SimCommSys: taking the errors out of error-correcting code simulations

Johann A. Briffa, Stephan Wesemeyer

Department of Computing, University of Surrey, Guildford GU2 7XH, England
E-mail: J.Briffa@surrey.ac.uk

Published in *The Journal of Engineering*; Received on 20th February 2014; Accepted on 12th May 2014

Abstract: In this study, we present SimCommSys, a simulator of communication systems that we are releasing under an open source license. The core of the project is a set of C++ libraries defining communication system components and a distributed Monte Carlo simulator. Of principal interest is the error-control coding component, where various kinds of binary and non-binary codes are implemented, including turbo, LDPC, repeat-accumulate and Reed–Solomon. The project also contains a number of ready-to-build binaries implementing various stages of the communication system (such as the encoder and decoder), a complete simulator and a system benchmark. Finally, SimCommSys also provides a number of shell and python scripts to encapsulate routine use cases. As long as the required components are already available in SimCommSys, the user may simulate complete communication systems of their own design without any additional programming. The strict separation of development (needed only to implement new components) and use (to simulate specific constructions) encourages reproducibility of experimental work and reduces the likelihood of error. Following an overview of the framework, we provide some examples of how to use the framework, including the implementation of a simple codec, the specification of communication systems and their simulation.

1 Introduction

Error-correcting codes are everywhere, from CDs (using Reed–Solomon codes), the now interstellar space communication between NASA and the Voyager 1 probe (using a convolutional code concatenated with a Golay code), to current terrestrial HDTV broadcasts (DVB-T2, using concatenated low-density parity-check (LDPC) and BCH codes). Shannon's seminal paper [1] showed that almost error-free communication is possible provided the rate of the information transmitted is below the capacity of the channel. Unfortunately, the proof is based on a probabilistic argument and does not provide a way of constructing codes that achieve the channel capacity while still being practical to encode and decode. Since then, the error-correction code community has developed a plethora of different codes and associated encoders and decoders. Modern codes, such as turbo [2], LDPC [3] and polar codes [4], have come very close to the channel capacity while still having encoders and decoders 'simple' enough to be useful.

As new codes continue to be developed whose performance need to be evaluated, most researchers in this field will at some point have written a simple test and performance measurement harness. To do this, it is likely they will have needed to implement or find at least: (a) a fast finite field representation for their code alphabet, (b) vector and matrix representations capable of handling finite fields for their encoder and decoder, (c) a channel implementation to simulate the transmissions of codewords, (d) a framework that glues together the encode, transmit, decode steps for a large number of codewords and computes the symbol error rate (SER) and frame error rate (FER) and (e) for more complex systems, a way of using this framework on a cluster to speed up the processing time. Although being able to code these things is arguably a good learning experience for any new researcher, most of these implementation tasks are peripheral to the main research problem of designing codes.

These elements are needed to test the code, but should be independent of the code and remain unchanged across codes. In fact, researchers having to implement these components separately is a likely source of additional errors which might hide problems in the performance of the designed code. For example, a channel which introduces less noise than required for a specific signal-to-noise ratio (SNR) might boost the claimed performance of a code. Consequently, having a framework of well-tested components that provide all the required features allows the researcher to concentrate on code design and testing it. It also provides other researchers with a means to reproduce the results more easily.

Current choices for researchers are limited, and each option has its caveats. Perhaps the most popular solution, MATLAB [5] and its Communications Toolbox includes implementations of a very wide range of current schemes. However, it comes at a considerable financial cost, particularly for parallel computation, limiting its availability to researchers. Furthermore, its usual workflow requires the user to write scripts to describe a given system, making it easy to introduce logical errors. The interpreted nature of the language also makes the implementation of novel code constructions inefficient, unless one makes use of the MATLAB to C/C++ interface (which requires considerable technical skill). An alternative open-source solution exists in the form of IT++ [6, 7], a library of mathematical, signal processing and communication system components. Although this library includes implementations of most current schemes, it is designed to be used by programmers, and the implementation of a system requires the user to write C++ code. Notably, the library does not include a simulation framework, so that the user is responsible to collect results and decide when a simulation has converged.

Our simulator of communication systems (SimCommSys) C++ framework helps address all these issues. It runs on both Windows and Linux and can also use NVIDIA GPUs using CUDA [8]. It has been developed over more than 15 years by the first author and was used successfully in a number of papers [9–13]. The source code has been released under the GNU General Public Licence version 3 (or later) and can be found, together with its documentation, at: https://github.com/jbresearch/simcommsys.

Researchers are provided with a host of existing codecs, channels, modulators and different performance measures to gather results quickly. Furthermore, the implementation is highly modular and every component is designed around simple-to-use interfaces which make it straightforward to extend the framework with new codecs as well as other components like channels, modulators etc. as required. Particular attention is also paid to the correctness of implementation and verifiability, for increased confidence in the results obtained. Internal checks are performed at multiple levels, and are available to the user through the debug build.

The rest of this paper is organised as follows. In Section 2, we describe the structure of the framework in more detail, highlighting the main components of interest. This is then followed by some examples of how to use SimCommSys in Section 3 and how to extend the framework in Section 4. We conclude with an invitation to other researchers to use and contribute to the project.

