CHALMERS

Evaluating SWIG using Turbo Codes

Master of Science Thesis in Communication Engineering

OSKAR ARVIDSSON TJÄDER and LINH TRAN

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2011
Master's Thesis EX029/2011

Evaluating SWIG using Turbo Codes
Oskar S. V. Arvidsson Tjäder and Linh M. Tran

Technical report no EX029/2011 Department of Signals and Systems
Chalmers University of Technology
SE-41296 Gothenburg
Sweden
Telephone +46-(0)31 772 1000

Department of Signals and Systems
Gothenburg, Sweden 2011

Abstract

Nowadays many larger companies have built up extensive libraries written in the system programming languages C or C++. These languages are capable of a wide variety of operations and have very fast execution times compared to scripting languages. The down-side is that the user friendliness and interactivity can become very low. Simplified Wrapper Interface Generator (SWIG) is a software development tool that integrates C/C++ modules with scripting languages to exploit the strengths of each language. The objective of this thesis is to evaluate whether SWIG can be used to wrap the Ericsson library Baseband Core Library (BCL) so that it can be used in a Python environment. The concept was proven by wrapping a turbo code class in the open source library IT++. To do this, two data conversion files were created to handle the passing of IT++ data structures between C++ and Python.

If the library to be wrapped has few dependencies to other libraries and only contains standard C++ data types, then wrapping with SWIG is rather straight forward. The libraries IT++ and BCL however use IT++ defined data structures which introduces additional complexity. To decide whether it is beneficial to wrap modules with converter files and glue code one should consider the amount of time and effort that is required to write the glue codes compared to the amount of usage.

"Experience is what you get when you didn't get what you wanted"

— *Anders Hansson, quoting Randy Pausch, referencing a quote used at Electronic Arts*

Contents

Chapter 1

Introduction

1.1 Background

Nowadays many larger companies have built up extensive libraries to be used for their development needs. At many times these libraries are crucial to achieve results that are of direct relevance for the company. They are often written in the system programming languages C or C++. These languages are capable of a wide variety of operations from low-level operations such as flipping bits and manipulating inputs as well as higher-level operations. This makes them suitable for building up libraries from scratch. C and C++ also have very fast execution times which is required for some applications. The down-side to all this is that the user friendliness and interactivity can become very low for users of the library. The sheer size and complexity of these libraries can make a trivial task complicated. The system programming languages are also not suitable when small code changes have to be performed often. After each change, no matter how small, the code will have to be re-compiled for the changes to come into effect.

Scripting languages on the other hand were not primarily meant for building programs from scratch but rather for plugging together existing programs and functions. For this task they are very flexible. Since scripting languages use interpreters and not compilers there is also no lengthy compilation time between code changes. They do however have longer execution times because of this.

As can be seen, both types of languages have their advantages and hence a way to combine these advantages would be valuable. One way of solving

this is by using Simplified Wrapper Interface Generator (SWIG). SWIG is a software development tool that generates wrappers for C and C++ modules so that these can be used by scripting languages.

Baseband Core Library (BCL) is a large library developed by Ericsson, containing functions for simulating baseband communication. It is written in C++ and is quite large and complex, around 500,000 lines of code in total. If this library could be wrapped using SWIG, it would greatly aid in its development and testing.

For Ericsson the desired scripting language to integrate with BCL is Python. Python has a syntax which is similar to MATLAB and supports object orientation and advanced data structures which makes it suitable for Ericsson's needs.

1.2 Objective and Approach

The objective of this master thesis work is to evaluate whether SWIG can be used to wrap BCL so that it can be used in a Python environment. A large and complex library like BCL is however not the ideal starting library to wrap with SWIG. The details of BCL are also confidential and hence cannot be covered in this report. However, BCL uses data structures from the open source library IT++ developed at Chalmers University of Technology. IT++ is a library which contains mathematical, signal processing and communication classes and functions intended to be used for communications related simulations. It is also a library, which compared to BCL, contains fewer dependencies to other libraries. This reduces the complexity of wrapping it considerably. As a proof of concept, the IT++ library will therefore be wrapped. This will show how SWIG can be used to wrap a library that is applicable to Ericsson's needs. Given how BCL uses IT++ data structures, successfully wrapping IT++ will also be a step towards wrapping BCL. To the best of our knowledge this has not been accomplished before in a way that allows data to be passed back and forth between Python and IT++ using SWIG.

A good choice for showcasing how SWIG can be used by Ericsson is to wrap the turbo encoder and decoder in IT++. This is a non-trivial class which can be troublesome to implement. It also requires many computations which makes it preferable to implement in a language with fast execution time. On top of this there are few dependencies to other classes which might

otherwise cause difficulties. With the Turbo class wrapped, it can be used interactively in a Monte Carlo loop in Python. The generation of information bits, modulation, addition of noise, and other trivial communication operations can be done in Python while the actual turbo details are done in the wrapped IT++ turbo encoder and decoder. Once this configuration is implemented it will be possible to manipulate the inputs and outputs of the turbo class functions as well as store and plot data using only Python. In this report we will showcase this by running the Monte Carlo loop and generating Bit Error Rate (BER) plots for a range of Signal to Noise Ratio (SNR) values. A turbo encoder and decoder will be implemented in Python to test and verify the Monte Carlo loop until the IT++ decoder is successfully wrapped.

In order to do this, however, we need to understand how a turbo decoder works within a communication system. To be able to combine it with Python and give the various functions the correct inputs and outputs an explanation of the theory behind turbo codes is covered.

Once IT++ is successfully wrapped the task of wrapping BCL can be started. As in the IT++ case a single class from BCL will be wrapped. As mentioned before, the details of this class cannot be presented due to Ericsson confidentiality but the lessons learned will be explained in this report.

1.2.1 Scope

A comparable solution for intregrating IT++ or BCL with a scripting language is to use MEX to integrate the library with MATLAB. Since Ericsson was interested in Python such a solution was never considered in any detail and will hence not be covered by this report.

There exists many other programs, such as Pyrex, SIP, and Boost, which perform similar functions to SWIG. In this master thesis SWIG is covered. The analyzed version of SWIG is 2.0 which is integrated with Python 3.1.1.

Since BCL and IT++ are both based on C++, modules written in this language will be wrapped. Focus is mainly put on wrapping a turbo code.

Chapter 2

IT++

IT++ is an open source C++ library of mathematical, signal processing and communication classes and functions [1]. It is mainly used for simulation and research in the communications area. The library was originally developed at the former department of Information Theory at Chalmers University of Technology in Sweden.

The library contains definitions of classes with associated routines which makes it possible to create data structures in C++ in a similar way as in MATLAB or GNU Octave. It covers a wide span of modules in a communication system, from basic components such as vectors, matrices and classes for random number generation to more advanced classes for channel modeling, interleaving, encoding, and decoding.

To demonstrate how IT++ can be used we give two short examples in the following sections.

2.1 Vector Class

IT++ defines a class Vec<Num_T> which utilizes the template feature in C++. It makes it possible to create vectors of different types, such as int, double, etc. For example, we can create a vector of type double with the following line

```
vec my_vector;
```

As in MATLAB, many operations can be performed on the vector. For instance we can create a vector of zeros with length n with

```
my_vector = zeros(n);
```

We can also insert, remove or shift elements in a vector, or concatenate several vectors.

IT++ provides an Application Programming Interface (API) that describes all classes and functions. In a similar way matrices can be created and operations performed according to the API.

2.2 Turbo Code Class

One of the more advanced classes implemented in IT++ is Turbo_Codec, the class that will be wrapped in this thesis. It implements a turbo encoder and decoder which has several parameters such as the choice of decoding method, interleaver length, encoder generator vectors, number of iterations, etc. The details of these parameters will be discussed in Chapter 4. Many of the functions in Turbo_Codec takes different types of Vec<Num_T> as input. How these types can be passed from python to C++ are described in section 5.1.

Chapter 3

SWIG

SWIG is a software development tool that connects programs written in C/C++ with scripting languages such as Python, Perl, and Tcl. By taking the declarations in the C/C++ header files it generates a glue code (wrapper) that enables the target languages to call functions within the C/C++ code. Figure 3.1 illustrates the basic idea behind SWIG.

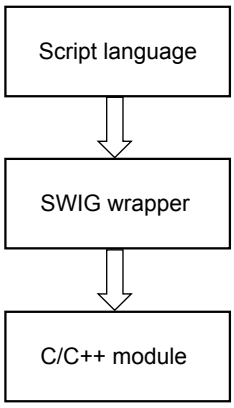

Figure 3.1: Basic idea behind SWIG.

For simplicity's sake, in the following sections the examples are written for C++ only.

3.1 Comparison of C++ and Python

Combining a scripting language, such as Python, with a system programming language like C++ is very powerful since it can exploit the strengths of each language. In this section the strengths and weaknesses relevant for building and using large libraries with these languages will be covered.

3.1.1 C++

C++ is a language intended to function on a mid-level of programming, meaning that it has to be capable of both low- and high-level operations. This results in a very flexible programming language that can interact with hardware with few restrictions but also function on a higher level of abstraction [2]. This allows for almost any complex data structure or algorithm to be implemented and the flexibility makes it suitable for building a library such as BCL from scratch. The down-side to this is that the code can often become quite large and complex. Since the library is built from scratch the programmer also needs to understand both high- and low-level programming.

C++ is based on static typing which means that the type is checked when the code is compiled and not during run-time. This allows the compiler to detect certain type related errors. It also allows the compiler to generate specialized code. For example, since it already knows the type it can just generate type specific instructions without any additional instructions for type checking. This optimization gives very fast execution times. It does however mean that code cannot be easily re-used. In addition, the code has to be re-compiled between each change, no matter how insignificant it may be. For large codes or libraries this can be a considerable delay in development time [3].

3.1.2 Python

Python on the other hand is a scripting language, which has somewhat different characteristics than C++. The first, and probably greatest, difference is that unlike C++ a scripting language assumes that useful components, written in other languages, already exist. In other words it is not designed to build programs from scratch but rather plugging or "gluing" together existing components. For this reason scripting languages are also referred to as

system integration languages or glue languages. In order to allow this connectivity between components, scripting languages are usually typeless. This means that there are no prior restrictions forced upon the data. Any component or value can thereby be used in any situation. This allows for code to be re-used for tasks the programmer never intended. This does however require instructions for type checking. Another major difference compared to C++ is that Python is not compiled but interpreted, which eliminates compilation time. The price one pays for these advantages is execution time. This is the result of using interpreters instead of compilers as well as having the components designed for ease of use instead of efficient mapping onto underlying hardware.

Python is a higher level programming language than C++ regarding the number of operations a single statement performs on average. Statements in a scripting language, such as Python, can produce hundreds to thousands of machine instructions, whereas a typical statement in languages like C++ executes five machine instructions on average [4]. The main cause for this is that the primitive operations in scripting languages are much more powerful than in languages like C++ [4].

As a scripting language, Python distinguishes itself in a number of ways which is of interest for Ericsson. It has a syntax that can be very similar to MATLAB, which is a language already used at Ericsson. Additionally, Python also allows for more advanced data structures than MATLAB. It also supports object orientation which can be useful in numerous applications. Another advantage with Python are the many already existing functions available. Besides the standard libraries there are also many third party modules.

3.1.3 Combining the Languages

As the two previous sections show there are advantages and disadvantages of both languages. The purpose of SWIG can hence be seen as combining the advantages of both languages. By wrapping C++ libraries to Python the speed and flexibility of C++ can be combined with the interactivity and ease of use of Python. In theory, this can be used to reduce the development time.

3.2 SWIG Structure

In this section we will go through the structure and components of SWIG. Figure 3.2 illustrates the key components and their inter-dependencies to generate the final extension module.

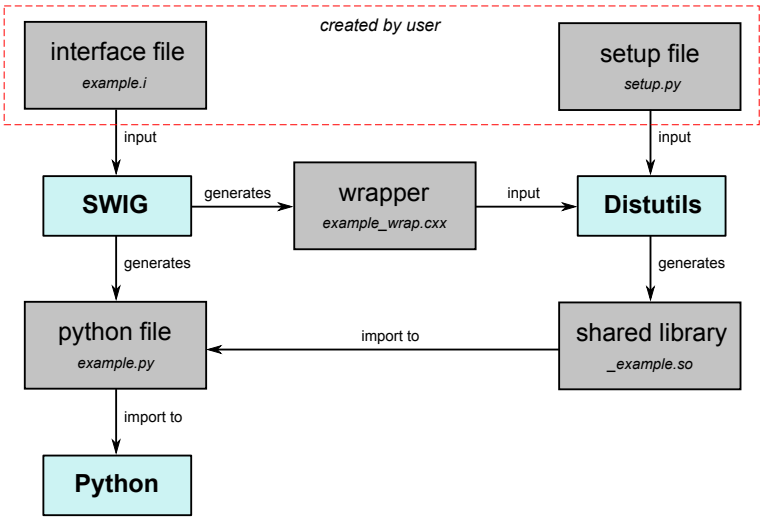

Figure 3.2: SWIG dependencies.

3.2.1 Interface File

The input to SWIG is a file containing ANSI C/C++ and special SWIG declarations. Normally this file is a special SWIG interface file denoted with a .i or .swg suffix. Below follows an example of the most common appearance of a basic SWIG interface file

```
%module example
%{
#include "example.h"
%}
// ANSI C/C++ declarations
int func(int x, int y);
int another_func(int x);
...
```

The name of the module generated by SWIG is given after the %module directive. In the %{ ... %} block all header files and declarations that are needed to make the wrapper code compile should be included. This text is not parsed or interpreted by SWIG but only copied to the generated wrapper file. At the end of the file all functions that should be accessible from Python should be declared.

3.2.2 SWIG Output

The output from SWIG consist of two files: a Python source file and a C++ source file that contains all the wrapper code that is required to build the extension module.

The command line

```
swig -c++ -python example.i
```

will produce a C++ source file example_wrap.cxx and a Python source file example.py. The name of the wrapper comes from SWIG taking the name of the interface file and appending _wrap to it. To create the final extension module the C++ source file needs to be compiled and linked with the rest of the C++ program. The Python source file is the file that will be imported into Python in order to use the module.

The recommended way to build an extension module for python is to use distutils. Distutils is a package in Python which provides support for building and installing additional modules into Python installations. It makes sure that the extension is built with the correct flags and headers etc. for the Python version that is being used. Distutils compiles the SWIG generated wrapper code into a shared library (.so-file for Linux and .pyd-file for Windows) that is included in the Python source file. The compilation is done by using a configuration file that describes the related extensions. Such a configuration file is conventionally called setup.py but can assume any name. Below follows an example

```
from distutils.core import setup, Extension

example_module = Extension('_example',
                           sources=['example_wrap.cxx',
                                    'example.cpp'],
                           )
```

```
setup (name = 'example',
       version = '1.0',
       author      = "Linh and Oskar",
       description = """Simple swig example""",
       ext_modules = [example_module],
       py_modules = ["example"],
       )
```

The line example_module = Extension(...) generates an extension module object using the source code files example_wrap.cxx and example.cpp (the original target source file) and sets the name of the module to _example. Extension modules to SWIG are normally named by using the same name as the python module prefixed with an underscore. In this case the name of the library would then be _example.so for Linux. The command to build the shared library using the setup file and distutils is

```
python setup.py build_ext --inplace
```

where

- python - is the version of Python we want to use
- setup.py - is the name of the configuration file
- build_ext - tells distutils to build extensions
- --inplace - tells distutils to put the extension lib in the current directory.

An alternative way to create the extension module is to let distutils handle the generation of the wrapper code and Python source file. Distutils supports creation of SWIG extension modules and so by modifying the configuration file slightly we can generate the extension module by only running the last distutils command. The setup.py file would then look like this

```
from distutils.core import setup, Extension

example_module = Extension('_example',
                           sources=['example.i',
                                    'example.cpp'],
                           swig_opts=['-c++']
                          )
```

```
setup (name = 'example',
       version = '1.0',
       author      = "Linh and Oskar",
       description = """Simple swig example""",
       ext_modules = [example_module],
       py_modules = ["example"],
       )
```

A second argument, swig_opts, has now been added to the Extension constructor which specifies the target language for the generated wrapper. Also, instead of giving the wrapper file as source we take the interface file. The build_ext command will then run SWIG on the interface file and compile the resulting C++ file into the final extension module [5].

Other necessary arguments to the Extension constructor are

- include_dirs — the paths to directories where include files are stored
- library_dirs — the paths to directories where libraries are stored
- libraries — names of included libraries.

Due to circular dependency the listed order of the libraries in the setup file is important. That is, if library B depend on library A, then A should be put before B in the list. Also, note that all libraries included in the setup file should be built with the same compiler version as the one used for SWIG.

