# Using pyAerial to run a PUSCH link simulation

# Table of contents

This example shows how to use the pyAerial cuPHY Python bindings to run a PUSCH link simulation. PUSCH transmitter is emulated by PDSCH transmission with properly chosen parameters, that way making it a 5G NR compliant PUSCH transmission. Building a PUSCH receiver using pyAerial is demonstrated in two ways, first by using a fully fused, complete, PUSCH receiver called from Python using just a single function call. The same is then achieved by building the complete PUSCH receiver using individual separate Python function calls to individual PUSCH receiver components.

The NVIDIA Sionna library is utilized for simulating the radio channel based on 3GPP channel models.

# Imports

[1]:

```
%matplotlib widget import datetime from collections import defaultdict import os os.environ["CUDA_VISIBLE_DEVICES"] = "0" os.environ['TF_CPP_MIN_LOG_LEVEL'] = "3" # Silence TensorFlow. import numpy as np import matplotlib.pyplot as plt import sionna import tensorflow as tf from aerial.phy5g.pdsch import PdschTx from aerial.phy5g.pusch import PuschRx from aerial.phy5g.algorithms import ChannelEstimator from aerial.phy5g.algorithms import ChannelEqualizer from aerial.phy5g.algorithms import NoiseIntfEstimator from aerial.phy5g.algorithms import Demapper from aerial.phy5g.ldpc import get_mcs from aerial.phy5g.ldpc import random_tb from aerial.phy5g.ldpc import LdpcDeRateMatch from aerial.phy5g.ldpc import LdpcDecoder from aerial.phy5g.ldpc import code_block_desegment from aerial.phy5g.types import PuschLdpcKernelLaunch from aerial.util.cuda import get_cuda_stream # Configure the notebook to use only a single GPU and allocate only as much memory as needed. # For more details, see https://www.tensorflow.org/guide/gpu. gpus = tf.config.list_physical_devices('GPU') tf.config.experimental.set_memory_growth(gpus[0], True)
```

# Parameters

Set simulation parameters, numerology, PUSCH parameters and channel parameters here.

[2]:

```
# Simulation parameters.
esno_db_range = np.arange(-5.4, -4.4, 0.2)
num_slots = 10000
min_num_tb_errors = 250
# Numerology and frame structure. See TS 38.211.
num_ofdm_symbols = 14
fft_size = 4096
cyclic_prefix_length = 288
subcarrier_spacing = 30e3
num_guard_subcarriers = (410, 410)
num_slots_per_frame = 20
num_tx_ant = 1 # UE antennas
num_rx_ant = 2 # gNB antennas
cell_id = 41 # Physical cell ID
rnti = 1234 # UE RNTI
scid = 0 # DMRS scrambling ID
data_scid = 0 # Data scrambling ID
layers = 1 # Number of layers
mcs = 1 # MCS index as per TS 38.214 table
dmrs_port = 1 # Used DMRS port.
start_prb = 0 # Start PRB index.
num_prbs = 273 # Number of allocated PRBs.
start_sym = 2 # Start symbol index.
num_symbols = 12 # Number of symbols.
dmrs_scrm_id = 41 # DMRS scrambling ID
dmrs_position = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# Indicates which symbols are used for DMRS.
dmrs_max_len=1
dmrs_add_ln_pos=0
num_dmrs_cdm_grps_no_data = 2
enable_pusch_tdi = 0
# Enable time interpolation for equalizer coefficients
eq_coeff_algo = 1
# Equalizer algorithm
# Channel parameters
carrier_frequency = 3.5e9 # Carrier frequency in Hz.
delay_spread = 100e-9
# Nominal delay spread in [s]. Please see the CDL documentation
# about how to choose this value.
link_direction = "uplink"
channel_model = "Rayleigh"
# Channel model: Suitable values:
# "Rayleigh" - Rayleigh block fading channel model (sionna.channel.RayleighBlockFading)
# "CDL-x", where x is one of ["A", "B", "C", "D", "E"] - for 3GPP CDL channel models
# as per TR 38.901.
speed = 0.8333 # UE speed [m/s]. The direction of travel will chosen randomly within the x-y plane.
```

# Create the PUSCH pipelines

As mentioned, PUSCH transmission is emulated here by the PDSCH transmission chain. Note that the static cell parameters and static PUSCH parameters are given upon creating the PUSCH transmission/reception objects. Dynamically (per slot) changing parameters are however set when actually running the transmission/reception, see further below.