2 SimCommSys: the framework

2.1 Communication system model

Fig. 1 shows a simplified version of the communication system model that is supported by our framework. The codec block represents the code that is to be tested, and consists of an encoder and its corresponding decoder. To create a new code, the user simply needs to inherit from the abstract class `codec` or, if the code can provide soft output, `codec_softout`. In turn this inherits from the `codec` class; it is provided for convenience, and also allows iterative decoding between the codec and modems that support this.

The remaining components are straightforward to explain. After encoding the information with the encoder, it is the responsibility of the mapper component to translate, if required, the output symbols of the encoder to the symbols that can be modulated by the chosen modulation scheme. For example, this translation is necessary when the code alphabet is over $\mathbb{F}_q$ with $q = 2^k$ for some $k > 1$, and the channel modulation scheme is binary. The mapper can also be used to interleave the codec output and to puncture it to increase the code rate. Obviously, at the receiving end any interleaving or puncturing needs to be undone and the received symbols need to be mapped back to the alphabet that the codec understands.

The modem is responsible for translating the abstract symbols to their equivalent channel representation. A number of different modems are available, including commonly used ones for channels with a signal-space representation. These include M-ary phase-shift keying (PSK) with variable M [including $M = 2$ for binary PSK (BPSK) etc.] and quadrature amplitude modulation (QAM). Null modems are also available for abstract channels.

Finally, the channel represents the medium over which the modulated signal is transmitted and exposed to noise or corruption. Again a number of channels are available, including the commonly used additive white Gaussian noise (AWGN) channel. Abstract channels are also available, including the q-ary erasure channel, q-ary symmetric channel and also channels with insertion, deletion and substitution errors. In the case of such synchronisation error channels, the length of the transmitted sequence (i.e. entering the channel) might not be the same as that of the received sequence.

A summary of the principal communication system components available in the SimCommSys code base is given in Table 1. This is followed by a list of codec sub-components in Table 2.

2.2 Monte Carlo simulator model

In addition to the ease of setting up different combinations of codecs, mappers, modems and channels, the framework also provides a feature-rich simulator. This can be configured to gather numerous performance measurements of the described system, including SER, FER and the timings of various components. An overview of the simulator model implemented by our framework is shown in Fig. 2, where the communication system is considered as a block box that performs the complete cycle of events of Fig. 1. In SimCommSys, the simulator object defines the input sequences to be cycled through the communication system object; a results collector component compares the input and output of the communication system and computes the required statistics. This modular architecture allows the user to simulate any given communication system under different input conditions and to collect a range of possible results. The simulator object implements a substantial part of the experiment interface: that concerned with computing a single sample. The accumulation of aggregate statistics from multiple samples is typically performed by the binomial experiment object; this is suitable for error-rate experiments, which can be seen as Bernoulli trials. Finally, the Monte Carlo object implements the necessary loops to obtain enough samples until convergence is achieved. A summary of classes providing simulation types and related facilities (such as results collectors) can be found in Table 3.

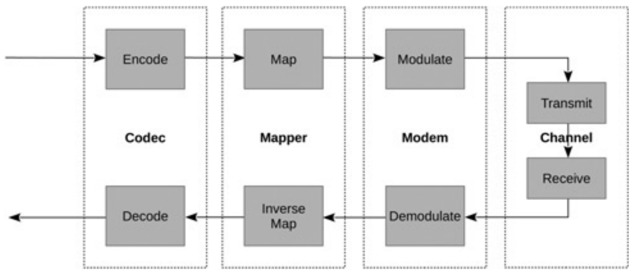

Fig. 1 *SimCommSys communication system model*

2.3 Local or distributed simulation

The simulator (through the Monte Carlo object) is designed to work in a distributed setup using a client–server model, but can also work as a single local process. The latter mode is useful, for example, for quick simulations or when gathering timings for benchmarking and optimisation.

In client–server mode, a server process controls the simulation and is responsible for gathering all the results. When starting a distributed simulation, the user specifies the port the server listens on. The user also passes a text-based configuration file that specifies the details of the components needed in the simulation (see Section 3 for examples). The user can then start any number of client processes either on the same machine or across any number of networked computers. The clients communicate with the server process using TCP/IP socket connections to the specified port. The server process keeps track of the number of clients and can accept new ones at any time. Additionally, if a client process dies or is otherwise disconnected, the server removes it from the list of connections and is otherwise unaffected. Together, these allow the user to dynamically scale the resources allocated to a particular simulation. Once the server process has finished the simulation, it will cleanly terminate all the clients before stopping. Should the server process stop inadvertently, for example if the user switched off the machine running the server process, all clients terminate automatically when their connection with the server process is lost. This behaviour prevents the client processes from using up resources on the client machines unneccessarily. Fig. 3 illustrates this principle. Note that in the distributed case, timings are only useful provided all the client machines are homogeneous both in hardware and software.

When starting a simulation, the user also specifies the range of channel conditions to be simulated and the convergence requirements for the simulation. The channel conditions are usually specified in terms of the SNR or error probabilities, with the range specified by the initial and final values and a step factor. Depending on the requirements of the channel used, the user can specify whether the step factor is applied additively (e.g. in the case of the AWGN channel) or multiplicatively (e.g. for abstract channels like insertion–deletion or erasure channels). Convergence requirements may be specified as the number of error events to be accumulated (conventionally 100) or by specifying the required confidence interval (as an error margin together with a confidence value). When a confidence interval is set, the simulation is considered to have converged when the true value is within the error margin of the current estimate, at the stated confidence level.