3.3 Templates

Templates can be of use in a wide variety of applications. In signal processing templates can be used for representing different bit widths or hard or soft information. One can for instance use a float or double that can contain 4 and 8 bytes of data respectively to represent the bit widths.

Templates allows one to create special functions or classes that can operate with generic types. These functions and classes can be adapted to more than one type without rewriting the entire code for each type by using template parameters.

3.3.1 Templates in C++

The syntax for declaring a function template with template parameters in C++ is

```
template <class identifier> function_declaration;
```

For example this could look like

```
template <class T> void function(T x);
```

where T could be an **int**, **double**, **float**, etc. To call this function we can write

```
int x;
function<int> (x);
```

3.3.2 Templates in SWIG

Compilers do not treat templates as normal functions or classes. Templates are compiled on demand, which means that they are not compiled until an instantiation with specific template arguments is required. When it is required the compiler generates a function specifically for the given arguments [2]. Because of this characteristic, information about a specific template instantiation needs to be provided to SWIG in order to wrap a template. Let us illustrate this with the following example code

```
// Header file: stack.h

template <class T>
class MyStack {
   MyStack();
   void push(T i);
   T pop();
   int top;
   T st[100];
};

// C++ file: stack.cc

#include "stack.h"

template <class T>
MyStack<T>::MyStack()
{
   top = -1;
}
```

```
template <class T>
void MyStack<T>::push(T i)
{
    st[++top] = i;
}

template <class T>
T MyStack<T>::pop()
{
    return st[top--];
}
```

These template declarations alone will be ignored by SWIG since it cannot generate any code until a definition of the type T is given. One way to create wrappers for templates is to use the %**template** directive as

```
%template(intStack) MyStack<int>;
%template(doubleStack) MyStack<double>;
```

where the argument to %**template**() is the name of the instantiation in the target language.

Since templates are compiled when required, the declaration and definition of a template function/class must be in the same file. Therefore, if we have a separate header file for the interface and one with the implementation (definition), then these must be included in the file that uses the template. This means that in SWIG we need to include both the .h-file and .cpp-file in the interface file where the %**template** directive is used. The interface file would then have the following look

```
%module stack
%{
    #include "stack.h"
    #include "stack.cpp"
%}

%include "stack.h"
%include "stack.cc"
%template(intStack) MyStack<int>;
```

With these directives included in the interface file we can create an instantiation of MyStack<**int**>::MyStack() directly from python as

```
from stack import *
s = intStack()
s.push(1)
```

3.4 Namespaces

SWIG provides support for C++ namespaces, however, the wrapping flattens the namespaces in the target language. That is, the contents of all namespaces are merged together in the resulting .py-module. Let us illustrate this with an example. Given the following code

```
namespace a {
  void func_a();
}
namespace b {
  void func_b();
}
```

SWIG will simply create two wrapper functions func_a and func_b in the target language without pre-pending a namespace prefix as a::func_a().

3.5 Type Conversion

Type conversion of data types between programming languages is one of the most problematic tasks in SWIG. For every C++ declaration, wrapper code must be generated in order to pass values back and forth between C++ and Python, Figure 3.3 shows the basic idea.

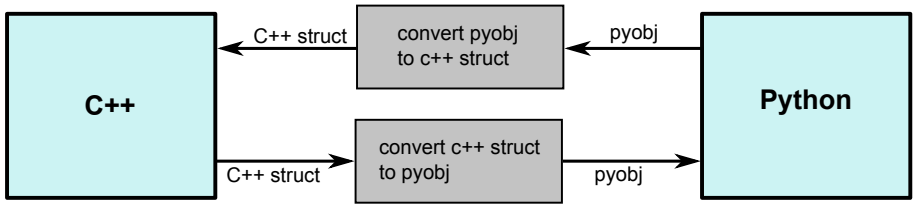

Figure 3.3: SWIG type conversion

Let us illustrate with an example how this works in more detail. Consider the following C++ function

```
int dummy(int n);
```

To access this function from Python, two Python API functions are used to convert the values. For each function written in C++, a corresponding function is generated in the wrapper code which handles the conversion. The pseudo code of such a function would look like this

```
arg = python_to_cpp(pyobj);
result = dummy(arg);
res_pyobj = cpp_to_python(result);
return res_pyobj;
```

The Python object is first converted into a C++ data type. With this data structure, that can be interpreted by C++, we can now input arg to dummy(). The result from the function is then converted back to a Python object so that it can be passed back into Python.

SWIG has a directive typemap which provides direct access to SWIG's low-level code generator. This gives the user possibility to manipulate the passing of different types between languages. However, use of typemaps is generally not required and is also not recommended to users unless they are absolutely confident and very familiar with SWIG [3]. Therefore we just mention the existence of typemaps and refer to [3] for more information.

3.6 SWIG Libraries

SWIG has its own libraries which contains support files for building extension modules that can be included in an interface file. These files often define new SWIG directives that enables access to parts of standard C++ libraries [6].

3.6.1 std_vector.i

The std_vector.i library provides support for use of the C++ std::vector class in the Standard Template Library (STL). To use the vector we need to include the library and instantiate vector for the different types we would like to use in the interface file. Below is an example of an interface file that uses the std_vector.i library.

```
%module example
%include "std_vector.i"
```

```
namespace std {
    %template(vectori) vector<int>;
    %template(vectors) vector<string>;
};
```

Here we have made two instantiations of the vector class. One is of the type **int** and the other of type string. Assume we have a C++ function that looks like the following

```
void print_vector(vector<int> v) {
    for(int i=0; i<v.size(); i++) {
        cout << v[i] << endl;
    }
}
```

we can then create a vectori object in Python, fill it with data and input it to print_vector. When a vector<**int**> is returned from C++ to Python it is interpreted as the Python data structure list.

3.6.2 std_string.i

Another library is std_string.i which enables passing of the object string in SWIG. To use the library it should be included in the interface file with the %include directive similar to the example in section 3.6.1.

Chapter 4

Turbo and Turbo-like Codes

Turbo codes revolutionized the coding community by achieving performance close to the Shannon limit. Today they are used in many communication systems. In this chapter we will explain the relevant theory for understanding the basics of turbo and turbo-like codes.

4.1 Discrete-time Equivalent Baseband Model

Consider the transmission system in Figure 4.1. This is the transmission system model that subsequent sections will be based upon. It is a discrete-time equivalent baseband model similar to the one Schlegel and Perez introduce in [7]. A way of obtaining this model is through matched filtering.

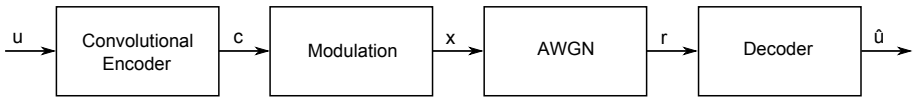

Figure 4.1: Reference Transmission System

The input to the system is the binary information sequence $\mathbf{u_1^K}$ of length K. Each bit, u_k, is assumed to be independently drawn from a uniform distribution $\{0, 1\}$. It can be interpreted as data subject to perfect source compression, i.e., no redundancy. This information sequence is encoded by a convolutional encoder, generating the code sequence

$$\mathbf{c}_1^K = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K) \tag{4.1}$$

where the length of each $\mathbf{c}_k$ is n_0. The code sequence is modulated to form the symbol sequence, $\mathbf{x}_1^S$ where each symbol, x_s, belongs to some finite discrete alphabet and the length of the sequence, S, depends on the chosen modulation. Any modulation could be used but for simplicity we will in this report limit us to only using Binary Phase Shift Keying (BPSK). This results in the modulated sequence, $\mathbf{x}_1^K$, and each symbol, x_k, taking on the values $\left\{-\sqrt{E_s}, +\sqrt{E_s}\right\}$.

The output sequence, $\mathbf{x}_1^K$, is the input to the Additive White Gaussian Noise (AWGN) channel. The white gaussian noise vector $\mathbf{n}_1^K$ is added to the signal forming the received sequence, $\mathbf{r}_1^K$ as

$$\mathbf{r}_1^K = \mathbf{x}_1^K + \mathbf{n}_1^K \tag{4.2}$$

where $\mathrm{E}\{n_k\} = 0$ and $\mathrm{E}\{n_k n_l\} = N_0/2\delta_{kl}$ where δ_{kl} is the Kronecker delta.

The decoder's task is based on decoding the received sequence, $\mathbf{r}_k^K$ outputting the best possible decision, $\hat{\mathbf{u}}_1^K$, of the input information sequence.

4.2 Convolutional Codes

Binary convolutional encoders are finite-memory systems which for every k_0 information bits inserted generate n_0 binary output bits. Hence the code rate, R_c, is defined as

$$R_c = k_0/n_0. \tag{4.3}$$

The encoder is composed of N shift registers containing k_0 bits, giving a total of Nk_0 stages inside the encoder, see Figure 4.2. The parameter N is called the convolutional code's constraint length. At each time instance t, the block of bits inside each shift register is shifted to the next one to the right and a new block of k_0 information bits is inserted into the first, and now empty, input shift register. The bit values of these positions are then used by n_0 modulo-2 adders to generate n_0 code bits that are fed to the output register. It can be concluded that such encoders' output code bits are not only dependent on the current input bits but on all $M = k_0(N-1)$ previous ones [8]. These bits correspond to the state of the encoder and will be defined as

$$\mathbf{S}_k = \left(s_k^{(1)}, s_k^{(2)}, \cdots, s_k^{(M)}\right) \tag{4.4}$$

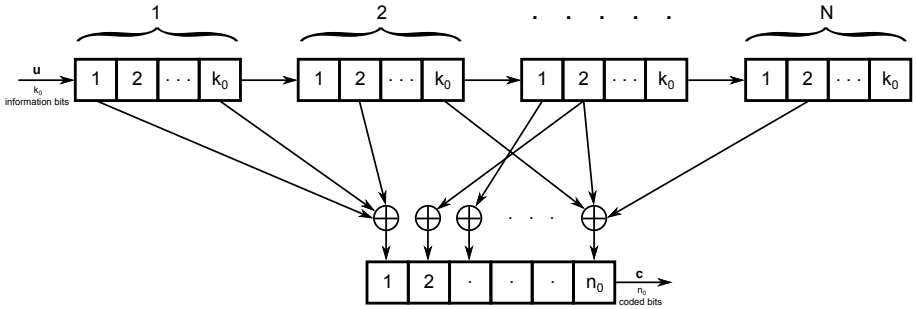

Figure 4.2: Block diagram of a convolutional encoder with n_0 inputs, k_0 outputs and constraint length N

where $s_t^{(1)}$ represents the contents of the first stage of the second shift register and the following elements the rest of the stages in the memory. Since the contents depend on the previous input bit sequence this can also be stated as

$$\mathbf{S}_k = (u_{k-1}, u_{k-2}, \cdots, u_{k-N}) \tag{4.5}$$

A way of describing a convolutional encoder is by specifying a set of n_0 generator vectors, one for each of the n_0 modulo-2 adders. Each vector has length Nk_0 corresponding to the connections from the stages of the encoder to the modulo-2 adder. Having a value of 1 in the ith position of the vector represents a connection between the ith stage of the encoder and the modulo-2 adder, similarly a value of 0 represents no connection [8]. An example is provided in Figure 4.3 below.

The binary convolutional encoder in Figure 4.3 has $N = 3$, $k_0 = 1$, and $n_0 = 2$ and can be represented by the two generator vectors

$$\mathbf{g_1} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \tag{4.6}$$

$$\mathbf{g}_2 = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}. \tag{4.7}$$

Out of convenience the generator vectors are often given in octal form and hence this encoder could also be presented as (7,5). This is also the format used in IT++.

If the input to an arbitrary binary convolutional encoder is **u** then the

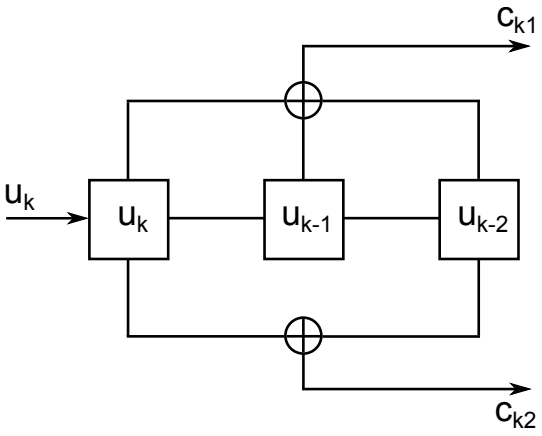

Figure 4.3: [7,5] encoder

output sequence is given by:

$$\begin{aligned} \mathbf{c}^{(1)} &= \mathbf{u} * \mathbf{g}_1 && (4.8)\\ \mathbf{c}^{(2)} &= \mathbf{u} * \mathbf{g}_2 && (4.9)\\ &\vdots \\ \mathbf{c}^{(n_0)} &= \mathbf{u} * \mathbf{g}_{n_0} && (4.10) \end{aligned}$$

where $*$ represents convolution using binary arithmetic. The output code sequence $\mathbf{c}$ is then formed in the following manner

$$\mathbf{c}_1^K = \left(c_1^{(1)}, c_1^{(2)}, \cdots, c_1^{(n_0)}, c_2^{(1)}, c_2^{(2)}, \cdots, c_2^{(n_0)}, \cdots, c_K^{(1)}, c_K^{(2)}, \cdots, c_K^{(n_0)}\right). \quad (4.11)$$

For our (7,5) encoder the output sequence will be formed as

$$\begin{aligned} c_k^{(1)} &= u_k + u_{k-1} + u_{k-2} && (4.12)\\ c_k^{(2)} &= u_k + u_{k-2} && (4.13)\\ \mathbf{c}_k &= (c_k^{(1)}, c_k^{(2)}) && (4.14) \end{aligned}$$

It is common that the D-transform is used to explain the operation of a convolutional encoder. This transform is equivalent to the z-transform except that the operator is $D = z^{-1}$. If the discrete time input to an arbitrary binary convolutional encoder is $\mathbf{u}_k$ then let the corresponding D-transform be

$$u(D) = \sum_k \mathbf{u}_k D^k \quad (4.15)$$

The generator vectors can also be represented using the D-transform. For the (7,5) example the generators would be expressed as

$$\begin{aligned} g_1(D) &= 1 + D + D^2 \\ g_2(D) &= 1 + D^2 \end{aligned} \tag{4.16}$$

It is now possible to express the encoder operation for this example as a function of the input and generator vectors as

$$c^{(1)}(D) = u(D)g_1(D) \tag{4.17}$$

$$c^{(2)}(D) = u(D)g_2(D). \tag{4.18}$$

$$\tag{4.19}$$

An alternative description of the binary convolutional encoder is the trellis diagram which connects different states over subsequent time steps. An example of such a trellis diagram is shown in Figure 4.4 which illustrates the before-mentioned (7,5) encoder. The nodes along the same vertical axis represent the various states at that discrete time k. In general there will be 2^M such states. Each of these states would have 2^{k_0} branches entering it and 2^{k_0} branches leaving it, one for each possible input segment. The branch from one state to another represents new inputs affecting the state of the encoder and generating the corresponding code bits. The code bits generated by each branch between two states given an input are written alongside these edges in the format input/code bits in 4.4. The branches between different states will generate a path through the trellis. This path is unique for a given input sequence and starting state.

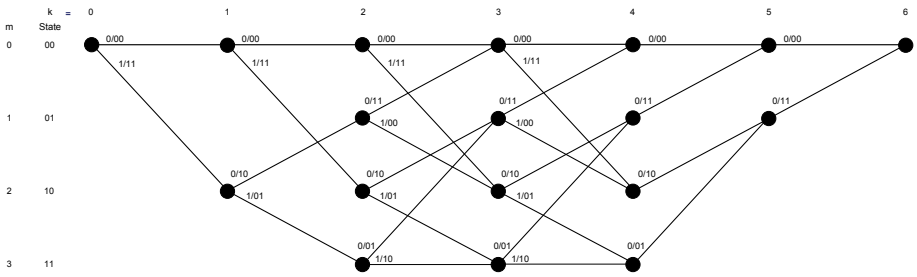

Figure 4.4: Trellis for (7,5) code

Customarily the code sequence is started and terminated in the $S = 0$ state of the trellis.