[3]:

```
pusch_tx = PdschTx(
    cell_id=cell_id,
    num_rx_ant=num_tx_ant,
    num_tx_ant=num_tx_ant,
) # This is the fully fused PUSCH receiver chain.
pusch_rx =
```

PuschRx( cell_id=cell_id, num_rx_ant=num_rx_ant, num_tx_ant=num_rx_ant, enable_pusch_tdi=enable_pusch_tdi, eq_coeff_algo=eq_coeff_algo, *# To make this equal separate PUSCH Rx components configuration:* ldpc_kernel_launch=PuschLdpcKernelLaunch.PUSCH_RX_LDPC_STREAM_SEQUENTIAL ) *# The PUSCH receiver chain built from separately called pyAerial Python components is defined here.* class PuschRxSeparate: """PUSCH receiver class. This class encapsulates the whole PUSCH receiver chain built using pyAerial components. """ def __init__(self, num_rx_ant, enable_pusch_tdi, eq_coeff_algo): """Initialize the PUSCH receiver.""" self.cuda_stream = get_cuda_stream() *# Build the components of the receiver.* self.channel_estimator = ChannelEstimator( num_rx_ant=num_rx_ant, cuda_stream=self.cuda_stream ) self.channel_equalizer = ChannelEqualizer( num_rx_ant=num_rx_ant, enable_pusch_tdi=enable_pusch_tdi, eq_coeff_algo=eq_coeff_algo, cuda_stream=self.cuda_stream ) self.noise_intf_estimator = NoiseIntfEstimator( num_rx_ant=num_rx_ant, eq_coeff_algo=eq_coeff_algo, cuda_stream=self.cuda_stream ) self.derate_match = LdpcDeRateMatch( enable_scrambling=True, cuda_stream=self.cuda_stream ) self.decoder = LdpcDecoder(cuda_stream=self.cuda_stream) def run( self, rx_slot, num_ues, slot, num_dmrs_cdm_grps_no_data, dmrs_scrm_id, start_prb, num_prbs, dmrs_syms, dmrs_max_len, dmrs_add_ln_pos, start_sym, num_symbols, scids, layers, dmrs_ports, rntis, data_scids, code_rates, mod_orders, tb_sizes ): """Run the receiver.""" *# Channel estimation.* ch_est = self.channel_estimator.estimate( rx_slot=rx_slot, num_ues=num_ues, slot=slot, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports ) *# Noise and interference estimation.* lw_inv, noise_var_pre_eq = self.noise_intf_estimator.estimate( rx_slot=rx_slot, channel_est=ch_est, num_ues=num_ues, slot=slot, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len, dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym, num_symbols=num_symbols, scids=scids, layers=layers, dmrs_ports=dmrs_ports ) *# Channel equalization and soft demapping. The first return value are the LLRs, # second*

```
are the equalized symbols. We only want the LLRs now. llr =
self.channel_equalizer.equalize( rx_slot=rx_slot, channel_est=ch_est, lw_inv=lw_inv,
noise_var_pre_eq=noise_var_pre_eq, num_ues=num_ues,
num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, start_prb=start_prb,
num_prbs=num_prbs, dmrs_syms=dmrs_syms, dmrs_max_len=dmrs_max_len,
dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym,
num_symbols=num_symbols, layers=layers, mod_orders=mod_orders )[0]
decoded_tbs = [] num_data_sym = (np.array(dmrs_syms[start_sym:start_sym +
num_symbols]) == 0).sum() tot_layers = 0 for ue_idx in range(num_ues): ue_layers =
range(tot_layers, tot_layers + layers[ue_idx]) tot_layers += layers[ue_idx] # De-rate
matching and descrambling. cinit = (rntis[ue_idx] << 15) + data_scids[ue_idx]
rate_match_len = num_data_sym * mod_orders[ue_idx] * num_prbs * 12 *
layers[ue_idx] coded_blocks = self.derate_match.derate_match( input_data=llr[0][:,
ue_layers, ...], tb_size=tb_sizes[ue_idx] * 8, code_rate=code_rates[ue_idx] / 1024.,
rate_match_len=rate_match_len, mod_order=mod_orders[ue_idx],
num_layers=layers[ue_idx], redundancy_version=0, ndi=1, cinit=cinit ) # LDPC
decoding of the derate matched blocks. code_blocks = self.decoder.decode(
input_llr=coded_blocks, tb_size=tb_sizes[ue_idx] * 8, code_rate=code_rates[ue_idx] /
1024., redundancy_version=0, rate_match_len=rate_match_len ) # Combine the code
blocks into a transport block. tb = code_block_desegment( code_blocks=code_blocks,
tb_size=tb_sizes[ue_idx] * 8, code_rate=code_rates[ue_idx] / 1024.,
return_bits=False, ) # Remove CRC - no checking, check TBs/bits directly. tb = tb[:-3]
decoded_tbs.append(tb) return decoded_tbs pusch_rx_separate =
PuschRxSeparate( num_rx_ant=num_rx_ant, enable_pusch_tdi=enable_pusch_tdi,
eq_coeff_algo=eq_coeff_algo )
```

