A Modular Parallel-Processing Link-Level Simulator on Python Programming Language

Rodrigo A. R. Fischer, João Paulo Leite, Bruno H. C. Faria

***Abstract*— Most link-level simulators implement a communication link that can be seen as a particular implementation of a more generic transmission-reception (TX-RX) chain. These simulators also share common simulation capabilities among them. This work presents a technology-independent modular link-level simulator that may be customized with such a generic TX-RX chain developed using the Python programming language. Parallel processing capabilities on its architecture allow the efficient use of the computer's hardware. Also, some processing modules can be further optimized with the use of a specific Python-to-C++ converter package, producing a lower execution time when compared to the use of Python-only features.**

***Keywords*— Simulation, communications systems, parallel computing, modular programming.**

I. Introduction

Over time, the complexity of newly developed communications systems has steadily increased. This comes from the changes on system requirements such as higher transmission rates, spectral efficiency and low power restrictions. One clear example is the oncoming 5G mobile technology, that scales up in 10 times the user experienced data rate, and incorporates IoT capabilities, which demand an increased number of devices accessing the network and operating in fields such as vehicle-to-road communication, agriculture and smart energy distribution systems [1]. To cope with such requirements, new systems must make use of more complex and advanced modulations, pulse shaping, robust signal processing algorithms and more efficient error control coding techniques, in the sense of closely approaching Shannon's channel capacity limit [2], [3].

In order to obtain insight into those emerging technologies and to obtain the performance of a given communication system standard, simulations may be performed. The link-level simulation is responsible for simulating the data transmission through the physical layer. Such approach is commonly adopted for problems and scenarios that are analytically intractable, and it has gained strength thanks to the striving advances on computation over the last decades. Along with hardware advances, high-level programming languages have also emerged, allowing for faster code deployment.

Two major challenges emerge when performing link-level simulations. The first one is execution time. Monte Carlo methods are most usually adopted to obtain metrics such as the bit error rate (BER) and frame error rate (FER) of a physical layer protocol. In doing such, many repetitions of an experiment must be carried out to obtain an accurate estimation of the required statistics, given that they will be estimated by generating draws from a certain probability distribution [2].

The second challenge is the deployment time. Every new technique or procedure must be coded when simulating a different system. High-level programming languages often let a shorter deployment time due to the higher abstraction level. As an example of such abstraction, variable types are deduced in programming languages such as Python. Type deduction simplifies the code and allows the programmer to focus on the algorithmic aspects of the system.

Solutions such as MATLABs Communications Toolbox$^{\text{TM}}$ [4] have no built in structure for link simulations. The usage examples on MATLABs$^{\text{TM}}$ help page [4] contain draft scripts for technology-specific campaigns.

This work presents a simulator that combines four main features that tackle both issues. The first one is the use of the Python programming language higher abstraction level along with the efficient numerical library NumPy [5]. The use of Python and the comprehensive NumPy package allows fast deployment time.

The second feature is its modularity. A modular simulator possesses core features that remain unchanged. This saves development time, since only new modules and functionalities need to be coded from scratch. This is done by implementing a generic TX-RX chain that allows the coding of virtually any physical layer procedure or communication standard into the simulator.

The third feature is the capability of performing parallel computations, in order to make efficient use of the computer's hardware. Since Monte Carlo methods consist of random experiments, they can be performed in parallel, independently from the technology implemented.

The fourth feature is the possibility of optimizing some modules using the Pythran [6] static optimization package, improving further the execution time. Pythran is a Python-to-C++ static compiler. Pythran doesn't encompass all of Python's built-in functionalities, nor all third party modules, but, most importantly, Pythran is compatible with NumPy.

This work is organized as follows: Section II presents the overall structure of the simulator. Section III presents two examples of the aforementioned features. The performance of polar codes and the modeling and behavior of a frequency-locked loop (FLL) are considered. Finally, Section IV presents the conclusions.