4.2.1 Code Weight and Minimum Distance

The weight of a codeword, $\mathbf{c} \in C$, is the number of nonzero components of that codeword and is expressed as $w(\mathbf{c})$. For linear codes the distance between two codewords can be written as $d(\mathbf{c}_1, \mathbf{c}_2) = w(\mathbf{c}_1 - \mathbf{c}_2)$. A useful metric for evaluating codes is the minimum distance, d_{min}, which is the the minimum of all possible distances between codewords and is expressed as

$$d_{min} = \min_{\mathbf{c}_1, \mathbf{c}_2 \in C \mathbf{c}_1 \neq \mathbf{c}_2} d(\mathbf{c}_1, \mathbf{c}_2) \tag{4.20}$$

4.2.2 Recursive Convolutional Encoders

A systematic encoder is an encoder where the input sequence is a direct part of the output sequence. It can be proven that systematic convolutional encoding, in general, will give lower minimum distances than nonsystematic encoders [8]. Using recursion, it is however possible to construct systematic encoders from any nonsystematic encoder of rate $1/n_0$ while maintaining the same minimum distance as the nonsystematic encoder. As can be seen in (4.17), in order to have the input sequence included directly in the output, one of the generator polynomials has to be equal to one. To achieve this, let us divide each encoder output by $g_1(D)$, which results in

$$\begin{aligned} c^{(1)}(D) &= u(D)\frac{g_1(D)}{g_1(D)} = u(D) \\ c^{(2)}(D) &= u(D)\frac{g_2(D)}{g_1(D)} \\ &\vdots \\ c^{(n_0)}(D) &= u(D)\frac{g_{n_0}(D)}{g_{n_0}(D)}. \end{aligned} \tag{4.21}$$

This corresponds to a convolutional encoder with feedback [8]. It is realized with shift registers and feedback. Such codes are commonly referred to as Recursive Convolution Codes (RCC). An example of how this can be implemented for the (7,5) encoder is shown in (4.21). It can be proven that such an encoder is equivalent to the original non-recursive non-systematic encoder since both will generate the same set of codewords, albeit for different input sequences [9].

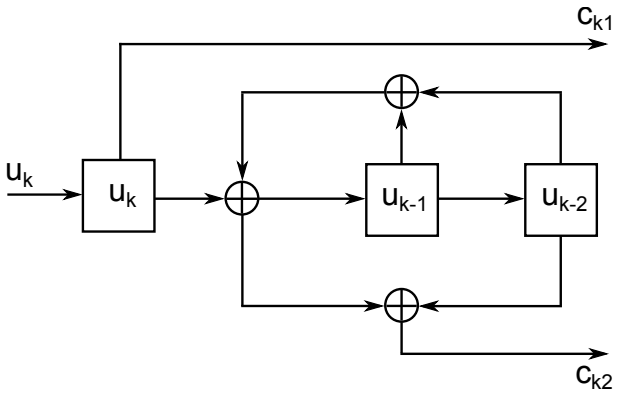

Figure 4.5: (7,5) recursive encoder.

In general the non-recursive encoder will perform better for medium-to-high SNR while the RCC will perform better for low SNR. This makes the non-recursive encoder the natural choice for a stand-alone encoder but when we aim for large coding gains at low SNR, as will be the case for turbo codes, RCC should be used [8].

4.3 Maximum Likelihood Sequence Decoding and the Viterbi Algorithm

Maximum Likelihood Sequence Decoding (MLSD) is a Maximum Likelihood (ML) method that for an AWGN channel seeks to minimize the minimum distance between the received signal, $\mathbf{r}_1^K$, and the set of possible transmitted code words. For convolutional codes this corresponds to finding the path through the trellis that is closest to the received signal sequence. For hard decoding the metric is the Hamming distance while for soft decoding the Euclidean distance is used.

A decoding algorithm that uses MLSD decoding is the Viterbi algorithm. It is only applicable to codes that can be represented by a trellis, e.g., convolutional codes. The Viterbi algorithm reduces the complexity of the decoding by using elimination to reduce the number of sequences that have to be compared. This is achieved by comparing the metrics for each branch entering a state. The one with the lowest metric is saved, called the survivor, and the others are discarded. This is done for all states and times of the trellis,

thereby greatly reducing the complexity of the decoder. This does not affect the optimality of the decoder since for any future branches through the trellis the survivor will still have the lowest metric according to the principle of optimality [10].

4.4 Maximum A-Posteriori Symbol Decoding

Another approach to decoding is to minimize the bit error probability of each received symbol. One should then aim to maximize the a posteriori probability (APP) for each bit of the transmitted sequence. This is called Maximum a Posteriori Probability (MAP) decoding and can be expressed as

$$\widehat{u}_k = \underset{u_k}{\operatorname{argmax}}\, p(u_k|\mathbf{r}_1^K) = \underset{u_k}{\operatorname{argmax}} \frac{\Pr(u_k)p(\mathbf{r}_1^K|u_k)}{p(\mathbf{r}_1^K)} \tag{4.22}$$

$$\propto \underset{u_k}{\operatorname{argmax}}\, \Pr(u_k)p(\mathbf{r}_1^K|u_k) \tag{4.23}$$

The main difference between the Viterbi algorithm and APP algorithms is the type of output. Whereas Viterbi outputs a hard decision of the transmitted information bits the APP algorithm gives the a posteriori probability. The a posteriori probabilities can be interpreted as a soft estimate of the probability that the information bits correspond to a certain input.

4.5 BCJR Algorithm

The BCJR algorithm, named after its inventors: Bahl, Cocke, Jelinek and Raviv, is an optimal decoding method for linear codes which minimizes the symbol error probability [11]. Unlike the Viterbi algorithm, which minimizes the probability of word error using ML decoding, BCJR uses the MAP criterion. The algorithm is sometimes also called Soft Input Soft Output (SISO) since the information to and from the decoder is soft. The algorithm is derived using the chain rule and Bayes rule iteratively.

Consider the trellis in Figure 4.4 again. A state transition between the states m' and m is controlled by the transition probability

$$\Pr(S_k = m|S_{k-1} = m') \tag{4.24}$$

and the output by

$$p(x_k = x|S_{k-1} = m'; S_k = m) \tag{4.25}$$

The algorithm is based on a forward recursion $\alpha_k(m)$, a backward recursion $\beta_k(m)$, and a branch metric $\gamma_k(m', m)$. The forward recursion computes the probability of being in a state, S_k, given that we know all state and transition probabilities up to a time k. The backward recursion is the probability of being in the state, S_k, given that we know all state and transition probabilities after time k, i.e. for $k+1$ to K. The branch metric is a measure of the cost of moving from one state S_k to another S_{k+1}. When these probabilities are calculated, the final step is to compute $\lambda_k(m)$ and $\sigma_k(m', m)$, where $\lambda_k(m)$ is the probability of being in state m at time k and $\sigma_k(m', m)$ is the probability of the transition from any state m' at time $k-1$ to a state m at k. In the following sections we will derive the just mentioned probabilities.

4.5.1 A Posteriori Information

The BCJR algorithm uses the observations of the received signal up to time K, to calculate the APP of the states and transitions [11]. The received signal is represented as K modulated code words

$$\mathbf{r}_1^K = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_K) \tag{4.26}$$

where the length of each $\mathbf{r}_k$ is n_0. Each node of the trellis in Figure 4.4 is associated with the APP $\Pr(S_k = m|\mathbf{r}_1^K)$ and each branch is associated with the corresponding $p(S_{k-1} = m'; S_k = m|\mathbf{r}_1^K)$. The aim of the decoder is to calculate these a posteriori probabilities.

It is easier to derive joint probabilities than to work with the conditional probabilities, therefore we introduce

$$\lambda_k(m) = p(S_k = m; \mathbf{r}_1^K) \tag{4.27}$$

and

$$\sigma_k(m', m) = p(S_{k-1} = m'; S_k = m; \mathbf{r}_1^K) \tag{4.28}$$

.

Since $p(\mathbf{r}_1^K)$ is constant for a given $\mathbf{r}_1^K$ ($p(\mathbf{r}_1^K) = \lambda_K(0)$ which is known in the decoder), we can divide $\lambda_k(m)$ and $\sigma_k(m', m)$ with $p(\mathbf{r}_1^K)$ to express the APPs as

$$\Pr(S_k = m|\mathbf{r}_1^K) = \frac{p(S_k = m; \mathbf{r}_1^K)}{p(\mathbf{r}_1^K)} \tag{4.29}$$

$$p(S_{k-1} = m'; S_k = m|\mathbf{r}_1^K) = \frac{p(S_{k-1} = m'; S_k = m; \mathbf{r}_1^K)}{p(\mathbf{r}_1^K)} \tag{4.30}$$

4.5.2 Derivation of α, β, γ

Let us define the forward and backward recursions and branch metric

$$\begin{aligned} \alpha_k(m) &= p(S_k = m; \mathbf{r}_1^k) && (4.31)\\ \beta_k(m) &= p(\mathbf{r}_{k+1}^K|S_k = m) && (4.32)\\ \gamma_k(m', m) &= p(S_k = m; \mathbf{r}_k|S_{k-1} = m') && (4.33) \end{aligned}$$

Now

$$\lambda_k(m) = p(S_k = m; \mathbf{r}_1^K)p(\mathbf{r}_{k+1}^K|S_k = m; \mathbf{r}_1^k) \tag{4.34}$$

Since events after time k are independent of $\mathbf{r}_1^k$ if S_k is known

$$\begin{aligned} \lambda_k(m) &= \alpha_k(m)p(\mathbf{r}_{k+1}^K|S_k = m)\\ &= \alpha_k(m)\beta_k(m). && (4.35) \end{aligned}$$

In the same way $\sigma_k(m', m)$ can be expressed as

$$\begin{aligned} \sigma_k(m', m) &= p(S_{k-1} = m'; \mathbf{r}_1^{k-1})p(S_k = m; r_k|S_{k-1} = m')p(\mathbf{r}_{k+1}^K|S_k = m)\\ &= \alpha_{k-1}(m')\gamma_k(m', m)\beta_k(m) && (4.36) \end{aligned}$$

In words, to calculate the probability of being in a state m at time k we need to calculate the cost of being at this state using both the forward and backward recursions. Similarly, the probability of transitioning from state m' at time $k-1$ to state m at time k is dependent on the cost of being in state m' at $k-1$ obtained using the forwards recursion, the cost of being in m at time k using the backwards recursion and the branch metric associated with this branch.

To derive the forward recursion, consider $\alpha_k(m)$ for $k = 1, 2, \ldots, K$

$$\alpha_k(m) = \sum_{m'=0}^{M-1} p(S_{k-1} = m'; S_k = m; \mathbf{r_1^K}) \tag{4.37}$$

Similarly as before, if S_{k-1} is known, then $\mathbf{r}_1^{k-1}$ does not affect events after time $k-1$, so the following can be obtained

$$\begin{aligned}\alpha_k(m) &= \sum_{m'} p(S_{k-1}=m';\mathbf{r}_1^{k-1})p(S_k=m;r_k|S_{k-1}=m') \\ &= \sum_{m'} \alpha_{k-1}(m')\gamma_k(m',m) \end{aligned} \tag{4.38}$$

Assuming the code is initialized in the zero state then for $k=0$ we have that

$$\alpha_0(0) = 1 \tag{4.39}$$
$$\alpha_0(m) = 0\ ,\ m \neq 0 \tag{4.40}$$

In the same way we can derive the backward recursion. Consider $\beta_k(m)$ for $k=1,2,\ldots,K-1$

$$\begin{aligned}\beta_k(m) &= \sum_{m'=0}^{M-1} p(S_{k+1}=m';\mathbf{r}_{k+1}^K|S_k=m) \\ &= \sum_{m'} p(S_{k+1}=m';r_{k+1}|S_k=m)p(\mathbf{r}_{k+2}^K|S_{k+1}=m') \\ &= \sum_{m'} \beta_{k+1}(m')\gamma_{k+1}(m,m') \end{aligned} \tag{4.41}$$

Assuming the code is terminated in the zero state then for $k=K$ we have the boundary conditions

$$\beta_K(0) = 1 \tag{4.42}$$
$$\beta_K(m) = 0\ ,\ m \neq 0 \tag{4.43}$$

Relations (4.38) and (4.41) show that $\alpha_k(m)$ and $\beta_k(m)$ are recursively obtained.

The branch metric can be expressed as

$$\gamma_k(m',m) = p(S_k=m;\mathbf{r}_k|S_{k-1}=m') \tag{4.44}$$
$$= p(S_k=m|S_{k-1}=m')p(\mathbf{r}_k|S_k=m,S_{k-1}=m') \tag{4.45}$$
$$= \Pr(u_k)p(\mathbf{r}_k|u_k) \tag{4.46}$$
$$= \Pr(u_k)p(\mathbf{r}_k|\mathbf{x}_k) \tag{4.47}$$

Now, assuming an AWGN channel, we can express

$$p(\mathbf{r}_k|\mathbf{x}_k) = \prod_{n=0}^{n_0-1} p(r_k^n|x_k^n) \tag{4.48}$$

$$= \prod_{n=0}^{n_0-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-|r_k^n - \widetilde{x}_k^n|^2 \frac{1}{2\sigma^2}\right) \tag{4.49}$$

where $\widetilde{x}_k^n$ is the hypothesized received signal belonging to the modulation alphabet. For BPSK modulation the variance is $\sigma^2 = N_0/2$.

4.5.3 Decoding

Minimization of the symbol probability of error is done by determining the most likely input bits, u_k from $\mathbf{r}_1^K$. Let A_k be the set of transitions corresponding to the input being $u_k = u$, where $u \in \{0, 1\}$. We can then calculate

$$p(u_k = u; \mathbf{r}_1^K) = \sum_{(m',m)\in A_k^{(j)}} \sigma_k(m', m) \tag{4.50}$$

This is calculated for both $p(u = 0; \mathbf{r}_1^K)$ and $p(u = 1; \mathbf{r}_1^K)$. In order to obtain the conditional probability of $u_k = 1$ given $\mathbf{r}_1^K$ we normalize

$$\Pr(u_k = 1|\mathbf{r}_1^K) = \frac{p(u_k = 1; \mathbf{r}_1^K)}{p(u_k = 0; \mathbf{r}_1^K) + p(u_k = 1; \mathbf{r}_1^K)} \tag{4.51}$$

We can now decode $u_k = 1$ if $\Pr(u_k = 1|\mathbf{r}_1^K) \geq 0.5$ and $u_k = 0$ otherwise.

The APP of the encoder output, $p(x_k^{(j)} = 0; \mathbf{r}_1^K)$ can also be calculated. Let $B_k^{(j)}$ be the set of all transitions from $S_{k-1} = m'$ to $S_k = m$ which correspond to the jth output digit , $x_k^{(j)}$, being 0. For time-invariant codes, which will be assumed here, $B_k^{(j)}$ is independent of k. Then

$$p(x_k^{(j)} = 0; \mathbf{r}_1^K) = \sum_{(m',m)\in B_k^{(j)}} \sigma_k(m', m) \tag{4.52}$$

and this can be normalized similarly to (4.51) in order to form $\Pr(x_k^{(j)} = 1|\mathbf{r}_1^K)$.

4.5.4 Outline of the Algorithm

Let us summarize the steps of the algorithm that need to be performed in the decoder to obtain $\lambda_k(m)$ and $\sigma_k(m', m)$.

1. First, $\alpha_0(m)$ and $\beta_K(m)$ for $m = 0, 1, \ldots, M-1$ are initialized as in (4.39) and (4.42).

2. When the whole sequence $\mathbf{r_1}^K$ is received, $\gamma_{k+1}(m, m')$, $\alpha_k(m)$ and $\beta_k(m)$ is computed recursively using (4.44), (4.38) and (4.41). The results are then stored for all times and states.

3. The decoder then has all information it needs to compute all $\lambda_k(m)$ and $\sigma_k(m', m)$ using (4.35) and (4.36).

4. Using the calculated values the decoder can output the APP of the input sequence, $\mathbf{u}_1^K$, using (4.50) from which hard decisions can be made. It can also output the APP of the encoder output sequence $\mathbf{x}_1^K$ using (4.52).

From the steps above it can be concluded that the algorithm will require a lot of memory to store all values α, β, and γ. Also since σ_k and λ_k are dependent on calculations based on all previous and subsequent k in $\mathbf{r}_1^K$ the amount of calculations will grow too many if the block lengths and constraint lengths are not kept low.

4.6 Turbo Codes

Berrou *et al.* introduced turbo codes in 1993 and achieved performance close to the Shannon limit [12]. Since then further research has resulted in other turbo-like codes. Turbo codes and turbo-like codes can be viewed as a refinement of a concatenated encoding structure. For decoding the BCJR algorithm is used iteratively. The purpose of this section is to explain the basics needed for understanding such codes.

