Aurimas ANSKAITIS

TELECOMMUNICATIONS THEORY

Project No
VP1-2.2-ŠMM-07-K-01-047

The Essential Renewal of Undergraduates Study Programs of VGTU Electronics Faculty
Aurimas ANSKAITIS

TELECOMMUNICATIONS THEORY

A Laboratory Manual

Vilnius “Technika” 2012

The book presents laboratory works in classical topics of telecommunications theory – signal generation, spectral analysis of signals, formation of baseband and passband modulated signals and transmission of modulated signals through band limited channel.

Also forward error correction methods using convolutional and cyclic codes are presented. Every laboratory work starts with theoretical presentation of corresponding topic. Also demonstrational programs, illustrating programming / simulation techniques for every laboratory work, are presented. The programs are written in Python programming language. Labs are formulated in such a way that student must write a short simulation program for every task.

The publication has been recommended by the Study Committee of VGTU Electronics Faculty.

Reviewed by:
Prof Habil Dr Algimantas Kajackas, VGTU Department of Telecommunications Engineering,
Prof Dr Šarūnas Paulikas, VGTU Department of Telecommunications Engineering

This publication has been produced with the financial assistance of Europe Social Fund and VGTU (Project No VP1-2.2-ŠMM-07-K-01-047). The book is a part of the project “The Essential Renewal of Undergraduates Study Programs of VGTU Electronics Faculty”.

This is an educational methodology book, No 1342-S, issued by VGTU Press TECHNIKA http://leidykla.vgtu.lt

Language editor Kristina Noreikienė
Typesetter Laura Petrauskiënė

doi:10.3846/1342-S

© Aurimas Anskaitis, 2012
© Vilnius Gediminas Technical University, 2012
## Contents

1. **Introduction to Simulation Environment**  
   1.1. The main features of *Python*  
   1.2. The *Python* interpreter  
   1.3. Basic *Python* syntax and data types  
   1.4. Control structures  
   1.5. Functions  
   1.6. Modules  
   1.7. Classes  
   1.8. Numpy library  
   1.9. Matplotlib library  
   1.10. Laboratory work  

2. **Signals and Spectra**  
   2.1. Classification of signals  
   2.2. Signal spectra  
   2.3. Laboratory work  

3. **Random Variables and Processes**  
   3.1. Random events and random variables  
   3.2. Random processes  
   3.3. Laboratory work  

4. **Data Transmission, Modulation and Detection**  
   4.1. Baseband data transmission  
   4.2. Passband modulation  

---

5  
6  
7  
12  
14  
16  
17  
19  
26  
27  
29  
33  
40  
45  
45  
52  
61  
65  
65  
70
4.2.1. BPSK modulation ............................... 71
4.2.2. BFSK modulation ............................... 73
4.3. Optimal receivers for AWGN channel ............... 75
4.4. Laboratory work ................................ 89

5. Channel Coding .................................. 91
   5.1. CRC coding fundamentals ...................... 92
   5.2. Convolutional codes ............................. 103
   5.3. Laboratory work ................................ 113

References ............................................ 117
Chapter 1

Introduction to Simulation Environment

The laboratories in this course will be done using Python programming language. This chapter introduces the programming language and associated libraries for mathematical computations. The main usage idioms are presented as well. In the Internet Python may be found at www.python.org.

Python is a powerful programming language and is used for many real world projects. Also it is one of the most popular programming languages today – according to http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html Python is 8th language in popularity as of today. One of the most appealing features of Python is that it is completely free and open source. Competing platforms such as MATLAB are costly. Python has high level data structures and a simple means for object oriented programming. Many finds Python’s syntax to be elegant and dynamic typing of the language allows for “quick and dirty” programming. Such programming style is valuable for quick scientific experiments. Python is also a multiplatform and works equally well on all systems.
1.1 The main features of *Python*

*Python* is a very simple programming language, but despite the simplicity it is a real programming language, not limited to some specific problem domains.

Programming in *Python* you can divide your programs into modules. The modules may be later reused in other programs. There is a large variety of standard modules that you can use in your programs. These modules provide things like file I/O, operating system calls, sockets for network programming and so on. There also exist many non standard (third party) modules. Two such modules (numpy and matplotlib) will be handy for our particular purpose of simulation of telecommunication systems.

The main *Python*’s implementation is interpreter. This saves a lot of time during program development as you don’t need to recompile your program after every small change. The interpreter also makes it easy to perform interactive experimentation with different language features. The interpreted nature of *Python* also has a noticeable drawback – the final programs are much slower compared to languages such as *C* or *Java*. But this is the only case if you fail to introduce high-level structures into your programs.

*Python*’s programs are usually compact. This is due to several features – high level data types makes it easy to express compound operations in one statement, no need to declare the type of variables and statement grouping is done by indentation instead of using curly braces.

1.2 The *Python* interpreter

Most Linux systems have *Python* preinstalled. In case you use Linux, the interpreter may be invoked by command “python”. If you are Windows user then you probably should install *Python* manually. 2.7.x version is used in this manual. When installed, *Python* interpreter may be invoked from “Start” menu.
The interpreter has two modes – interactive and batch mode. Here is a simple interactive session under Linux:

```python
>>> print 2+2
4
```n
Interactive mode is handy for testing of small functions but it is not designed for creation of real programs. If you want to write and execute real programs, you should use batch mode.

Under Linux *Python* scripts can be made executable if corresponding *.py* file starts with

```bash
#! /usr/bin/env python
```

and file permissions allow execution. Under Windows usually file extension “.py” is associated with *Python* interpreter and double clicking on such file causes its execution.

## 1.3 Basic *Python* syntax and data types

The following examples are presented in interactive *Python* shell.

One of the most important parts of any program is comments. Comments in *Python* start with the “#” character and extend to the end of the corresponding line. For example:

```python
# This is a comment line
a = 1    # assign value 1 to variable a
b = 2    # assign value 2 to variable b
# This is the last comment
```

*Python* operates with numbers in a simple and sensible way. The below given session illustrates this:
>>> 2+2
4
>>> (3+3)*(4+1)
30
>>> 10/5
2
>>> 9/4 #integer division
2
>>> 9.0/4 #real division
2

The important fact to note is if you want real division then at least one of the arguments must be floating point number. If all the operands are integer then integer division is performed.

Values to variables are assigned using "=" operator. You should not care about types (at least for now). These are deduced by the values on the right hand side of the assignment operator.

>>> a = 5
>>> b = 6
>>> area = a * b
>>> area
30

Variables should be assigned before they are used. Otherwise the error will be raised:

>>> a = 1
>>> a * b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined

Usually we should not be afraid of errors. In majority of cases they tell us exactly what is wrong with our program. For example, the above error trace tells that variable "b" is not defined at the point where it is used.

Python has a built in support for complex numbers. This feature is obviously beneficial for any mathematical computations:
>>> c = 2+3.2j
>>> 2*c
(4+6.4j)
>>> d = complex(2, 3)
>>> 2*d
(4+6j)

Complex numbers always consist of two floating point numbers. There is no such thing as integer complex number in Python. Imaginary and real parts of a complex number may be extracted by accordingly named operations. Magnitude of a complex number may be extracted using function “abs”.

>>> a = 1+2j
>>> a.imag
2.0
>>> a.real
1.0
>>> abs(a)
2.23606797749979

As and every other decent programming language, Python can operate on strings. Strings are enclosed in single or double quotes. In case string is enclosed in single quotes, double quotes do not need escaping. And vice versa. Some examples follow:

>>> "aaa"
'aaa'
>>> '"aaa"'
"'aaa'"
>>> '"aaa"'

You can access some special characters by using so called escape sequences. For example “tab” character may be accesses using the following string “\t”. If a string containing such character is printed, the tabulation is produced in an output.

There are many operations you can perform with strings. You can access separate characters using index notation, strings can be concatenated using “+” operator, substrings may be extracted using
special indexing notation; string length may be obtained and so on. By the way, concatenation operator for strings is the same as for numerical addition. This feature is called operator overloading. This means that operator (or function) may have different meaning for different data types. A few examples are given below:

```python
>>> s = "abcdef"
>>> s[1]  # indexing is zero based
'b'
>>> s[2:4]  # extract substring
'cd'
>>> s[2:]  # extract symbols till the end
'cdef'
>>> s[:2]  # extract symbols till the given index
'ab'
>>> s + "aaa"
'abcdefaaa'
>>> len(s)
6
```

In *Python* there is also so called multiline and raw strings. If you ever need such thing, details may be found using *Python* documentation.

Probably the most important data type in *Python* is a list. Lists in some sense are similar to resizeable arrays in other programming languages but they are much more powerful in many ways. List is a compound data type. In essence list is a collection of indexed objects. These objects are not necessarily of the same type. A simple example follows. Notice, that list “l” is an element of a list “ll”. This just shows that lists may contain elements of any type.

```python
>>> l = [1, 2, "abc"]
>>> l
[1, 2, 'abc']
>>> ll = [1, 2, "abc", l]
>>> ll
[1, 2, 'abc', [1, 2, 'abc']]```
Like strings, lists are indexed starting from zero. Separate elements may be extracted, list may be sliced (slice operation extracts a sublist), list length may be obtained:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[1]
2
>>> l[1:4]
[2, 3, 4]
>>> len(l)
5
```

There is a big difference between lists and strings – elements of a list can be modified and strings can not. Assigning to slices is also allowed. In this case expression on the right hand side should be list with a length equal to the length of the slice:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[2] = 10
>>> l
[1, 2, 10, 4, 5]
>>> l[0:2] = [10, 10]
>>> l
[10, 10, 10, 4, 5]
```

List’s length is not fixed. You can append and delete elements from the list. In general, deletion operator works not only with lists but with any data type.

```
>>> l = [1, 2, 3]
>>> l.append(10)
>>> l
[1, 2, 3, 10]
>>> del l[1]
>>> l
[1, 3, 10]
```
1.4 Control structures

Up till now our attention was concentrated on simple operations. In principle majority of such operations could be performed by hand. Computers are good at repetitive tasks. For this purpose there are looping constructs and other flow control structures.

“While” loop executes for as long as the condition is true. The condition is written directly after the “while” keyword. The following example prints all even numbers that are less than 10 or equal to 10.

```python
>>> i = 1
>>> while i <= 10:
...     if i % 2 == 0:
...         print i
...     i = i + 1
...
2
4
6
8
10
```

This snippet introduces a few new concepts. Operator “%” is modulo division. Indentation is mandatory and all the statements in the same block should be indented by the same amount of spaces. Instead of spaces it is possible to use tabs but this is not recommended. You should configure your text editor to expand tabs into spaces. Also notice that every statement starting new block of code ends with colon.

Conditional statement (“if” statement) probably is the foundation of programming. Take a note that all looping constructs include implicit conditional statement. This statement checks the truth value of a condition and executes the code only if this value is true. Otherwise the other condition is checked. The following decides if the number is zero, positive or negative.

```python
>>> x = float(input("Enter an number: 
"))
Enter an number: 10
```
```python
>>> if x == 0:
...     print "Zero"
... elif x < 0:
...     print "Negative"
... else:
...     print "Positive"

Positive

In Python statement is a little different than in more traditional languages like C. Instead of generating monotonically increasing numbers, Python’s “for” traverses any collection – list, string or user defined one. At every step of iteration, current element from the collection is extracted. For example:

```python
>>> l = [1, "aa", 2, "b", [1, 2]]
>>> for i in l:
...     print i
...
1
aa
2
b
[1, 2]
```

What if we want to have a more traditional “for” loop? In such case we need to form temporary collection with increasing numbers. For this purpose there is “range” function. A few examples follow:

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> range(1, 20, 3)
[1, 4, 7, 10, 13, 16, 19]
>>> for i in range(5):
...     print i
...
0
1
```
Python has “break” and “continue” statements which are roughly equivalent to corresponding C statements. “Break” statement exits the enclosing “for” loop. “Continue” statement goes directly to the next iteration of the enclosing “for” statement.

Python has one more unorthodox control statement – “pass”. This statement does nothing and it can be used in places where you need something syntactically.

### 1.5 Functions

Programs of any significant complexity are never written as a continuous block of code. We have some structure in programs. The most important way of introducing the structure is creation of functions. Functions are defined using the keyword “def”. Function name follows this keyword and formal parameters are written in parenthesis after the function name. Let us write a function which adds two of its arguments. After the function is defined, it can be called by supplying actual parameters. In our example below actual parameters are numbers 1 and 2 and strings “1” and “2”. This example also illustrates dynamic typing nature of Python. As long as the “+” operation is supported on supplied arguments, program just works. But you must provide arguments of the same types. If one of the arguments is a number and the other string then the error will be raised.

```python
>>> def mySum(a, b):
...     return a + b
...    
>>> mySum(1, 2)
3
>>> mySum("1", "2")
'12'
>>> mySum(1, "2")
```

14
It is also possible to define functions with a variable number of arguments but such need is quite rare in simple programs.

Functions may have a documentation string in their definition. It is the string in triple quotes directly after the function name. This string is simply skipped by the interpreter.

It is recommended to indent Python code using 4 spaces and no tab characters. Using tabs and spaces in the same file may lead to cryptic syntax errors.

Tuple is an important data structure. It is similar to list but has a fixed length. Elements of a tuple are written in bracket and separated by commas. Elements of a tuple may be accessed using indexing operator[]. The main use for tuples is returning multiply values from a function. The function below illustrates this. Not only the sum of list elements is calculated, but also list length is returned.