Rodrigo A. R. Fischer, Electrical Engineering Department, University of Brasília, Brasília, Brazil, e-mail: rodrigoarfischer@gmail.com; João Paulo Leite, Electrical Engineering Department, University of Brasília, Brasília, Brazil, e-mail: jpauloleite@unb.br; Bruno H. C. Faria, Ektrum Tecnologia, Brasília, Brazil, e-mail: bruno.faria@ektrum.com.

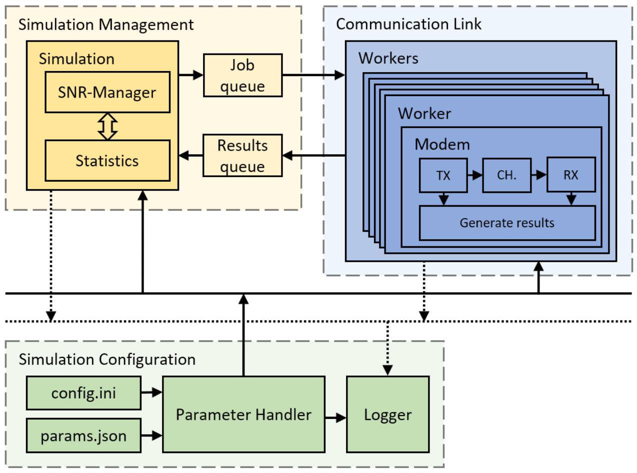

Fig. 1. Overall simulator structure.

II. Simulator Structure

The simulator's structure must be devised so as to embed and encompass all the proposed features. In addition, it must also allow it to be easily set up to implement different communication link configurations.

The architecture that was found to best suit all these basic requirements is shown in Figure 1. Three main task divisions are made: communication link, simulation management and simulation configuration.

This division is inspired by practical considerations: by doing so, these tasks are isolated from each other and can both be designed and function separately. By having the communication link implemented as a separate set of classes, one can design reusable modules of configuration and management so that different link technologies can be simulated by only modifying the communication link module, which is operated by the simulation management classes. Configuration parameters and log information are managed by the configuration classes. Each task subdivision will be discussed next.

A. Communication Link

In order to perform a Monte Carlo link simulation one must implement in code the desired communication link procedures such as modulation, pulse shaping, signal processing and error control coding. These procedures usually involve complex and computationally expensive algorithms. In contrast, setting up the simulation, performing its management and computing statistics require, in comparison to communication link procedures, much less resources, due to its simpler nature and to the fact that they must be executed only once.

There are two conflicting features desired for the simulator: high-level abstraction and fast efficient code. In order to enjoy both features while using the simulator, one may take advantage of its modular design. One can find the bottleneck blocks of the simulation and perform optimizations within these blocks, leaving the lower resource consumption blocks unchanged. Therefore, the extra deployment time needed to optimize the code is efficiently used on the most resource consuming blocks. With this in mind, the communication link was isolated from the configuration and simulation management blocks.

In order to execute multiple Monte Carlo experiments simultaneously, the Master/Worker parallel computing pattern was implemented [7]. This pattern was preferred to optimizing bottlenecks within the *Modem*, since this kind of optimization is technology-dependent and sometimes TX-RX operations can't be made in parallel within a single experiment. The communication link lies within a `Worker` structure that is able to function in parallel with other `Workers`.

1) Modem: All the communication link procedures are encapsulated on a generic structure called `Modem`. The `Modem` object consists of the methods that define the transmitter (TX), the communication channel (CH.) and the receiver (RX), all serially concatenated. These methods are abstract, meaning that the user chooses how the transmission occurs, how the channel operates and how the signal is received. One example is to use QAM modulation and root-raised cosine (RRC) pulse shaping at the transmitter, additive white Gaussian noise (AWGN) at the channel and matched filtering reception at the receiver. As will be seen on Section III, this abstraction is generic enough to encompass even more complex receiver models, such as the one shown in Figure 2.

2) Worker : The `Worker` abstraction allows several Monte Carlo experiments involving the Modem to be executed simultaneously. For each `Worker`, a Python process is started, using the built-in `multiprocessing` Python package.