4.6.1 Concatenated Codes with Interleaver

The performance of a convolutional code is mainly defined by its error correction capability which depends on the minimum distance of the code. Codes can have different lengths for a given code rate, R_c. Longer codes give larger minimum distance and thus offer better performance [9]. However, with increasing lengths the decoding complexity also increases, generally exponentially. A way of obtaining long block lengths and keeping reasonable complexity is to concatenate codes with shorter block lengths. The code

words of the first encoder will act as the input to the second, and so on, forming increasingly longer code words. In this report only two levels of concatenation will be considered. The first encoder will be called the outer encoder and it generates the outer code, c_o, and the second will be called the inner encoder and it generates the inner code, c_i. If the outer encoder has rate $R_c^o = k_o/n_o$ and the inner encoder $R_c^i = k_i/n_i$ then the overall rate of the concatenated code is simply the product of the two codes [8]:

$$R_c = \frac{k_o}{n_o}\frac{k_i}{n_i} = R_c^o R_c^i \tag{4.53}$$

The main advantage of concatenated codes is that the received, concatenated codewords can be decoded by subsequent decoders. The code is decoded in the reverse encoding order, i.e. first the inner decoder decodes the inner code and passes on the result to the outer decoder. This has the effect of breaking up the large code into several, simpler codes. Codes with such encoding and decoding scheme are often referred to as Serially Concatenated Convolutional Codes (SCCC), an example of which can be observed in Figure 4.6.

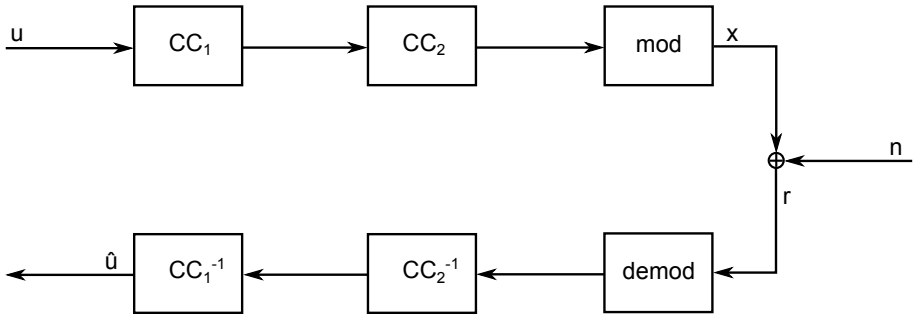

Figure 4.6: Communication system using Serially Concatenated Convolutional Codes

It has been shown that the optimal decoding solution for concatenated codes is to pass the APP from the inner decoder to the outer decoder [8]. To reach this optimum one therefore needs the demodulator and inner decoder to generate soft outputs.

When Berrou *et al.* introduced turbo codes one of the key components was a concatenated encoding scheme combined with an interleaver. By introducing a pseudo-random interleaver Parallel Concatenated Convolutional

Codes (PCCC) can be constructed as in Figure 4.7 as well as the more intuitive serial concatenated codes in Figure 4.8. In the traditional turbo code, introduced by [11], parallel concatenation was used but as interest grew in the coding community serial concatenation was also utilized in turbo-like codes [13]. In the PCCC case both encoders use RCC whereas for SCCCs the outer encoder is non-recursive, non systematic and the inner is RCC. As mentioned earlier RCCs are suitable when one aims to gain large code gains at low SNRs.

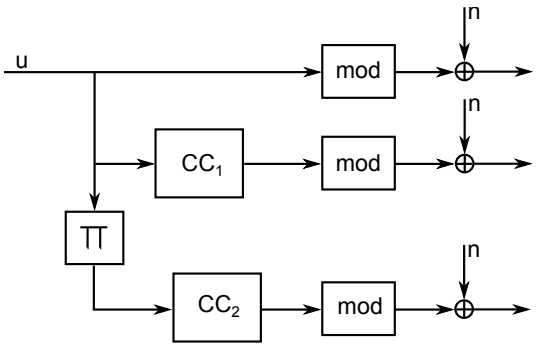

Figure 4.7: Transmission system using Parallel Concatenated Convolutional Codes with interleaver

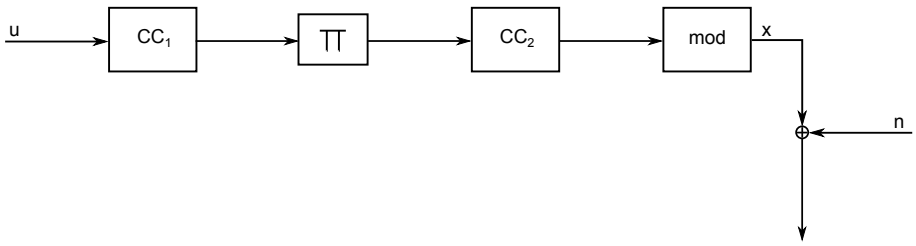

Figure 4.8: Transmission system using Serial Concatenated Convolutional Codes with interleaver

For both serial and parallel concatenation the interleaving has the effect of producing codes with very few low-weight codewords. This does not mean that the free distance of the code is thereby large but does result in the codewords being relatively sparse and having few nearest neighbors. In other words, the coding gain is achieved as result of the interleaver introducing this sparsity, called multiplicity, of codewords of low-weight.

The introduction of the interleaver has the side-effect of making the number of possible states for the entire codeword very large. This is the result of the interleaver randomly permuting the outer code and feeding this as input to the inner encoder. Any state diagram over the entire encoding procedure would hence have to take this into effect and will thereby become impossible to decode in an optimal way.

4.6.2 Extrinsic Information

An additional performance increasing factor introduced by Berrou *et al.* is extrinsic information [12]. Extrinsic information can, from an abstract level, be viewed as extra knowledge gained from the decoding process. In practice, this means that the information delivered from the outer to the inner encoder is no longer the APP but extrinsic information formed by the normalization of the computed APP by its corresponding a priori probability. In other words the values are based on information coming from all symbols, except the one corresponding to the same symbol. A trivial way of realizing this is by modifying the APP output as

$$\Pr(u_k = 1|\mathbf{r}_1^K)' = \Pr(u_k = 1|\mathbf{r}_1^K)/\Pr(u_k = 1). \tag{4.54}$$

$\Pr(u_k = 1|\mathbf{r}_1^K)'$ will then be the extrinsic output. The division by $\Pr(u_k = 1)$ can however result in numerical problems, especially for the last iterations where the a priori values can approach 0. An alternate solution to this is to not include the a priori values when calculating the $\gamma_k(m', m)$ used to calculate $\sigma_k(m', m)$ [14]. This is expressed as

$$\gamma_k'(m', m) = p(\mathbf{r}_k|\mathbf{x}_k) \tag{4.55}$$

and $\sigma_k'(m', m)$ can now be expressed as

$$\sigma_k'(m', m) = \alpha_{k-1}(m')\gamma_k'(m', m)\beta_k(m) \tag{4.56}$$

The extrinsic output can now be obtained by modifying (4.50) as

$$p(u_k = u; \mathbf{r}_1^K)' = \sum_{(m',m)\in A_k^{(j)}} \sigma_k'(m', m) \tag{4.57}$$

and from this $\Pr(u_k = 1|\mathbf{r}_1^K)'$ can be formed through normalization as in (4.51).

4.6.3 Iterative Decoding

Since the large number of states in turbo and turbo-like codes make optimal decoding impossible, Berrou *et al.* introduced a suboptimal iterative decoding algorithm, known as the turbo decoding algorithm. This algorithm was based on iteratively using concatenated and locally optimal decoders based on the BCJR algorithm. This eventually results in an APP estimate of the transmitted information bits. Since there exists both parallel and serial encoder schemes there are also parallel and serial decoder schemes. They both use the same decoding principles, albeit in different configurations, and hence only one will be explained in any detail in this report. The decoder for the serially concatenated code tends to be somewhat simpler to describe and is thereby chosen here.

The inner decoder feeds the the outer decoder with extrinsic information calculated from the received signal and a priori values. The outer decoder calculates the APP of the outer encoder output for each iteration and feeds this back to the inner decoder in the subsequent iteration, where it is used as a priori values.

The SCCC decoding scheme can be observed in Figure 4.9. If the feedback is disregarded one can observe that for each operation in the encoder there is an inverse operation in the decoder. The order of the decoding operations is also the reverse of the encoder.

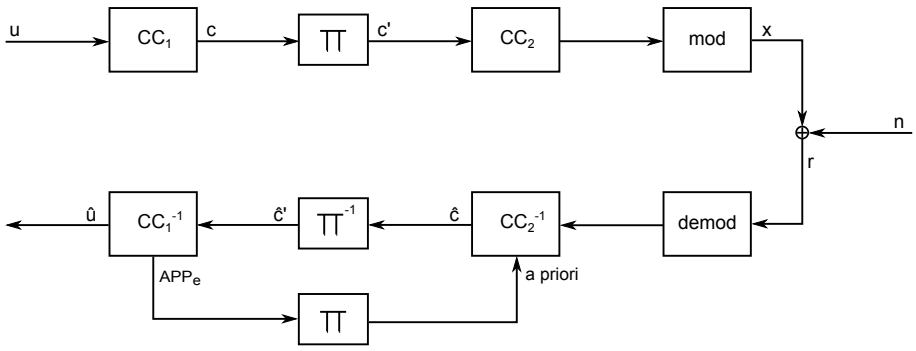

Figure 4.9: Serially Concatenated Convolutional Turbo-like code

The inner decoder receives the soft-valued symbol sequence, $\mathbf{r}_1^K$ and uses this information to compute all γ'-values according to (4.55). Since γ' is not dependent on the a priori probabilities this can be calculated once for each received sequence and then stored. With all γ' calculated the decoder can,

for each iteration, calculate the γ-values as

$$\gamma_k(m', m) = \Pr(u_k)\gamma'_k(m', m). \tag{4.58}$$

During the first iteration the a priori probability used for these calculations is unknown and is hence set to 0.5 if one uses the probability domain. The α- and β-values can now be calculated according to (4.38) and (4.41). Now σ' can be calculated according to (4.56). With σ' calculated the extrinsic information output, called $\hat{\mathbf{c}}'$ in Figure 4.9, can be computed according to (4.57).

Thereafter, $\hat{\mathbf{c}}'$ is de-interleaved to generate $\hat{\mathbf{c}}$ which is used by the outer decoder. Here one cannot use the BCJR algorithm exactly as in the inner decoder. Since the input now is a probability the branch cost has to be calculated differently. The γ-value is obtained according to

$$\gamma_k(m', m) = \prod_{n=0}^{n_0-1} \Pr(\widehat{c}_k^{(n)} = c^{(n)}(m', m)) \tag{4.59}$$

where $c^{(n)}(m', m) \in \{0, 1\}$ is the code bit n associated with the trellis transition (m', m). In other words

$$\Pr(\widehat{c}_k^{(n)} = c^{(n)}(m', m)) \quad = \quad \begin{cases} \widehat{c}_k^{(n)}, & c^{(n)} = 1 \\ 1 - \widehat{c}_k^{(n)}, & c^{(n)} = 0 \end{cases} \tag{4.60}$$

This γ will have to be re-calculated for each iteration since it is dependent on $\hat{\mathbf{c}}$ which varies for each iteration. After this the decoding algorithm can function as in the inner decoder and calculate α, β, and σ. From σ the APP-values of the inserted information bits can be formed using (4.50). From these the most likely bits for this iteration can be computed. The γ values are also used to calculate the APP of the outer encoder output according to (4.52). This is then interleaved to form the a priori information and a new iteration is begun. In the new iteration these priors are now used by the inner decoder in (4.44). Except for this difference everything else is done in the same manner as in the first iteration. This iterative process continues until a fixed number of iterations have been performed or some stopping criterion has been met. After this the APP-values of the input bits for this iteration are returned and decisions can be performed on these to form $\hat{\mathbf{u}}_1^K$.

For PCCC turbo codes the basics of the algorithm are similar but there are some differences to take into consideration. These differences are related

to the different decoding scheme that PCCC encoding requires, see Figure 4.10.

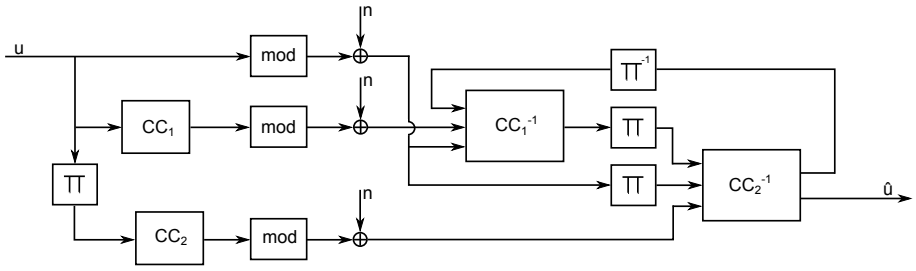

Figure 4.10: Parallel Concatenated Convolutional Turbo code

Performance of Parallel and Serial Turbo and Turbo-like Codes

While both PCCC and SCCC achieve performance close to the Shannon limit there are some differences. For the same interleaver length and rate PCCC performs better at high values of BER. However, in the low BER region PCCC the slope of the BER curve has a considerable change of slope, which is commonly called the "error floor," which SCCC does not have. This results in SCCC outperforming PCCC in the low BER region.

Chapter 5

Implementation and Results

The IT++ turbo decoder that is wrapped using SWIG is not a stand alone class. It requires other functions such as modulation, adding noise and counting errors in order for any results to be obtained. It also requires the correct parameters to be passed to it, such as the noise variance and the generator vectors. Since it functions at such low BER one also needs to send it large amounts of information bits in order for the simulation results to be statistically sound. To make sure all this would not cause problems once the IT++ turbo decoder was successfully wrapped, a Monte Carlo loop with an SCCC turbo-like decoder was implemented in Python beforehand. The main file is called turbo.py, see Appendix B.2.1.

The Monte Carlo loop has two stopping criteria: it has to have processed at least a certain amount of blocks and a certain amount of errors. The first criterion will ensure that at least some blocks are used when the BER is high, otherwise all errors could occur in a single block. The second criterion guarantees that for low BER, enough blocks are sent so that the results are statistically meaningful.

With the Monte Carlo loop tested and completed the SCCC turbo-like encoder and decoder were implemented. What we wanted to achieve was a Python code that shows the BER for each SNR value and each iteration the decoder performs.

The Python code is implemented in such a manner that it is possible to replace and re-use the different functions. In order to show how SWIG can be used interactively with Python the encoder and decoder were replaced with the IT++ encoder and decoder functions. These functions are a part of the class Turbo_Codec and in order to integrate them with Python the

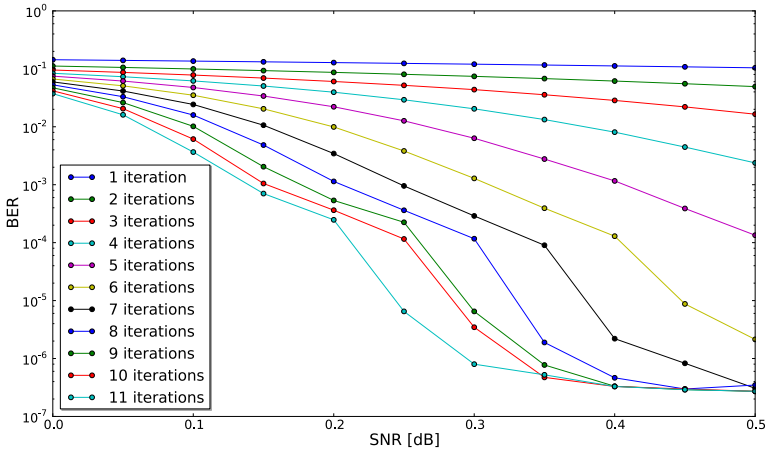

Figure 5.1: BER plot for recursive [14,15] turbo code.

class needed to be wrapped. The results of running this Monte Carlo loop are presented in Figure 5.1. The simulated code is a recursive [14,15] turbo code with interleaver size 16384.

5.1 Wrapping IT++

The main problem encountered when trying to wrap the class Turbo_Codec was to wrap the special data structures that IT++ defines. Let us illustrate the problem with an example.

Turbo_Codec contains an encoder whose function declaration looks like

```
void encode(const bvec &input, bvec &output)
```

The arguments to the function are references to the type bvec which is a binary vector. This vector is not a standard C++ data structure but is defined by IT++. Therefore we cannot pass the values from Python to SWIG as is the case when we have an **int** or **double**, etc. Since SWIG does not recognize these special data types and cannot perform a standard type conversion they have to be handled separately.