Table 1 Summary of the principal communication system components available in the code base

Base	Class	Description
codec	reedsolomon	Reed-Solomon code over $\mathbb{F}_q$ of length $n \in \{q, q-1\}$ and dimension $1 < k < n-1$ with Berlekamp decoder
	ldpc	LDPC code over $\mathbb{F}_q$ of length n and dimension m
	mapcc	convolutional code with BCJR decoder [14]
	repacc	repeat-accumulate code with BCJR decoder
	sysrepacc	systematic repeat-accumulate code with BCJR decoder
	turbo	parallel concatenated convolutional code with variable interleavers and BCJR decoder
	memoryless	simple mapping, with or without repetition
	uncoded	uncoded transmission (output is copy of input)
	codec_multiblock	meta-codec that concatenates a number of blocks of the underlying codec (for interleaving across blocks)
	codec_concatenated	meta-codec that concatenates a sequence of codecs (with intermediate mappers)
blockmodem[a]	dminner	sparse inner codes with distributed marker sequence and Davey–MacKay decoder [15]
	marker	marker codes with bit-level MAP decoder [16]
	tvb	time-varying block codes with GPU-enabled symbol-level MAP decoder [9, 17]
mapper	map_straight	each modulation symbol encodes exactly one encoder symbol
	map_interleaved	random interleaving of symbols within the block
	map_permuted	random permutation of symbols at each index
	map_aggregating	each modulation symbol encodes more than one encoder symbol (e.g. binary codecs on q-ary modems)
	map_dividing	each encoder symbol is represented by more than one modulation symbol (e.g. q-ary codecs on binary modems)
	map_stipple	punctured mapper for turbo codes, with all information symbols transmitted and parity symbols taken from successive sets; equivalent to odd/even puncturing for two-set turbo codes
	map_concatenated	meta-mapper that concatenates a sequence of mappers
blockmodem	direct_blockmodem	abstract q-ary channel modulation
	M-ary phase shift keying	MPSK modulation with Grey code mapping of adjacent symbols in constellation
	qam	quadrature amplitude modulation for square constellations with Grey code mapping of adjacent symbols
channel	awgn	additive white Gaussian noise channel (for signal-space modulations)
	laplacian	additive Laplacian noise channel (for signal-space modulations)
	qec	q-ary erasure channel (for abstract modulations)
	qsc	q-ary symmetric substitution channel (for abstract modulations)
	qids	q-ary insertion, deletion and substitution channel (for abstract modulations)
	bpmr	bit-patterned media recording channel of [18]

[a]The encoders/decoders in this section are implemented using the blockmodem interface because of the required access to the channel, which is only available through the blockmodem interface.

Once the server process is up, all that any client process requires is the hostname or IP address of the server process and the port number it is listing on. On connection, the server sends the system configuration to the client together with the channel parameter to simulate. Each client seeds its random generator with a true random value from the OS. This ensures that all clients simulate different random input sequences, noise patterns and (for applicable systems) different random system components

Table 2 Summary of available codec sub-components

Base	Class	Description
fsm[a]	dvbcrsc	circular recursive systematic convolutional code from the DVB standard [19]
	gnrcc	non-recursive convolutional code over $\mathbb{F}_q$ with encoder polynomials expressed in controller-canonical form
	grscc	recursive convolutional code over $\mathbb{F}_q$ with encoder polynomials expressed in controller-canonical form
	nrcc	binary non-recursive convolutional code with encoder polynomials expressed in controller-canonical form
	rscc	binary recursive convolutional code with encoder polynomials expressed in controller-canonical form
	zsm	a zero-state machine or in other words a repeater
	cached_fsm	meta-fsm that pre-computes and caches the input/output table of its component fsm
interleaver[b]	flat	null interleaver (usually used for the first parity sequence)
	berrou	the original turbo code interleaver of [2]
	helical	helical interleaver [20]
	rectangular	simple rectangular interleaver
	shift_lut	circular-shifting interleaver
	named_lut	a general interleaver specified as a look-up table (generated externally)
	uniform_lut	random interleaver (with uniform distribution)
	rand_lut	random interleaver with simile property [20]
	onetimepad	interleaver that performs symbol-by-symbol modular addition between input and a random sequence
	padded	meta-interleaver that concatenates any interleaver with a onetimepad

[a]This class implements a finite state machine interface, which is used to specify the encoder in a convolutional code. This is needed for obvious reasons in the mapcc and turbo classes; it is also used to specify the accumulator in repacc and the mapping in memoryless.
[b]This class implemented the interface for the interleaver used in parallel concatenated convolutional codes, creating diversity between the parity sequences.