With the purpose of allowing efficient parallel computing, the `Workers` interface with queues only. Two main queues are required: the `job queue` and the `results queue`, in which the execution instructions and the generated results are placed, respectively. This feature allows for many parallel computing configurations. As an example, the user may instruct all the `Workers` to simulate a scenario with the same signal-to-noise ratio (SNR) and then retrieve the results. The `Workers` can also be assigned so that each one simulates a different SNR value from a given range. Further, each `Worker` may be assigned to a different set of `Modem` parameters, such as code-rate.

B. Simulation Management

The simulation management objects manages the `Workers` operation and collects the generated results. The fact that the `Workers` only interface with the queues allows the user to create custom simulation management objects for different simulation types. One example is the multiple parallel configurations possible presented on Section II-A.2.

One may already presume that the assignment of the next SNR to be simulated is an important task that can be done in multiple ways. In the same manner, the computation of the desired statistics is a relevant task of the simulator. Another example would be a variable step SNR range option in which a greater SNR sampling resolution could be applied where the metric values such as BER or FER seem to change rapidly. Therefore, a `SNR-Manager` and a `Statistics` objects were created as a subdivision of the simulation management task.

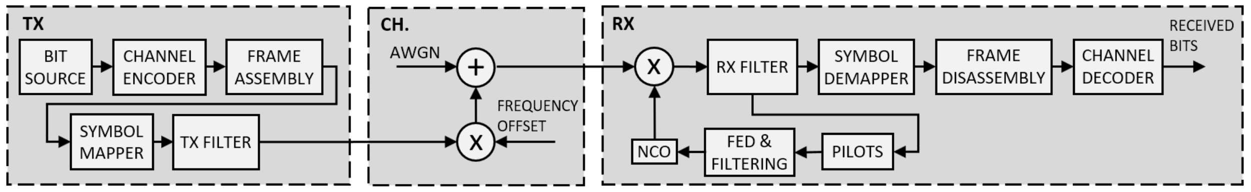

Fig. 2. An example of a digital communication link. The link is composed of three modules: the transmitter (TX), the communication channel (CH.) and the receiver (RX). Each one of these modules is composed of sub-components that make up the link.

1) Simulation: The `Simulation` object is responsible for assigning tasks to the `Workers` and for managing the `SNR-Manager` and `Statistics` objects.

2) SNR-Manager: The `SNR-Manager` object determines the range of SNR values to be simulated. One way of doing so is by defining starting and ending values along with the steps between them. This object receives commands from the `Simulation` object, such as a command for it to return the next SNR value. The `SNR-Manager` can also output directives, such as to halt the simulation. One possibility is to halt the simulation only at the end of the SNR range. Another possibility is to halt the simulation when a frame batch of a certain minimum size is processed without errors.

3) Statistics: The `Statistics` object retrieves the results from the `results queue` and computes the desired statistics, such as BER and FER. These statistics are available to the `SNR-Manager` object.

C. Simulation Configuration

A link-level simulation has many parameters to be set up, such as the number of transmitted bits or the SNR range to simulate.

Due to the simulator's modularity, the setup operation is best done by a centralized parameters manager. This allows the simulator to be fully configured using only one interface and the parameters to be distributed to every simulation module.

The simulator's configuration structure was also designed so that the simulator can be executed in a campaign-like fashion, where its basic setup remains unchanged while some parameters vary from campaign to campaign.

1) Parameter Handler: The `Parameter Handler` object parses the parameters and distributes them across the simulation. The provided `.json` file defines all the parameter's names, types, default values, options and descriptions. This file is fully customizable and is provided by the user. The `.ini` file contains the campaign specific parameter values, which are parsed and checked against the data provided by the `.json` file.

2) Logger: Logging capabilities are essential on any simulator. As an example, the `Logger` object can be used to track error events, warnings or the simulation progress. The modular structure of the simulator also requires that the `Logger` is centralized, being able to log any event generated by all the other modules into a file or into the console.

III. Proof of Concept

On this section two examples are used to illustrate the aforementioned features. On Subsection III-A the simulator communication link is implemented with polar channel coding and this scenario is used to illustrate the parallel computing gain that can be obtained by using the proposed architecture and the static optimization gain provided by Pythran. On Subsection III-B the DVB-S2 framing structure is used with a frequency-locked loop implemented on the communication link receiver so as to illustrate the flexibility of the simulator modular architecture.