Assume we have generated a bit sequence in Python that is of the data structure list. Now, we want to pass this to the IT++ encoder but in order to do that the list first has to be converted to a standard C++ data type

that has the same kind of structure as a list. In section 3.6.1 it was described how std::vector can be used. We can utilize this class to pass values back and forth between Python and SWIG, however, in order to complete a call to encode, two extra conversions are required. A first conversion is made in Python by converting the list to a vectori (following the declaration in 3.6.1) like

```
# Convert from a "python list" to vectori (vector<int>)
def list_to_veci(v1):
    v2 = vectori(len(v1))
    for i in range(len(v1)):
        v2[i] = v1[i]
    return v2
```

The binary sequence is now a vector<**int**> which must be converted into a bvec before it can be passed as an argument to encode. This conversion is done as

```
/* Convert vector<int> to bvec */
bvec veci_to_bvec(vector<int> v1) {
  bvec v2 = zeros_b(v1.size());
  for (int i=0; i<v1.size(); i++) {
    v2[i] = v1[i];
  }
  return v2;
}
```

Finally, the returned value from veci_to_bvec can be used as input to encode. To pass the output from encode back to Python the values need to be converted back in a similar way.

Now, to handle the required conversions in C++ we have created an additional glue code which is wrapped instead of the target file. The created glue code for the encode function looks like

```
vector<int> encode(vector<int>& info_bits) {
    itpp::bvec output_bits;
    turbo.encode(veci_to_bvec(info_bits), output_bits);
    return bvec_to_veci(output_bits);
}
```

It takes the input, does the needed conversion, calls the target function and converts the output before it is returned to Python. Each IT++ function called from Python has its corresponding glue code function written in a file

called demo.cc. Two files, converters.h and converters.py, with all required converters have been created, see Appendix B.1.7 and B.1.8. Figure 5.2 shows how these files are used by the C++ and Python modules.

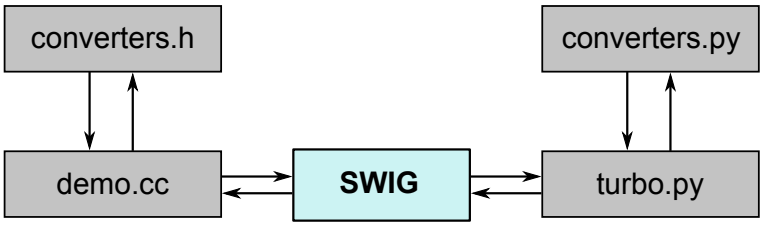

Figure 5.2: Dependencies between converters and C++ and Python modules.

Let us now summarize what we have done. Since IT++ uses its own defined data structures we cannot wrap the IT++ classes and functions directly. Instead we have created a glue code called demo.cc which is wrapped instead. This glue code contains functions that call the IT++ functions and includes converters.h.

The file which implements the Monte Carlo loop is turbo.py. It is also the file that includes the SWIG generated Python module which enables calls to demo.cc. It uses converters.py for handling non-standard C++ data structures, such as list(), to demo.cc.

An alternative and more complex way to handle the IT++ data types is to write a library similar to std_vector.i. However, we will begin with using converters.h and converters.py since the main task is to determine whether BCL can be wrapped by SWIG or not.

5.1.1 Libraries

IT++ uses the libraries Basic Linear Algebra Subprograms (blas), Complex Basic Linear Algebra Subprograms (cblas), Linear Algebra PACKage (LAPACK), and Fastest Fourier Transform in the West (fftw), therefore these have to be included in the setup file. The libraries blas, cblas, and LAPACK all give support for linear algebra operations, whereas fftw is used for computing discrete Fourier transforms.

5.2 Wrapping BCL

The previous section showed how classes in IT++ can be wrapped. The goal of this thesis however is to wrap a class in BCL.

Due to the Ericsson confidentiality policy we will not mention the real name of the class or any variables but only show examples that describe the principles behind certain code sections of the class. For the sake of simplicity we will refer to the class as bcl_class.

We have already seen how classes can be wrapped using converters.h. However, writing glue code can be somewhat inefficient and cumbersome. A better way would be to wrap a BCL class without using any glue code. Such a solution would require a library similar to std_vector.i, as mentioned in section 3.6.1, but written for IT++ data types. This is however not in the scope of this thesis. Instead, the aim is to examine whether it is possible at all to wrap a BCL class. An attempt to wrap the BCL class directly was made but this was prevented by **protected** sections in bcl_class. Generally, SWIG does not support **protected** or **private** declarations and simply ignores these kind of sections [3]. The program will run until the protected variable is used and then generate an error message about an unknown variable.

Since the BCL class cannot be wrapped without glue code it has been wrapped in the same way as for IT++ where converters.h and converters.py are used for passing IT++ data structures between C++ and Python.

The main problem encountered when wrapping bcl_class was to wrap namespaces. To illustrate, consider the following header file

```
/* Header file ns_example.h */

#include "b.h"

namespace a {
  using namespace b;
}
```

In normal C++ code this will function without any problem if the declaration of b is written in b.h. However, in SWIG this would give the error message

```
Error: Nothing known about namespace 'b'
```

The preprocessor simply ignores the namespace declaration in b.h. A work-around to this problem is to add %include b.h before %include ns_example.h

in the interface file. This solution is based on making sure that SWIG sees the declaration of **namespace** b before it processes ns_example.h.

The same error will be generated if **namespace** b is not defined. This can be solved by defining the namespace in the interface file like

```
/* Interface file ns_example.i */

%module ns_example

%{
#include "ns_example.h"
%}

namespace b {};

%include "ns_example.h"
```

With the solutions converters.h, converters.py, and glue code it is possible to wrap BCL.

Chapter 6

Discussion and Conclusion

The result of having a C++ function wrapped using SWIG and having it accessible in Python can be very useful. The interactivity of Python is combined with the flexibility and fast execution time of C++. As can be seen from our example of wrapping the IT++ turbo class, SWIG enables a very convenient way of interacting with a C++ class from Python. The parameters can be quickly altered without the need to recompile. The resulting data of the C++ class is then available and can be manipulated and presented in Python.

If the library to be wrapped has few dependencies to other libraries and only contains standard C++ data types, then wrapping with SWIG is rather straight forward. The libraries IT++ and BCL however use IT++ defined data structures which introduces additional complexity. A favorable solution to handle these data structures would be a library similar to `std_vector.i`. However, BCL includes protected sections which SWIG does not support, therefore such a solution is not applicable in this case. Instead we have implemented a solution to cope with the restraints of BCL. It was achieved, not by wrapping the functions/classes themselves but rather by wrapping written glue code for each function/class.

In order to pass special data types between Python and C++ the created files `converters.h` and `converters.py` are used. This solution is somewhat cumbersome. Since Python is a typeless language the user needs to understand what data types are used by the functions and make the appropriate type conversions. Although writing this glue code might seem ponderous, it only needs to be done once. To decide whether it is beneficial to wrap modules in this way one should consider the amount of time and effort that is required

to write the glue codes compared to the amount of usage. To give an example, consider the IT++ turbo code. If we would have implemented the Monte Carlo loop in C++, re-compilation would have been needed even after the slightest change, such as changing generator vectors. Since Python is interpreted, by implementing the Monte Carlo loop in Python and importing the wrapped IT++ turbo code, we would avoid these re-compilations. If the turbo code will have to changed only a couple of times then maybe wrapping it will not be beneficial. However, if it will be altered often, perhaps in a testing environment, then using SWIG can have great advantages.

SWIG gives support for most standard C++ directives. There are however some limitations, such as that SWIG generally cannot handle private or protected sections. This can be seen as a flaw of the tool, given that these kind of data types are commonly used. There are work-arounds to this problem, but it requires additional effort and knowledge from the user (such as writing glue code). Ideally SWIG should handle this for the user. Apart from protected sections, if we consider the fact that BCL uses non standard C++ data types a solution similar to std_vector.i should be possible. This would remove the need for any glue code to be written by the user and would thereby bring out the full potential of SWIG.

Bibliography

[1] "About it++." http://itpp.sourceforge.net/current/index.html [Accessed 19 May 2011].

[2] "Templates." http://www.cplusplus.com/doc/tutorial/templates/ [Accessed 19 May 2011].

[3] D. M. Beazley, "Swig-2.0 documentation." http://www.swig.org/Doc2.0/SWIGDocumentation.pdf [Accessed 1 June 2011].

[4] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer magazine*, 1998.

[5] "Distutils." http://docs.python.org/distutils/setupscript.html [Accessed 19 May 2011].

[6] D. M. Beazley, "Swig-2.0 library." http://www.swig.org/Doc2.0/Library.html [Accessed 1 June 2011].

[7] C. Schlegel and L. C. Perez, *Trellis Coding*. Lightning Source Inc, 1997.

[8] S. Benedetto and E. Biglieri, *Principles of Digital Transmission With Wireless Applications*. Kluwer Academic/Plenum Publishers, 1999.

[9] J. G. Proakis and M. Salehi, *Digital Communications*. McGraw-Hill, 5th ed., 2008.

[10] U. Madhow, *Fundamentals of Digital Communications*. Cambridge University Press, 2008.

[11] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, vol. 20, pp. 284–287, March 1974.

[12] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding," in *Proc. IEEE International Conference on Communications*, pp. 1064–1070, May 1993.

[13] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: Performance analysis, design, and iterative decoding," *IEEE Trans. Inform. Theory*, vol. 44, pp. 909–926, 1998.

[14] S. Benedetto, D. Divsalar, F. Ieee, G. Montorsi, and F. Pollara, "A soft-input soft-output app module for iterative decoding of concatenated codes," *IEEE Commun. Lett*, vol. 1, pp. 22–24, 1997.

[15] "Install it++." http://itpp.sourceforge.net/stable/installation.html [Accessed 6 June 2011].

[16] "Install swig." http://sourceforge.net/projects/swig/files/swig/ [Accessed 6 June 2011].

[17] "install python." http://diveintopython3.org/installing-python.html #ubuntu [Accessed 9 June 2011].

[18] "Numpy." http://sourceforge.net/projects/numpy/files/NumPy/ [Accessed 9 June 2011].

Acronyms

API Application Programming Interface

APP a posteriori probability

AWGN Additive White Gaussian Noise

BCL Baseband Core Library

BER Bit Error Rate

blas Basic Linear Algebra Subprograms

BPSK Binary Phase Shift Keying

cblas Complex Basic Linear Algebra Subprograms

fftw Fastest Fourier Transform in the West

LAPACK Linear Algebra PACKage

MAP Maximum a Posteriori Probability

ML Maximum Likelihood

MLSD Maximum Likelihood Sequence Decoding

OS Operating System

PCCC Parallel Concatenated Convolutional Codes

RCC Recursive Convolution Codes

SCCC Serially Concatenated Convolutional Codes

SISO Soft Input Soft Output

SNR Signal to Noise Ratio

STL Standard Template Library

SWIG Simplified Wrapper Interface Generator

Appendix A

User's Guide

This guide describes the steps required to wrap the encode() function in IT++ with SWIG and integrate it with Python. Installations vary depending on the used Operating System (OS). These instructions are written for Linux Ubuntu.

A.1 Setup

Before writing any code we need to install all required software.

1. Install IT++ following the instructions in [15].

2. Install SWIG. Most popular Linux distributions come with SWIG but the version may vary depending on which distribution. The version used in this thesis is 2.0.1 and it can be downloaded from [16].

3. Install Python. Most popular Linux distributions come with Python but the version may vary depending on which distribution. The version used in this thesis is 3.1.1 Follow the instructions in [17] to install. The numpy package is needed in order to use functions such as rand(), sqrt(), etc. Version 1.6.0 is compatible with Python 3.x and can be found at [18].

A.2 Wrap

With all tools installed we can now start to wrap.

1. Create the interface file, see section 3.2.1 and appendix B.1.3. Depending on what type of data structures we want to pass between Python

and C++ other instantiations of std::vector can be made. New converters can be written using std::vector.

2. Create the glue code and the corresponding header file for the chosen target function, see section 5.1, appendix B.1.1 and B.1.2. Make sure converters.h is included in the .cc-file.

3. Write the setup file, see section 3.2.2 and appendix B.1.4. As mentioned in section 3.2.2 there are two ways to generate the extension module. Here we use the second mentioned method, therefore, give the interface file, target file and converters file as input to sources. When including libraries, make sure they are built with the same gcc compiler version as the one used for running SWIG.

4. Run the following command to generate the wrapper, python module and shared library:

```
python setup.py build_ext --inplace
```

5. We can now import the python module into Python and use it together with converters.py to pass data to the wrapped function. This can be done in the command prompt like

```
>>>from demo import *
>>>from converters import *
```

or it can be included into a .py-file, see appendix B.1.5.

Appendix B

Code

B.1 SWIG modules

B.1.1 demo.cc

```
/** The functions in this file acts as glue code for calling IT
    ++ functions.
*/

#include <iostream>
#include <string>
#include <vector>
#include <itbase.h>
#include <bcl.h>
#include "turbo.h"
#include "modulator.h"
#include "error_counters.h"
#include "demo.h"
#include "converters.h"

using namespace std;

// Global objects
itpp::Turbo_Codec turbo;



/** Calls the IT++ function decode
    @param rec A vector containing received data
    @return A vector containing the decoded received data
```

```
*/
vector<int> decode(vector<double> rec) {
  itpp::bvec dec_bits;
  itpp::ivec iter;
  turbo.decode(vecd_to_vec(rec), dec_bits, iter);
  return bvec_to_veci(dec_bits);
}


/** Calls the IT++ function encode
    @param info_bits A vector containing information bits
    @return A vector containing the encoded information bits
*/
vector<int> encode(vector<int>& info_bits) {
  itpp::bvec output_bits;
  turbo.encode(veci_to_bvec(info_bits), output_bits);
  return bvec_to_veci(output_bits);
}


/** Calls the IT++ function wcdma_turbo_interleaver_sequence
    @param n Interleaver length
    @return A vector containing the interleaver sequence
*/
vector<int> get_interleaver(int n){
  return ivec_to_veci(itpp::wcdma_turbo_interleaver_sequence( n
      ));
}


/** Calls the IT++ function set_parameters. Initialize turbo for
    log max decoding
    @param gen1 A vector containing the generator polynomials
       for the first constituent encoder
    @param gen2 A vector containing the generator polynomials
       for the second constituent encoder
    @param constraint_length The constraint length of the two
       constituent encoders
    @param interleaver Interleaver sequence
    @param in_iterations The number of decoding iterations
    @param in_logmax_scale_factor This parameter allows for a
       down-scaling of the extrinsic information that will be
       passed on to the next decoder
*/
void init_turbo_max(vector<string>& gen1,vector<string>& gen2,
```

```
    int constraint_length ,vector<int>& interleaver, int
    iterations, double in_logmax_scale_factor) {
  itpp::ivec gen1_ivec = vecs_to_ivec(gen1);
  itpp::ivec gen2_ivec = vecs_to_ivec(gen2);
  itpp::ivec interleaver_ivec = veci_to_ivec(interleaver);
  turbo.set_parameters(gen1_ivec, gen2_ivec, constraint_length,
      interleaver_ivec, iterations, "LOGMAX",
      in_logmax_scale_factor);
}


/** Calls the IT++ function set_parameters. Initialize turbo for
    log map decoding
    @param gen1 A vector containing the generator polynomials
       for the first constituent encoder
    @param gen2 A vector containing the generator polynomials
       for the second constituent encoder
    @param constraint_length The constraint length of the two
       constituent encoders
    @param interleaver Interleaver sequence
    @param in_iterations The number of decoding iterations
*/
void init_turbo_map(vector<string>& gen1,vector<string>& gen2,
    int constraint_length,vector<int>& interleaver, int
    iterations) {
  itpp::ivec gen1_ivec = vecs_to_ivec(gen1);
  itpp::ivec gen2_ivec = vecs_to_ivec(gen2);
  itpp::ivec interleaver_ivec = veci_to_ivec(interleaver);
  turbo.set_parameters(gen1_ivec, gen2_ivec, constraint_length,
      interleaver_ivec, iterations, "LOGMAP");
}


/** Set scaling factor
    @param Lc Scaling factor
*/
void set_scale_factor(double Lc){
  turbo.set_scaling_factor(Lc);
 }
```

B.1.2 demo.h

```
#include <vector>
using namespace std;
```

```
vector<int> decode(vector<double> rec);
vector<int> encode(vector<int>& info_bits);
vector<int> get_interleaver(int n);
void init_turbo_max(vector<string>& gen1,vector<string>& gen2,
    int constraint_length,vector<int>& interleaver, int
    iterations, double in_logmax_scale_factor);
void init_turbo_map(vector<string>& gen1,vector<string>& gen2,
    int constraint_length,vector<int>& interleaver, int
    iterations);
void set_scale_factor(double Lc);
```

