gnpy Documentation

Telecom InfraProject - OOPT PSE Group

Jun 02, 2021

CONTENTS

GNPy is an open-source, community-developed library for building route planning and optimization tools in real-world mesh optical networks. It is based on the Gaussian Noise Model.

CHAPTER

ONE

INSTALLING GNPY

There are several methods on how to obtain GNPy. The easiest option for a non-developer is probably going via our *Docker images*. Developers are encouraged to install the *Python package in the same way as any other Python package*. Note that this needs a *working installation of Python*, for example *via Anaconda*.

1.1 Using prebuilt Docker images

Our Docker images contain everything needed to run all examples from this guide. Docker transparently fetches the image over the network upon first use. On Linux and Mac, run:

```
$ docker run -it --rm --volume $(pwd):/shared telecominfraproject/oopt-gnpy
root@bea050f186f7:/shared/example-data#
```

On Windows, launch from Powershell as:

```
PS C:\> docker run -it --rm --volume ${PWD}:/shared telecominfraproject/oopt-gnpy
root@89784e577d44:/shared/example-data#
```

In both cases, a directory named `example-data/` will appear in your current working directory. GNPy automatically populates it with example files from the current release. Remove that directory if you want to start from scratch.

1.2 Using Python on your computer

> **Note**: *gnpy* supports Python 3 only. Python 2 is not supported. *gnpy* requires Python 3.6
>
> **Note**: the *gnpy* maintainers strongly recommend the use of Anaconda for managing dependencies.

It is recommended that you use a "virtual environment" when installing *gnpy*. Do not install *gnpy* on your system Python.

We recommend the use of the Anaconda Python distribution which comes with many scientific computing dependencies pre-installed. Anaconda creates a base "virtual environment" for you automatically. You can also create and manage your `conda` "virtual environments" yourself (see: https://conda.io/docs/user-guide/tasks/manage-environments.html)

To activate your Anaconda virtual environment, you may need to do the following:

```
$ source /path/to/anaconda/bin/activate # activate Anaconda base environment
(base) $                                # note the change to the prompt
```

You can check which Anaconda environment you are using with:

```
(base) $ conda env list                          # list all environments
# conda environments:
#
base                  *  /src/install/anaconda3

(base) $ echo $CONDA_DEFAULT_ENV                 # show default environment
base
```

You can check your version of Python with the following. If you are using Anaconda's Python 3, you should see similar output as below. Your results may be slightly different depending on your Anaconda installation path and the exact version of Python you are using.

```
$ which python                    # check which Python executable is used
/path/to/anaconda/bin/python
$ python -V                       # check your Python version
Python 3.6.5 :: Anaconda, Inc.
```

1.2.1 Installing the Python package

From within your Anaconda Python 3 environment, you can clone the master branch of the *gnpy* repo and install it with:

```
$ git clone https://github.com/Telecominfraproject/oopt-gnpy # clone the repo
$ cd oopt-gnpy
$ python setup.py develop
```

To test that *gnpy* was successfully installed, you can run this command. If it executes without a `ModuleNotFoundError`, you have successfully installed *gnpy*.

```
$ python -c 'import gnpy' # attempt to import gnpy

$ pytest                  # run tests
```

CHAPTER

TWO

JSON INPUT FILES

GNPy uses a set of JSON files for modeling the network. Some data (such as network topology or the service requests) can be also passed via *XLS files*.

2.1 Equipment Library

Design and transmission parameters are defined in a dedicated json file. By default, this information is read from gnpy/example-data/eqpt_config.json. This file defines the equipment libraries that can be customized (EDFAs, fibers, and transceivers).

It also defines the simulation parameters (spans, ROADMs, and the spectral information to transmit.)

2.1.1 EDFA

The EDFA equipment library is a list of supported amplifiers. New amplifiers can be added and existing ones removed. Three different noise models are available:

1. `'type_def': 'variable_gain'` is a simplified model simulating a 2-coil EDFA with internal, input and output VOAs. The NF vs gain response is calculated accordingly based on the input parameters: `nf_min`, `nf_max`, and `gain_flatmax`. It is not a simple interpolation but a 2-stage NF calculation.
2. `'type_def': 'fixed_gain'` is a fixed gain model. *NF == Cte == nf0* if *gain_min < gain < gain_flatmax*
3. `'type_def': None` is an advanced model. A detailed JSON configuration file is required (by default gnpy/example-data/std_medium_gain_advanced_config.json). It uses a 3rd order polynomial where NF = f(gain), NF_ripple = f(frequency), gain_ripple = f(frequency), N-array dgt = f(frequency). Compared to the previous models, NF ripple and gain ripple are modelled.