```python
def sumList(l):
    s = 0
    # loop sums the elements in the list
    for i in l:
        s = s + i
    return (s, len(l))
```

```python
>>> sumList([1, 2, 3, 4])
(10, 4)
>>> sumList([1, 2, 3, 4])[0] #access only sum
10
>>> sumList([1, 2, 3, 4])[1] #access only length
4
```

Python has much more data structures – different sequences, sets, dictionaries and so on. But these are rarely required for simple scien-
1.6 Modules

Modules are collections of related functions, data types and variables. You can use already predefined modules, third party modules or you can define your own. An example of a standard module is “cmath” module which contains mathematical functions on complex numbers. Before using any functions or data from a module you must import that module. There are several ways of importing. They differ by the level of access to names defined in the module provided.

Let’s construct an example module. This module contains our summation functions. The module is named “sums.py”. It should be placed in current directory. The contents of a module is given below:

```
#!/usr/bin/env python

def sumNumbers(a, b):
    return a + b

def sumList(l):
    s = 0
    for i in l:
        s = s + i
    return s
```

Now we can use the functions in the module in several ways. The first way imports only module name into current symbol table and function names must be qualified by module name. An example is below:

```python
>>> import sums
>>> sums.sumNumbers(1, 2)
3
```

The other way imports directly all function names from a module to the current symbol table:
Both ways of importing have their advantages and disadvantages. The first way is longer but it is clear where from the used functions are coming. And the second way is shorter but the origin of function names is not clear and sometimes name collisions can happen.

Also it is possible to selectively import desired function names. This is illustrated below:

```python
>>> from sums import sumNumbers
>>> sumNumbers(2, 3) # no need for qualification
5
```

Modules can also import other modules. The syntax of importing is the same as described above.

When Python interpreter encounters an import statement, it searches for the module using simple algorithm. First, current directory is searched, then system directories. All third party software is installed into those system directories and you should not care where precisely they are. It is sufficient to know certain module names you want to import.

Python functions are grouped in modules. In a similar fashion modules may be grouped in packages. If a few modules are located in one directory then they are in the package which name coincides with directory name. Importing from modules located in packages is also quite simple. For example “from sound.wav import *” imports all the names located in module “wav” which is located in package “sound”.

1.7 Classes

Class is the main method of abstraction in modern imperative programming. Many real world entities are well encoded using this abstraction. Classes is the base on which object oriented programming
is built. The following review is just a short glimpse into Python's means of object oriented programming.

The following example constructs a simple class named “Counter”. There is a method named __init__ in this class. This is constructor – this method is called when an instance of a class is created. An instance of a class is called object. We can see that every method in a class is declared the same way as functions are declared. The main difference is that the first argument is always “self”. This argument is necessary to differentiate between regular functions and class methods. Every variable which is introduced preceded by keyword self is accessible the same way as methods.

```python
class Counter:
    def __init__(self, initialCount):
        self.count = initialCount

    def increase(self):
        self.count = self.count + 1

    def decrease(self):
        self.count = self.count - 1

    def printValue(self):  
        print self.count
```

Objects are constructed from class. We can construct as many objects as we want from one class. The example given below constructs on object of type “Counter”, initializes counter to 10, two times increases the counter and prints the resulting counter value. The result of evaluation of this code is printed number 12.

```python
c = Counter(10)
c.increase()
c.increase()
c.printValue()
```

Python also supports all the things which are considered necessary for object oriented programming – inheritance, multiply inheritance, private variables, etc.
This short review ends our general presentation of Python. The following writings are about Python libraries dedicated to mathematical computing. In particular we are going to review numpy (http://www.scipy.org/) and matplotlib (http://matplotlib.sourceforge.net/) libraries. Numpy gives Matlab like environment where the main data type is a vector. Matplotlib is a library for scientific plotting. Under Windows operating system you can install both libraries by downloading them and following standard steps of window program installation. Under Linux use your system’s package manager.

Question can be raised why not MATLAB. The main reason is the price of MATLAB. The other reason is that Python is general purpose programming language and familiarity with it may be beneficial for further professional carrier of a student.

1.8 Numpy library

The main features of numpy library are explored. Sometimes comparisons to MATLAB equivalents are made.

The main object in numpy library is homogeneous multidimensional array. In our course usually only vectors will be required. In this context vector is a synonym for one dimensional array. So, numpy array is a table of indexed elements, all of the same type. Every element in an array is indexed by a tuple of non-negative integers. Indexing always starts from zero. In numpy’s terminology dimensions are called axes. The dimensionality of an array is called rank. We will be mostly concerned with “rank 1” arrays – vectors. For example, a signal representing a sinus wave is an array of rank 1.

Array class in numpy is called ndarray. You almost never should construct the array using its constructor. There are a number of functions which construct various types of arrays in a more convenient fashion. Nonetheless, every constructed array has a number of useful attributes. Some of them are listed here:
- \textit{ndim} – number of dimensions of an array;

- \textit{shape} – tuple of integers showing the number of elements in every dimension. The length of this tuple is equal to \textit{ndim};

- \textit{size} – the number of elements in an array. It is equal to the product of elements of \textit{shape}.

An example below demonstrates the creation of a single dimensional array in different ways. Also, properties of the array are requested in several ways. Notice that the code starts with importing all names from numpy package. Such method of import is standard in numpy usage. This way we get access to a vast variety of array manipulation functions.

```python
>>> from numpy import *
>>> a1 = arange(10)
>>> a1
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a2 = zeros(10)
>>> a2
array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> a3 = ones(10)
>>> a3
array([ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> len(a1)
10
>>> a2.shape
(10,)
>>> a3.size
10
```

The following example demonstrates the creation of a two dimensional array. We are not going to analyze multidimensional arrays in depth but majority of operations on arrays work the same way on vectors and multidimensional arrays.
>>> from numpy import *
>>> a = zeros(10).reshape(2, 5)
>>> a
array([[ 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0.]]
>>> a.size
10
>>> a.shape
(2, 5)

It is also possible to create arrays from lists:

>>> b = array([1, 2, 3], dtype = float)
>>> b
array([ 1., 2., 3.])

Keyword argument “dtype” shows that data types of the elements should be interpreted as floating point numbers. We can achieve the same result writing one of numerical literals in the list as a floating point number.

Numpy array is a sequence (this is ensured by underlying design). So these arrays may be used as arguments in “for” loops. For example:

>>> for i in arange(5):
...    print i
... 0
1
2
3

Arithmetic operations and standard transcendental functions on arrays are defined element-wise. Every binary operation is applied to corresponding elements of the argument arrays. It means that argument arrays must have the same length. A demonstration of some of these arithmetic operations is given below.

>>> from numpy import *
>>> a = arange(5)
```python
>>> b = array([2, 3, 4, 5, 6])
>>> a+b
array([ 2, 4, 6, 8, 10])
>>> a*b
array([ 0, 3, 8, 15, 24])
>>> a/b
array([0, 0, 0, 0, 0])

The last line of the example may seem surprising. But remember that elements of the arrays are integers thus integer division is performed. If we want real division, we must declare one of the arrays to have floating point values. For example “a = arange(5, dtype = float)” would suffice.

Functions on arrays also operate element-wise:

```python
>>> from numpy import *
>>> a = arange(5, dtype = float)
>>> exp(a)
array([ 1., 2.71828183, 7.3890561 , 20.08553692, 54.59815003])
>>> a**2  # squaring operation
array([ 0., 1., 4., 9., 16.])
>>> a**2+1
array([ 1., 2., 5., 10., 17.])
```

There exist many other predefined functions on arrays. For example – finding minimal and maximal elements of an array, summation of array elements and so on:

```python
>>> from numpy import *
>>> a = sin(arange(10, dtype = float))
>>> a
array([ 0. , 0.84147098, 0.90929743, 0.14112001, -0.7568025 , -0.95892427, -0.2794155 , 0.6569866 , 0.98935825, 0.41211849])
>>> min(a)
-0.95892427466313845
>>> max(a)
0.98935824662338179
```
>>> sum(a)
1.9552094821073804

>>> a.argmax()  # find the index of maximal element
8

>>> a.argmin()  # find the index of minimal element
5

We have already discussed slicing when talking about lists. Numpy arrays have even more powerful constructs. Slicing is a way to extract a subvector from a given vector. There are many ways to do this. The following example illustrates a few of such possibilities.

```python
>>> from numpy import *
>>> a = arange(10)
>>> a[2:5]
array([2, 3, 4])
# take every second element between 1 and 8
>>> a[1:8:2]
array([1, 3, 5, 7])

>>> a[1:3] = 0  # modify elements
>>> a
array([0, 0, 0, 3, 4, 5, 6, 7, 8, 9])

>>> a[::-1]  # reverse elements
array([9, 8, 7, 6, 5, 4, 3, 0, 0, 0])
```

There are many methods to manipulate shape of array. One of the most frequently used methods is array concatenation. The following example shows one a few possible ways of array concatenation. Function names in the example can probably be deciphered as “horizontally stack” and “vertically stack”.

```python
>>> from numpy import *
>>> a = arange(5)
>>> b = arange(5)

>>> hstack((a, b))
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])

>>> vstack((a, b))
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```
It is also possible to split one big array into smaller ones. The functions for this task are “hsplit” and “vsplit”.

Array assignment and copying are important topics because neglecting them may result in erroneous programs.

Simple assignment does not create a copy of the data. Following example shows that variables $a$ and $b$ share the same data. This strategy is computationally efficient in most cases but sometimes we want a new copy of the existing data.

```python
>>> from numpy import *
>>> a = array([1, 2, 3, 4])
>>> b = a
>>> b[0] = 100  # it seems we change only b
>>> a  # but a is also changed
array([100, 2, 3, 4])
>>> b
array([100, 2, 3, 4])
```

If we really know that we need a copy of the data then copy method should be used. The same logic applies to slices. If an assignment is made to a slice then the variable assigned “looks” at the data in the original array. Copy method should be used if we need a new copy of the sliced data. The example below illustrates the impact of `copy` method.

```python
>>> from numpy import *
>>> a = array([1, 2, 3, 4])
>>> b = a.copy()  # make copy of the data
>>> b[0] = 100
>>> a  # this vector is not changed
array([1, 2, 3, 4])
>>> b  # only this is changed
array([100, 2, 3, 4])
```

Arrays by itself are useful for signal representation but usually we need linear algebra operations on two dimensional arrays. At least solving of linear equations should be supported. Numpy supports linear algebra operations in “numpy.linalg” package. The example given below shows a few such operations.
>>> a = arange(9, dtype = float).reshape((3, 3))
>>> a[1, 1] = 10
>>> a
array([[ 0.,  1.,  2.],
        [ 3., 10.,  5.],
        [ 6.,  7.,  8.]])
>>> a.transpose()  # transponation of the matrix
array([[ 0.,  3.,  6.],
        [ 1., 10.,  7.],
        [ 2.,  5.,  8.]])
>>> inv(a)  # matrix inversion
array([[-0.625 , -0.08333333,  0.20833333],
        [-0.08333333,  0.16666667, -0.08333333],
        [ 0.54166667, -0.08333333,  0.04166667]])
>>> dot(a, a)  # matrix product
array([[ 15.,  24.,  21.],
        [ 60., 138.,  96.],
        [ 69., 132., 111.])]
>>> y = array([1, 2, 3])
>>> solve(a, y)  # solve system of linear equations
array([-0.16666667,  0. ,  0.5])
>>> eig(a)  # eigenvalues and eigenvectors
>>> of a matrix
(array([ 15.81523497, -1.30467231,  3.48943733]),
array([[-0.13388152, -0.83159272,  0.22458934],
        [-0.68489511, -0.02471549, -0.65651488],
        [-0.71623629,  0.55483574,  0.72010266]]))

There is also special class for matrix operations. Every two dimensional array can be converted to a matrix. For this class many operators are overloaded to have semantics of usual linear algebra operations. For simple tasks two dimensional arrays are sufficient. Only in case your algorithm relies on matrixes heavily, you should consider converting you data to matrix class.
1.9 Matplotlib library

Scientific computing environments are unimaginable without data visualization tools. Matplotlib is a library for this particular purpose. It has many usages but here we are going to look at a stateful interface which provides very similar means of plotting as MATLAB. Installation procedure is exactly the same as for numpy library. Matplotlib depends on numpy so the later should already be installed.

The main type of the plot in “matplotlib” is a simple line plot. In this case you provide data points in two vectors. The entries of the first vector mark $x$ values, while the entries of the second vector mark $y$ values of the data. These points are connected by lines. The example below plots a graph of a parabola. Also some possibilities of graph annotation are illustrated – labels on axes are added and the graph is titled.