B.1.3 demo.i

```
%module demo

%include "std_string.i"
%include "std_vector.i"

%{
#include <itbase.h>
#include "demo.h"
%}

// Instantiations of vector (std_vector.i)
namespace std {
  %template(vectori) vector<int>;
  %template(vectord) vector<double>;
  %template(vectors) vector<std::string>;
};

%include "demo.h"
```

B.1.4 setup.py

```
from distutils.core import *
demo_module = Extension(
    '_demo',
    sources = ['demo.i', 'demo.cc', 'converters.cc'],

    include_dirs = ['example/path/to/include/file/'],

    library_dirs=['example/path/to/library/'],

    libraries=['lapack','blas','fftw3','cblas','itpp-external','
        lapack-noopt','gfortran','itpp'],
```

```
    swig_opts=["-c++"]
)
setup(
    name = 'demo',
    version = '3.0',
    author      = "Oskar Arivdsson Tjader, Linh Tran",
    description = "swig demo using IT++",
    ext_modules = [demo_module],
    py_modules = ["demo"]
)
```

B.1.5 turbo.py

```
# Note: The IT++ decoder does not return the different BER for
    each iteration but only for the last. Since we want to show
    its performance for each iteration and SNR value we need to
    instantiate a new decoder for each iteration of each \ac{SNR}
     to get around this

from demo import *
from converters import *

from help_funcs import *
from com_funcs import*
from help_funcs import*

from numpy import array

def main():

    BLOCK_SIZE = 16384
    constraint_length = 4
    gen = ('013', '015')
    metric = 'LOGMAP'
    log_scale_factor = 0.7

    SNR_dB_min      = 0         # min SNR
    SNR_dB_max      = 50        # max SNR
    SNR_dB_step     = 5         # SNR granularity
    Eb              = 1         # Bit (symbol) energy

    MIN_NO_ERRORS   = 100       # Limit for number of errors (
        monte carlo)
    MIN_NO_BLOCKS   = 100       # Limit for number of
```

```
    transmitted blocks (monte carlo)
ITERATIONS      = 12          # No of iterations in turbo
    decoder


SNR_dB_vec = range(SNR_dB_min, SNR_dB_max + SNR_dB_step,
    SNR_dB_step)
SNR_dB_vec_real = array(SNR_dB_vec,float)/100

# generate interleaver
interleaver =  list_to_veci(get_interleaver(BLOCK_SIZE))

for SNR_idx in SNR_dB_vec_real:
    len_sym=49164
    rate = BLOCK_SIZE/len_sym
    SNR_linear = pow(10, SNR_idx/10)
    N0 = (1/SNR_linear)/rate
    var=N0/2

    for iteration in range(ITERATIONS):

            random.seed(1)
            gen_veci = list_to_vecs(gen)
            if  (metric =='LOGMAX'):
                    init_turbo_max(gen_veci, gen_veci,
                        constraint_length, interleaver,
                        iteration+1, metric, log_scale_factor
                        )
            else:
                    init_turbo_map(gen_veci, gen_veci,
                        constraint_length, interleaver,
                        iteration+1, metric)

            # Generate information bits
            info_bits =list_to_veci(create_data(BLOCK_SIZE))

            # Encode information bits
            output_bits = list_to_veci(encode(info_bits))

            # Modulate bits
            symbols = modulate(output_bits)

            rate = (float(BLOCK_SIZE)/len(symbols))
            No_err = 0
            no_blocks = 0
```

```
while (No_err < MIN_NO_ERRORS) or (no_blocks <
    MIN_NO_BLOCKS):

        noise = create_AWGN(len_sym,var)
        rec = list_to_vecd(symbols + noise)

        #set scale factor
        scale_factor = 4/N0
        set_scale_factor(scale_factor)

        # Decode recived symbols
        dec_bits = decode(rec)

        # Count errors
        No_err += count_errors(info_bits,
            dec_bits)
        no_blocks += 1

ber = No_err/(no_blocks*BLOCK_SIZE)
print( "SNR : " ,SNR_idx)
print( " iteration : ",iteration+1)
print ( " BER : ", ber)
filename='BER'+metric+'.txt'
with open(filename, mode='a', encoding='utf-8')
    as a_file:
     a_file.write(" " + str(ber))
```

B.1.6 converters.cc

```
/** This file contains functions for convertation between itpp::
    Vec<Type> and std::vector<Type>
*/

#include <vector>
using namespace std;
#include "converters.h"


/** Convert vector of type itpp::bvec to std::vector<int>
 *  @param v1 vector of binaries
 *  @return vector
*/
vector<int> bvec_to_veci(itpp::bvec v1) {
  vector<int> v2;
  for (int i=0; i<v1.size(); i++) {
```

```
    v2.push_back(v1[i]);
  }
  return v2;
}

/** Convert vector of type itpp::ivec to std::vector<int>
 *  @param v1 vector of integers
 *  @return vector
*/
vector<int> ivec_to_veci(itpp::ivec v1) {
  vector<int> v2;
  for (int i=0; i<v1.size(); i++) {
    v2.push_back(v1[i]);
  }
  return v2;
}

/** Convert vector of type itpp::vec to std::vector<double>
 *  @param v1 vector of decimal values
 *  @return vector
*/
vector<double> vec_to_vecd(itpp::vec v1) {
  vector<double> v2;
  for (int i=0; i<v1.size(); i++) {
    v2.push_back(v1[i]);
  }
  return v2;
}

/** Convert vector of type std::vector<int> to itpp::bvec
 *  @param v1 vector of integers
 *  @return vector
*/
itpp::bvec veci_to_bvec(vector<int> v1) {
  itpp::bvec v2 = itpp::zeros_b(v1.size());
  for (int i=0; i<v1.size(); i++) {
    v2[i] = v1[i];
  }
  return v2;
}

/** Convert vector of type std::vector<int> to itpp::ivec
 *  @param v1 vector of integers
 *  @return vector
*/
```

```
itpp::ivec veci_to_ivec(vector<int> v1) {
  itpp::ivec v2 = itpp::zeros_i(v1.size());
  for (int i=0; i<v1.size(); i++) {
      v2[i] = v1[i];
  }
  return v2;
}

/** Convert vector of type std::vector<string> to itpp::ivec
 *  @param v1 vector of strings
 *  @return vector
*/
itpp::ivec vecs_to_ivec(vector<string> v1){
  itpp::ivec v2 = itpp::zeros_i(v1.size());
  string tmp;
  for(int j=0; j< v1.size(); j++){
      tmp = "";
      for (int i=0; i<v1[j].size(); i++) {
          tmp += v1[j][i];
      }
      v2[j] = atoi(tmp.c_str());
  }
  return v2;
}

/** Convert vector of type std::vector<int> to itpp::vec
 *  @param v1 vector of integers
 *  @return vector
*/
itpp::vec veci_to_vec(vector<int> v1) {
  itpp::vec v2 = itpp::zeros(v1.size());
  for (int i=0; i<v1.size(); i++) {
    v2[i] = v1[i];
  }
  return v2;
}

/** Convert vector of type std::vector<double> to itpp::vec
 *  @param v1 vector of double
 *  @return vector
*/
itpp::vec vecd_to_vec(vector<double> v1) {
  itpp::vec v2 = itpp::zeros(v1.size());
  for (int i=0; i<v1.size(); i++) {
    v2[i] = v1[i];
```

```
  }
  return v2;
}
```

B.1.7 converters.h

```
#include <itbase.h>
#include <vector>
using namespace std;

vector<int> bvec_to_veci(itpp::bvec v1);          // itpp::bvec
    to vector<int>
vector<int> ivec_to_veci(itpp::ivec v1);          // itpp::ivec
    to vector<int>
vector<double> vec_to_vecd(itpp::vec v1);         // itpp::vec
    to vector<int>
itpp::bvec veci_to_bvec(vector<int> v1);          // vector<int>
     to itpp::bvec
itpp::ivec veci_to_ivec(vector<int> v1);          // vector<int>
     to itpp::ivec
itpp::ivec vecs_to_ivec(vector<string> v1);       // vector<
    string> to itpp::ivec
itpp::vec veci_to_vec(vector<int> v1);            // vector<int>
     to itpp::vec
itpp::vec vecd_to_vec(vector<double> v1);         // vector<
    float> to itpp::vec
```

B.1.8 converters.py

from demo import vectori from demo import vectord from demo import vectors

```
# Convert a "python_list" to vectori (vector<int>)
# inparam: a list of integers
# return: a vector<int>
def list_to_veci(v1):
    v2 = vectori(len(v1))
    for i in range(len(v1)):
        v2[i] = v1[i]
    return v2

# Convert a "python_list" to vectorf (vector<float>)
# inparam: a list of floats
# return: a vector<float>
def list_to_vecd(v1):
    v2 = vectord(len(v1))
```

```
    for i in range(len(v1)):
        v2[i] = v1[i]
    return v2

# Convert a "python_list" to vectors (vector<string>)
# inparam: a list of strings
# return: a vector<string>
def list_to_vecs(v1):
    v2 = vectors(len(v1))
    for i in range(len(v1)):
        v2[i] = v1[i]
    return v2
```

B.1.9 com_funcs.py

```
from help_funcs import *

from numpy import array
from numpy import random
from numpy import sqrt
from numpy import zeros
from numpy import transpose
from math import ceil


#
    ------------------------------------------------------------------------


# BPSK modulation.
# inparam: a bit
# return: an int (-1 or 1)
def BPSK(bit):
    return -bit*2 + 1
#
    ------------------------------------------------------------------------


# Generate stream of ones and zeros, (normally distributed), e.g
    . array([1,0,1,1,0])
# inparam: length of generated stream
# return: array of ones and zeros
def create_data(n):
    return random.randint(0,2,(n,)).tolist()
#
    ------------------------------------------------------------------------
```

```
# Generate array with AWGN
# inparam1: length of generated stream
# inparam2: noise variance
# return: array of floats
def create_AWGN(n, variance):
    return sqrt(variance)*random.randn(n)
#
    ------------------------------------------------------------------------

# Hard decision -1 -> 0 ,  1 -> 1
# inparam: list of -1 and 1
# return: list of 1 and 0
def hard_dec(data):
    for i in range(len(data)):
        if (data[i]>0): data[i]=1
        else: data[i]=0
    return data
#
    ------------------------------------------------------------------------

# Interleaver
# inparam: length of array
# return: list or array with interleaved elements and array with
    indecies
def gen_interleaver(n):
    interleaver=range(n)
    random.shuffle(interleaver)
    return interleaver
#
    ------------------------------------------------------------------------

# Interleaver
# inparam: list or array of ints or floats
# return: list or array with interleaved elements and array with
    indecies
def interleave(bits, interleaver):
    bits=str2array(bits)
    interleaved_bits=bits[interleaver]
    return interleaved_bits.tolist()
#
    ------------------------------------------------------------------------

# Deinterleaver
# inparam1: list or array of ints or floats
```

```
# inparam2: array with indecies
# return: list or array with deinterleaved elements
def deinterleaver(interleaved_bits, interleaver):
    str2array(interleaved_bits)
    deinterleaved_bits=zeros(len(interleaved_bits), float)
    deinterleaved_bits[interleaver]=interleaved_bits

    return deinterleaved_bits.tolist()
#
    ------------------------------------------------------------------------

# Modulation with BPSK
# inparam: list of strings e.g ['1','1','0']
# return: list of ints [1,1,-1]
def modulate(data):
    mod_data = []
    for d in data:
        mod_data.append(BPSK(int(d)))
    return mod_data
#
    ------------------------------------------------------------------------

# Normalize along columns in a matrix
# inparam: a matrix with ints or floats
# return: column-normalized matrix
def norm_mtrx_col(mtrx):
    for col in range(shape(mtrx)[1]):
        if (sum(mtrx[:,col]) != 0):
            mtrx[:,col]=mtrx[:,col]/sum(mtrx[:,col])
    return mtrx
#
    ------------------------------------------------------------------------

# Count number of errors
def count_errors(info_data, hard_data):
        return len(transpose((array(info_data)-array(hard_data))
            .nonzero()))
#
    ------------------------------------------------------------------------

# Normalize along columns in a matrix
# inparam: an array
# return: normalized array
def norm(array):
    sum_array=sum(array)
```

```
    if (sum_array != 0):
        for idx in range(len(array)):

            array[idx]=float(array[idx])/sum_array
    return array
```