TABLE I
Communication Link Configuration for Proof of Concept Scenarios

Parameters	Configuration	
	Polar Coding Scenario	**FLL Scenario**
Channel Coding	Rate 1/2 Polar Coding Bhattacharyya Constr.	-
Framing Structure	-	DVB-S2
Symbol Mapping	BPSK	QPSK
TX Filter	-	RRC
Samples per Symbol	1	8
Payload Size (bits)	2048	64800
Frame Size (symbols)	2048	33282
Communication Channel	AWGN	AWGN & Frequency Offset
RX Filter	-	RRC
Synchronization	Ideal	FLL

Both configurations can be simulated by modifying the `Modem` object and the corresponding `.json` and `.ini` files. Figure 2 shows a generic communication link that encompasses both proposed communication link scenarios. To obtain the specific communication link for each scenario, the setup presented on Table I was considered.

A. Polar Coding

Polar codes are a family of block codes proposed by Arikan [8]. Given a certain construction method [9], the code is completely specified by the block length and code rate. The polar coding configuration is as present on Table I using the Bhattacharyya construction method [9].

Two different computing machines were used in order to compare the simulator performance on different computer architectures. The test machines' setup is shown on Table II, where HW/SW identifies the Hardware and Software items, whereas OS identifies the operational system. Machine 1 refers to a personal computer, while Machine 2 refers to a Google Cloud Platform virtual machine with 16 virtual CPUs.

1) Static Optimization Gain: A profiling campaign was designed to asses the gain provided by the use of Pythran. The

TABLE II
MACHINE CONFIGURATION

HW/SW	Specification	
	Machine 1	**Machine 2**
OS	Windows 10 Pro	Debian GNU/Linux 9.12
Processor	AMD FX(tm)-8350 Eight-Core Processor 4.00 GHz	16 Intel(R) Xeon(R) Virtual CPUs @ 2.00GHz
RAM	16.0 GB	60.0 GB
Python	3.7.4	3.7.4
Compiler	Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24245 for x86	gcc version 6.3.0 20170516 (Debian 6.3.0-18+deb9u1)

TABLE III
SIMULATION CONFIGURATION FOR PROOF OF CONCEPT SCENARIOS

Parameters	Configuration		
	Campaign 1	**Campaign 2**	**Campaign 3**
SNR Range	Noiseless	-3 to 0 dB with a 0.5 dB step	0 dB
Frequency Offset	0%	0%	10%
Number of TX Frames	10^4	10^5 for each SNR	1200
Number of Workers	1	1 to 512	1

TABLE IV
PROFILING RESULTS WITH 10^4 POLAR CODED FRAMES

Executed Code	Execution Time (s)	
	Machine 1	**Machine 2**
Pure Python	3899.2	2605.9
Pythran Generated	129.2	63.2
Speedup	30.2	41.26

campaign configuration can be seen identified as Campaign 1 on Table III. This campaign consisted of a single `Worker` initialized on a script that was responsible for creating a loop to iterate though the frame count and to assign the `Worker` jobs. This is an example of how the simulator modularity allows it to operate with different simulation designs. In this case, the simulation management modules were substituted by a instruction script.

Using this campaigns, two different scenarios were simulated: one using the polar encoder and decoder pure Python implementation, and another using the static optimization of both provided by Pythran.

The profiling results are shown on Table IV. These results account only for the time elapsed during the encoding and decoding operations. The substitution of the pure Python implementation with the compiled module module led to a 30-fold speedup for Machine 1 and a 40-fold speedup for Machine 2.