```python
>>> from numpy import *
>>> import pylab as P #import plotting functions
>>> x = arange(-3, 3, 0.1)
>>> y = x**2
>>> P.figure() #always create new figure
<matplotlib.figure.Figure object at 0x90f4f0c>
>>> P.plot(x, y) #main plotting function
[<matplotlib.lines.Line2D object at 0x8f8358c>]
>>> P.xlabel("Time, s")
<matplotlib.text.Text object at 0x8f80b4c>
>>> P.ylabel("Amplitude, V")
<matplotlib.text.Text object at 0x8f71eac>
>>> P.title("Strange dependence")
<matplotlib.text.Text object at 0x8f914cc>
>>> P.show() #draw the result
```

The resulting graph is shown in fig. 1.1. Majority of the plotting functions in matplotlib are quite flexible. For example you can supply only one argument to the “plot” function. In such case that argument would be regarded as $y$ data and $x$ data would be generated as increasing integers from 0 to “\text{len}(x)-1". 

26
There are many types of possible plots in matplotlib. Every plot may be annotated in different ways. All these possibilities may be looked up in the documentation if need arises.

### 1.10 Laboratory work

#### Lab purpose

Purpose of the lab is to get acquainted with the basic elements of Python programming and to obtain basic skills of numpy and matplotlib library usage.

#### Lab task

1. Implement sorting routine using bubble sort and selection sort algorithms. You should not use any additional library functions for this – just pure Python. What is the computational complexity of these algorithms? What Python constructs do...
you need in order to implement them?

2. Use the implemented sorting routine on the above generated list of random numbers. Can you use the implemented algorithms on numpy arrays? Why?

3. Using only numpy functions write an algorithm for polynomial data approximation. Given $N$ data points as pairs $x_i, y_i$ (these data points may be given in numpy arrays) and the order of approximating polynomial $M (M < N)$, the algorithm should find the coefficients of the polynomial.

4. Plot the data points and approximating graph from the above clause on the same graph. The graph should have legends for both data sets.

Control questions

- What is the dynamic typing? How is it different from static typing?

- What are pros and cons of the dynamic typing (compared to static one)?

- Should Python lists be homogeneous?

Lab report

The report should contain answers to the questions presented in the lab task and heavily commented programs, written for the task.
Chapter 2

Signals and Spectra

Many types of signals are present in telecommunication systems. Some of these signals are important and desired – in this class falls information bearing signals. Other signals cause information reception problems – noise and interference signals belong to this class. Also signals may be analog or discrete. More and more operations are performed in discrete domain in modern telecommunications. Despite this big variety of signals it is important to have apparatus for their description. This chapter focuses on deterministic signals and their spectra.

2.1 Classification of signals

The physical nature of a signal is not important for our discussion. In many cases unit of measure for the signal is voltage. Signals may be classified in many ways. Several classification schemes are presented below. Frequently classification criteria divides all possible signals into two nonoverlapping classes.

Energy signals. Signal is classified as an energy signal if it has finite energy. For continuous signal energy is calculated using expression:
Energy of a discrete time signal is calculated similarly:

$$ E = \sum_{n=-\infty}^{+\infty} x(n)^2. \quad (2.2) $$

Every real physical signal is an energy signal. In theory there are many signals with infinite energy. For example, any constant non zero signal, any signal which vanishes slower than 1/t, any harmonics all are not energy signals.

**Power signals.** Power is the energy divided by time during which that energy was created. Average power for a continuous time signal is defined as:

$$ P = \lim_{T \to +\infty} \frac{1}{2T} \int_{-T}^{T} x(t)^2 \, dt. \quad (2.3) $$

For discrete time signals power is defined as:

$$ P = \lim_{N \to +\infty} \frac{1}{2N} \sum_{n=-N}^{N} x(n)^2. \quad (2.4) $$

Signal is defined as a power signal if it has infinite energy but finite average power. There is a broad class of such signals. For example all signals that have infinitine energy and satisfy $|x(t)| < M$ for all $t$ and some positive constant $M$.

What does this classification of signals into energy and power signals give us? Usually if we have energy signal, we can find its Fourier transforms. For the power signal calculation of Fourier series sometimes is possible (additional requirements should be satisfied). If signal is neither energy, nor power, then analysis of such signal
is complicated. For many practical situations such signal is useless. And fortunately, there are no such signals in practice.

**Deterministic signals.** This class of signals may be described by some formula. In other words deterministic signal is a functional relation between some independent variable (usually time) and its value. Examples of such signals are all well known mathematical functions. These functions may also be combined in a piecewise manner. In telecommunications such signals are not suitable for information transmission – information bearing signal should be random to some degree. Deterministic signals in telecommunications are used for system testing.

**Nondeterministic signals.** If signal cannot be described deterministically, it is call non-deterministic or random. Aparatus of probability theory and statistics is used for analysis of such signals. These signals are very important in telecommunications because they are everywhere – from information bearing signals to different noise and interference sources. Separate laboratory work will be prepared for this topic.

**Continuous time and discrete time signals.** We already have discussed these signals a little. More precisely, continuous time signal is the one which is determined for any real value of its argument. On the contrary, discrete time signals are only defined for integer values of its argument. Discrete time signal may be used to approximate continuous time signals. For this purpose the process of discretization is performed on the continuous time signal. If discretization frequency is high enough then the process of discretization is reversible – continuous time signal may be restored without any distortions.

**Periodic and nonperiodic signals.** Many signals in telecommunications are periodic. For example carrier signals usually are harmonic.

Formally it is said that continuous time signal is periodic if the following equality holds for all integer $n$:

$$x(t) = x(t + nT). \tag{2.5}$$
Any $T$ for which this equation is satisfied is called a period. The least positive value of $T$ is called a prime period.

It is easy to see that any periodic signal which is not zero, has infinite energy. Really, in this case the energy over single period is some positive number. And we have infinitely many periods, so signal’s energy is not finite.

The most well known periodic signal is harmonic:

$$x(t) = A \cos(\omega t + \phi).$$

(2.6)

Here $\omega$ is angular frequency of harmonic and $\phi$ – initial phase. Prime period of this signal is $\omega/(2\pi)$.

All “normal” periodic signals may be expanded into Fourier series. In this context “normal” means that the signal satisfies Dirichlet conditions.

For discrete time signals definition of periodicity is identical only all the variables are integers instead of real numbers.

Below given example shows one of the possible ways of harmonic signal generation in Python. In similar way other kinds of signals may be generated. The example also calculates signal energy. This calculation is done using rectangle method of integration. Samples in the signal as regarded as sampled from corresponding continuous time signal. This way the energy is calculated from formula:

$$E = \Delta t \sum_{n=0}^{N-1} x^2(n).$$

(2.7)

Here $\Delta t$ is sampling interval (0.01 in our example), $x(n)$ are sampled values of the continuous time signal (this signal is cosinus in our example) and $N$ is the number of sample points in the signal.

```python
from numpy import *
import pylab as P

# time vector
x = cos(2*pi*3*t)  # harmonic signal

P.figure()
```
Figure 2.1. Example of generated harmonic signal

```
P.plot(t, x)
P.xlabel("time, s")
P.ylabel("Amplitude")
P.show()
E = 0.01*sum(x ** 2)
```

The result of the above example is a fig. 2.1 which shows the harmonic signal. We can also create general function for signal energy calculation. This function may be handy in this and following laboratory works. The code of the function is given below.

2.2 Signal spectra

Fourier transform is one of the most important tools in signal analysis. Spectral (Fourier) domain is dual to time domain. And so every operation in time domain may be performed in spectral domain.

A periodic signal satisfying Dirichlet conditions may be represented as a sum of harmonics of different functions. Dirichlet conditions in a non-strict sense require the signal to be “drawable” by
Fourier series representation of a signal is usually written in complex form:

$$x(t) = \sum_{n=-\infty}^{+\infty} c_n e^{2j\pi nf_0 t}.$$  \hspace{1cm} (2.8)

Here $f_0$ is equal to $1/T$ and $T$ is signal period. This expression is called Fourier synthesis form – signal is synthesized from a number of harmonics.

In case we have the signal $x(t)$ and want to obtain coefficients $c_n$, the formula is:

$$c_n = \frac{1}{T} \int_{-T/2}^{T/2} x(t) e^{-2j\pi nf_0 t} \, dt.$$  \hspace{1cm} (2.9)

This is analysis part of Fourier transform for periodic signals. As an example consider periodic rectangular pulse train given as:

$$x(t) = \sum_{n=-\infty}^{+\infty} p_{T_1/2}(t - nT_0).$$  \hspace{1cm} (2.10)

Here

$$p_T(t) = \begin{cases} 
1, & \text{if } |t| \leq T \\
0, & \text{if } |t| > T
\end{cases}$$

Fourier series expansion of this signal is given by:

$$c_n = f_0 T_1 \text{sinc}(nf_0 T_1), \quad f_0 = \frac{1}{T_0}.$$  \hspace{1cm} (2.11)

Amplitude spectrum of this complex signal is shown in fig. 2.2. The program, generating the picture of this spectrum, is given below.

```python
from numpy import *
import pylab as P
```
Figure 2.2. Fourier series of rectangular pulse train

\[ T_0 = 1 \]
\[ T_1 = 0.1 \]
\[ f_0 = \frac{1}{T_0} \]
\[ n = \text{arange}(-60, 61, 1.0) \] #Numbers of harmonics
\[ X = f_0 T_1 \text{sinc}(n f_0 T_1) \] #Fourier coefficients

P.figure()
P.stem(n, abs(X))
P.xlabel("Number of harmonics")
P.ylabel("Value of spectrum component")
P.show()

It is clear that the energy of a non-trivial periodic signal is infinite. So only average power may be calculated – it equals to the energy over one period divided by the length of the period. The average power may also be expressed using coefficients of Fourier series expansion. This possibility to calculate signal’s power from Fourier coefficients is described by Parseval’s theorem. The theorem states that:
General aperiodic signals cannot be expanded into Fourier series. In this case Fourier transform is used. Direct and inverse Fourier transforms are given by (2.13) and (2.14) formulas:

\[
X(f) = \int_{-\infty}^{+\infty} x(t) e^{-2j\pi ft} \, dt. \tag{2.13}
\]

\[
x(t) = \int_{-\infty}^{+\infty} X(f) e^{2j\pi ft} \, df. \tag{2.14}
\]

In practice the above given integrals may be calculated using numerical methods. To obtain the spectrum of a signal such calculation should be performed for many points in frequency domain. Computational complexity of this task is usually high – to be precise, the order of operations required is \(N^2\), where \(N\) is the number of the points in the signal. Fast Fourier transform comes to the rescue in such situations.

Discrete Fourier transforms are designed for calculation of harmonic expansion of discrete periodic signal. But in practice, if an analog signal is sampled with high enough sampling frequency, discrete Fourier transform gives good representation for signal’s spectrum. Original formulation of discrete Fourier transform also has quadratic computational complexity. Fast Fourier transform is an algorithm which gives the same result but in acceptable time (with \(N \log_2 N\) complexity).

An example which computes amplitude spectrum of a pulse signal is presented below. Signal itself is plotted in fig. 2.3 and spectrum is depicted in fig. 2.4. Notice that frequency range in fig. 2.4 is marked in such a way as to correspond to physical spectrum. This
is easy; we know that maximum physical frequency represented by FFT is \( fs/2 \). If you need more spectrum resolution in your research, zero signal may be appended to the original signal. Additional zeros will make frequency points closer to each other. Also observe that obtained approximation of the spectrum is quite precise – for example zeros of the spectrum are located at frequency points \( 10n \) Hz, where \( n \) is integer. Also, the shown spectrum is truncated – you can observe the full view as a result of the program execution.

```python
from numpy import *
import pylab as P

fs = 1000.0  # sample frequency
dT = 1/fs    # sampling time interval

# form single square pulse
x = zeros(1000)
x[0:100] = 1
X = abs(fft.fft(x))

P.figure()
P.plot(arange(len(x))/fs, x)
P.xlabel("Time, s")
P.draw()

P.figure()
P.plot(arange(len(X), dtype=float)/len(X)*fs, X)
P.xlabel("Frequency, Hz")
P.show()
```

Spectral analysis is especially useful when analyzing radio signals – signals that have a clear high frequency component. Majority of such signals may be expressed as:

\[
s(t) = u(t) \cos(2\pi ft).
\]  

(2.15)

Here \( u(t) \) is low frequency signal usually carrying information.
**Figure 2.3.** Rectangular pulse

**Figure 2.4.** Spectrum approximation of rectangular pulse
And frequency $f$ is relatively high. In telecommunications such signals should fit into some frequency band. Spectral analysis helps to determine what band is occupied by the signal.

The example given below shows how we can generate radio frequency signal and one possible method of bandwidth calculation is also shown. The bandwidth is calculated only from spectrum of positive frequencies. The assumption is made that power spectrum maximum is achieved at the central frequency. Algorithm computes the bandwidth such that all spectrum values outside it are smaller then 0.025 of maximum value. The limit of 0.025 is chosen arbitrary. This can be modified in an obvious way. The power spectrum of generated signal and the bandwidth is shown in fig 2.5.

```python
from numpy import *
import pylab as P

#X is assumed to be input power (energy) spectrum
#X should be single sided - no negative
#frequencies function returns the indices of
#points such that spectrum values outside these
#indices is smaller then 0.025 max spectrum value

def findIndicesOfBandwidth(X):
    #Find maximum value of power spectrum
    m = max(X)

    #Find the first index counting from
    #zero, such that spectrum(ind+1) > 0.025
    ind1 = 0
    while X[ind1+1] < 0.025*m:
        ind1 = ind1 + 1