# Channel generation using Sionna

Simulating the transmission through the radio channel takes advantage of the channel model implementations available in NVIDIA Sionna. In Sionna, the transmission can be simulated directly in frequency domain by defining a resource grid. In our case, reference signal patterns and data carrying resource elements are defined elsewhere within the Aerial code, hence we define resource grid as a simple dummy grid containing only data symbols.

See also: Sionna documentation

[4]:

*# Define the resource grid.* resource_grid = sionna.ofdm.ResourceGrid( num_ofdm_symbols=num_ofdm_symbols, fft_size=fft_size, subcarrier_spacing=subcarrier_spacing, num_tx=1, num_streams_per_tx=1, cyclic_prefix_length=cyclic_prefix_length, num_guard_carriers=num_guard_subcarriers, dc_null=False, pilot_pattern=None, pilot_ofdm_symbol_indices=None ) resource_grid_mapper = sionna.ofdm.ResourceGridMapper(resource_grid) remove_guard_subcarriers = sionna.ofdm.RemoveNulledSubcarriers(resource_grid) *# Define the antenna arrays.* ue_array = sionna.channel.tr38901.Antenna( polarization="single", polarization_type="V", antenna_pattern="38.901", carrier_frequency=carrier_frequency ) gnb_array = sionna.channel.tr38901.AntennaArray( num_rows=1, num_cols=int(num_rx_ant/2), polarization="dual", polarization_type="cross", antenna_pattern="38.901", carrier_frequency=carrier_frequency ) if channel_model == "Rayleigh": ch_model = sionna.channel.RayleighBlockFading( num_rx=1, num_rx_ant=num_rx_ant, num_tx=1, num_tx_ant=num_tx_ant ) elif "CDL" in channel_model: cdl_model = channel_model[-1] *# Configure a channel impulse reponse (CIR) generator for the CDL model.* ch_model = sionna.channel.tr38901.CDL( cdl_model, delay_spread, carrier_frequency, ue_array, gnb_array, link_direction, min_speed=speed ) else: raise ValueError(f"Invalid channel model{channel_model}!") channel = sionna.channel.OFDMChannel( ch_model, resource_grid, add_awgn=True, normalize_channel=True, return_channel=False ) def apply_channel(tx_tensor, No): """Transmit the Tx tensor through the radio channel.""" *# Add batch and num_tx dimensions that Sionna expects and reshape.* tx_tensor = tf.transpose(tx_tensor, (2, 1, 0)) tx_tensor = tf.reshape(tx_tensor, (1, -1))[None, None] tx_tensor = resource_grid_mapper(tx_tensor) rx_tensor = channel((tx_tensor, No)) rx_tensor = remove_guard_subcarriers(rx_tensor) rx_tensor = rx_tensor[0, 0] rx_tensor = tf.transpose(rx_tensor, (2, 1, 0)) return rx_tensor

# Helper class for simulation monitoring

This helper class plots the simulation results and shows simulation progress in a table.