2) Parallel Computing Gain: A campaign designed to asses the gain provided by the proposed parallel computing architecture was set up. Identified as Campaign 2, the simulator configuration can be seen on Table III. The polar encoding and decoding operations are optimized using the Pythran module. This campaign was then performed multiple times on both machines with the same base setup, varying only the number of `Workers` in a exponential manner. The simulation execution times were used to compute the parallel computing speedup, where the reference execution time is the time corresponding

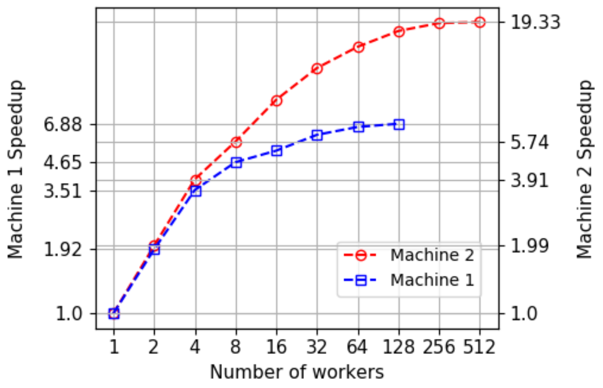

Fig. 3. Logarithmic scale plot of the speedup for both test machines compared to the base scenario of only one worker for each machine.

to only one worker for each machine.

A logarithmic scale plot of the obtained speedups is shown on Figure 3. The curves for both machines exhibit the same behavior; a linear start with later saturation. The linear start corresponds to the range on which if the `Workers` are doubled, the execution time is halved, and thus the speedup is also doubled. Due to other OS concurrent processes, the curve deviates from the linear behavior before reaching the number of `Workers` equivalent to the machine's CPU count. It is worth pointing that even after the behavior turns nonlinear there are still parallel computing gains. In order to obtain the best performance for a given machine, a reasonable procedure would be to perform a profiling campaign that gradually increases the number of workers until the machine's processing capacity is attained. This value is capable of providing the best parallel computing gain. For Machine 2, an almost 20-fold gain was obtained by massively assigning `Workers` to the campaign.

All campaigns, regardless of the number of `Workers`, led to the same statistical results, as can be seen on Figure 4. These results also match the expected performance for polar coding simulations with the Bhattacharyya construction parameter [9], thus validating the simulator implementation. The deviation among the curves on the last SNR value illustrates the Monte Carlo variance behavior when not enough events are simulated.

B. Frequency-Locked Loop (FLL)

The simulator architecture is versatile enough to incorporate more complex elements inside the `Modem` abstraction, which opens up the possibility to asses the performance of signal

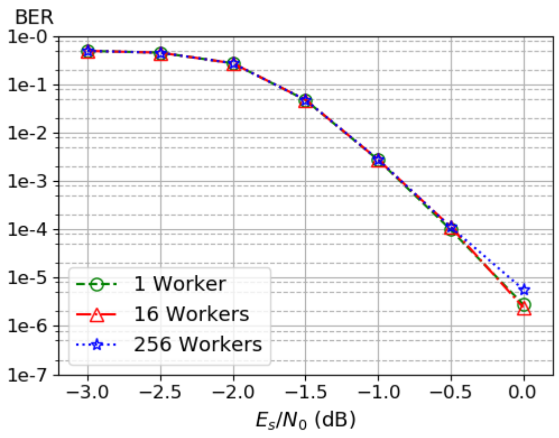

Fig. 4. Performance in BER for Bhattacharyya-constructed rate-1/2 polar codes with block size 4096 obtained by different campaigns with different number of workers.

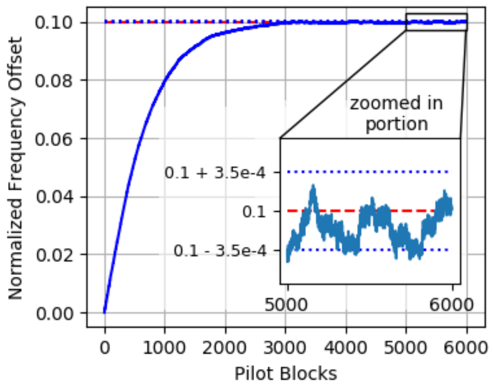

Fig. 5. Pilot-aided FLL estimated normailzed frequency error for the DVB-S2 frame structure.

processing algorithms and receiver architectures as well. The CH. method can also be changed to model other impairments such as frequency offset error, among other RF imperfections. Correspondingly, on the RX method, a frequency locked loop can be implemented in order to recover the frequency and phase of the transmitted signal. Three blocks compose the FLL: the pilot extraction block, the forward error detection (FED) and the FED filter, and the numerically controlled oscillator (NCO). The feedback loop is formed when the frequency offset estimation obtained by the received pilots is used to drive the NCO, which in turn has its output multiplied by the incoming signal in order to correct the frequency offset.