    #Find the first index counting from
    #max frequency such that
    #spectrum(ind-1) > 0.025
    ind2 = len(X)-1
    while X[ind2-1] < 0.025*m:
```

39
ind2 = ind2 - 1

return (ind1, ind2)

fs = 1000.0  # sample frequency
dT = 1/fs  # sampling time interval

# form a single sinus pulse
x = zeros(10000)
x[0:500] = sin(2*pi*100*arange(0, 0.5, 0.001))

X = (abs(fft.fft(x)))**2 # compute power spectrum
X = X[0:len(X)/2]  # take only positive
# frequencies we don't need symmetric part

limitIndices = findIndicesOfBandwidth(X)
freqVector=arange(len(X), dtype=float)/len(X)*fs/2
limitedFreqVector = \
freqVector[limitIndices[0]:limitIndices[1]+1]

fig = P.figure()
ax = fig.add_subplot(111)
ax.plot(freqVector, X)
ax.fill_between(limitedFreqVector, 
X[limitIndices[0]:limitIndices[1]+1], color = ""r")
P.xlabel("Frequency, Hz")
P.show()

2.3 Laboratory work

Lab purpose

The purpose of the work is to get acquainted with the computer aided signal generation, their properties, spectrums of signals.
Lab task

1. Generate sinus and cosinus signals according to formulas 
   \[ s_C(t) = A \cos(2\pi ft), \quad s_S(t) = B \sin(2\pi ft) \]
   where \( t \) is discrete time with elements equally increasing from 0 to 1 (note that the length of vector \( t \) is inverse of sample frequency), \( f \) is frequency of the signals. Choose the sample frequency which ensures smooth representation of the signals. Observe the graphs of these functions when parameters \( A, B \) and \( f \) change. What happens if \( f \) is equal to half of length of \( t \)? Why?

2. Calculate energies of the signals \( s_C(t) \) and \( s_S(t) \) using formula 2.7. Does the energy change with changing frequency? Does the energy depend on the length of \( t \)?

3. Create the complex signal described as 
   \[ A_1 \cos(2\pi 1t) + B_1 \sin(2\pi 1t) + A_2 \cos(2\pi 2t) + B_2 \sin(2\pi 2t) + A_3 \cos(2\pi 3t) + B_3 \sin(2\pi 3t) \]. There should be possibility to change amplitude coefficients.
4. Estimate amplitude and power spectrum of the complex signal. Plot them (remember you may need to append zeros to original signal to obtain good resolution of spectrum frequencies). Describe the relation between amplitude coefficients of the complex signal and the corresponding spectrum.

5. Create phase modulated signal from pure \( \sin \) and \( \cos \) components. Initial phase of phase modulated signal should be one of 0, \( \pi/4 \), \( \pi/3 \), \( \pi/2 \). General phase modulated signal is written as \( C \cos(2\pi ft + \phi) \) where \( \phi \) is initial phase and amplitude \( C \) of this signal is not important for our purposes. So, you should choose \( A \) and \( B \) in \( A \cos(2\pi ft) + B \sin(2\pi ft) \) to satisfy equation \( C \cos(2\pi ft + \phi) = A \cos(2\pi ft) + B \sin(2\pi ft) \) or any given initial phase \( \phi \).

6. Create signals described by general formula 2.15. Information bearing signal \( u(t) \) should be a) rectangle pulse; b) triangle pulse; c) slow sinusoid. Determine bandwidth for each of these signals. Which of them has the narrowest band? The widest band?

**Control questions**

- What is the signal spectrum? What information it gives?

- How the energy of continuous time signal is calculated?

- How the energy of discrete time signal is calculated?

- What is the power (energy) spectrum? What information this spectrum carries?

- How can you determine the bandwidth of a given signal? Propose at least a few different methods.
Lab report

The report should contain answers to every question formulated in lab task. Plots of time-domain signals and their spectra should be made and explained.
Chapter 3

Random Variables and Processes

3.1 Random events and random variables

Probability theory describes events that have a component of a chance. It is a very practical discipline as many technological phenomena could not be described by deterministic models. Also, some of the current physical theories are inherently probabilistic, for example, quantum mechanics. In telecommunications probability theory is very important because random signals in the form of noise and interference are always present in real systems.

The notion of event is central to probability theory. Event is something what could occur as a consequence of an experiment. Let’s denote the event A. For example, in the coin-tossing experiment, event A may represent “number”. Suppose that \( n \) trials of the experiment are done and event A occurs \( k \) times. We may then calculate the relative frequency of the event A as \( k/n \). This relative frequency is obviously a non-negative real number between zero and one. In case \( A \) is impossible event, it will never occur and the relative frequency of such event is 0. On the contrary, if \( A \) is a certain event then relative frequency is 1. The limit of relative frequency is called the probability
of event A. It is expressed as:

\[ P(A) = \lim_{n \to +\infty} \frac{k}{n}. \] (3.1)

Probabilities of events may also be defined axiomatically. Axiomatic definition is theoretically more sound but is not as intuitive. Probability of the event represents likelihood that the event is going to happen as a result of an experiment.

Probabilities of events have some properties.

**Property 1.** \( P(\overline{A}) = 1 - P(A) \). \( \overline{A} \) is the event complementary to event A. This property is valuable when investigating the probability of non-occurrence of event A.

**Property 2.** If \( A_1, A_2, \ldots, A_M \) are mutually exclusive events and they exhaust all the event space then \( \sum_{i=1}^{M} P(A_i) = 1 \). From this property it follows that if all events are equiprobable then the probability of any such event is \( 1/M \).

**Property 3.** If A and B are any events then \( P(A + B) = P(A) + P(B) - P(AB) \). Here event A+B means “either A or B” and “AB” means “A and B together”.

Random events are well suited for description of classical, discrete phenomenons. Random variables do not possess these limitations and are a convenient tool for describing any random values.

Random variable takes some value as a result of an experiment. Random variables may be continuous or discrete. Discrete random variables may take only a discrete set of values. For example, random variable which describes coin tossing experiment. Continuous random variable may take any real value (we can also describe complex random variables).

To make some use of random variables, we need a method to describe them. Preferentially such method should work equally well for discrete as well as continuous random variables. Let us consider the random variable \( X \) and the probability of the event \( P(X < x) \). This event means the probability that random variable \( X \) takes a value smaller than \( x \) as a result of the experiment. It is clear we can
calculate this probability for any value of dummy variable $x$. This can be written as:

$$F_X(x) = P(X < x). \quad (3.2)$$

This function is called cumulative distribution function (cdf) of random variable $X$. Sometimes word “cumulative” is skipped.

Cumulative distribution function depends on the random variable $X$. Also, this function gives us all the information concerning the random variable. For any value of $x$, cdf is simply probability. Taking this into account it is easy to see that cdf has the following properties:

1. $F_X(x)$ as probability is bounded between 0 and 1.

2. $F_X(x)$ is monotonically non-decreasing.

For continuous random variables cumulative distribution function is not very informative. These variables may be much better characterized using so called probability distribution function (pdf) of the variable. Pdf is obtained from cdf by differentiating the latter:

$$f_X(x) = \frac{dF_X(x)}{dx}. \quad (3.3)$$

This function is much more intuitive because of the following property:

$$P(x_1 < X \leq x_2) = \int_{x_1}^{x_2} f_X(x) \, dx. \quad (3.4)$$

This property states that probability of random variable falling between $x_1$ and $x_2$ is equal to the area below $f_X(x)$ between these points. So the graph of pdf gives a clear view which values are the most probable for this random variable.

The main and quite obvious property of pdf is:
\[
\int_{-\infty}^{+\infty} f_X(x) \, dx = 1. \tag{3.5}
\]

This property follows directly from formula 3.4 and the fact that event \( P(-\infty < X \leq +\infty) \) is certain.

The notions of cdf and pdf may be generalized to multidimensional case.

In fig. 3.1 it is show the pdf of the most popular distribution in existence – Gaussian distribution. This distribution, also called normal distribution, is described by formula:

\[
f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m)^2}{2\sigma^2}}. \tag{3.6}
\]

It can be seen that Gaussian distribution depends on two parameters – the first one is a mean of the distribution – \( m \) and the second is a standard deviation of the distribution – \( \sigma \). In fig. 3.1 the mean is equal to 0 and the standard deviation is 1.

Every time we have a collection of numbers drawn from a distribution, we can empirically evaluate that distribution. Histogram may be used for this purpose. Histogram constructs an approximation of the theoretical pdf from which the numbers were drawn. This is done by calculating the number of points in bins. The example of code of generation of random numbers from Gaussian distribution is given below, and the histogram of numbers is constructed and drawn. The graph is in fig 3.2.

```python
from numpy import *
import pylab as P

#generate 10000 Gaussian random numbers with mean #= 0 and standard deviation = 1
x = random.normal(0, 1, 10000)
#bins for histogram construction
```
Figure 3.1. Gaussian probability distribution function

```
bins = arange(-5, 5, 0.2)
P.figure()
P.hist(x, bins, normed = True)
P.xlabel("x")
P.ylabel("empirical cdf")
P.show()
```

It is easy to see that empirical approximation of Gaussian pdf is very similar to theoretical pdf. The similarity depends on the number of factors – the number of points drawn from distribution and the number of bins used for histogram construction. Clearly, larger number of point gives better approximation. Wider bins give more continuous approximation but with less details.

Probability distribution function fully describes any random variable. But sometimes we do not need such a thorough description and only some parameters are sufficient.

An expected value or mean of random variable is defined as:
The mean is a center of the gravity of the figure formed by probability density function. Of course, this definition is purely theoretical. In practice we often have some values drawn from the distribution and these values may be used for approximate calculation of the mean:

$$m = \frac{1}{N} \sum_{i=0}^{N-1} x_i. \quad (3.8)$$

Here $N$ is the number of collected values and $x_i$ is the $i$-th value. In a similar way we can obtain a mean of any transformed random variable. If the random variable is transformed using function $g$ then the mean of transformed variable is:
\[ E_{g(X)} = \int_{-\infty}^{+\infty} g(x) f_X(x) \, dx. \] (3.9)

In this case there is no need to have pdf of random variable \( g(X) \). The task of obtaining pdf of \( g(X) \) may be quite complex analytically. Computer simulation may help in this case. For example fig. 3.3 shows approximation of pdf of \( X^2 \) when \( X \) is a Gaussian random variable. The algorithm of this calculation is given below. It is quite simple and consists of a few steps:

- Construct a big number of random variables from distribution \( f_X(x) \).
- Transform these variables using function \( g \).
- Draw the histogram of the transformed values. The histogram is an approximation of desired pdf.

Higher order moments may be defined in a similar fashion. Only second order moments will be discussed here. Of particular importance is the second central moment. It is given by:

\[ \text{var}[X] = E[(X - m)^2] = \int_{-\infty}^{+\infty} (x - m)^2 f_X(x) \, dx. \] (3.10)

In the formula \( m \) is the average of random variable \( X \). The defined quantity is also called variance. Square root of the variance is called standard deviation. The latter measures how wide the corresponding distribution is. The measurement unit of standard deviation is the same as that of the original random variable. If variance approaches zero, the random variable becomes conventional constant value.

Natural question may arise, why Gaussian distribution is everywhere. The answer is given by central limit theorem. Informally, it
states that the sum of a big number of similar random variables is distributed normally. Consider any random physical process. In many cases the outcome is influenced by many factors, so Gaussian value may arise as a result.

3.2 Random processes

Random process can be written as $X(t)$. It means that for every value of argument $t$ we have some random variable.

It is possible to look at random process from the other point. We can observe the outcome of the process for some time and the result is deterministic time function (it is called realization of the process). It is important to understand that the realization is deterministic only after observation. Before observation we can not predict the exact form of realization function.

How can we describe a random process? The most basic and theoretical way of description is using joint probability density functions for every possible argument. These joint distribution functions may

\textbf{Figure 3.3.} Histogram of squared Gaussian random numbers
be written as:

\[ f_{X(t_1), X(t_2), \ldots, X(t_n)}(t_1, t_2, \ldots, t_n), \]

(3.11)

and we need such functions for every \( n \) and for every combination of \((t_1, t_2, \ldots, t_n)\). It is clear that in general this is impractical. Fortunately in practice many random processes may be described using much simpler methods.

Some random processes may be described analytically using only some random variables. For example, consider digital binary phase shift keying modulation. In this case binary “0” is transmitted using waveform \( \cos(2\pi f_0 t) \) and binary “1” is transmitted using waveform \( \cos(2\pi f_0 t) \). We can fully describe such process only knowing probabilities of “0” and “1”. In telecommunications such processes are common but they do not exhaust all interesting processes.

Probably the widest class of interesting processes may be described using average values of various kinds. In many cases these average values don’t give us a full description of the process, but often we even don’t need that full description.

Mean value of the process is calculated as:

\[ m(t) = E[X(T)]. \]

(3.12)

Notice how mean is a function of time variable.

Autocorrelation of the process is two dimensional function defined as:

\[ R_{XX}(t_1, t_2) = E[X(t_1)X(t_2)]. \]

(3.13)

Autocorrelation function characterizes how similar are process values at two different time moments.

In case our process operates in discrete time, continuous time variable \( t \) should be substituted for discrete time variables \( n \), and the definitions remain the same.

Sometimes it is clear that statistical properties of the process do not change with time. Such process is stationary in a strict sense.
There is also not so strict stationarity condition. Process is called wide sense stationary if it has a constant mean value and autocorrelation function depends only on time difference between moments $t_1$ and $t_2$. Such autocorrelation function may be written as:

$$R_{XX}(\tau) = E[X(t)X(t + \tau)].$$ (3.14)

Wide sense stationary processes are very important in telecommunications because majority of noises are described as such processes.

Often we only have one or a few realizations of the processes and from these some characteristics should be calculated. Not every stationary process is suitable for this. In case the realization of the process may be used for calculation of statistical averages, the process is called ergodic. When the process is ergodic, process mean value is calculated using formula:

$$m = \frac{1}{T} \int_0^T x(t) \, dt.$$ (3.15)

Here $x(t)$ is realization function of the process and $T$ is measurements interval over which the realization was observed.

Similarly, autocorrelation function is also obtained using time average:

$$R(\tau) = \frac{1}{T} \int_0^{T-\tau} x(t)x(t + \tau) \, dt.$$ (3.16)

When we deal with discrete variables, formulas 3.15 and 3.16 become:

$$m = \frac{1}{N} \sum_{n=0}^{N} x(n).$$ (3.17)

and:
\[ R(k) = \frac{1}{T} \sum_{n=0}^{N-k} x(n)x(n+k). \] (3.18)

Full correlation function may be obtained repeating the calculation from 3.18 for every value of \( k \). The continuous process is called white noise process if its autocorrelation function is Dirac delta impulse (\( \delta(t) \)). Discrete time process (noise) is considered to be white if its autocorrelation function is simply unit impulse.

The code example given below illustrates generation of correlated random process (to be exact, only one realization of the process is generated) and calculation of the autocorrelation function.

```python
from numpy import *
import pylab as P

N = 1000  # length of the signals

# generate non correlated random process
x = random.normal(0, 1, N)

# generate correlated random process for this purpose uncorrelated process is filtered using simple IIR filter vector y contains correlated process
y = zeros(N, dtype = float)
for i in xrange(1, N):
    y[i] = x[i] + 0.95*y[i-1]

# calculate correlation function for the process
R=zeros(N, dtype=float)  # contains function points
for k in xrange(N):
    sum = 0  # holds temporary sum value
    for n in xrange(N-k):
        sum = sum + y[n]*y[n+k]
    R[k] = 1.0/N * sum

# now we can plot the process itself and its
```
# autocorrelation function

```python
P.figure()
P.plot(y)
P.xlabel("discrete time")
P.ylabel("signal value")
P.draw()

P.figure()
P.plot(R)
P.xlabel("discrete time")
P.ylabel("autocorrelation value")
P.show()
```

Take notice that the given algorithm for autocorrelation computation has quadratic complexity of the input data size. So be careful when increasing \( N \) – calculation time may become unacceptable. The results of the program shown above are given in fig. 3.4 and 3.5. Fig. 3.4 shows the realization of a random process and fig. 3.5 shows autocorrelation function of the same realization. In theory autocorrelation function should be much smoother, but any quantity obtained from finite data has some randomness. The same randomness may be observed in this empirical (calculated from data) correlation function. Even from such a graph we can draw a conclusion that the process is highly correlated until time moment 20.

Autocorrelation function has some interesting properties:

- Value of autocorrelation function at time moment zero is equal to average power of the signal. Also this value is non-negative:
  \[
  R_{XX}(0) = E[X^2(t)] = \text{Average power} \geq 0.
  \]

- Autocorrelation is an even function (symmetric):
  \[
  R_{XX}(\tau) = R_{XX}(-\tau).
  \]

- Autocorrelation function has absolute maximum at zero:
  \[
  R_{XX}(0) \geq R_{XX}(\tau) \text{ for all } \tau.
  \]

Autocorrelation gives the important information about random
Figure 3.4. Realization of correlated random process

Figure 3.5. Example autocorrelation function
process in time domain. In frequency domain similar role is played by power spectral density.

Power spectral density is defined as the direct Fourier transform of autocorrelation function:

\[
S_{XX}(f) = \int_{-\infty}^{+\infty} R_{XX}(\tau) e^{\textcolor{red}{-}j2\pi f \tau} d\tau. \quad (3.19)
\]

The inverse relation is also true:

\[
R_{XX}(\tau) = \int_{-\infty}^{+\infty} S_{XX}(f) e^{j2\pi f \tau} df. \quad (3.20)
\]

Evaluating 3.20 at time moment 0 we get:

\[
R_{XX}(0) = \int_{-\infty}^{+\infty} S_{XX}(f) df = \text{Average power.} \quad (3.21)
\]

From this formula it is clear that power spectral density gives the distribution of power in the process. These formulas also give the algorithm for power spectral density calculation – find the autocorrelation function and apply Fourier transform to it. The result will be power spectral density.

Similar to autocorrelation function, power spectra density (PSD) has some dual properties:

- PSD is real and non-negative function.
- PSD is symmetric function: \( P_{XX}(f) = P_{XX}(-f) \).
- The power of the process in a finite bandwith between \( f_1 \) and \( f_2 \) is given by: \( P[f_1, f_2] = \int_{f_1}^{f_2} P_{XX}(f) df \).
When dealing with discrete signals, principles are the same and the form of formulas also remains very similar. In discrete case, the PSD may be evaluated directly from the signal using periodogram method. This method is based on the fast Fourier transform and may be described as follows:

- Divide the signal into overlapping intervals of fixed length. Usually 50 % overlap is chosen.
- Apply windowing function to every segment of the signal (for example Hamming window function).
- Calculate fast Fourier transform for every segment of the signal.
- Square the amplitudes of calculated FFTs.
- Average all the squared spectrums.

The code implementing this algorithm is given below. The periodogram calculation algorithm is located in function “periodogram”. The function is tested on the same type of the signal as used for the previous example. The plot of periodogram is given in fig. 3.6. It can be seen that the result is quite smooth power spectrum estimation and clearly low frequencies are dominating in the signal. The same domination of low frequencies may be seen from time graph of the signal. Also note, that frequencies in the plot of periodogram do not have any physical meaning. The physical meaning to them could be assigned if sample rate of time-domain signal would be known.