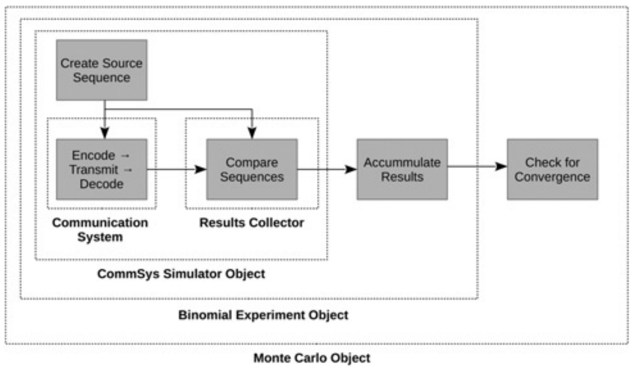

Fig. 2 *SimCommSys simulator model*

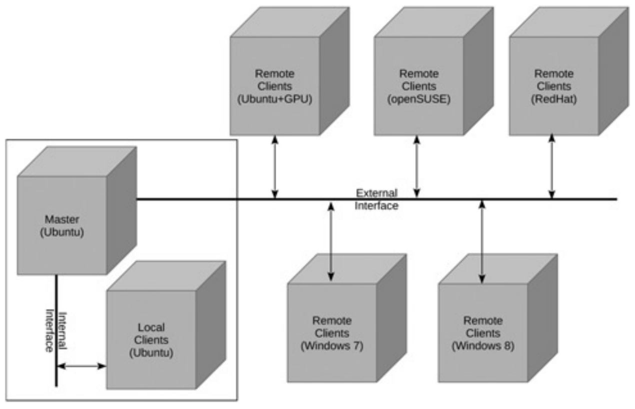
Remote Clients (Ubuntu+GPU)
Remote Clients (openSUSE)
Remote Clients (RedHat)
Master (Ubuntu)
External Interface
Internal Interface
Local Clients (Ubuntu)
Remote Clients (Windows 7)
Remote Clients (Windows 8)

Fig. 3 *SimCommSys client–server model*

(e.g. random interleavers). The client instantiates all the necessary components and runs the simulation sending results back to the server process regularly. The server process aggregates results from all clients, and stores these in a human-readable text file. Intermediate results (i.e. aggregate results that have not yet converged) are stored regularly on file, with the full state of the simulation. This allows the server to continue a previously terminated simulation, and mitigates the risk of server failure on long-running simulations. The code base provides a python module and a simple script which can be easily modified to present the results graphically using the matplotlib library [22].

In all of the above, the only programming required by the user is the implementation of any new components and a simple adaptation of the python script to visualise the results if required. All the other aspects of the simulation are taken care of by the framework. In the next section, we will demonstrate how to set up a system and run a simulation.

3 Using SimCommSys

3.1 *Running a quick simulation with a simple codec*

The most common use case is to set up and simulate a communication system under a range of channel conditions. This starts with the specification of a simulation based on a communication system, defining the mapper, modem and channel to be used as well as what results we want to gather. The file shown in Fig. 4 is an example of how to achieve this for a very simple setup: an uncoded BPSK transmission over AWGN. As can be seen from the config file, each component has its own heading followed by a number of parameters. A version number is usually included to allow old configuration files to be read provided the newer version of the serialisation code can provide default values for any missing or changed parameters. The file format allows the inclusion of comments, indicated by lines starting with a `#`, which are skipped when the file is read.

Table 3 Summary of classes providing simulation types and related facilities (such as results collectors)

Base	Class	Description
`experiment_binomial`[a]	`commsys_simulator`	generic simulator of communication systems, supporting random, all-zero or user-specified input sequences and a modular results collection interface; simulation parameter specifies channel conditions
	`commsys_stream_simulator`	variation on `commsys_simulator` that simulates stream transmission and reception, where the start and end of each frame are not assumed to be known a priori and are instead estimated by the receiver
	`commsys_threshold`	variation on `commsys_simulator` where the simulation parameter specifies the modem threshold setting; the channel conditions are fixed to a value specified in this object
`experiment_normal`[b]	`commsys_timer`	meta-experiment to determine timings of individual components of a given communication system
results collector[c]	`errors_hamming`	conventional SER and FER computed using the Hamming distance metric
	`errors_levenshtein`	as for `errors_hamming`, with additional SER computed using the Levenshtein metric [21]
	`fidelity_pos`	computation of the fidelity metric at frame and codeword boundary positions, for synchronisation error-correcting codes
	`hist_symerr`	computes histogram of symbol error count for each block simulated
	`prof_burst`	determines the error probabilities for the first symbol in a frame, a symbol following a correctly-decoded one, and a symbol following an incorrectly decoded one; used to determine the error burstiness profile
	`prof_pos`	computes symbol-error histogram as dependent on position within block
	`prof_sym`	computes symbol-error histogram as dependent on source symbol value

[a]Implements the interface for an experiment where a binomial proportion is estimated, approximating the error with a normal distribution.
[b]Implements the interface for an experiment where the samples take a normal distribution, and the mean of the distribution needs to be estimated.
[c]Although there is no base class, the interface is specified in `commsys_simulator`, where objects of this type are used to specify the results to be collected in a given simulation.

```
commsys_simulator<sigspace,errors_hamming>
# Version
2
# Input mode (0=zero, 1=random, 2=user[seq])
1
# Communication system
commsys<sigspace,vector>
# Version
1
# Single channel?
1
## Channel
awgn
## Modem
mpsk
# Alphabet size in symbols
2
## Mapper
map_straight<vector,double>
## Codec
uncoded<double>
# Version
1
# Alphabet size
2
# Block length
16320
```

Fig. 4 *Uncoded BPSK transmission over AWGN*

In this example, we are simulating a communication system with a signal-space channel representation and gathering error rates using the Hamming distance. Within the simulator we can specify whether an all-zero, random or user-specified information sequence should be used. The actual communication system starts with the `uncoded` class defining a binary code of length 16 320 bits. This is passed through a straight mapper (i.e. left unmodified) and modulated with a PSK modem of alphabet size 2 (i.e. a BPSK modem). The result is transmitted on an AWGN channel. Note that the communication system allows us to define separate channel objects for the transmit and receive functions. This is unused here, and would allow us to simulate the system with a mismatched receiver.