For all amplifier models:

field	type	description
`type_variety`	(string)	a unique name to ID the amplifier in the JSON/Excel template topology input file
`out_voa_auto`	(boolean)	auto_design feature to optimize the amplifier output VOA. If true, output VOA is present and will be used to push amplifier gain to its maximum, within EOL power margins.
`allowed_for_design`	(boolean)	If false, the amplifier will not be picked by auto-design but it can still be used as a manual input (from JSON or Excel template topology files.)

2.1.2 Fiber

The fiber library currently describes SSMF and NZDF but additional fiber types can be entered by the user following the same model:

field	type	description
`type_variety`	(string)	a unique name to ID the fiber in the JSON or Excel template topology input file
`dispersion`	(number)	(s.m-1.m-1)
`dispersion_slope`	(number)	(s.m-1.m-1.m-1)
`gamma`	(number)	2pi.n2/(lambda*Aeff) (w-1.m-1)
`pmd_coef`	(number)	Polarization mode dispersion (PMD) coefficient. (s.sqrt(m)-1)

2.1.3 Transceiver

The transceiver equipment library is a list of supported transceivers. New transceivers can be added and existing ones removed at will by the user. It is used to determine the service list path feasibility when running the path_request_run.py routine.

field	type	description
`type_variety`	(string)	A unique name to ID the transceiver in the JSON or Excel template topology input file
`frequency`	(number)	Min/max as below.
`mode`	(number)	A list of modes supported by the transponder. New modes can be added at will by the user. The modes are specific to each transponder type_variety. Each mode is described as below.

The modes are defined as follows:

field	type	description
`format`	(string)	a unique name to ID the mode
`baud_rate`	(number)	in Hz
`OSNR`	(number)	min required OSNR in 0.1nm (dB)
`bit_rate`	(number)	in bit/s
`roll_off`	(number)	Pure number between 0 and 1. TX signal roll-off shape. Used by Raman-aware simulation code.
`tx_osnr`	(number)	In dB. OSNR out from transponder.
`cost`	(number)	Arbitrary unit

2.1.4 Simulation parameters

Auto-design automatically creates EDFA amplifier network elements when they are missing, after a fiber, or between a ROADM and a fiber. This auto-design functionality can be manually and locally deactivated by introducing a `Fused` network element after a `Fiber` or a `Roadm` that doesn't need amplification. The amplifier is chosen in the EDFA list of the equipment library based on gain, power, and NF criteria. Only the EDFA that are marked `'allowed_for_design': true` are considered.

For amplifiers defined in the topology JSON input but whose `gain = 0` (placeholder), auto-design will set its gain automatically: see `power_mode` in the `Spans` library to find out how the gain is calculated.

2.1.5 Span

Span configuration is not a list (which may change in later releases) and the user can only modify the value of existing parameters:

field	type	description
power_mode	(boolean)	If false, gain mode. Auto-design sets amplifier gain = preceding span loss, unless the amplifier exists and its gain > 0 in the topology input JSON. If true, power mode (recommended for auto-design and power sweep.) Auto-design sets amplifier power according to delta_power_range. If the amplifier exists with gain > 0 in the topology JSON input, then its gain is translated into a power target/channel. Moreover, when performing a power sweep (see power_range_db in the SI configuration library) the power sweep is performed w/r/t this power target, regardless of preceding amplifiers power saturation/limitations.
delta_power_range_db	(number)	Auto-design only, power-mode only. Specifies the [min, max, step] power excursion/span. It is a relative power excursion w/r/t the power_dbm + power_range_db (power sweep if applicable) defined in the SI configuration library. This relative power excursion is = 1/3 of the span loss difference with the reference 20 dB span. The 1/3 slope is derived from the GN model equations. For example, a 23 dB span loss will be set to 1 dB more power than a 20 dB span loss. The 20 dB reference spans will *always* be set to power = power_dbm + power_range_db. To configure the same power in all spans, use *[0, 0, 0]*. All spans will be set to power = power_dbm + power_range_db. To configure the same power in all spans and 3 dB more power just for the longest spans: *[0, 3, 3]*. The longest spans are set to power = power_dbm + power_range_db + 3. To configure a 4 dB power range across all spans in 0.5 dB steps: *[-2, 2, 0.5]*. A 17 dB span is set to power = power_dbm + power_range_db - 1, a 20 dB span to power = power_dbm + power_range_db and a 23 dB span to power = power_dbm + power_range_db + 1
max_fiber_lineic_loss_for_raman	(number)	Maximum linear fiber loss for Raman amplification use.
max_length	(number)	Split fiber lengths > max_length. Interest to support high level topologies that do not specify in line amplification sites. For example the CORONET_Global_Topology.xlsx defines links > 1000km between 2 sites: it couldn't be simulated if these links were not split in shorter span lengths.
length_units	"km"	Unit for max_length.
max_loss	(number)	Not used in the current code implementation.
padding	(number)	In dB. Min span loss before putting an attenuator before fiber. Attenuator value Fiber.att_in = max(0, padding - span_loss). Padding can be set manually to reach a higher padding value for a given fiber by filling in the Fiber/params/att_in field in the topology json input [1] but if span_loss = length * loss_coef + att_in + con_in + con_out < padding, the specified att_in value will be completed to have span_loss = padding. Therefore it is not possible to set span_loss < padding.
EOL	(number)	All fiber span loss ageing. The value is added to the con_out (fiber output connector). So the design and the path feasibility are performed with span_loss + EOL. EOL cannot be set manually for a given fiber span (workaround is to specify higher con_out loss for this fiber).
con_in, con_out	(number)	Default values if Fiber/params/con_in/out is None in the topology input description. This default value is ignored if a Fiber/params/con_in/out value is input in the topology for a given Fiber.

```
{
    "uid": "fiber (A1->A2)",
    "type": "Fiber",
    "type_variety": "SSMF",
    "params":
    {
            "length": 120.0,
            "loss_coef": 0.2,
            "length_units": "km",
            "att_in": 0,
            "con_in": 0,
            "con_out": 0
    }
}
```

2.1.6 ROADM

The user can only modify the value of existing parameters:

field	type	description
`target_pch_out_db`	(number)	Auto-design sets the ROADM egress channel power. This reflects typical control loop algorithms that adjust ROADM losses to equalize channels (eg coming from different ingress direction or add ports) This is the default value Roadm/params/target_pch_out_db if no value is given in the `Roadm` element in the topology input description. This default value is ignored if a params/target_pch_out_db value is input in the topology for a given ROADM.
`add_drop_osnr`	(number)	OSNR contribution from the add/drop ports
`pmd`	(number)	Polarization mode dispersion (PMD). (s)
`restrictions`	**(dict of** strings)	If non-empty, keys `preamp_variety_list` and `booster_variety_list` represent list of `type_variety` amplifiers which are allowed for auto-design within ROADM's line degrees. If no booster should be placed on a degree, insert a `Fused` node on the degree output.

2.1.7 SpectralInformation

The user can only modify the value of existing parameters. It defines a spectrum of N identical carriers. While the code libraries allow for different carriers and power levels, the current user parametrization only allows one carrier type and one power/channel definition.

field	type	description
f_min, f_max	(number)	In Hz. Carrier min max excursion.
baud_rate	(number)	In Hz. Simulated baud rate.
spacing	(number)	In Hz. Carrier spacing.
roll_off	(number)	Pure number between 0 and 1. TX signal roll-off shape. Used by Raman-aware simulation code.
tx_osnr	(number)	In dB. OSNR out from transponder.
power_dbm	(number)	Reference channel power. In gain mode (see spans/power_mode = false), all gain settings are offset w/r/t this reference power. In power mode, it is the reference power for Spans/delta_power_range_db. For example, if delta_power_range_db = *[0,0,0]*, the same power=power_dbm is launched in every spans. The network design is performed with the power_dbm value: even if a power sweep is defined (see after) the design is not repeated.
power_range_db	(number)	Power sweep excursion around power_dbm. It is not the min and max channel power values! The reference power becomes: power_range_db + power_dbm.
sys_margins	(number)	In dB. Added margin on min required transceiver OSNR.