```python
from numpy import *
import pylab as P

#the function calculated periodogram x - input
#signal M - the number of points to use for
#signal segments
def periodogram(x, M):
    #this variable holds accumulated
# periodogram data
sumPeriodogram = zeros(M)
currentIndex = 0
# check if we need to take one more segment
while currentIndex + M <= len(x):
    data = x[currentIndex:currentIndex+M].copy()
    # apply Hamming window function to data
    data = data * hamming(M)
    # calculate amplitude spectrum squared
    sp = abs(fft.fft(data)) ** 2
    # update periodogram data
    sumPeriodogram = sumPeriodogram + sp
    # update loop variable
    currentIndex = currentIndex + M/2

# average periodogram data
sumPeriodogram = sumPeriodogram/(len(x)/M*2)
return sumPeriodogram

N = 10000 # length of the signals

# generate non correlated random process
x = random.normal(0, 1, N)

# generate correlated random process for this purpose uncorrelated process is filtered using simple IIR filter
y = zeros(N, dtype = float)
for i in xrange(1, N):
    y[i] = x[i] + 0.95*y[i-1]

# calculate periodogram of correlated data
periodogramOfData = periodogram(y, 1000)

P.figure()
P.plot(periodogramOfData)
P.xlabel("Frequency")
3.3 Laboratory work

Lab purpose

The purpose of this lab is to get acquainted with the computer generation of random variables and random signals, measurement of statistical parameters.

Lab task

1. Form an array (let it be $x$) of normally distributed random values. Standard deviation should equal $A$ and mean should be 0. Make it possible to change $A$ and the length of data array (let it be $N$). Plot the histogram of the data when $N =$
10, 100, 1000, 10000. What conclusion can be made about the influence of the amount of data on histogram quality?

2. Estimate and plot the power spectrum density of \( x \), when \( N = 100000 \). How would you describe the estimated PSD?

3. Empirically prove the central limit theorem. For this purpose form \( n \) arrays of uniformly distributed random numbers. Call the \( i \)-th array \( X_i \). Calculate the sum of these arrays: \( X = \sum_{i=0}^{n-1} X_i \) and estimate the histogram of \( X \). What is the smallest \( n \) when you can not distinguish the estimated histogram from Gaussian distribution?

4. Form a correlated sequence according to formula:
   \[
y(k) = \sum_{i=0}^{K-1} x(k - i)h(i).
\]
   The number of coefficients \( h(i) \) may vary. Consider \( x(i) = 0 \) if \( i < 0 \). Note that this is signal filtering using finite impulse response filter. What is the distribution of \( y \)?

5. Estimate power spectral density of signal \( y \). For this purpose the length of the signal should be above 10000.

6. Estimate autocorrelation function of the correlated sequence \( y \). How many non zero elements are there? Hint – there is no need to estimate elements, which are zero theoretically.

7. Can you choose \( h(i) \) in such a way that the resulting PSD would contain only low frequencies or only high frequencies? What are these coefficients for both cases?

8. Estimate mean and standard deviation of the correlated sequence \( y \). Repeat this estimation 1000 times and gather the results. Plot the histogram of these results (separately for mean and standard deviation). Does the shape of the histogram depend on the length of \( y \)? If so, what is the dependence?
Control questions

- What is the difference between random and deterministic signals?
- What are the definitions of mean, sample mean, variance and sample variance?
- What information gives the power spectral density?
- What is the difference between spectral analysis using deterministic and random signals?
- What is normal random process? What is white noise? Should white noise be normally distributed?
- What is the shape of autocorrelation function of white noise?
- What is the relation between autocorrelation function and power spectral density function?
- What influence does the amount of data have on sample statistical estimates?
- Does linear filtering change the shape of autocorrelation function?

Lab report

The report should contain answers to every question in the lab task. Plots of the key signals, data, etc. should be made and explained.
Chapter 4

Data Transmission, Modulation and Detection

4.1 Baseband data transmission

There are two distinct data transmission schemes – baseband and passband transmission. Baseband transmission directly transmits digital information in baseband channel. Baseband channel by definition includes zero frequency. Passband transmission, on the other hand, uses digital modulation techniques and the passband channel is used – this channel does not include zero frequency and in practice is narrow band channel.

Digital data usually have a significant low-frequency component. Therefore a low-pass channel with a wide enough bandwidth is necessary to accommodate the main frequency content of the data. Typically the channel has limited bandwidth – in practice its frequency response may be considered zero outside of some frequency. Such channel results in distortion of the shape of transmitted pulse. Moreover, adjacent pulses affect each other and this undesired influence is called intersymbol interference (ISI). Intersymbol interference is one of the main problems in baseband data transmission. To control ISI, the pulse transmitted must be shaped accordingly.
Channel noise also affects the quality of data reception. This influence is relatively small (compared to passband transmission) because higher signal to noise ratios may be obtained relatively easy.

Baseband and passband modulation schemes have many in common. The main principle of modulation is the same. It is assumed that some stream of symbols is present at the transmitter input. Let it be \( \{s_0, s_1, ..., s_n, \ldots\} \). Every symbol may have one of \( M \) values. In practice symbols are formed from a stream of bits and thus \( M = 2^k \) for some value \( k \). This means that \( k \) adjacent bits are grouped into one symbol.

Further, symbol stream at the input is mapped onto the signal suitable for transmission. Different signal forms are used for different symbols. Every signal form lasts \( T \) second. It means that every \( 1/T \) seconds one symbol is transmitted. Thus, bit rate is \( k/T \). Modulation schemes differ in a way the symbols are mapped onto corresponding symbols.

We will take a look at PAM (Pulse Amplitude Modulation) modulation scheme for baseband transmission. Sometimes word “modulation” is reserved only for passband transmission and word “signalling” is used for baseband transmission. We will use these words interchangeably.

PAM modulation defines symbol waveforms as:

\[
s_m(t) = A_mp(t). \tag{4.1}
\]

Here \( A_m, 0 \leq m \leq M - 1 \) are different amplitudes. Usually the amplitudes take discrete values \( \pm 1, \pm 3, ..., \pm M - 1 \). It should be clear that the overall shape of a waveform depends on \( p(t) \). PAM modulation signal is in baseband only if \( p(t) \) is in baseband too. The most frequently used form for \( p(t) \) (at least in theory) is a rectangular pulse. The program which generated such binary PAM signal is given below. 10 random bits are generated and the corresponding PAM signal is plotted. Fig. 4.1 shows the signal.

```python
from numpy import *
import pylab as P
```
#!/usr/bin/env python

#bits - the stream of input bits T - duration of one bit dT - sample time interval dt should be smallest than T function returns a tuple of time vector and full PAM signal

def generateBinaryPam(bits, T, dT):
    t = arange(0.0, len(bits)*T, dT) #time vector

    pamSignal = zeros(len(t), dtype = float) #for every bit generate a signal and put it at the appropriate place #in the output signal
    for i in xrange(len(bits)):
        if bits[i] == 0:
            pamSignal[int(i*T/dT):int((i+1)*T/dT)]= -ones(int(T/dT))
        else: #bits[i] == 1
            pamSignal[int(i*T/dT):int((i+1)*T/dT)]=ones(int(T/dT))
    return (t, pamSignal)

#generate 10 random bits
randomBits = random.random_integers(0, 1, 10)

#generate PAM signal with 1 Hz frequency
(t, signal)=generateBinaryPam(randomBits,1,0.02)

#plot the signal
P.figure()
P.plot(t, signal)
P.xlabel("Time, s")
P.ylabel("Signal amplitude")
P.show()

The binary PAM signals have a property that $S_0(t) = -S_1(t)$. Such signals are called antipodal signals. These signals are the most resistant to noise.

Binary PAM signal is conceptually one of the simplest signals
Figure 4.1. Binary PAM signal

for information transmission. Majority of questions about the signal are answered just by looking at the time plot of it. The only other interesting question is what bandwidth does this signal occupy? The answer to this question may be given analytically. We could calculate the correlation function of the signal and the inverse Fourier transform of it would give the power spectrum. In the lab our preferred tool is computer simulation. So, we will try to estimate the bandwidth using FFT and a simple algorithm for bandwidth estimation. Similar algorithm was analyzed in Chapter 2, but this time our signal has power spectrum around zero. And it is not symmetric. So, that algorithm should be changed a bit. All the code of simulation is given below. The code uses modified periodogram function from Chapter 3. The function “generateBinaryPam” is used but in the listing it is not shown – it can be copied and added from the previous example. The result is a plot of the power spectrum of PAM signal and the bandwidth plotted. It is shown in fig. 4.2. From the picture we can see that in order to transmit binary PAM with 1 bit/s, the bandwidth needed is a little less that 2 Hz.
from numpy import *
import  pylab as P

#X is assumed to be input power (energy) spectrum
#X is assumed to be spectrum of baseband signal,
#so, bandwidth starts at zero frequency function
#returns the indice of point such that all
#further indices have smaller value then 0.025
#of maximum spectrum value

def findIndexOfBandwidth(X):
    #Find maximum of power spectrum
    m = max(X)

    #Find the first index from the
    #right such that spectrum(ind-1) > 0.025
    ind = len(X)-1
    while X[ind-1] < 0.025*m:
        ind = ind - 1
    print ind
    return ind

#the function calculated periodogram x - input
#signal M - the number of points to use for
#signal segments
def periodogram(x, M):
    sumPeriodogram = zeros(M/2, dtype = float)
    currentIndex = 0
    while currentIndex + M <= len(x):
        data=\
        x[currentIndex:currentIndex+M].copy()
        #apply Hamming window function to data
        data = data * hamming(M)
        #calculate amplitude spectrum squared
        sp = abs(fft.fft(data)) **2
        sp = sp[0:M/2]
        #update periodogram data
        sumPeriodogram = sumPeriodogram + sp
        currentIndex = currentIndex + M/2
# average periodogram data
sumPeriodogram = sumPeriodogram/(len(x)/M*2)
return sumPeriodogram

# generate 100 random bits
randomBits = random.random_integers(0, 1, 1000)
# generate PAM signal with 1 Hz frequency
dT = 0.02  # sample interval
(t,signal)=generateBinaryPam(randomBits,1,dT)
X = periodogram(signal, 1000)
limitIndex = findIndexOfBandwidth(X)
print len(X), limitIndex

# form the frequency vector for labelling of axes
# max frequency should be 1/2 of sample frequency
freqVector=arange(len(X),dtype=float)/len(X)*(0.5/dT)
limitedFreqVector = freqVector[0:limitIndex+1]

fig = P.figure()
ax = fig.add_subplot(111)
ax.plot(freqVector, X)
ax.fill_between(limitedFreqVector, X[0:limitIndex+1], color = "r")
P.xlabel("Frequency, Hz")
P.show()

4.2 Passband modulation

In this part we are going to review binary phase shift keying (BPSK) and binary frequency shift keying (BFSK). Analytic expressions of these modulations will be given, time graphs will be plotted and power spectra will be estimated.
4.2.1 BPSK modulation

In BPSK modulation we have two signals – one for bit zero and the other for bit one. These signals are presented in formulas 4.2 and 4.3:

\[ s_0(t) = A \cos(2\pi f_0 t), \]  
\[ s_1(t) = A \cos(2\pi f_0 t + \pi). \]

Here \( f \) is the carrier frequency of modulated signals, \( 0 \leq t < T \) and \( T \) is the duration of one bit. Carrier frequency does not have any influence on the performance of modulation. It is easy to see that BPSK signals are antipodal.

The frequency shift keying modulation scheme may be easily adapted to multilevel modulation. In such case we would require multiply different phase angles – one for every symbol in transmission alphabet.

A function for BPSK generation is given below. It is very similar to PAM generating function. The graph of BPSK symbol is presented.
in fig. 4.3. The parameters of the plotted signal are: frequency – 3 Hz, duration of bit – 1 s.

```python
from numpy import *
import pylab as P

#X is assumed to be input power (energy) spectrum
#X is assumed to be spectrum of baseband signal,
#so, bandwidth starts at zero frequency function
#returns the indice of point such that all
#further indices have smaller value then 0.025
#of maximum spectrum value

def findIndexOfBandwidth(X):
    #Find maximum of power spectrum
    m = max(X)
    #Find the first index from the
    #right such that spectrum(ind-1) > 0.025
    ind = len(X)-1
    while X[ind-1] < 0.025*m:
        ind = ind - 1
    print ind
    return ind

#the function calculated periodogram x - input
#signal M - the number of points to use
#for signal segments
def periodogram(x, M):
    sumPeriodogram = zeros(M/2, dtype = float)
    currentIndex = 0
    for i in range(len(x)-1):
        sumPeriodogram[currentIndex] = x[i]
        currentIndex = currentIndex + 1

    return sumPeriodogram
```

Power spectrum of BPSK signal is estimated in the same way as it was done for PAM signal. The only difference is that bandwidth is centered around carrier frequency, so function for bandwidth estimation from chapter 1 is used. For power spectrum estimation, signal parameters were: carrier frequency – 10 Hz, and bit duration – 1 s. The estimated power spectrum is plotted in fig. 4.4. It can be seen
that one side of power spectrum of BPSK is almost exactly the same as that for PAM modulated signal. Can you explain this phenomenon (consider this explanation a question for lab)? Also compare fig. 4.2 and fig. 4.4. Bandwidth of BPSK modulated signal is approximately twice the bandwidth of PAM signal. Why (this is also an additional question for the lab)?

4.2.2 BFSK modulation

BFSK modulation changes the carrier frequency in every bit interval. BFSK signals (corresponding to 0 and 1) are described as:

\[ s_0(t) = A \cos(2\pi f_0 t), \]  
\[ s_1(t) = A \cos(2\pi f_1 t) \]  
and \( f_i \) are chosen using formula:
Figure 4.4. Power spectrum of binary BPSK

\[ f_i = \frac{n + i + 1}{T}. \]  \hspace{1cm} (4.6)

Here \( i = 0, 1 \) and \( n \) is some fixed integer, \( T \) is duration of one bit. From formula 4.6 it is clear that both frequencies are separated by \( 1/T \) Hz. For example, for 1 Mbit/s bitrate, carriers should be separated by 1 Mhz. Really this separation could be multiply of \( 1/T \), but this only expands spectrum of modulated signal without any additional benefits. Question may be raised, whether smaller separation is possible? The answer is no. Because the condition 4.6 guarantees that signals representing 0 and 1 are orthogonal. Orthogonality is necessary for effective BFSK data transmission.