The user can test the system file by simply running a short simulation using the 'QuickSimulation' command as shown in Fig. 5. This will run the simulation at an SNR of 6.8 dB for ten seconds. The final output should look something like Fig. 6. This output also allows us to determine the speed at which a simulation of this code runs. For this simple codec, the simulator computed 135.8 frames of 16 320 bits each per second, equivalent to 2.22 Mbit/s (on an Intel Core i5-3570K central processing unit using a single core at 3.4 GHz).

3.2 Quick simulation of a more complex system

Consider next a more complex system with a Reed–Solomon code in concatenation with a convolutional code, as used in the NASA Voyager mission [23, 24]. The corresponding configuration file is shown in Fig. 7. This defines a serially concatenated construction, as follows. The outer code is a (255, 223) Reed–Solomon code defined over $\mathbb{F}_{256}$ (using the `reedsolomon` component). A random interleaver operates over four successive outer codewords, so that any burst errors are distributed across four codewords

```
quicksimulation.master.release -t 10 -r 6.8 -i
    Simulators/errors_hamming-random-awgn-bpsk-
    uncoded.txt >Results/sim.errors_hamming-random-
    awgn-bpsk-uncoded.txt
```

Fig. 5 *Running a short simulation*

```
System Used:
~~~~~~~~~~~~
Simulator for Communication System: Uncoded
    Representation (16320x2), Straight Mapper (
    Vector) [16320], BPSK blockmodem, AWGN channel,
    random input
Confidence Level: 99.9%
Convergence Mode: Margin of error within +/-0.1% of
    result
Date: 02 Dec 2013, 18:00:42
Simulating at system parameter = 6.8

Results:
~~~~~~~~
SER_0 0.00098744  [+/-2.22%]
FER_0 1 [+/-0%]

Build: master
Version: v1.0.0-12-g70a9b29-dirty
Statistics: 1365 samples in 10.00s.
Simulation Speed: 136.5 samples/sec
```

Fig. 6 *Simulation output for uncoded BPSK transmission over AWGN*

(achieved using the `map_interleaved` component). The inner code is a rate-1/2 convolutional code, specified by the feedforward polynomials $1+z^{-2}+z^{-3}+z^{-5}+z^{-6}$ and $1+z^{-1}+z^{-2}+z^{-3}+z^{-6}$. In the serialised file, these are represented by the strings 1011011 and 1111001. The Reed–Solomon encoder output is converted from $\mathbb{F}_{256}$ to $\mathbb{F}_2$ (binary) using the `map_dividing` component. The output of four Reed–Solomon codewords is equivalent to 8160 bits; this is terminated with six tail bits before encoding with the convolutional code. The `mapcc` component uses the BCJR algorithm [14], minimising SER. The output for this code at an SNR of 2.2 dB after ten seconds is shown in Fig. 8. Note that this corresponds to a decoding speed of 36.9 kbit/s which is about sixty times slower than the uncoded transmission.

3.3 Running a proper simulation

Having tested the simulator configuration files and verified that they are working, it is now a simple step to start a simulation. For the uncoded system, this can be done as shown in Fig. 9. This starts the server process, listening on port 9000. The simulation starts with the channel SNR at 1.5 dB and works its way up to 11 dB in steps of 0.1 dB. The point at which the simulation switches to the next SNR value is determined by the convergence requirements, which can be specified either in terms of the number of error events encountered or as a confidence interval. In this case, we specify an error margin of $\pm 5\%$ at a 95% confidence level.

In this example, the simulation stops completely when at least one of the measures (SER or FER in this case) has fallen below 10^{-5}. Note that this can and usually should occur well before all noise values have been explored. Alternatively, the user can set `--floor-max` which would require all measures to fall below this threshold before the simulation stops. As SER $\leq$ FER, 'floor-min' is usually used if the SER is more important to measure while 'floor-max' is used if the FER is the main focus.

Any number of clients can now be started using the command shown in Fig. 10, where the `server_address` could be `localhost` if the client is started on the same machine or alternatively the IP address or the DNS-resolvable name of the computer where the server process is running.

Repeating the same process for the concatenated system of Section 3.2 and for a system with the inner convolutional code alone, we now have the necessary results to plot a performance comparison, as shown in Fig. 11. The theoretical error rate for the uncoded AWGN channel is also shown, clearly coinciding with our simulation. For comparison, the figure also includes previously published results from [24]. As expected, our simulation of the

```
commsys_simulator<sigspace,errors_hamming>
# Version
2
# Input mode (0=zero, 1=random, 2=user[seq])
1
# Communication system
commsys<sigspace,vector>
# Version
1
# Single channel?
1
## Channel
awgn
## Modem
mpsk
# Alphabet size in symbols
2
## Mapper
map_straight<vector,double>
## Codec
codec_concatenated<double>
# Version
1
# Number of concatenated codecs
2
# Codec 1
codec_multiblock<double>
# Version
1
# Underlying codec
reedsolomon<gf256>
# Length of the code (n)
255
# Dimension of the code (k)
223
# Number of blocks to aggregate
4
# Codec 2
mapcc<double,double>
# Encoder
nrcc
#: Generator matrix (k x n bitfields)
1 2
1011011 1111001
# Message length (including tail, if any)
8166
# Terminated?
1
# Circular?
0
# Mapper 1
map_concatenated<vector,double>
# Version
1
# Number of concatenated mappers
2
# Mapper 1
map_interleaved<vector,double>
# Interface size
256
# Mapper 2
map_dividing<vector,double,double>
```

Fig. 7 *Serially concatenated Reed–Solomon and convolutional codes*

same convolutional code shows better performance; this is because we use a bit-optimal (maximum a posteriori (MAP)) decoder rather than the Viterbi decoder. On the other hand, our simulation of the concatenated system shows slightly worse performance since we use a random interleaver covering four Reed–Solomon codewords rather than an infinite interleaver.