[5]:

```
class SimulationMonitor: """Helper class to show the progress and results of the simulation."""  markers = ["d", "o", "s"] linestyles = ["-", "--", ":"] colors = ["blue", "black", "red"] def __init__(self, cases, esno_db_range): """Initialize the SimulationMonitor. Initialize the figure and the results table. """ self.cases = cases self.esno_db_range = esno_db_range self.current_esno_db_range = [] self.start_time = None self.esno_db = None self.bler = defaultdict(list) self._print_headers() def step(self, esno_db): """Start next Es/No value."""  self.start_time = datetime.datetime.now() self.esno_db = esno_db self.current_esno_db_range.append(esno_db) def update(self, num_tbs, num_tb_errors): """Update current state for the current Es/No value."""  self._print_status(num_tbs, num_tb_errors, False) def _print_headers(self): """Print result table headers."""  cases_str = " " * 21 separator = " " * 21 for case in self.cases: cases_str += case.center(20) + " " separator += "-" * 20 + " " print(cases_str) print(separator) title_str = "Es/No (dB)".rjust(12) + "TBs".rjust(8) + " " for case in self.cases: title_str += "TB Errors".rjust(12) + "BLER".rjust(8) + " " title_str += "ms/TB".rjust(8) print(title_str) print(("=" * 20) + " " + ("=" * 20 + " ") * len(self.cases) + "=" * 8) def _print_status(self, num_tbs, num_tb_errors, finish): """Print simulation status in a table."""  end_time = datetime.datetime.now() t_delta = end_time - self.start_time if finish: newline_char = '\n' else: newline_char = '\r' result_str = f"{self.esno_db:9.2f}".rjust(12) + f"{num_tbs:8d}".rjust(8) + " " for case in self.cases: result_str += f"{num_tb_errors[case]:8d}".rjust(12) result_str += f"{(num_tb_errors[case] / num_tbs):.4f}".rjust(8) + " " result_str += f"{(t_delta.total_seconds() * 1000 / num_tbs):6.1f}".rjust(8) print(result_str, end=newline_char) def finish_step(self, num_tbs, num_tb_errors): """Finish simulating the current Es/No value and add the result in the plot."""  self._print_status(num_tbs, num_tb_errors, True) for case_idx, case in enumerate(self.cases): self.bler[case].append(num_tb_errors[case] / num_tbs) def finish(self): """Finish simulation and plot the results."""  self.fig = plt.figure() for case_idx, case in enumerate(self.cases): plt.plot( self.current_esno_db_range, self.bler[case], marker=SimulationMonitor.markers[case_idx], linestyle=SimulationMonitor.linestyles[case_idx], color=SimulationMonitor.colors[case_idx], markersize=8, label=case ) plt.yscale('log') plt.ylim(0.001, 1) plt.xlim(np.min(self.esno_db_range), np.max(self.esno_db_range))
```

```
plt.title("Receiver BLER Performance vs. Es/No") plt.ylabel("BLER") plt.xlabel("Es/No [dB]") plt.grid() plt.legend() plt.show()
```

## Run the actual simulation

Here we loop across the Es/No range, and simulate a number of slots for each Es/No value. A single transport block is simulated within a slot. The simulation starts over from the next Es/No value when a minimum number of transport block errors is reached.