BFSK modulation can be easily generalized for \( M \)-ary modulation. In this case \( M \) different frequencies should be used for signal formation.

Function for BFSK generation is very similar to BPSK generating one. It is given below.

\#bits - the stream of input bits  \ T - duration of
#one bit dT - sample time interval dt should be
#smallest than T f0 - carrier frequency of "0" bit
#f1 - carrier frequency of "1" bit abs (f0-f1)
#should be equal to 1/T function returns a tuple
#of time vector and full BFSK signal

def generateBinaryBFSK(bits, T, dT, f0, f1):
    t = arange(0.0, len(bits)*T, dT) #time vector
    bfskSignal = zeros(len(t), dtype = float) #for every bit generate a signal and put
    #it at the appropriate place in output signal
    for i in xrange(len(bits)):
        if bits[i] == 0:
            bfskSignal[int(i*T/dT):int((i+1)*
            T/dT)] = cos(2*pi*f0*t[0:T/dT])
        else: #bits[i] == 1
            bfskSignal[int(i*T/dT):int((i+1)*
            T/dT)] = cos(2*pi*f1*t[0:T/dT])
    return (t, bfskSignal)

BFSK signal is shown in fig. 4.5. Bit duration is 1 s and frequencies used are 2 Hz and 3 Hz. The first and third bits are different from the second bit. It can be seen noticing different frequencies in corresponding time intervals.

Power spectrum is evaluated the same way as for BPSK signal. The resulting power spectrum is plotted in fig. 4.6. Parameters for this estimation were: bit duration – 1 s and frequencies used – 9 Hz and 10 Hz. The bandwidth of frequency modulated signal is narrower than that of phase modulated signal. Can we conclude that BFSK is unambiguously better modulation scheme?

4.3 Optimal receivers for AWGN channel

AWGN (Additive White Gaussian Noise) channel is the most widely adopted channel model in telecommunications. Despite the fact, that
Figure 4.5. BFSK modulated signal

Figure 4.6. Power spectrum of binary BFSK
real channels are not AWGN, at least the part of receivers is often duplicated in real world equipment.

AWGN channel has infinitely wide bandwidth and the only distortion is Gaussian white noise. Consider we have an alphabet of $N$ symbols. A waveform is assigned to each symbol. Then, for every symbol interval, the following signal is received at the receiver:

$$x(t) = s_i(t) + w(t). \quad (4.7)$$

Here $x(t)$ is the received signal, $s_i(t)$ – one of $N$ possible transmitted signals, $w(t)$ – white Gaussian noise with power spectral density of $N_0/2$. The task of the receiver is to make decision, which one of $N$ symbols has been transmitted. There are two popular optimal schemes for this task. The first one is called correlation receiver and it is depicted in fig. 4.7.

The algorithm of correlation receiver may be described as follows. Multiply the input signal with the signal corresponding to $i$-th symbol and calculate the integral of this product over symbol period:

Figure 4.7. Correlation receiver
Here $d_i$ are decision variables. The value of transmitted symbol is decoded by choosing the symbol, corresponding to the biggest $d_i$:

$$\text{Decoded symbol} = \arg\max_i (d_i).$$

(4.9)

There is also an equivalent optimal decoding scheme called matched filter. It is shown in fig. 4.8.

Matched filter uses $N$ analog filters with impulse responses defined as:

$$h_{MFi} = s_i(T - t).$$

(4.10)

Impulse response of every matched filter is reversed and translated waveform of the $i$-th signal. It is easy to see, that at time mo-
Figure 4.9. Matched filter receiver for binary band-limited transmission

ments \( kT \), where \( k \) is integer, the output from the matched filter is exactly the same as that from the correlation receiver. So, the two decision schemes are equivalent.

Algorithm of optimal reception is exactly the same for baseband and passband signals. Further in the chapter we are going to focus on baseband transmission model. Moreover, every passband transmission model has a baseband equivalent – it should be no surprise as passband transmission usually only shifts baseband data to higher frequencies.

The channel model in derivation of correlation receiver (matched filter receiver) has infinite bandwidth. But even when channel has limited bandwidth, the same method of receiver design is used – impulse response of the receiver should be matched to the signal after channel. Let us note one more fact – when there are PAM signals, only one receive filter is sufficient. For example for binary PAM decision is made in favor of “0” if the output from receive filter is below zero. Otherwise the “1” is selected. Such more realistic system is shown in fig. 4.9.

The presented model has only one new element – channel filter. Input bits are assumed to be polar delta impulses, so the signal after
transmit filter (sometimes called shaping filter) has the same shape as \( g(t) \). I.e. \( g(t) = s(t) \). In the ideal world, receive filter is matched to signal \( x_0(t) \).

Further, we are going to show the impact of the channel filter on resulting intersymbol interference.

At first, let’s take a look at the operation of scheme in fig. 4.9 when channel filter is unlimited and transmitted pulses are rectangular also, there is no noise in this simulation. The program code for such simulation is given below. The code uses already used function “generateBinaryPam”. The new function is “firFilter”, which filters the signal using given impulse response. This function is needed for simulation of matched filter.

```python
from numpy import *
import pylab as P

# filter signal x using impulse response h use 
# zero initial conditions, i.e. x(-1) = 
# x(-2) = ... = x(-n) = 0
def firFilter(x, h):
    y = zeros(len(x), dtype = float)
    for i in xrange(len(y)):
        if i < len(h):
            # initial conditions
            # still "active"
            y[i] = sum(x[0:i]*h[0:i[::-1]])
        else:
            y[i] = sum(x[i-len(h):i]*h[::-1])
    return y
```

```python
randomBits = random.random_integers(0, 1, 5)

dT = 0.01  #sample interval
(t,signal)=generateBinaryPam(randomBits,1,dT)
```

```python
#for square pulse signal matched filters impulse
#response is also square pulse multiplyer 0.02
#only needed for a visually clear plot multiplyer
```
# does not change the shape of a filter so it has zero impact on performance of the receiver

\[
\text{matchedImpResponse} = \text{ones(100)}*0.02
\]

\[
\text{signalAfterMatchedFilter} = \text{firFilter} (\text{signal}, \text{matchedImpResponse})
\]

The results of program execution are shown in fig. 4.10. Receiver makes decision using signal from matched filter at time instants 1, 2, ..., 5 s. It is clear that there is no ISI in such circumstances. Presence (or absence) of ISI may be confirmed looking at so called eye diagram. Eye diagram is generated by repeatedly plotting one or several periods of matched filter output. In real world oscilloscope may be used for this purpose. We, of course, generate eye diagram in software. The function for eye diagram plotting is given below:

```python
# function plots eye diagram x - signal for plotting t - time vector for one plotting period # usually should contain multiply of the symbol periods
def plotEyeDiagram(x, t):
    P.figure()
    for i in xrange(len(x)/len(t)):
        dataForOnePeriod = x[i*len(t):(i+1)*len(t)].copy()
        P.plot(t, dataForOnePeriod, 'b')
    P.xlabel("Time, s")
    P.show()
```
The eye diagram of previously described scenario is given in fig. 4.11. It can be seen that the eye is wide open. This means zero ISI. Parameters for this plot were – waveform rectangular with 1 s duration for every bit, 1000 random bits were used.

Up till now infinite bandwidth was used in simulations. Almost all real channels have finite bandwidth. How are we going to simulate finite bandwidth? By using low pass filter. Assume we have a signal $x$ and want to filter it. In the first step ideal frequency response is constructed and coefficients of impulse response are obtained from it. Then, using these coefficients, filtering in frequency domain is performed. This involves padding impulse response and the signal with zeros (padding is necessary to avoid circular convolution), computing FFTs of both padded signals, multiplying FFTs and calculation of inverse FFT transform. Only middle part of this signal is desired filtered signal.

Implementation of FFT filtering procedure is given in the code below:

```python
#low pass filter x - input signal f0 - cutoff
```
def fftFilter(x, f0, dT):
    #design impulse response using window
    #method rectangular windos is good for
    #our purposes length of impulse response
    #is the same as that of signal it is
    #assumed even. Runtime error will be
    #generated otherwise
    fcut = f0/(1.0/dT)
    n = arange(len(x), dtype = float)
    h = sin(2*pi*fcut*(n-(len(n)-1.0)/2.0))/
    (pi*(n-(len(n)-1.0)/2.0))

    #extend signal and impulse response by
    #zeros this is done in order to avoid
    #the effects of circular convolution

Figure 4.11. Eye diagram of rectangular PAM and infinite bandwidth

#frequency (physical) dT - sampling interval of
#signal maximum physical frequency is 0.5/dT
#function returns filtered signal with high
#frequencies removed
xpadded = hstack((x, zeros(len(x))))
hpadded = hstack((h, zeros(len(h))))

#perform filtering in frequency domain
X = fft.fft(xpadded)
H = fft.fft(hpadded)
xfilt = real(fft.ifft(X*H))

#we need only part of filtered signal
#the one which takes into account delay
#introduced by filter taking this part
#of the signal we compensate for the delay
xfilt = xfilt[len(x)/2:
len(x)+len(x)/2].copy()
return xfilt

What happens when PAM signal is transmitted through the band limited channel? The answer is presented in fig. 4.12 and the code for this simulation is given below. We should notice that the receive filter is still rectangular, because we assume naive design of the system. And this experiment tries to show the drawbacks of such naive design.

randomBits = random.random_integers(0, 1, 1000)
#generate PAM signal with 1 Hz frequency
dT = 0.01    #sample interval
(t,signal)=generateBinaryPam(randomBits, 1, dT)

#the important part - rectangular pulses are filtered
signal = fftFilter(signal, 0.6, dT)

matchedImpResponse = ones(100)*0.02
signalAfterMatchedFilter = firFilter(signal, \matchedImpResponse)

#don't take the first few symbols to allow
dissapear the effect of initial conditions
**Figure 4.12.** Eye diagram of rectangular PAM when bandwidth is finite

```python
plotEyeDiagram(
    signalAfterMatchedFilter[int(1/dT)*5:-1], 
    t[0:int(2/dT)])
```

From fig. 4.12 we can see that this time ISI is significant. Notice that this simulation was also performed using zero additional noise. As it can be seen from above given code, bandwidth used for simulation is 0.6 Hz. And it is a known result that zero ISI signals may be designed if bandwidth is 0.5 Hz for 1 bit/s datarate.

How is it possible to design the signals with zero (or very small) ISI? The theory concerning this question is quite involved but the answer lies in choosing the right form for impulse response of filter $g(t)$ in matched filter receiver. And we know that this filter is responsible for the shape of transmitted pulse.

One possible choice for $g(t)$ is so called square root raised cosine filter. Receive filter should be matched to $g(t)$, so it also has the same impulse response. Square root raised cosine impulse response is physically non-realizable, so some truncation and delay should be
used. Square root raised cosine impulse response is described using formula:

\[
g(t) = \frac{2\beta}{\pi \sqrt{T}} \cdot \frac{\cos [(1 + \beta)\pi \frac{t}{T}] + \sin \left( \frac{(1-\beta)\pi \frac{t}{T}}{4\beta \frac{t}{T}} \right)}{1 - (4\beta \frac{t}{T})^2}.
\]  (4.11)

Here \( T \) is duration of the symbol and \( \beta \) – bandwidth parameter.

This pulse for a few different values of \( \beta \) is shown in fig. 4.13. Parameter \( \beta \) determines the bandwidth of the pulse – the spectrum of the pulse is zero for frequencies outside of bandwidth \((1 + \beta)/(2T)\). For \( \beta = 0 \) the pulse becomes sinc function.

It may seem that sinc function is the best one for pulse design, because the spectrum of it is the narrowest. But sinc has a very slow decay (with the rate \( 1/t \)) and this causes bigger ISI if synchronization is not perfect.

Now let us examine whether square root raised cosine filter improves ISI situation in the experiment identical to the previous
The only difference is that pulses are not rectangular, but has a waveform of square root raised cosine. For this experiment we need a few additional functions. The first is needed for the generation of pulse waveform. And the second is needed for generation of the signal corresponding to given bit stream. These functions are given below. Also, don’t forget that impulse response of matched filter should also be the same as $g(t)$ and this is the only change in the main program (not shown below). Also, in data detection, delay should be introduced. It depends on the duration of generated waveform.

```python
# t - time vector for function generation T -
# duration of symbol (bit) beta - frequency
# coefficient of squre root raised cosine waveform

def squareRootRaisedCosine(t, T, beta):
    nominator = cos( (1+beta)*pi*t/T ) + \
    sin( (1-beta)*pi*t/T )/(4*beta*t/T)
    denominator = (1 - (4*beta*t/T)**2)
    g = 2*beta/(pi*sqrt(T)) *\n    nominator / denominator
    return g

# bits - the stream of input bits T - duration of
# one bit dT - sample time interval dt should be
# smallet than T function returns a tuple of time
# vector and full PAM signal

def generateBinaryPamRaisedCos(bits, T, dT):
    # waveform corresponding to single bit
    waveform = squareRootRaisedCosine\n    (arange(-5*T, 5*T, dT), T, 0.2)

    t = arange(0.0, len(bits)*\n    T+10*T, dT) #time vector
    pamSignal = zeros(len(t), dtype = float)

    #for every bit generate a signal and
    #put it at the appropriate place in
```
Figure 4.14. Eye diagram of square root raised cosine PAM when bandwidth is finite

```python
# the output signal
for i in xrange(len(bits)):
    if bits[i] == 0:
        pamSignal[int(i*T/dT):
        int((i+10)*T/dT)] += -waveform
    else: # bits[i] == 1
        pamSignal[int(i*T/dT):
        int((i+10)*T/dT)] += waveform

return (t, pamSignal)
```

Eye diagram when for transmission is used square root raised cosine pulses, is shown in fig. 4.14. The parameters of simulation are identical to the previous ones. The only new parameter is the shape of square root raised cosine pulse. Its duration is 10 s (it is truncated outside of $-5$ s and 5 s) and . From the picture it is clearly seen, that the ISI is almost completely eliminated.
4.4 Laboratory work

Lab purpose

The purpose of this lab is to get acquainted with digital baseband and passband modulation and demodulation techniques. When studying and experimenting, emphasis should be placed on channels with limited bandwidth.