B.2 Turbo Code

B.2.1 turbo.py

```
#
    ------------------------------------------------------------------------------


BLOCK_SIZE = 5
GENERATORS      = ('11','1101')
RATE_INNER      = 1.0/2
RATE_OUTER      = 1.0/2
RATE_TOT= RATE_OUTER*RATE_INNER

SNR_dB_min      = -25       # min SNR
SNR_dB_max      = -25       # max SNR
SNR_dB_step     = 5         # SNR granularity
Eb              = 1         # Bit (symbol) energy

MIN_NO_ERRORS   = 0         # Limit for number of errors (monte
    carlo)
MIN_NO_BLOCKS   = 1         # Limit for number of transmitted
    blocks (monte carlo)
ITERATIONS      = 7         # No of iterations in turbo decoder

SNR_dB_vec = range(SNR_dB_min, SNR_dB_max + SNR_dB_step,
    SNR_dB_step)
#
    ------------------------------------------------------------------------------

import scipy.io as sio
from outer_encoder import *
from inner_encoder import *
from BCJR_inner import *
from BCJR_outer import *
from help_funcs import *
from com_funcs import*

from numpy import array
```

```
from numpy import random
from numpy import sqrt
from numpy import zeros
from numpy import ones
from numpy import transpose
from math import ceil
import matplotlib.pyplot as plt

#
-------------------------------------------------------------------------------

# Main
def main():


    n = len(GENERATORS)          # length of one code word
    k = int(RATE_OUTER*n)        # length of one input
    mem = 0
    for g in GENERATORS:
        mem = max(mem, len(g))

    BER_mtrx=zeros((ITERATIONS, len(SNR_dB_vec)))
    info_bits = create_data(BLOCK_SIZE)     # generate
        information bits
    info_bits_array = array(info_bits.tolist() + zeros(mem-1,int
        ).tolist()) # append tail
    info_bits=array2str(info_bits_array)

    outer_enc = OUTER_ENC(GENERATORS,RATE_OUTER,mem) #
        instantiate outer encoder
    ff_table_outer, fb_table_outer, code_table_ones_outer,
        code_table_zeros_outer = outer_enc.create_trans_table() #
         create outer alpha, beta tables
    coded_bits_outer = outer_enc.encode(info_bits) # outer
        encode

    interleaver = gen_interleaver(len(coded_bits_outer))
    coded_bits_outer=interleave(coded_bits_outer, interleaver) #
         interleave coded bits
    coded_bits_outer=array2str(array(coded_bits_outer,int))

    inner_enc = INNER_ENC(GENERATORS,RATE_INNER,mem) #
        instantiaten inner encoder
    ff_table_inner, fb_table_inner, code_table_ones,
        code_table_zeros = inner_enc.create_trans_table()  #
```

```
    create inner alpha, beta tables
coded_bits, state = inner_enc.encode(coded_bits_outer, ''.
    zfill(mem-1)) # inner encode

# inner encoder is recursive -> terminating bits depend on
    state
if (state[0]=='1'):
    term_bits='100'
else:
    term_bits='000'
coded_term, state_term=inner_enc.encode(term_bits, state) #
    encode terminating bits
coded_bits=coded_bits+coded_term # append tail

mod_data = modulate(coded_bits) # modulate

t_tot_inner = int(ceil(len(mod_data)*RATE_INNER)) # number
    of transmitted symbols per block

BER_vec = []
BER_vec_uncod = []
BER_vec_turbo = []
BER_turbo_iter = zeros(ITERATIONS)
no_err_turbo_iter = zeros(ITERATIONS)
SNR_dB_vec_real = array(SNR_dB_vec, float)/100

bcjr_inner = BCJR_inner(GENERATORS,RATE_INNER, t_tot_inner,
    mem) # instantiate BCJR inner
bcjr_outer = BCJR_outer(GENERATORS,RATE_OUTER, int(ceil(len(
    coded_bits_outer)*RATE_OUTER)),mem) # instantiate BCJR
    outer

for SNR_idx in range(len(SNR_dB_vec_real)):

    SNR_linear = pow(10, (float(SNR_dB_vec_real[SNR_idx]))
        /10)

    N0 = (float(Eb)/SNR_linear)/(float(BLOCK_SIZE)/len(
        mod_data))
    var = float(N0)/2     # noise variance (for BPSK)
    N0_uncod = float(Eb)/SNR_linear
    var_uncod = float(N0_uncod)/2
    no_err = 0            # no of errors for a given SNR
    no_err_uncod = 0
```

```
no_blocks = 0          # number of transmitted blocks for
    a given SNR
print "************************ SNR_dB: " + str(float
    (SNR_dB_vec_real[SNR_idx])) + "
    *************************"

while (no_err < MIN_NO_ERRORS) or (no_blocks <
    MIN_NO_BLOCKS):
    noise = create_AWGN(len(mod_data),var)
    rec_data = mod_data + noise

    uncod_data = modulate(info_bits) + create_AWGN(len(
        info_bits),var_uncod)
    app_list = (0.5*ones((t_tot_inner))).tolist() # init
        a priori vector to inner decoder

    cost_mtrx_ex = bcjr_inner.calc_cost_mtrx_ex(rec_data
        ,ff_table_inner,var) # gamma extrinsic
    print "*************** MONTE_CARLO " + str(
        no_blocks+1) + " ***************"
    for u in range(ITERATIONS):
        print "--------- ITERARTION " + str(u+1) + "
            ---------"
        bcjr_inner = BCJR_inner(GENERATORS,RATE_INNER,
            t_tot_inner,mem) # instantiate BCJR inner
        bcjr_outer = BCJR_outer(GENERATORS,RATE_OUTER,
            int(ceil(len(coded_bits_outer)*RATE_OUTER)),
            mem) # instantiate BCJR outer

        cost_mtrx_inner, Pr_FF_inner = bcjr_inner.
            ff_decode(rec_data,ff_table_inner,var,
            app_list, cost_mtrx_ex) # alpha recursion,
            calculate gamma
        Pr_FB_inner = bcjr_inner.fb_decode(rec_data,
            fb_table_inner,cost_mtrx_inner) # beta
            recursion
        sigma_mtrx_inner = bcjr_inner.calc_sigma(
            cost_mtrx_ex, Pr_FB_inner, Pr_FF_inner,
            ff_table_inner) # calculate sigma extrinsic
        prob_list = bcjr_inner.calc_prob(
            sigma_mtrx_inner, app_list) # calculate a
            posteriori to outer decoder (extrinsic info)
        prob_list=deinterleaver(prob_list, interleaver)
            # deinterleave a posteriori
```

```
            cost_mtrx, Pr_FF = bcjr_outer.ff_decode(
                ff_table_outer, prob_list) # alpha recursion,
                 calculate gamma
            Pr_FB = bcjr_outer.fb_decode(prob_list,
                fb_table_outer,cost_mtrx) # beta recursion
            sigma_mtrx = bcjr_outer.calc_sigma(cost_mtrx,
                Pr_FB, Pr_FF, ff_table_outer) # calculate
                sigma
            app_list = bcjr_outer.calc_app(sigma_mtrx,
                code_table_ones_outer, code_table_zeros_outer
                , prob_list) # calculate a priori to inner
                decoder
            app_list=interleave(app_list, interleaver) #
                interleave a priori

            app_list = padd(app_list,0.5,mem-1) # append
                tail

            hard_turbo = bcjr_outer.hard_decision(sigma_mtrx
                , BLOCK_SIZE) # hard decision for this
                iteration

            no_err_turbo = sum(hard_turbo != info_bits_array
                [0:BLOCK_SIZE])
            print "no_err_turbo : " + str(no_err_turbo)
            BER_turbo = float(no_err_turbo)/BLOCK_SIZE
            print "BER: " + str(BER_turbo)
            BER_mtrx[u, SNR_idx] += no_err_turbo

        no_blocks += 1
        hard_uncod = hard_dec(uncod_data) # hard decision
            for this SNR and block
        no_err += no_err_turbo
        print "no_err : " + str(no_err)
        BER = float(no_err)/(no_blocks*BLOCK_SIZE)
        print "BER: " + str(BER)
        no_err_uncod += sum(hard_uncod != info_bits_array)
        BER_uncod = float(no_err_uncod)/(no_blocks*
            BLOCK_SIZE)
        BER_turbo_iter=array(no_err_turbo_iter, float)/(
            no_blocks*BLOCK_SIZE)


    BER_vec_turbo.append(BER_turbo_iter/(no_blocks))
```

```
            BER_vec.append(BER)
            BER_vec_uncod.append(BER_uncod)
            BER_mtrx[:,SNR_idx] /= (no_blocks*BLOCK_SIZE)

            print "BER_mtrx: " + str(BER_mtrx)
            print "  no_blocks  :  " +str(no_blocks)

        sio.savemat('ber_mtrx.mat', {'BER_mtrx':BER_mtrx})
        plt.figure(1)
        plt.subplot(211)
        plt.semilogy(SNR_dB_vec_real,BER_vec,'-o')
        plt.semilogy(SNR_dB_vec_real,BER_vec_uncod,'-x')
        plt.ylabel('BER')
        plt.xlabel('SNR [dB]')
        plt.title('BER vs SNR')
        plt.legend(('coded','uncoded'))

        plt.subplot(212)
        for vec in BER_mtrx[:,]:
            plt.semilogy(SNR_dB_vec_real,vec,'-o')

        plt.ylabel('BER 1- ' + str(len(BER_vec_turbo)) + '
            iterations')
        plt.xlabel('SNR [dB]')
        plt.title('BER vs SNR')
        plt.show()
```

B.2.2 com_funcs.py

```
from help_funcs import *

from numpy import array
from numpy import random
from numpy import sqrt
from numpy import zeros
from numpy import transpose


#
    ------------------------------------------------------------------------------

# BPSK modulation.
# inparam: a bit
# return: an int (-1 or 1)
def BPSK(bit):
```

```
    return bit*2 - 1
#
    ------------------------------------------------------------------------

# Generate stream of ones and zeros, (normally distributed), e.g
    . array([1,0,1,1,0])
# inparam: length of generated stream
# return: array of ones and zeros
def create_data(n):
    return random.randint(0,2,(n,))
#
    ------------------------------------------------------------------------

# Generate array with AWGN
# inparam1: length of generated stream
# inparam2: noise variance
# return: array of floats
def create_AWGN(n, variance):
    return sqrt(variance)*random.randn(n)
#
    ------------------------------------------------------------------------

# Hard decision -1 -> 0 ,  1 -> 1
# inparam: list of -1 and 1
# return: list of 1 and 0
def hard_dec(data):
    for i in range(len(data)):
        if (data[i]>0): data[i]=1
        else: data[i]=0
    return data
#
    ------------------------------------------------------------------------

# Interleaver
# inparam: length of array
# return: list or array with interleaved elements and array with
    indecies
def gen_interleaver(n):
    interleaver=range(n)
    random.shuffle(interleaver)
    return interleaver
#
    ------------------------------------------------------------------------

# Interleaver
```

```
# inparam: list or array of ints or floats
# return: list or array with interleaved elements and array with
    indecies
def interleave(bits, interleaver):
    bits=str2array(bits)
    interleaved_bits=bits[interleaver]
    return interleaved_bits.tolist()
#
    ------------------------------------------------------------------------

# Deinterleaver
# inparam1: list or array of ints or floats
# inparam2: array with indecies
# return: list or array with deinterleaved elements
def deinterleaver(interleaved_bits, interleaver):
    str2array(interleaved_bits)
    deinterleaved_bits=zeros(len(interleaved_bits), float)
    deinterleaved_bits[interleaver]=interleaved_bits

    return deinterleaved_bits.tolist()
#
    ------------------------------------------------------------------------

# Modulation with BPSK
# inparam: list of strings e.g ['1','1','0']
# return: list of ints [1,1,-1]
def modulate(data):
    mod_data = []
    for d in data:
        mod_data.append(BPSK(int(d)))
    return mod_data
#
    ------------------------------------------------------------------------

# Normalize along columns in a matrix
# inparam: a matrix with ints or floats
# return: column-normalized matrix
def norm_mtrx_col(mtrx):
    for col in range(shape(mtrx)[1]):
        if (sum(mtrx[:,col]) != 0):
            mtrx[:,col]=mtrx[:,col]/sum(mtrx[:,col])
    return mtrx
#
    ------------------------------------------------------------------------
```

```
# Count number of errors
# inparam1: original information bits
# inparam2: estimated bits
def count_errors(info_data, hard_data):
        return len(transpose((info_data-hard_data).nonzero()))
#
    ------------------------------------------------------------------------------

# Normalize along columns in a matrix
# inparam: an array
# return: normalized array
def norm(array):
    sum_array=sum(array)
    if (sum_array != 0):
        for idx in range(len(array)):

            array[idx]=float(array[idx])/sum_array
    return array
```

B.2.3 help_funcs.py

```
from numpy import array
from numpy import random
from numpy import sqrt
from numpy import zeros
from numpy import shape

#
    ------------------------------------------------------------------------------

# Append a value (int or float) to the end of a list
# inparam1: a list
# inparam2: number of elements to be appended
# return: padded list
def padd(l,v,n):
    for i in range(n):
        l.append(v)
    return l
#
    ------------------------------------------------------------------------------

# Append zeros to the end of a string
# inparam1: a string
# inparam2: number of zeros to be appended
# return: zero padded string
```

```
def append_zeros(string,n):
    for i in range(n):
        string = string + '0'
    return string
#
    ------------------------------------------------------------------------------

# Binary to decimal conversion
# inparam1: a binary, string or list
# return: decimal representation of binary
def bin2dec(number):
    if isinstance(number, str):
        return int(number, 2)
                                                        # e.g.,
            '110' becomes 6
    else:
        return sum(number * pow(2, array(range(len(number)-1,
            -1, -1)))) # e.g., [1, 1, 0] becomes 6

#
    ------------------------------------------------------------------------------

# Decimal to binary conversion
# inparam1: decimal number
# inparam2: length of the binary output
# return: binary representation of decimal
def dec2bin(dec_num, bin_length):
    return bin(int(dec_num)).lstrip('-0b').zfill(bin_length)

#
    ------------------------------------------------------------------------------

# Returns all binary combinations for an integer e.g. 4 ->
    ['00','01','10','11']
# inparam1: an integer
# inparam2: length of the binary output
# return: list of binary combinations (strings)
def bin_range(bin_range, bin_length):
    l = []
    for i in range(bin_range):
        l.append(dec2bin(i, bin_length))
    return l

#
    ------------------------------------------------------------------------------
```

```
# Converts an array to string e.g. array([1,0,1]) -> '101'
# inparam1: array
# return: string
def array2str(a):
    string = ''
    for elem in a:
        string = string + str(elem)
    return string

#
    ------------------------------------------------------------------------

# Converts a string to array e.g. '101' -> array([1,0,1])
# inparam1: string
# return: array
def str2array(string):
    array=zeros(len(string))
    for i in range(len(string)):
        array[i]=(string[i])
    return array

#
    ------------------------------------------------------------------------

# Converts a string to list of strings e.g '001' ->
    ['0','0','1']
# inparam1: string
# return: list of strings
def str2list(string):
    return str(string)[1:-1].split(',')

#
    ------------------------------------------------------------------------

# Finds the index of the maximum value for each column in a
    matrix
# inparam: matrix
# return: list of indecies
def maxidx_mtrx_col(Mtrx):
    max_ind=zeros(((shape(Mtrx))[1]),int)
    for col in range((shape(Mtrx))[1]):
        max_ind[col]=((Mtrx[:,col]).tolist()).index(max(Mtrx[:,
            col]))
    return max_ind
```

B.2.4 outer_encoder.py

```
from numpy import dot
from numpy import transpose
from numpy import zeros
from math import log
from numpy import append
from help_funcs import *
from operator import xor

#
-------------------------------------------------------------------------------

class OUTER_ENC:

    # Contructor
    def __init__(self,generators,rate,mem):
        self.n = len(generators)                          # number
            of codewords
        self.k = int(rate*self.n)                         # number
            of input bits
        self.generators = generators                      #
           generator polynomials
        self.rate = rate                                  # code
           rate = k/n
        self.mem = mem                                    # number
            of memory elements of the shift register

        self.no_states = pow(2,self.mem-self.k)           # number
            of states in trellis
        self.state_len = int(log(self.no_states,2))       # number
            of bits representing one state
        self.no_inputs = pow(2,self.k)                    # number
            of inputs to encoder

        self.gen_list = []                                # list
           containing all generators
        for g in self.generators:
            g_padded = g.ljust(self.mem,'0')
            self.gen_list.append(g_padded)                # zero
               pad so that all generators have same length
```

```
# Create transition tables
# feed forward table (alpha): e.g. {0: [{'to': 0, 'in': 0, '
    out': '00'}, {'to': 2, 'in': 1, 'out': '11'}]}
# The number before ':' indicates from which state we are
    going.
# feed backward table (beta): e.g. {0: [{'out': '00', 'from
    ': 0, 'in': '0'}, {'out': '11', 'from': 1, 'in': '0'}]}
# The number before ':' indicates to which state we are
    going.
# code_table_ones indicates which branches give a '1' as
    first output bit.
# code_table_zeros indicates which branches give a '0' as
    first output bit.
# return: feed forward table, feed backward table,
    code_table_ones, code_table_zeros
def create_trans_table(self):
    ff_table = dict()
    fb_table = dict()
    code_table_ones = dict()
    code_table_zeros = dict()

    inputs = bin_range(pow(2,self.k),self.k)        # get
        all input combinations, pow(2,self.k) = number of
        inputs to encoder
    for state in range(self.no_states):
        ff_table[state] = list()
        fb_table[state] = list()

    for n in range(self.n):
        code_table_ones[n] = list()
        code_table_zeros[n] = list()

    for old_state in bin_range(self.no_states,self.state_len
        ):            # Iterate over all states
        for in_bits in inputs:
            output, new_state = self.encoder(in_bits,
                old_state)
            old_idx = bin2dec(old_state)
            new_idx = bin2dec(new_state)
            ff_table[old_idx].append({'to':new_idx, 'in':
                in_bits, 'out':output})
            fb_table[new_idx].append({'from':old_idx, 'in':
                in_bits, 'out':output})
            for i,o in zip(range(len(output)),output):
                if(o == '1'):
```

```
                    code_table_ones[i].append(int(old_state
                        ,2)*self.no_inputs+int(in_bits,2))
                else:
                    code_table_zeros[i].append(int(old_state
                        ,2)*self.no_inputs+int(in_bits,2))
        return ff_table, fb_table, code_table_ones,
            code_table_zeros



    # Encodes a stream of info bits and returns the output
    # inparam: a string of information bits
    # return: a string of coded bits
    def encode(self, info_bits):
        outputs = ""
        old_state = ''.zfill(self.mem-1)
        for i in range(0,len(info_bits), self.k):
            output, old_state = self.encoder(info_bits[i:i+self.
                k], old_state)
            outputs = outputs + output
        return outputs



    # Encodes the input bits for one transition in trellis and
        returns the output from one transition in trellis and the
         new state
    # inparam1: a string of information bits
    # inparam2: the state of encoder (a string)
    # return: output of encoder (string), new state of encoder (
        string)
    def encoder(self,in_bits, state):
        states_internal = in_bits+state
        coded_bits=""
        for gen in self.gen_list:
            coded_bits =coded_bits + str(reduce(xor,map(int,list
                (bin(int(gen,2)&int(states_internal,2)))[2:])))
        state = states_internal[:-self.k:]
        return coded_bits, state
```