CHAPTER

THREE

EXCEL (XLS, XLSX) INPUT FILES

`gnpy-transmission-example` gives the possibility to use an excel input file instead of a json file. The program then will generate the corresponding json file for you.

The file named 'meshTopologyExampleV2.xls' is an example.

In order to work the excel file MUST contain at least 2 sheets:

- Nodes
- Links

(In progress) The File MAY contain an additional sheet:

- Eqt
- Service

3.1 Nodes sheet

Nodes sheet contains nine columns. Each line represents a 'node' (ROADM site or an in line amplifier site ILA or a Fused):

```
City (Mandatory) ; State ; Country ; Region ; Latitude ; Longitude ; Type
```

- **City** is used for the name of a node of the graph. It accepts letters, numbers,underscore,dash, blank… (not exhaustive). The user may want to avoid commas for future CSV exports.

  City name MUST be unique

- **Type** is not mandatory.
  - If not filled, it will be interpreted as an 'ILA' site if node degree is 2 and as a ROADM otherwise.
  - If filled, it can take "ROADM", "FUSED" or "ILA" values. If another string is used, it will be considered as not filled. FUSED means that ingress and egress spans will be fused together.
- *State*, *Country*, *Region* are not mandatory. "Region" is a holdover from the CORONET topology reference file CORONET_Global_Topology.xlsx. CORONET separates its network into geographical regions (Europe, Asia, Continental US.) This information is not used by gnpy.
- *Longitude*, *Latitude* are not mandatory. If filled they should contain numbers.
- **Booster_restriction** and **Preamp_restriction** are not mandatory. If used, they must contain one or several amplifier type_variety names separated by ' | '. This information is used to restrict types of amplifiers used in a ROADM node during autodesign. If a ROADM booster or preamp is already specified in the Eqpt sheet , the field is ignored. The field is also ignored if the node is not a ROADM node.

There MUST NOT be empty line(s) between two nodes lines

3.2 Links sheet

Links sheet must contain sixteen columns:

```
                <--           east cable from a to z                                  
↪ --> <--                   west from z to                                  -->
NodeA ; NodeZ ; Distance km ; Fiber type ; Lineic att ; Con_in ; Con_out ; PMD ;
↪Cable Id ; Distance km ; Fiber type ; Lineic att ; Con_in ; Con_out ; PMD ; Cable Id
```

Links sheets MUST contain all links between nodes defined in Nodes sheet. Each line represents a 'bidir link' between two nodes. The two directions are represented on a single line with "east cable from a to z" fields and "west from z to a" fields. Values for 'a to z' may be different from values from 'z to a'. Since both direction of a bidir 'a-z' link are described on the same line (east and west), 'z to a' direction MUST NOT be repeated in a different line. If repeated, it will generate another parrallel bidir link between the same end nodes.

Parameters for "east cable from a to z" and "west from z to a" are detailed in 2x7 columns. If not filled, "west from z to a" is copied from "east cable from a to z".

For example, a line filled with:

```
node6 ; node3 ; 80 ; SSMF ; 0.2 ; 0.5 ; 0.5 ; 0.1 ; cableB ;  ;  ; 0.21 ; 0.2 ;  ;  ;
```

will generate a unidir fiber span from node6 to node3 with:

```
[node6 node3 80 SSMF 0.2 0.5 0.5 0.1 cableB]
```

and a fiber span from node3 to node6:

```
[node6 node3 80 SSMF 0.21 0.2 0.5 0.1 cableB] attributes.
```

- **NodeA** and **NodeZ** are Mandatory. They are the two endpoints of the link. They MUST contain a node name from the **City** names listed in Nodes sheet.
- **Distance km** is not mandatory. It is the link length.
  - If filled it MUST contain numbers. If empty it is replaced by a default "80" km value.
  - If value is below 150 km, it is considered as a single (bidirectional) fiber span.
  - If value is over 150 km the *gnpy-transmission-example`* program will automatically suppose that intermediate span description are required and will generate fiber spans elements with "_1","_2", ... trailing strings which are not visible in the json output. The reason for the splitting is that current edfa usually do not support large span loss. The current assumption is that links larger than 150km will require intermediate amplification. This value will be revisited when Raman amplification is added"
- **Fiber type** is not mandatory.