Lab task

1. Create rectangular PAM signals with added white Gaussian noise. Demodulate them and calculate BER (Bit Error Rate). Repeat the previous experiment for different noise level. Plot the graph noise level vs BER. It is up to you to choose suitable representation of noise level but there is no requirement to calculate SNR (Signal to Noise Ratio). In the graph BER should vary between 0 and at least 0.3.

2. Repeat experiment of clause 1 using multilevel PAM (using 4 levels for example). How does the detection strategy change?

3. Repeat the items 1 and 2 using phase shift keying modulations.

4. Repeat the items 1 and 2 using frequency shift keying modulations.

5. Repeat the experiment in clause 1 with the only difference that the channel of limited bandwidth is added to simulation scenario. Is BER increased for the same level of noise?

6. Repeat the experiment in clause 5 using square root raised cosine pulses. Is there any improvement in BER?

7. Investigate the relation between ISI and bandwidth width for different pulse shapes. For this purpose create a sensible qualitative measure of ISI and, changing the width of the channel,
observe the relation. Make this experiment for rectangular, and square root raised cosine ($\beta$ should be 0, 0.5 and 1) pulses. What conclusion can be made?

**Control questions**

- What is the purpose of modulation?
- What are the types of modulation?
- What are the main methods of optimal detection in AWGN channel?
- What is the effect of limited bandwidth channel on signal waveform?
- What are the sources of intersymbol interference?

**Lab report**

The report should contain the answers to the questions posed in the lab task. Plots of the key signals and simulation results should be made and explained.
Chapter 5

Channel Coding

In this chapter the methods of channel coding will be analyzed. Channel coding is an introduction of redundant information into data stream with the purpose to protect original data. Two main categories of channel codes may be distinguished – error detecting codes and error correcting codes. In this lab CRC (Cyclic Redundancy Check) codes will be investigated as the error detecting codes. And convolutional codes will be our example for error correcting codes.

While dealing with a coding system, we always have to opposite operations – coding and decoding. Coding introduces information redundancy into data and decoding restores original data. The result of decoding is original symbols in case of error correction or the decision about data integrity in case of error detection. The success of decoding action is probabilistic in nature. The purpose of code designer is to maximize the probability of success.

What is failure in decoding? For error detection operation failure is the conclusion “data is not changed” while in reality the data were changed. Error correction failure is the output bit stream which is not equal to original data stream.
5.1 CRC coding fundamentals

Using CRC method, a predetermined number of check bits, called a checksum, are appended to the message being transmitted. This way the redundancy is introduced. The receiver then determines whether or not the check bits are in sync with the data. Of course, this algorithm gives correct answer only with a certain degree of probability. We seek this probability to be as close to one as possible. If an error is indicated, the receiver may send a request for data retransmission. But whether this request is made is determined by the protocol used by the system.

The similar technique is also used in data storage devices. In case the error is detected, reread of the data is initiated.

There are many techniques to generate check bits. The simplest example is a parity check bit appended to the data frame. If the number of ones is even, then “0” is appended to the data as a check bit. Otherwise “1” is appended. It is obvious that this technique detects the error if exactly one bit is flipped (lost in the process of transmission). Two randomly flopped bits are undetected.

In general we want to append arbitrary number of control bits. If the algorithm is sound, then more check bits give better error detection performance. As it is always in telecommunications there is a compromise – the higher number of CRC bits eats more of the usable channel.

General CRC computation algorithms are based on polynomial arithmetic. The polynomials used are the ones over GF(2) field (Galois field with 2 elements). The main operation on such polynomials is modulo division.

A polynomial of degree \( n \) over GF(2) is a function of \( x \) of the form:

\[
   a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0. \tag{5.1}
\]

Here \( a_n, a_{n-1}, \ldots, a_1, a_0 \) are coefficients of the polynomial and they can take only two values – “1” or “0”. And we are not really
interested in the nature of variable $x$. It is a just a formal variable. And it is possible to define polynomials without any such variables but the usage of such formal variable is convenient, especially for people familiar with general polynomials.

Notice how the number of coefficient in the polynomial is equal to the degree of polynomial plus one. Because we don’t care about the nature of $x$, polynomials may be represented only by their coefficients. So we have a bijective relation between all polynomials and all vectors with binary elements. For example, a polynomial $x^3 + x + 1$ is written as a bit vector (1011).

Two main operations are defined on binary polynomials. Addition (which informally is the same as subtraction) is performed adding corresponding coefficients modulo 2 (xor-ing them). For example let’s add polynomial $x^4 + 1$ with $x^3 + x + 1$. This operation in binary vector notation may be written as:

```
10001
01011
11010
```

The second main operation is polynomial modulo division. The result of such operation is the remainder of dividing polynomial $g(x)$ by polynomial $h(x)$. This division is very similar to long division on standard polynomials. An example of such long division is given below. All polynomials are written in binary form.

```
110010111 |1011
1011
11110111
1011
1000111
1011
11111
1011
1001
1011
|10| - remainder
```
So, the result of binary polynomial division is the remainder. In majority of cases we are not interested in quotient.

Binary polynomial multiplication is defined and calculated in a very similar manner as for “normal” polynomials. Only intermediate addition is substituted by addition modulo 2. Binary polynomial multiplication is not that important in coding theory. Probably the only important result is this – multiplication of a polynomial by $x^n$ is the same as appending $n$ zeros to the original polynomial.

CRC coding is, in essence, a polynomial division. It is important to have hardware implementation of this operation. The scheme for polynomial division using linear shift registers is shown in fig. 5.1. With every clock cycle these registers shift their content to the outgoing line and accept the data from the incoming line.

The algorithm of polynomial division may be described as follows:

1. Initialize the shift registers to all zeros.
2. Get first (next) message bit from the input data stream (m).
3. If the highest bit of CRC is 1:
   - Shift CRC and input bit to the left and XOR the result with the G.
4. Otherwise:
• Just shift the content of registers and input bit to the left.

5. If there are more message bits, go back to get the next one.

6. The result of division is in the registers after all input bits have been processed.

A software implementation of the polynomial division algorithm is given below. The implementation of some helper functions necessary for further investigation is given also. These helper functions are necessary for conversion between used data structures. In particular, binary polynomials are used either as integers (binary representation of integer are regarded as coefficients of the polynomial) or as vectors of zeros and ones. Polynomial division also has two implementations. In both of them the divisor is represented as an integer but the input polynomial in one version is represented as bit vector and in the other one – as the integer.

```python
# helper function finds the number of bits in a
given number really, we are interested in the
position of the highest bit, counting from the
lowest bit notice recursive implementation

def findNoOfBitsInInteger(n):
    if n == 0:
        return 0
    else:
        return 1 + findNoOfBitsInInteger(n/2)

# helper function converts from numerical
# representation of bit vector to vector of ones
# and zeros

def convertIntegerToVector(num):
    ans = []
    tmp = num
    while tmp > 0:
        ans.insert(0, tmp%2)
```

95
tmp = tmp/2
return asarray(ans)

#helper function converts from vector representation of bit vector to a number (binary representation)
def convertVectorToInteger(vec):
    ans = 0
    for i in xrange(len(vec)):
        if vec[i] == 1:
            ans = ans + 2**(len(vec)-i-1)
    return ans

#functions divides binary polynomial x modulo g. The result is remainder of the division. x is represented as numpy array or list of ones and zeros g is represented as integer the result is an integer representing remainder
def polynomialDivide(x, g):
    #we need a variable to hold the state of the registers
    stateOfRegisters = 0

    #We also must know the number of shift registers It is equal to length of bits in g minus 1
    m = findNoOfBitsInInteger(g)-1

    for i in xrange(len(x)):
        #is the last register zero? in such a case just shift register contest and the fill the last register with incoming bit
        if stateOfRegisters & (2**(m-1)) == 0:
            stateOfRegisters = \
            (stateOfRegisters << 1)+x[i]
        else: #feedback takes place
stateOfRegisters = \
((stateOfRegisters<<1)%
(2**(m))) +x[i]) ^ (g % 2**(m))

return stateOfRegisters

#functions divides binary polynomial x modulo g.
The result is remainder of the division. x and g
#are represented as integer the result is an
#integer which represents remainder

def polynomialDivideBin(x, g):
    #check for limit cases
    if g == 1:
        return 0

    #we need a variable to hold the state
    #of the registers
    stateOfRegisters = 0

    #length of shift registers
    m = findNoOfBitsInInteger(g)-1

    #length of input data
    l = findNoOfBitsInInteger(x)

    #the bit for feeding division scheme will
    #be calculated from this number
    tmp = x

    #loop starting from the highest bit
    for i in range(l[::-1]):
        currentInputBit = tmp / 2**(i)
        #is the last register zero? in such a
        #case just shift register contest and
        #the fill the last register with incoming bit
        if stateOfRegisters & (2**(m-1)) == 0:
            stateOfRegisters = (stateOfRegisters\n                << 1) + currentInputBit
        else: #feedback takes place
stateOfRegisters = \\
((stateOfRegisters << 1)% (2**m))\ 
+ currentInputBit) ^ (g % 2**m)

# shift input data by one bit

tmp = tmp % 2**(i)

return stateOfRegisters

It can be easily checked that these functions give the same results as division by hand.

CRC is formed using the following simple algorithm. Let us assume that the code has format \((n, k)\). This means there are \(k\) informational bits and \(n - k\) control bits in a codeword. The information bits are presented as degree \(k - 1\) polynomial. These bits are multiplied by polynomial \(x^{n-k}\), meaning, that \(n - k\) zeros are appended to information bits. The resulting polynomial is modulo divided by so called generator polynomial (let it be \(g(x)\)), which is of degree \(n - k\). The result of division (remainder) is our checksum bits. It is clear that there are \(n - k\) checksum bits. In one formula this may be written as:

\[
c(x) = m(x)x^{n-k} \mod g(x). \tag{5.2}
\]

Here \(m(x)\) is polynomial corresponding to the data bits, \(g(x)\) is generator polynomial and \(c(x)\) is polynomial corresponding to control bits.

The simplified circuit for CRC calculation is shown in fig. 5.2. It only slightly differs from polynomial division circuit. And the difference is in the position, where input data is fed. This displacement of data feeding position is the same as pre-multiplying \(m(x)\) by \(x^{n-k}\).

In software CRC encoding may be realized by simple division of information bits with appended zeros by generator polynomial.

Up to this moment the discussion was only about CRC encoding. And what about error detection with the help of CRC codes? The algorithm of error detection is really simple. Consider the CRC code \((n, k)\). It means that we receive \(n\) bits and some of them could have changed during transmission. We can represent these bits as
Figure 5.2. Simplified CRC generation circuit for $g(x) = x^3 + x + 1$

polynomial $r(x)$ and calculate $r(x) \mod g(x)$. If the result of such
calculation is zero, it means that no errors have occurred. If the result
of such division is not zero, then error has been indicated.

Imagine that during transmission, error vector $e(x)$ was generated
(in general the degree of $e(x)$ is equal to the degree of $r(x)$). In such
case the received data is:

$$r(x) = m(x)x^{n-k} + c(x) + e(x). \quad (5.3)$$

Error detection algorithm executed on such data gives:

$$r(x) \mod g(x) = e(x) \mod g(x). \quad (5.4)$$

This result means that in order to investigate the performance of
CRC code, we should not care about generation of information bits
as any information bits give the same answer. The only thing that
matters is error polynomial $c(x)$. So, if we want to check whether
the particular error vector is detected, the following should be done –
generate error vector $e(x)$ and calculate $e(x) \mod g(x)$. If $e(x)$ is not
zero and the result of division is zero, then error detection has failed.

The introduced quantity $e(x)$ is called error syndrome.

The simulation example which checks the probability of error
detection miss when the code used is $(20, 17)$, generator polynomial
is $g(x) = x^3 + x + 1$ and the errors in received data are generated
independently with probability 0.05 is given below.

#helper function for generation of error vector

99
# where 1 is generated with probability p n -
# length of generated vector

def generateErrorVector(n, p):
    # vector for output data
    ret = zeros(n, dtype = int)

    # random numbers between 0 and 1
    r = random.rand(n)
    for i in xrange(n):
        if r[i] < p:
            ret[i] = 1
    return ret

# function calculates the probability that CRC
# misses the error in data
# n - length of data frame
# g - generator polynomial as integer
# p - probability of independent error in received data

def calculateMissRateOfCode(n, g, p):
    # variable holds the number of missdetected symbols
    errorCount = 0

    # assume that 10000 repetitions gives reliable statistics
    for i in xrange(10000):
        # generate error vector
        e = generateErrorVector(n, p)
        eInt = convertVectorToInteger(e)

        # calculate error syndrome
        syndrome = polynomialDivideBin(eInt, g)
        if (eInt != 0) and (syndrome == 0):
            errorCount = errorCount + 1

    return errorCount/10000.0
The presented function may be called using statement “calculateMissRateOfCode(20, 11, 0.05)”. The result of function execution tells us the percentage of data block that were indicated as “good”, though in reality they contained errors. For this simulation this percentage is 2.84 %. This is a relatively high mark but clearly the CRC is very short and the result is expected.

Up until now we have discussed coding and decoding of CRC. But how do we choose generator polynomial? The answer is simple. Partial answer is in so called primitive polynomials. Majority of standard generator polynomials include primitive polynomials in one form or another. For example, generator polynomial is often chosen as \( p(x)(x + 1) \), where \( p(x) \) is primitive.

There are many definitions for primitive polynomials. One of them is this – if closed shift register circuit, corresponding to given polynomial, generates all \( 2^n \) elements from initial non-zero state, then the polynomial is primitive. In other words, imagine there is no input data in fig. 5.1 and the state of registers is not zero. Then clocking such circuit will generate all elements between (001) and (111) in pseudorandom order.

Naive algorithm for primitivity testing is based on the above presented definition of primality. The code given below (function “isPolynomialPrimitive”) tests whether a given binary polynomial (given as an integer) is primitive. The implementation is quite straightforward and should be clear from the comments in code.

```python
# Functions imitates one cycle of linear shift
# register Register state is given in integer
# "state" Polynomial is given in integer "g"
# Functions returns the state of registers after
# one cycle The function is necessary for naive
# primitivity testing

def shiftRegisterCycle(state, g):
    # number of shift register
    m = findNoOfBitsInInteger(g) - 1
```
# if the last register is zero then just shift
if state & (2**(m-1)) == 0:
    return state << 1
else:
    return ((state << 1) % 2**m) ^ (g % 2**m)

# naive check for primitivity if there is no shorter cycle than 2^n-1, where n is the degree of polynomial then g is primitive otherwise it is not primitive

def isPolynomialPrimitive(g):
    # degree of polynomial
    n = findNoOfBitsInInteger(g) - 1

    # initial state of registers
    state = 1

    # assume polynomial is primitive
    # we try to falsify this hypothesis
    primitive = True

    # in this array "1" means that the element in corresponding index was already generated
    statesGenerated = zeros(2**n, dtype = int)
    statesGenerated[1] = 1
    count = 0
    while (count < 2**n-2) and primitive:
        state = shiftRegisterCycle(state, g)
        if statesGenerated[state] == 1:
            primitive = False
        else:
            statesGenerated[state] = 1
            count = count + 1
    return primitive
5.2 Convolutional codes

Convolutional codes are designed for error correction. They are used in almost all modern wireless systems in some form. Turbo codes are more powerful but they are based on convolutional codes. Because of this, convolutional codes are a good base for further study of modern coding theory. In this lab we are going to study convolutional coding and decoding algorithms and some properties of these codes.