B.2.5 inner_encoder.py

```
from numpy import dot
from numpy import transpose
from numpy import zeros
```

```
from math import log
from numpy import append
from help_funcs import *
from com_funcs import *
from operator import xor

#
-------------------------------------------------------------------------------

class INNER_ENC:

    # Contructor
    def __init__(self,generators,rate,mem):
        self.n = len(generators)                    # number
            of codewords
        self.k = int(rate*self.n)                   # number
            of input bits
        self.generators = generators                #
           generator polynomials
        self.rate = rate                            # code
           rate = k/n
        self.mem = mem                              # number
            of memory elements of the shift register

        self.no_states = pow(2,self.mem-self.k)     # number
            of states in trellis
        self.state_len = int(log(self.no_states,2)) # number
            of bits representing one state
        self.no_inputs = pow(2,self.k)              # number
            of inputs to encoder

        self.gen_list = []                          # list
           containing all generators
        for g in self.generators:
            g_padded = g.ljust(self.mem,'0')
            self.gen_list.append(g_padded)          # zero
               pad so that all generators have same length



    # Create transition tables
    # feed forward table (alpha): e.g. {0: [{'to': 0, 'in': 0, '
       out': '00'}, {'to': 2, 'in': 1, 'out': '11'}]}
    # The number before ':' indicates from which state we are
       going.
```

```
    # feed backward table (beta): e.g. {0: [{'out': '00', 'from
        ': 0, 'in': '0'}, {'out': '11', 'from': 1, 'in': '0'}]}
    # The number before ':' indicates to which state we are
        going.
    # code_table_ones indicates which branches give a '1' as
        first output bit.
    # code_table_zeros indicates which branches give a '0' as
        first output bit.
    # return: feed forward table, feed backward table,
        code_table_ones, code_table_zeros
    def create_trans_table(self):
        ff_table = dict()
        fb_table = dict()
        code_table_ones = dict()
        code_table_zeros = dict()

        inputs = bin_range(self.no_inputs,self.k)
        for state in range(self.no_states):
            ff_table[state] = list()
            fb_table[state] = list()

        for n in range(self.n):
            code_table_ones[n] = list()
            code_table_zeros[n] = list()

        for old_state in bin_range(self.no_states,self.state_len
            ):
            for in_bits in inputs:
                output, new_state = self.encoder(in_bits,
                    old_state)
                old_idx = bin2dec(old_state)
                new_idx = bin2dec(new_state)
                ff_table[old_idx].append({'to':new_idx, 'in':
                    in_bits, 'out':output, 'out_dec':modulate(
                    output)})
                fb_table[new_idx].append({'from':old_idx, 'in':
                    in_bits, 'out':output, 'out_dec':modulate(
                    output)})
                for i,o in zip(range(len(output)),output):
                    if(o == '1'):
                        code_table_ones[i].append(int(old_state
                            ,2)*self.no_inputs+int(in_bits,2))
                    else:
                        code_table_zeros[i].append(int(old_state
                            ,2)*self.no_inputs+int(in_bits,2))
```

```
        return ff_table, fb_table, code_table_ones,
            code_table_zeros



    # Recursiely encodes a stream of info bits and returns the
        output and new new state of the encoder
    # inparam1: a string of information bits
    # inparam2: the initial state of encoder (a string)
    # return: a string of coded bits, new state of encoder (
        string)
    def encode(self, info_bits, init_state):
        old_state = init_state
        outputs=''
        for i in range(0,len(info_bits), self.k):
            output, old_state = self.encoder(info_bits[i:i+self.
                k], old_state)
            outputs = outputs+output
        return outputs, old_state



    # Recursively encodes the input bits for one transition in
        trellis and returns the output and the new state
    # inparam1: a string of information bits
    # inparam2: the state of encoder (a string)
    # return: output of encoder (string), new state of encoder (
        string)
    def encoder(self,in_bits,state):

        state_inter=in_bits+state
        nom=reduce(xor,map(int,list(bin(int(self.gen_list[0],2)&
            int(state_inter,2)))[2:]))
        state_r= bin(nom)[2:]+state
        coded_bits = str(in_bits[0])
        for gen in self.gen_list[1:]:
            coded_bits = coded_bits + str(reduce(xor,map(int,
                list(bin(int(gen,2)&int(state_r,2)))[2:])))
        state = state_r[:-self.k:]
        return coded_bits, state
```

B.2.6 BCJR_outer.py

```
from numpy import dot
```

```
from numpy import transpose
from numpy import zeros
from numpy import ones
from math import log
from math import exp
from numpy import append
from help_funcs import *
from com_funcs import *

#
```

```
class BCJR_outer:

    # Contructor
    def __init__(self,generators,rate,data_len,mem):
        self.n = len(generators)          # number of codewords
        self.k = int(rate*self.n)         # number of input bits
        self.generators = generators      # generator polynomials
        self.rate = rate                  # code rate = k/n
        self.mem = mem                    # number of memory
            elements of the shift register

        self.no_states = pow(2,self.mem-self.k)      # number of
            states in trellis
        self.t_total = data_len+1                    # number of
            'time states' in trellis
        self.state_len = int(log(self.no_states,2)) # number of
            bits representing one state
        self.no_branches = pow(2,self.k)             # number of
            branches in trellis

        self.Pr_FF = zeros((self.no_states,self.t_total), float)
            # matrix containing probabilites for each state
        self.Pr_FF[0,0] = 1
        self.Pr_FB = zeros((self.no_states,self.t_total), float)
            # matrix containing probabilites for each state
        self.Pr_FB[0,self.t_total-1] = 1



    # Decode received data
    # inparam1: feed forward table (alpha) e.g. {0: [{'to': 0, '
        in': '0', 'out': '00'}, {'to': 2, 'in': '1', 'out':
        '11'}]}
```

```
    # inparam2: vector with a priori fed from inner decoder
    # return: matrix with calculated costs, feed forward
        probability matrix
    def ff_decode(self,ff_table, prob_list):
        cost_mtrx = zeros((self.no_states*self.no_branches,self.
            t_total-1), float)
        for t in range(self.t_total-1):
            prob_output = prob_list[t*self.n:t*self.n+self.n]
            for s in range(self.no_states):
                for b in range(self.no_branches):
                    to_state = ff_table[s][b].get('to')
                    output = ff_table[s][b].get('out')
                    cost = self.branch_cost_outer(output,
                        prob_output)
                    cost_mtrx[s*self.no_branches+b][t] = cost
                    self.Pr_FF[to_state, t+1] += self.Pr_FF[s,t
                        ]*cost
            self.Pr_FF[:,t+1] = norm(self.Pr_FF[:,t+1])
            cost_mtrx[:,t] = norm(cost_mtrx[:,t])
        return cost_mtrx, self.Pr_FF



    # Decode received data
    # inparam1: list containing received data
    # inparam2: feed backward table (beta) e.g. {0: [{'out':
        '00', 'from': 0, 'in': '0'}, {'out': '11', 'from': 1, 'in
        ': '0'}]}
    # inparam3: matrix with costs of all branches in trellis
        based on the received data and a priori
    # return: feed backward probability matrix
    def fb_decode(self,rec_bits,fb_table,cost_mtrx):
        for t in range(self.t_total)[::-1][:self.t_total-1]:
            for s in range(self.no_states):
                for b in range(self.no_branches):
                    from_state = fb_table[s][b].get('from')
                    inputs = bin2dec(fb_table[s][b].get('in'))
                    output = modulate(fb_table[s][b].get('out'))
                    rec_output = rec_bits[t*self.n-self.n:t*self
                        .n]
                    self.Pr_FB[from_state, t-1] += self.Pr_FB[s,
                        t]*cost_mtrx[from_state*self.no_branches+
                        inputs,t-1]
            self.Pr_FB[:,t-1] = norm(self.Pr_FB[:,t-1])
        return self.Pr_FB
```

```
# Calculate sigma, formed extrinsically by discarding a
    priori information
# inparam1: matrix with costs of all branches in trellis
    based on the received data only
# inparam2: feed backward probability matrix
# inparam3: feed forward probability matrix
# inparam4: feed forward table (alpha)
# return: sigma matrix
def calc_sigma(self, cost_mtrx, Pr_fb, Pr_ff, ff_table):
    sig_mtrx=zeros(shape(cost_mtrx))
    for col in range(shape(cost_mtrx)[1]):
        for s in range(self.no_states):
            for b in range(self.no_branches):
                to_state = ff_table[s][b].get('to')
                row = s*self.no_branches+b
                sig_mtrx[row, col] = Pr_ff[s,col]*Pr_fb[
                    to_state, col+1]*cost_mtrx[row,col]
        sig_mtrx[:,col]=norm(sig_mtrx[:,col])
    return sig_mtrx


# Make hard decisions based on sigma matrix
# inparam1: sigma matrix
# inparam2: block size
# return: list with hard bits
def hard_decision(self,sigma_mtrx, BLOCK_SIZE):
    for t in range(shape(sigma_mtrx)[1]):
        max = 0
        col = sigma_mtrx[:,t]
    prob_zeros=zeros(shape(sigma_mtrx)[1])
    prob_ones=ones(shape(sigma_mtrx)[1])
    for t in range(shape(sigma_mtrx)[1]):
        prob_zeros[t]=sum(sigma_mtrx[:,t][0::self.
            no_branches])
        prob_ones[t]=sum(sigma_mtrx[:,t][1::self.no_branches
            ])
    prob_zeros=prob_zeros
    prob_ones=prob_ones
    decision_vec=prob_ones[0:BLOCK_SIZE]
    hard_bits=1*array(decision_vec > 0.5)
    return hard_bits
```

```
# Calculate cost for a branch transition
# inparam1: theoretical output from inner decoder
# inparam2: probability of an output from inner decoder
# return: cost
def branch_cost_outer(self,outputs , prob_outer):
    cost = 1
    for idx,output in zip(range(len(outputs)), outputs):
        if (output=='1'):
            cost*= prob_outer[idx]
        else:
            cost *= 1 - prob_outer[idx]
    return cost



# Calculate the output of outer decoder
# inparam1: sigma matrix
# inparam2: code_table_ones indicates which branches give a
    '1' as first output bit
# inparam3: code_table_zeros indicates which branches give a
     '0' as first output bit
# inparam4: vector with a priori fed back from outer decoder
# return: output of outer decoder
def calc_app(self, sigma_mtrx, code_table_ones,
    code_table_zeros, prob_list):
    app_list = zeros((self.t_total-1)*self.n, float)
    app_list_ones=zeros((self.t_total-1)*self.n, float)
    app_list_zeros=zeros((self.t_total-1)*self.n, float)
    for t in range(self.t_total-1):
        for c in range(self.n):
            app_list_ones[t*self.n+c]=sum(sigma_mtrx[
                code_table_ones[c],t])
            app_list_zeros[t*self.n+c]=sum(sigma_mtrx[
                code_table_zeros[c],t])
    for i in range(len(app_list_ones)):
        if prob_list[i] != 0 and prob_list[i] != 1:
            app_list_ones[i] = app_list_ones[i] / prob_list[
                i]
            app_list_zeros[i] = app_list_zeros[i] / (1-
                prob_list[i])
    s = app_list_ones + app_list_zeros
    for i in range(len(s)):
        if s[i] != 0:
```

```
            app_list[i] = app_list_ones[i] / (app_list_ones[
                i] + app_list_zeros[i])
        return app_list
```

B.2.7 BCJR_inner.py

```
from numpy import dot
from numpy import transpose
from numpy import zeros
from numpy import ones
from math import log
from math import exp
from numpy import append
from help_funcs import *
from com_funcs import *

#
```

```
class BCJR_inner:

    # Contructor
    def __init__(self,generators,rate,data_len,mem):
        self.n = len(generators)          # number of codewords
        self.k = int(rate*self.n)         # number of input bits
        self.generators = generators      # generator polynomials
        self.rate = rate                  # code rate = k/n
        self.mem = mem                    # number of memory
            elements of the shift register

        self.no_states = pow(2,self.mem-self.k)      # number of
            states in trellis
        self.t_total = data_len+1                    # number of
            'time states' in trellis
        self.state_len = int(log(self.no_states,2)) # number of
            bits representing one state
        self.no_branches = pow(2,self.k)             # number of
            branches in trellis

        self.Pr_FF = zeros((self.no_states,self.t_total), float)
            # matrix containing probabilites for each state
        self.Pr_FF[0,0] = 1
        self.Pr_FB = zeros((self.no_states,self.t_total), float)
            # matrix containing probabilites for each state
        self.Pr_FB[0,self.t_total-1] = 1
```

```
# Calculates the costs of all branches in trellis based on
    the received data only (not dependent of a priori).
# inparam1: string of received data
# inparam2: feed forward table (alpha)
# inparam3: noise variance
# return: matrix with calculated costs
def calc_cost_mtrx_ex(self,rec_bits,ff_table,var):
    cost_mtrx = zeros((self.no_states*self.no_branches,self.
        t_total-1), float)
    for t in range(self.t_total-1):
        for s in range(self.no_states):
            for b in range(self.no_branches):
                output = ff_table[s][b].get('out_dec')
                rec_output = rec_bits[t*self.n:t*self.n+self
                    .n]
                cost = self.branch_cost(output, rec_output,
                    var)
                cost_mtrx[s*self.no_branches+b][t] = cost
    return cost_mtrx



# Decode received data
# inparam1: list containing received data
# inparam2: feed forward table (alpha) e.g. {0: [{'to': 0, '
    in': '0', 'out': '00'}, {'to': 2, 'in': '1', 'out':
    '11'}]}
# inparam3: noise variance
# inparam4: vector with a priori fed back from outer decoder
# inparam5: matrix with costs of all branches in trellis
    based on the received data only
# return: matrix with calculated costs, feed forward
    probability matrix
def ff_decode(self,rec_bits,ff_table,var, app_list,
    cost_mtrx_ex):
    cost_mtrx = zeros((self.no_states*self.no_branches,self.
        t_total-1), float)
    for t in range(self.t_total-1):
        for s in range(self.no_states):
            for b in range(self.no_branches):
                to_state = ff_table[s][b].get('to')
                output = ff_table[s][b].get('out_dec')
```

```
                rec_output = rec_bits[t*self.n:t*self.n+self
                    .n]
                input_bit = ff_table[s][b].get('in')
                if (input_bit=='1'):
                    app=app_list[t]
                else:
                    app= 1 - app_list[t]
                cost=app*cost_mtrx_ex[s*self.no_branches+b,t
                    ]+1e-175
                cost_mtrx[s*self.no_branches+b][t] = cost
                self.Pr_FF[to_state, t+1] += self.Pr_FF[s,t
                    ]*cost
        self.Pr_FF[:,t+1] = norm(self.Pr_FF[:,t+1])
    return cost_mtrx, self.Pr_FF



# Decode received data
# inparam1: list containing received data
# inparam2: feed backward table (beta) e.g. {0: [{'out':
    '00', 'from': 0, 'in': '0'}, {'out': '11', 'from': 1, 'in
    ': '0'}]}
# inparam3: matrix with costs of all branches in trellis
    based on the received data and a priori
# return: feed backward probability matrix
def fb_decode(self,rec_bits,fb_table,cost_mtrx):
    for t in range(self.t_total)[::-1][:self.t_total-1]:
        for s in range(self.no_states):
            for b in range(self.no_branches):
                from_state = fb_table[s][b].get('from')
                inputs = bin2dec(fb_table[s][b].get('in'))
                rec_output = rec_bits[t*self.n-self.n:t*self
                    .n]
                self.Pr_FB[from_state, t-1] += self.Pr_FB[s,
                    t]*cost_mtrx[from_state*self.no_branches+
                    inputs,t-1]
        self.Pr_FB[:,t-1] = norm(self.Pr_FB[:,t-1])
    return self.Pr_FB



# Calculate sigma, formed extrinsically by discarding a
    priori information
```

```
# inparam1: matrix with costs of all branches in trellis
    based on the received data only
# inparam2: feed backward probability matrix
# inparam3: feed forward probability matrix
# inparam4: feed forward table (alpha)
# return: sigma matrix
def calc_sigma(self, cost_mtrx, Pr_fb, Pr_ff, ff_table):
    sig_mtrx=zeros(shape(cost_mtrx))
    for col in range(shape(cost_mtrx)[1]):
        for s in range(self.no_states):
            for b in range(self.no_branches):
                to_state = ff_table[s][b].get('to')
                row = s*self.no_branches+b
                sig_mtrx[row, col] = Pr_ff[s,col]*Pr_fb[
                    to_state, col+1]*cost_mtrx[row,col]
    return sig_mtrx


# Make hard decisions based on sigma matrix
# inparam: sigma matrix
# return: list with hard bits
def hard_decision(self,sigma_mtrx):
    hard_bits = []
    for t in range(shape(sigma_mtrx)[1]):
        max = 0
        col = sigma_mtrx[:,t]
        for i in range(self.no_branches):
            Pr_sum = sum(col[i::self.no_branches])
            if Pr_sum > max:
                max = Pr_sum
                input_dec = i
        hard_bits.append(int(dec2bin(input_dec,self.k)))
    return hard_bits



# Calculate the output of inner decoder
# inparam1: sigma matrix
# inparam2: vector with a priori fed back from outer decoder
# return: output of inner decoder
def calc_prob(self,sigma_mtrx, app_list):
    app_list=app_list[:-(self.mem-1)]
    prob_list = zeros(len(app_list), float)
    prob_list_ones=zeros(len(app_list), float)
    prob_list_zeros=zeros(len(app_list), float)
```

```
        for t in range(len(app_list)):
            col = sigma_mtrx[:,t]
            prob_list_zeros[t] = sum(col[0::self.no_branches])
            prob_list_ones[t] = sum(col[1::self.no_branches])
        s = prob_list_zeros + prob_list_ones
        for i in range(len(s)):
            if s[i] != 0:
                prob_list[i] = prob_list_ones[i] / (
                    prob_list_zeros[i] + prob_list_ones[i])
        return prob_list



    # Calculate cost for a branch transition
    # inparam1: theoretical data
    # inparam2: received data
    # return: cost
    def branch_cost(self,data , rec_data, var):
        cost = 1
        for idx in range(len(data)):
            cost *= exp(-(pow(data[idx]-rec_data[idx],2))
                *(1.0/(2*var)))
        cost = cost + 1e-175
        return cost
```