh, Mathew:** Photovoltaic Energy Systems, Mc Graw-Hill, New York, 1983

[10] **Chauhan, Ankush:** Modelling of diesel engine bearing wear under steady state and transient conditions, Rutherford Appleton Laboratory, paper ERU-92-002, 1992

[11] **Coleman, Clint:** Hybrid power system operational test results: Wind/ PV/ Diesel system documentation, Telecommunication Energy Conference, Vol. 2, pp. 1-7, 1989

[12] **Child, Duncan:** MPhil thesis, Loughborough University of Technology, 1993

[13] **Facinelli, W. A.:** Modeling and Simulation of Lead Acid Batteries for Photovoltaic Systems, 18th Intersociety Energy Conference, 1983, Vol. 4, pp 1582-1588

[14] **Feller, W.:** An introduction to probability theory and its applications, Vol.1, Wiley, 1957

[15] **Fletcher, R.:** Practical methods of Optimization I, Chichester, 1980

[16] **Freris, L.L.:** The control of wind turbines, Imperial College London

[17] **Gopinathan, K. K.:** Solar sky radiation estimation techniques, Solar Energy Vol. 49, pp. 9 - 11, 1992

[18] **Gumbel, E.J.:** Distribution à plusieurs variables dont les marges sont données, C.R. Académie des Sciences Paris, Vol. 246, pp. 2717-2720, 1958; in: Emanuel Parzen: Modern probability Theory and its applications, John Wiley and Sons, New York, 1992

[19] **Hassan, U.; Sykes, D.M.:** Wind structure and statistics; in: Freris, L.L. (editor): Wind Energy Conversion Systems, Prentice- Hall, 1990

[20] **Helstrom, Carl W.:** Probability and Stochastic Processes for Engineers, Macmillan, 1991

[21] **Hill, Martin; Mc Carthy, Sean:** PV Battery Handbook, University of Cork, Ireland, 1990

[22] **Horst, Emil W. ter; Blok, Kornelis, Turkenburg, Wim C.:** Battery Modelling for Photovoltaic Applications, Photovoltaic Solar Energy 8th EC Conference, 1988 Florence, pp. 1564-1568

[23] **Jantsch, M., Stoll, W., Schmid, J.:** The effect of tilt angle and voltage conditions on PV system performance. An experimental investigation, 10 th European Photovoltaic Solar Energy Conference, Lisbon, Portugal, 1991

[24] **Khouzam, Kame Y.:** Optimum matching of a photovoltaic array to a storage battery, Photovoltaic Specialists Conference 1991 (22 nd) IEEE, Vol. 1, pp. 706-711, 1991

[25] **Lipman, N.H.; Infield, D.G.:** Wind- diesel systems; in: Freris, L.L. (editor): Wind Energy Conversion Systems, Prentice- Hall, 1990

[26] **Magnus, Wilhelm:** Formulas and Theorems for the Special Functions of Mathematical Physics, Springer Verlag Berlin, 3rd ed., 1966

[27] **Manwell, James F.; Mv Gowan, Jon G.:** Lead Acid Battery Storage Model for Hybrid Energy Systems, Solar Energy, Vol. 50, No. 5, pp 399-405, 1993

[28] **Nayar, C. V.:** Solar/ Wind/ Diesel Hybrid energy systems for remote areas, Energy Conversion Engineering Conference IECEC, Vol. 4, pp. 2029-2034, 1989

[29] **Orgill, J.F.:** Correlation equation for hourly diffuse radiation on a horizontal surface, Solar Energy Vol. 19, pp. 357 - 359, 1977

[30] **Papoulis, Athanasios:** Probability, Random Variables and Stochastic Processes, McGraw-Hill, 1984

[31] **Paynter, R.J.H; Lipman, N.H.; Foster, J.E.:** The potential of hydrogen and electricity production from wind energy, Report, Rutherford Appleton Laboratory, UK, 1991

[32] **Paynter, R.J.H.:** Predictive control of a wind diesel generation set, Rutherford Appleton Laboratory, 1994

[33] **Press, William H.:** The Art of Scientific Computing, 2nd edition, Cambridge University Press, 1992

[34] **Risken, H.; Vollmer, D.:** Methods for solving Fokker- Planck equations with applications; in: Frank Moss (editor): Noise in nonlinear dynamical systems, Vol. 1: Theory of continuous Fokker- Planck systems, Cambridge University Press

[35] **Salameh, Ziyad M.:** Step-up maximum power point tracker for photovoltaic arrays, Solar Energy, Vol. 44, No. 1, pp. 57-61, 1990

[36] **Salameh, Ziyad M.:** Step-down maximum power point tracker for photovoltaic systems, Solar Energy, Vol. 46, No. 5, pp. 279-282, 1991

[37] **Salameh, Ziyad M.:** A mathematical model for lead- acid batteries, IEEE Transactions on Energy Conversion, Vol. 7, No. 1, pp. 93-97, 1992

[38] **Shepherd, C. M.:** Design of Primary and Secondary Cells .II. An Equation Describing Battery Discharge, Journal of the Electrochemical Society, Vol. 112, pp. 657-664, July 1965

[39] **Sheridan, Norman R.:** Batteries for autonomous renewable energy systems, Journal of Power Sources Vol. 35, pp. 371 - 375, 1991

[40] **Skartveit, A.:** The probability and autocorrelation of short- term global and beam irradiance, Solar Energy Vol. 49, No. 6, pp. 477 - 487, 1992

[41] **Spanier, J.; Oldham, K.B.:** Atlas of functions, Springer- Verlag Berlin, 1987

[42] **Tsubota, Masaharu:** Development of lead- acid batteries for photovoltaic power systems, Journal of Power Sources Vol. 35, pp. 355 - 358, 1991

[43] **Weiss, R.; Appelbaum, J.:** Battery State of Charge Determination in Photovoltaic Systems, J. Electrochem. Soc, Vol. 129, No. 9, pp 1928-1933