A collection of example systems is included in the repository. This includes the systems shown in this section, together with the simulation results and plot script. For further details on constructing simulations for specific components, please consult the user documentation.

```
System Used:
~~~~~~~~~~~~
Simulator for Communication System: Concatenated
    codec (2 codecs, 1 mappers) [C1: Multi-block
    codec (4 blocks) [RS code [255, 223] ], C2:
    Terminated, Non-circular, MAP-decoded
    Convolutional Code (16332,8160) - NRC code (nu
    =6, rate 1/2, G=[1011011, 1111001]), M1:
    Concatenated mapper (2 mappers) [M1: Interleaved
     Mapper, Interface size=256, M2: Dividing Mapper
     (Vector) [1020<->8160]]], Straight Mapper (
    Vector) [16332], BPSK blockmodem, AWGN channel,
    random input
Confidence Level: 99.9%
Convergence Mode: Margin of error within +/-0.1% of
    result
Date: 02 Dec 2013, 18:00:52
Simulating at system parameter = 2.2

Results:
~~~~~~~~
SER_0 0.000969433 [+/-24.6%]
FER_0 0.0483092 [+/-102%]

Build: master
Version: v1.0.0-12-g70a9b29-dirty
Statistics: 207 samples in 10.02s.
Simulation Speed: 20.67 samples/sec
```

Fig. 8 *Simulation output for serially concatenated Reed–Solomon and convolutional codes*

```
simcommsys.master.release -i Simulators/
    errors_hamming-random-awgn-bpsk-uncoded.txt -o
    Results/errors_hamming-random-awgn-bpsk-uncoded.
    txt -e :9000  --start 1.5 --stop 11 --step 0.1
    --floor-min 1e-5 --confidence 0.95 --relative-
    error 0.05
```

Fig. 9 *Running the server process for a proper simulation*

```
simcommsys.master.release -e server_address:9000
```

Fig. 10 *Running the client process*

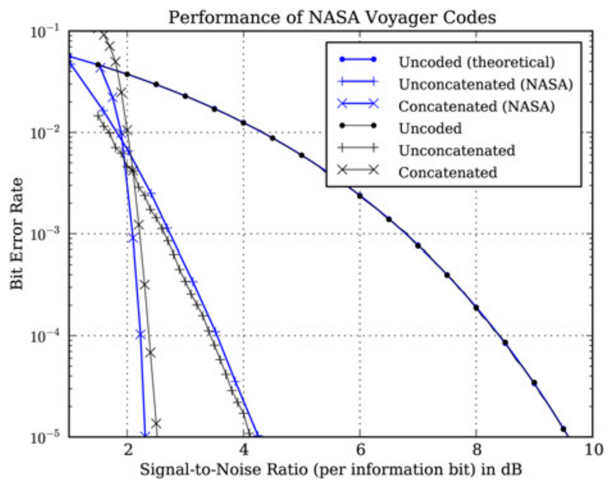

Fig. 11 *Performance comparison of two codes used in the NASA Voyager mission: previously published results from [24] and theoretical results for the uncoded channel are shown in blue*

4 Extending SimCommSys

To illustrate how the framework can be extended we will implement the simplest of codes, the 'uncoded' code, where the information sequence is transmitted as is and the received sequence is simply decoded by taking a hard decision. This will illustrate several features and concepts of the framework without getting bogged down in any implementation issues of an actual codec. To make this exercise slightly more interesting, our implementation extends the `codec_softout` abstract class rather than the plain `codec` abstract class. This class defines the interface for a soft-input, soft-output codec, and allows us to look at some of the features used by most modern codes.

The main method that needs to be implemented for the encoding is the `do_encode` method of Fig. 12. Note the use of templates in the code fragment. Throughout the framework, templates are used to provide support for a range of code alphabets and numerical precisions with very little extra programming effort. The actual code alphabet and numerical precision are defined when the templates are instantiated, and can be chosen at run-time as part of the serialisation process.

Decoding is implemented as a two-step process. First, the decoder in initialised with the probabilities of the received sequence. For a soft-input, soft-output codec, the methods that need to be implemented for this are the `do_init_decoder` methods of Fig. 13. Note that this method is overloaded, having two implementations with different parameters. Therefore two variants of this method need to be implemented: one where only the channel statistics for the received sequence are available and another where prior probabilities for the transmitted sequence are also given. The latter interface is required for systems involving iteration between the `modem` and `codec` components.