The FLL implementation is exemplified using the Digital Video Broadcasting - Satellite - Second Generation (DVB-S2) framing structure using pilots [10]. The pilot blocks are repeated every 1440 data symbols, which yields a maximum residual frequency offset of $\pm 3.5 \times 10^{-4}$ that can be later corrected by a phase correction algorithm [11]. The FED and the FLL filter used are as in [11].

The simulator is also equipped with probing capabilities that allow the extraction of data from any block. This functionality was used to retrieve the estimated frequency offset generated by the FLL. A campaign was set up, as identified by Campaign 3 in Table III. The frequency offset output by the FED & Filtering block is shown on Figure 5. The frequency estimation converges to the value present at the channel, as expected. A zoomed in portion of the curve is shown depicting also the frequency correction algorithm targets.

IV. Conclusion

This work presents a flexible modular link-level simulator that explores parallel computing and static optimization features available in Python in order to perform time efficient Monte Carlo simulations.

The simulator's performance was tested using a robust cloud computing platform and a personal computer. Considerable parallel computing gains were observed for both architectures, yielding a 20-fold speedup for the cloud computing platform. However, this simulator is suitable even for small scale simulations on a personal computer, with an observed 6-fold parallel computing speedup. Also, the static optimization allowed a 40-fold speedup on the cloud computing platform. This improves even further the overall gains when combined with the use of parallel computing.

The modular and expansible framework of the link-level simulator was demonstrated by analyzing two different communication link scenarios: the BER performance of polar coding and the behavior of a FLL in a DVB-S2 receiver chain. The results also made it possible to evaluate the correctness of the coded algorithms and proposed interfaces.

References

[1] ITU, "IMT Vision - Framework and overall objectives of the future development of IMT for 2020 and beyond," ITU, Recommendation ITU-R M.2083-0, Accessed: Apr. 19, 2020. [Online]. Available: https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2083-0-201509-I!!PDF-E.pdf

[2] W. H. Tranter, K. S. Shanmugan, T. S. Rappaport, K. L. Kosbar, *Principles of Communication Systems Simulation with Wireless Applications*. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[3] D. J. Costello and G. D. Forney, "Channel coding: The road to channel capacity," in Proceedings of the IEEE, vol. 95, no. 6, pp. 1150-1177, June 2007, doi: 10.1109/JPROC.2007.895188.

[4] MathWorks, (2018). Communications Toolbox: User's Guide (R2018b). Retrieved October 5, 2020 from https://www.mathworks.com/help/releases/R2018b/comm/index.html

[5] Travis E. Oliphant, *A guide to NumPy*. USA: Trelgol Publishing, (2006).

[6] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, A. Raynaud, "Pythran: Enabling static optimization of scientific Python programs," *Computational Science & Discovery*, vol. 8, no. 1, p. 014001, 2015.

[7] T. Mattson, B. Sanders, B. Massingill, "The Master/Worker Pattern," in *Patterns for Parallel Programming*, First ed. Boston, Massachusetts, EUA: Addison-Wesley Professional, 2004, ch. 5, sec. 5.

[8] E. Arikan, "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels," in IEEE Transactions on Information Theory, vol. 55, no. 7, pp. 3051-3073, July 2009, doi: 10.1109/TIT.2009.2021379.

[9] B. Tahir, "Construction and Performance of Polar Codes for Transmission over the AWGN Channel," M.S. thesis, Inst. of Telecommunications, TU Wien, Vienna, Austria, 2017. [Online]. Available: https://publik.tuwien.ac.at/files/publik_262980.pdf

[10] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2*, ETSI Standard EN 302 307-1, Nov. 2014.

[11] *Digital Video Broadcasting (DVB); Implementation guidelines for the second generation system for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2*, ETSI Standard TR 102 376-1, Nov. 2015.