[6]:

```
cases = ["Fused", "Separate"] monitor = SimulationMonitor(cases, esno_db_range) # Loop the Es/No range. bler = [] for esno_db in esno_db_range: monitor.step(esno_db) num_tb_errors = defaultdict(int) # Run multiple slots and compute BLER. for slot_idx in range(num_slots): slot_number = slot_idx % num_slots_per_frame # Get modulation order and coderate. mod_order, coderate = get_mcs(mcs) tb_input = random_tb(mod_order, coderate, dmrs_position, num_prbs, start_sym, num_symbols, layers) # Transmit PUSCH. This is where we set the dynamically changing parameters. # Input parameters are given as lists as the interface supports multiple UEs. tx_tensor = pusch_tx.run( tb_inputs=[tb_input], # Input transport block in bytes. num_ues=1, # We simulate only one UE here. slot=slot_number, # Slot number. num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, # DMRS scrambling ID. start_prb=start_prb, # Start PRB index. num_prbs=num_prbs, # Number of allocated PRBs. dmrs_syms=dmrs_position, # List of binary numbers indicating which symbols are DMRS. start_sym=start_sym, # Start symbol index. num_symbols=num_symbols, # Number of symbols. scids=[scid], # DMRS scrambling ID. layers=[layers], # Number of layers (transmission rank). dmrs_ports=[dmrs_port], # DMRS port(s) to be used. rntis=[rnti], # UE RNTI. data_scids=[data_scid], # Data scrambling ID. code_rates=[coderate], # Code rate x 1024. mod_orders=[mod_order] # Modulation order. )[0] # Channel transmission using TF and Sionna. No = pow(10., -esno_db / 10.) rx_tensor = apply_channel(tx_tensor, No) rx_tensor = np.array(rx_tensor) # Run the fused PUSCH receiver. # Note that this is where we set the dynamically changing parameters. tb_crcs, tbs = pusch_rx.run( rx_slot=rx_tensor, num_ues=1, slot=slot_number, num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data, dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs,
```

```
dmrs_syms=dmrs_position, dmrs_max_len=dmrs_max_len,
dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym,
num_symbols=num_symbols, scids=[scid], layers=[layers], dmrs_ports=[dmrs_port],
rntis=[rnti], data_scids=[data_scid], code_rates=[coderate], mod_orders=
[mod_order], tb_sizes=[len(tb_input)] ) num_tb_errors["Fused"] +=
int(np.array_equal(tbs[0][:-3], tb_input) == False) # Run the receiver built from
separately called components. tbs = pusch_rx_separate.run( rx_slot=rx_tensor,
num_ues=1, slot=slot_number,
num_dmrs_cdm_grps_no_data=num_dmrs_cdm_grps_no_data,
dmrs_scrm_id=dmrs_scrm_id, start_prb=start_prb, num_prbs=num_prbs,
dmrs_syms=dmrs_position, dmrs_max_len=dmrs_max_len,
dmrs_add_ln_pos=dmrs_add_ln_pos, start_sym=start_sym,
num_symbols=num_symbols, scids=[scid], layers=[layers], dmrs_ports=[dmrs_port],
rntis=[rnti], data_scids=[data_scid], code_rates=[coderate], mod_orders=
[mod_order], tb_sizes=[len(tb_input)] ) num_tb_errors["Separate"] +=
int(np.array_equal(tbs[0], tb_input) == False) monitor.update(num_tbs=slot_idx + 1,
num_tb_errors=num_tb_errors) if (np.array(list(num_tb_errors.values())) >=
min_num_tb_errors).all(): break # Next Es/No value.
monitor.finish_step(num_tbs=slot_idx + 1, num_tb_errors=num_tb_errors)
monitor.finish()
```

```
Fused Separate ---------- ---------- Es/No (dB) TBs TB Errors BLER TB
Errors BLER ms/TB ========== ========== ========== ========== ========
-5.40 250 250 1.0000 250 1.0000 96.4 -5.20 259 252 0.9730 250 0.9653 95.1
-5.00 539 250 0.4638 252 0.4675 95.8 -4.80 10000 177 0.0177 175 0.0175 95.4
-4.60 10000 0 0.0000 0 0.0000 111.5
```