Next, the actual decoding takes place; this may be repeated a number of times in an iteratively decoded code, where each successive decoding makes use of information from previous decodings. For a soft-input, soft-output codec, the methods that need to be implemented for this are the `softdecode` methods of Fig. 14. This method is also overloaded: the first implementation computes the posterior probabilities of the decoded sequence only, whereas the second one also computes the posterior probabilities of the encoded sequence. It is up to the calling class to decide which of these methods to use as there may be a computational cost in computing both values when only one is needed. For example, in the case of turbo codes, only the probabilities of the information symbols are required as input to the next decoding stage, whereas the probabilities of the parity check symbols are not required. On the other hand, with LDPC codes the sum–product algorithm needs to compute the probability of all symbols as everything is used in the next iteration.

```
template <class dbl>
void uncoded<dbl>::softdecode(array1vd_t& ri)
   {
   // Set input-referred stats to stored values
   ri = R;
   }

template <class dbl>
void uncoded<dbl>::softdecode(array1vd_t& ri,
    array1vd_t& ro)
   {
   // Set input-referred stats to stored values
   ri = R;
   // Set output-referred stats to stored values
   ro = R;
   }
```

Fig. 14 *uncoded.cpp (decode)*

The last two required functions provide serialisation support to the codec. These allow the codec to be read in from file when setting up a simulation, and also allow the server to send a serialised version of the system to be simulated to its clients. To achieve this, each component of the system needs to implement the following two methods of the `serialisable` abstract class. In this example, the `serialisable` interface is inherited through the `codec` and thus the `codec_softout` class. The first method writes the details of the codec to an output stream (e.g. a file or a socket), whereas the second method reads the necessary class parameters (and components, where applicable) from an input stream. Both are shown in Fig. 15. Note that the `libbase:eatcomments` manipulator filters out any comments in the input stream. There are some more boiler-plate methods that need to be implemented, including methods to return a short description of the codec and methods to return the length, dimension and alphabet size of the code.

For such a templated class, the implementation file needs to contain explicit instantiations of the different combinations of template parameters of the codec. Fig. 16 shows how Boost preprocessor metaprogramming [25] is used to create the various instances of the templated codec which can then be serialised using the name of

```
template <class dbl>
void uncoded<dbl>::do_encode(const array1i_t& src, array1i_t& enc)
   {
   // Copy input to output
   enc = src;
   }
```

Fig. 12 *uncoded.cpp (encode)*

```
template <class dbl>
void uncoded<dbl>::do_init_decoder(const array1vd_t&
    ptable)
   {
   // R is a private variable of the uncoded class
   R = ptable;
   }

template <class dbl>
void uncoded<dbl>::do_init_decoder(const array1vd_t&
    ptable, const array1vd_t& app)
   {
   // Initialize results to received statistics
   do_init_decoder(ptable);
   // Multiply with prior statistics
   R *= app;
   }
```

Fig. 13 *uncoded.cpp (init_decoder)*

```
template <class dbl>
std::ostream& uncoded<dbl>::serialize(std::ostream&
    sout) const
   {
   sout << "# Version" << std::endl;
   sout << 1 << std::endl;
   sout << "# Alphabet size" << std::endl;
   sout << q << std::endl;
   sout << "# Block length" << std::endl;
   sout << N << std::endl;
   return sout;
   }

template <class dbl>
std::istream& uncoded<dbl>::serialize(std::istream&
    sin)
   {
   // get format version
   int version;
   sin >> libbase::eatcomments >> version;
   // read the alphabet size and block length
   sin >> libbase::eatcomments >> q >> libbase::
       verify;
   sin >> libbase::eatcomments >> N >> libbase::
       verify;
   return sin;
   }
```

Fig. 15 *uncoded.cpp (serialise)*

```
namespace libcomm {

// Explicit Realizations
#include <boost/preprocessor/seq/for_each.hpp>
#include <boost/preprocessor/stringize.hpp>

using libbase::serializer;
using libbase::mpreal;
using libbase::mpgnu;
using libbase::logreal;
using libbase::logrealfast;

#define REAL_TYPE_SEQ \
   (float)(double) \
   (mpreal)(mpgnu) \
   (logreal)(logrealfast)

/* Serialization string: uncoded<real>
 * where:
 *       real = float | double | mpreal | mpgnu |
     logreal | logrealfast
 */
#define INSTANTIATE(r, x, type) \
   template class uncoded<type>; \
   template <> \
   const serializer uncoded<type>::shelper( \
      "codec", \
      "uncoded<" BOOST_PP_STRINGIZE(type) ">", \
      uncoded<type>::create); \

BOOST_PP_SEQ_FOR_EACH(INSTANTIATE, x, REAL_TYPE_SEQ)

} // end namespace
```

Fig. 16 *uncoded.cpp (boost)*

the class. Finally, we must ensure that serialisation system 'sees' an object of this type on startup: this is done by adding an object of this type as a field or parent class of `serializer_libcomm`. For templated classes, it is enough to include an object with any of the explicitly instantiated template parameters.

5 Conclusions

In this paper, we have presented SimCommSys, a simulator of communication systems released under an open-source license. An overview was given of the core of the project, a set of C++ libraries defining a number of communication system components and a distributed Monte Carlo simulator. The more common use case, where a communication system is defined and then simulated using the framework, was demonstrated. Finally, a tutorial for extending the framework was given, using the implementation of an 'uncoded' codec as an example.

SimCommSys fills a current void, providing a reliable platform for simulating communication systems without any additional programming (as long as the required components are already available in SimCommSys). Development is only necessary when extending the framework by implementing new components. The strict separation of development and the framework's use to simulate specific constructions encourages reproducibility of experimental work and reduces the likelihood of error.