In theory convolutional codes are stream codes – there is continuous input data stream and this data is coded, producing the coded stream with introduced redundancy. We will study only 1/2 rate codes, which means that for every input bit, the coder produces two output bits. In practice convolutional codes are also used in packet mode. In this case the number of bits transmitted is doubled compared to uncoded data.

Convolutional coder is shown in fig. 5.3. The coder for every input bit $u$ generates two output bits $v^0$ and $v^1$. Every input bit is generated using the state of shift registers and current input bit. Shift registers contain previous input data. Particular output is determined by generator polynomials of convolutional encoder. These generator polynomials have nothing in common with generator polynomials of cyclic codes, except that they are binary polynomials. They are usually designated using $D$ as a variable – the $D$ stands for “delay”.

\[ u ightarrow s_0 \rightarrow v^0 \]
\[ s_1 \rightarrow v^1 \]

**Figure 5.3.** Convolutional encoder with generator polynomials $1 + D + D^2$ and $1 + D^2$
Generator polynomials may be viewed as impulse responses of the separate branches of the encoder. The reaction of the branch to input sequence (100....) is given by polynomial coefficients, when these are written in increasing order.

Operation of the encoder is simple and one cycle of the encoding may be described as:

- Form the bit sequence $S = (u, s_0, s_1, ..., s_n)$.
- Calculate the output $v^0$ from the first branch by xor-ing those bits from $S$ that are present in generator polynomial $g_1$.
- Calculate the output $v^1$ from the second branch by xor-ing those bits from $S$ that are present in generator polynomial $g_2$.
- Change the state of registers (shift them) to $(u, s_0, s_1...s_{n-1})$.

Software implementation of the encoder is given below. The code is pretty straightforward and the comments should help to understand it.

```python
from numpy import *
import pylab as P

# helper function counts the number of bits in a
given argument "num" the implementation in naive
# (slow) notice the function is recursive and
# recursive implementation is much more elegant
# then sequential

def bitCount(num):
    if num == 0:
        return 0
    else:
        return num % 2 + bitCount(num/2)

# function computes the output bit from one branch
# of convolutional encoder g - generator
```
# polynomial represented as integer. The least significant bit means zero'th order state - represents the state of shift registers. the least significant bit contains newest data # inp - input bit. It should be 0 or 1 # m - memory length (the number of shift registers)

def encodeOneBit(g, state, inp, m):
    # data bits in registers plus the incoming bit
dataAffectingOut = ((int(state) << 1) + inp)

    # multiply with polynomial
dataToXor = dataAffectingOut & g

    # count weather the number of ones is even or odd and according to this return the output bit
    numOfSetBits = bitCount(dataToXor)
    if numOfSetBits % 2 == 1:
        return 1
    else:
        return 0

# convolutional encoder generator polynomials are g1 and g2 they are represented as integers with the least significant bit meaning no delay
# initial state is all zeros length of output data is 2x of input data bits - input data. It should be python list or numpy array of zeros and ones
# m - memory length
# The procedure is: output bits are generated and after this the state is changed

def encodeDataFrame(bits, g1, g2, m):
    codedData = zeros(2*len(bits), dtype = int)
    currentState = 0 # initialize state
    for i in xrange(len(bits)):
        codedData[2*i] = encodeOneBit\
Transition between states is determined by the incoming informational bit. Also, given transition between two states, corresponding bit can be unambiguously determined – it is equal to the first bit of the transitioned state.

The trellis diagram is used in optimal convolutional data decoding (Viterbi decoding algorithm). Notice, that at some time moment
the system is in state \( s \), then there is no difference, how it has arrived to that state. Also, every state has two incoming paths. This means, that at every time interval, one path can be discarded.

Suppose the data length before coding is \( N \). This means that \( 2^N \) different data sequences exist. Optimal decoding algorithm may be described as:

1. For every \( i \) in \( 0..2^N \).
   - Generate different data sequence \( V_i \).
   - Encode this sequence obtaining \( U_i \).
   - Compute the Hamming distance between \( V_i \) and the received sequence \( R \). Let’s call this distance \( d_i \).

2. The optimally decoded sequence is \( V_k \), where \( k = \arg\max(d_i) \).

The huge problem with the above described algorithm is its computational complexity. Usually \( N \) is at least 100 bits in real systems, so the problem becomes intractable. The trellis structure depicted above lets us apply dynamic programming principles and to construct practical decoding algorithm. Viterbi algorithm at every time moment has \( 2^M \) (remember that \( M \) is the length of memory) paths and the Hamming distance from each of these paths to the received sequence until the analyzed time. Going in time these paths and associated distances are updated. When all data is processed, the path with the smallest Hamming distance is chosen and tracebacked. Traceback is a procedure where the bits transmitted are restored from the given path. It can be easily done as noticed earlier. Before full description of Viterbi algorithm, some notations will be introduced.

- \( t \) – events are analyzed at this time.
- \( t - 1 \) – previous time moment.
- \( R_t \) – data received at time \( t \) (when transition between time \( t - 1 \) and \( t \) happens).
• $s_j$ – state $j$, $j$ is one of $0..2^M - 1$.

• $V_{ij}$ – data generated when transition between states $i$ and $j$ happens. Length of this data is 2 bits.

• $d_H$ – Hamming distance function.

• $D_j(t)$ – Hamming distance between received data and the best path to state $j$ at time $t$.

Viterbi algorithm itself is given below:

• For every state $j$ in $0..2^M - 1$.
  – Determine two previous states $s_k$ and $s_l$.
  – Find the output data when transitions between previous and current states happen – it is obviously $V_{kj}$ and $V_{lj}$.
  – Calculate the Hamming distances $d_H(R_t, V_{kj})$ and $d_H(R_t, V_{lj})$.
  – Find the minimum full distance to state $j$:
    $$D_j(t) = \min(D_k(t - 1) + d_H(R_t, V_{kj}), D_l(t - 1) + d_H(R_t, V_{lj}))$$
  – Discard the path from the above item which is not the minimal one. Update the minimal path.

• Repeat the previous step for the next bit pair received.

The algorithm step described above should be repeated for every bit in the data frame we want to decode. After this we traceback decoded data from the best path.

The implementation of Viterbi algorithm is given below:

```python
#Viterbi decoder bits - input data g1, g2 -
#generator polynomials representation is the same #as for the encoder
#m - length of memory (nuber of shift registers)
#function return decoded data
```
def decodeDataFrame(bits, g1, g2, m):
    # full metric used for branch selection
    # i-th element means the full distance of
    # path to i-th state initialization takes
    # into account that initially only zeroth
    # state is possible
    fullMetrics = 1e10*ones(2**m, dtype = float)
    fullMetrics[0] = 0
    # temporary variable for metrics calculation
    tempFullMetrics = zeros(2**m, dtype = float)

    # matrix for traceback decoding element in
    # i-th row and j-th column shows the state
    # from which it was transitioned to the
    # i-th state (at decoding time instant j)
    stateTracebackMatrix = \
    zeros((len(bits)/2, 2**m), dtype = int)

    # loop for every pair of input bits
    # equivalently - for every state transition
    for i in xrange(len(bits)/2):
        # loop for every state
        for j in xrange(2**m):
            # calculate previous states to
            # current state j
            prevState0 = (j >> 1) + 0
            prevState1 = (j >> 1) + 2**(m-1)
            inpBitToPreState = j % 2
            # Calculate outputs from
            # previous states
            tmpOut0 = zeros(2, dtype = int)
            tmpOut1 = zeros(2, dtype = int)
            tmpOut0[0] = encodeOneBit(g1, \n            prevState0, inpBitToPreState, m)
            tmpOut0[1] = encodeOneBit(g2, \n            prevState0, inpBitToPreState, m)
tmpOut1[0] = encodeOneBit(g1, \
prevState1, inpBitToPreState, m)
tmpOut1[1] = encodeOneBit(g2, \
prevState1, inpBitToPreState, m)

#Calculate the Hamming distance
#between the received bit sequence
#and sequence, generated when
#transitioning between previous
#and current state happens
tempDist0 = \nsum(tmpOut0 != bits[i*2:i*2+2])
tempDist1 = \nsum(tmpOut1 != bits[i*2:i*2+2])

#compare and select step also update
#traceback matrix
if fullMetrics[prevState0]\n+tempDist0 < fullMetrics\n[prevState1]+tempDist1:
    tempFullMetrics[j] = \n    fullMetrics[prevState0]+tempDist0
    stateTracebackMatrix[i][j] =\n    prevState0
else:
    tempFullMetrics[j] = \n    fullMetrics[prevState1]+tempDist1
    stateTracebackMatrix[i][j] = \n    prevState1
fullMetrics = tempFullMetrics.copy()

decodedData = zeros(len(bits)/2, dtype = int)
#traceback data from state transitions
bestLastState = fullMetrics.argmin()
for i in range(len(bits)/2)[::-1]:
    decodedData[i] = bestLastState % 2
    bestLastState = \n    stateTracebackMatrix[i][bestLastState]
Now the performance of convolutional codes can be checked. This is done while simulating data transmission, encoding, bit error addition and decoding. Bit error rate (BER) is given in the data before decoding and in decoded data. This is a little non standard metric, but it gives the impression of code performance. The code of such simulation is given below. The parameters are: BER in coded stream 3 %, generator polynomials $1 + D + D^2$ and $1 + D^2$. The BER in decoded data stream (as compared to original input bits) was measured as 0.115 % during simulation.

# helper function for generation of error vector
# where 1 is generated with probability p n -
# length of generated vector

def generateErrorVector(n, p):
    # vector for output data
    ret = zeros(n, dtype = int)

    # random numbers between 0 and 1
    r = random.rand(n)
    for i in xrange(n):
        if r[i] < p:
            ret[i] = 1

    return ret

# given that BER in input data is p what is BER in
# output data function returns the BER in output
# stream

def hardDecisionDecodingPerformance(p):
    errorCount = 0
    for i in xrange(10):
        dat = random.random_integers(0, 1, 10000)
        encDat = encodeDataFrame(dat, 7, 5, 2)

        # introduce error into encoded data
        e = generateErrorVector(len(encDat), p)
for k in xrange(len(e)):
    if e[k] == 1:
        # flip the bit
        encDat[k] = abs(encDat[k]-1)
    decDat = decodeDataFrame(encDat, 7, 5, 2)

# count the error in this frame (i)
# and add to the total count
errorCount=errorCount+sum(dat != decDat)
return errorCount/100000.0

There are many possible variations and modifications which may be applied to convolutional codes. The decoding may be soft, the rate may be 1/3, puncturing may be used; data interleaving may be used to fight with bursty error and so on. There will be a chance to investigate some of these possibilities during the lab.

One more unanswered question is how generator polynomials are chosen. It is impossible to give definite answer but some necessary conditions may be stated. For example, generator polynomials should be relatively prime. By definition, polynomials are relatively prime if the highest degree of their greatest common divisor is zero.

Greatest common divisor of two numbers or polynomials may be found using Euclidean algorithm. The code of this algorithm for polynomials is given below.

```python
# Euclidean algorithm for binary polynomials gcd
# stand for greatest common divisor a, b - input
# polynomials represented as integers
# output is gcd polynomial represented as integer

def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, polynomialDivideBin(a, b))
```
5.3 Laboratory work

Lab purpose

The purpose of this lab is to get acquainted with error correcting and error detecting codes, to understand coding and, especially, decoding algorithms. After the lab is done, it should be clear what performance can be expected from different coding techniques.

Lab task

1. Write the program for generation of all primitive polynomials of degree \( n, n = 3, 4, \ldots, 10 \). What is the computational complexity of the program? Is it possible to generate all primitive polynomials of degree 30 using this program? Estimate the time needed for generation of all primitive polynomials of degree 31.

2. Check the hypothesis “every primitive polynomial is also irreducible polynomial”. Irreducible polynomial is an equivalent of a prime number. It has only to divisors – itself and zero degree polynomial (which is polynomial “1”). All irreducible polynomials up to relatively small degrees may be computed using sieve of Eratosthenes algorithm.

3. Construct two CRC code with generator polynomials of degree 7, and codeword length of 20. The first one should have the generator polynomial which is primitive polynomial. And the second polynomials should not be even irreducible. Compare error detection performance of these codes. For this purpose construct and plot the graphs of “probability of independent errors in input data vs. probability of undetected bad frames in output data” type. Which of the two codes has better performance?

4. Construct CRC code (20, 7) with generator polynomial which is primitive. Investigate whether bursty bit errors are detected
better or worse then independent errors. You should choose the
model for bursty errors, and the overall BER for both cases
investigated should be equal.

5. Generate all pairs of relatively prime binary polynomials of
degree 5. How many such pairs are there?

6. For convolutional code with generator polynomials (111, 101)
create the plot “BER in coded data vs. BER in decoded data”.
Choose some relatively prime generator polynomials with
memory length 5 and generate the same graph. Which code
performs better? Why?

7. Modify the hard decision convolutional decoder to soft de-
cision decoder. For this purpose only two things should be
changed – bits (0, 1) should be transformed to (−1, +1) and
distance metrics becomes Euclidean squared distance instead
of Hamming distance.

8. Test, whether soft decoding performs better or worse then hard
decoding under the same SNR.

Control questions

• What is binary polynomial?

• How operations are defined on binary polynomials?

• What are the irreducible and primitive polynomials?

• How is CRC computed given a generator polynomial?

• How can it be determined which one of given two generator
polynomials is better? Is there an unambiguous answer to this
question?

• How is the convolutional encoder constructed?
• What parameters fully describe the convolutional encoder?

• Describe the optimal decoding strategy for convolutionally coded data.

• What is Euclidean algorithm? How can it be implemented?

• Why do generator polynomials of convolutional encoder can not divide each other?

• Why is soft decision decoding usually better than hard decision?

**Lab report**

The report should contain answers to questions given in the lab task. Plots of the simulation results should be explained. Modified simulation programs should be presented.
References