The project has been in development for many years, and has evolved to support changing requirements. For example, most recently the framework has been used to simulate codes for synchronisation error channels, which as far as we know is not supported by any other publicly available software. We expect the project to continue to evolve as the needs of its users change. The wider its user base, the more comprehensive the software will become. We encourage potential users to download and use the project in their own research. Support is available through the project forum, whereas feature requests and bug reports may be submitted through the project tracker. Developers who wish to contribute to the project are asked to contact the maintainer; community support is welcome.

6 References

[1] Shannon C.E.: 'A mathematical theory of communication', *Bell Syst. Tech. J.*, 1948, **23**, pp. 379–423, 623–656

[2] Berrou C., Glavieux A., Thitimajshima P.: 'Near Shannon limit error-correcting coding and decoding: turbo-codes'. Proc. IEEE Int. Conf. Communications, Geneva, Switzerland, May 1993, pp. 1064–1070

[3] MacKay D.J.C., Neal R.M.: 'Near Shannon limit performance of low density parity check codes', *Electron. Lett.*, 1997, **33**, (6), pp. 457–458

[4] Arıkan E.: 'Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels', *IEEE Trans. Inf. Theory*, 2009, **55**, (7), pp. 3051–3073

[5] Matlab: The language of technical computing' (MathWorks, Inc., 2012). [Online]. Available at http://www.mathworks.co.uk/products/datasheets/pdf/matlab.pdf

[6] de Lima C.H.M., Stancanelli E.M.G., Rodrigues E.B., da S.Maciel J.M., Cavalcanti F.R.P.: 'A software development framework based on C++ OOP language for link-level simulation tools'. Int. Telecommunications Symp., 2006, pp. 597–602

[7] Cristea B.: 'Turbo receivers with IT++'. Second Int. ICST Conf. Simulation Tools and Techniques. ACM, May 2010

[8] NVIDIA CUDA C Programming Guide, NVIDIA Corporation, October 2012, version 5.0

[9] Briffa J.A.: 'A GPU implementation of a MAP decoder for synchronization error correcting codes', *IEEE Commun. Lett.*, 2013, **17**, (5), pp. 996–999

[10] Buttigieg V., Briffa J.A.: 'Codebook and marker sequence design for synchronization-correcting codes'. Proc. IEEE Int. Symp. Information Theory, St. Petersburg, Russia, 31 July–5 August 2011

[11] Briffa J.A., Schaathun H.G.: 'Improvement of the Davey–MacKay construction'. Proc. IEEE Int. Symp. Information Theory and its Applications, Auckland, New Zealand, 7–10 December 2008, pp. 235–238

[12] Briffa J.A., Schaathun H.G.: 'Non-binary turbo codes and applications'. Proc. IEEE Int. Symp. Turbo Codes & Related Topics, Lausanne, Switzerland, 1–5 September 2008, pp. 294–298

[13] Briffa J.A., Buttigieg V.: 'Interleavers and termination in unpunctured symmetric turbo codes', *IEE Proc. Commun.*, 2002, **149**, (1), pp. 6–12

[14] Bahl L.R., Cocke J., Jelinek F., Raviv J.: 'Optimal decoding of linear codes for minimizing symbol error rate', *IEEE Trans. Inf. Theory*, 1974, **20**, (2), pp. 284–287

[15] Davey M.C., MacKay D.J.C.: 'Reliable communication over channels with insertions, deletions, and substitutions', *IEEE Trans. Inf. Theory*, 2001, **47**, (2), pp. 687–698

[16] Ratzer E.A.: 'Marker codes for channels with insertions and deletions', *Ann. Telecommun.*, 2005, **60**, pp. 29–44

[17] Briffa J.A., Schaathun H.G., Wesemeyer S.: 'An improved decoding algorithm for the Davey–MacKay construction'. Proc. IEEE Int. Conf. Communications, Cape Town, South Africa, 23–27 May 2010

[18] Iyengar A.R., Siegel P.H., Wolf J.K.: 'Write channel model for bit-patterned media recording', *IEEE Trans. Magn.*, 2011, **47**, (1), pp. 35–45

[19] Digital Video Broadcasting (DVB); Interaction channel for satellite distribution systems, ETSI, 5th September 2005, eN 301 790 v1.4.1

[20] Barbulescu S.A., Pietrobon S.S.: 'Terminating the trellis of turbo-codes in the same state', *Electron. Lett.*, 1995, **31**, (1), pp. 22–23

[21] Levenshtein V.I.: 'Binary codes capable of correcting deletions, insertions and reversals', *Sov. Phys. Dokl.*, 1966, **10**, (8), pp. 707–710

[22] Hunter J.D.: 'Matplotlib: a 2d graphics environment', *Comput. Sci. Eng.*, 2007, **9**, (3), pp. 90–95

[23] McEliece R.J., Swanson L.: 'Reed-Solomon codes and the exploration of the solar system'. Technical Report, Jet Propulsion Laboratory, California Institute of Technology, 20th August, 1993. [Online]. Available at http://www.hdl.handle.net/2014/34531

[24] Miller R.L., Deutsch L.J., Butman S.A.: 'On the error statistics of Viterbi decoding and the performance of concatenated codes', *NASA STI/Recon Tech. Rep.*, 1981, **81**, (9)

[25] Abrahams D., Gurtovoy A.: 'C++ template metaprogramming: concepts, tools, and techniques from boost and beyond' (Addison-Wesley Professional, 2004)