  If filled it must contain types listed in eqpt_config.json in "Fiber" list "type_variety". If not filled it takes "SSMF" as default value.
- **Lineic att** is not mandatory.

  It is the lineic attenuation expressed in dB/km. If filled it must contain positive numbers. If not filled it takes "0.2" dB/km value
- *Con_in*, *Con_out* are not mandatory.

They are the connector loss in dB at ingress and egress of the fiber spans. If filled they must contain positive numbers. If not filled they take "0.5" dB default value.

- *PMD* is not mandatory and and is not used yet.

  It is the PMD value of the link in ps. If filled they must contain positive numbers. If not filled, it takes "0.1" ps value.

- *Cable Id* is not mandatory. If filled they must contain strings with the same constraint as "City" names. Its value is used to differenate links having the same end points. In this case different Id should be used. Cable Ids are not meant to be unique in general.

(in progress)

3.3 Eqpt sheet

Eqt sheet is optional. It lists the amplifiers types and characteristics on each degree of the *Node A* line. Eqpt sheet must contain twelve columns:

```
                <--           east cable from a to z                  --> <--
↪west from z to a                              -->
Node A ; Node Z ; amp type ; att_in ; amp gain ; tilt ; att_out ; delta_p ; amp type ;
↪ att_in ; amp gain ; tilt ; att_out ; delta_p
```

If the sheet is present, it MUST have as many lines as egress directions of ROADMs defined in Links Sheet.

For example, consider the following list of links (A,B and C being a ROADM and amp# ILAs)

```
A    - amp1
amp1 - amp2
Amp2 - B
A    - amp3
amp3 - C
```

then Eqpt sheet should contain:

- one line for each ILAs: amp1, amp2, amp3
- one line for each degree 1 ROADMs B and C
- two lines for ROADM A which is a degree 2 ROADM

```
A    - amp1
amp1 - amp2
Amp2 - B
A    - amp3
amp3 - C
B    - amp2
C    - amp3
```

In case you already have filled Nodes and Links sheets create_eqpt_sheet.py can be used to automatically create a template for the mandatory entries of the list.

```
$ cd $(gnpy-example-data)
$ python create_eqpt_sheet.py meshTopologyExampleV2.xls
```

This generates a text file meshTopologyExampleV2_eqt_sheet.txt whose content can be directly copied into the Eqt sheet of the excel file. The user then can fill the values in the rest of the columns.

- **Node A** is mandatory. It is the name of the node (as listed in Nodes sheet). If Node A is a 'ROADM' (Type attribute in sheet Node), its number of occurence must be equal to its degree. If Node A is an 'ILA' it should appear only once.
- **Node Z** is mandatory. It is the egress direction from the *Node A* site. Multiple Links between the same Node A and NodeZ is not supported.
- **amp type** is not mandatory. If filled it must contain types listed in eqpt_config.json in "Edfa" list "type_variety". If not filled it takes "std_medium_gain" as default value. If filled with fused, a fused element with 0.0 dB loss will be placed instead of an amplifier. This might be used to avoid booster amplifier on a ROADM direction.
- **amp_gain** is not mandatory. It is the value to be set on the amplifier (in dB). If not filled, it will be determined with design rules in the convert.py file. If filled, it must contain positive numbers.
- *att_in* and *att_out* are not mandatory and are not used yet. They are the value of the attenuator at input and output of amplifier (in dB). If filled they must contain positive numbers.
- *tilt* –TODO–
- **delta_p**, in dBm, is not mandatory. If filled it is used to set the output target power per channel at the output of the amplifier, if power_mode is True. The output power is then set to power_dbm + delta_power.

to be completed

(in progress)

3.4 Service sheet

Service sheet is optional. It lists the services for which path and feasibility must be computed with `gnpy-path_request`.

Service sheet must contain 11 columns:

```
route id ; Source ; Destination ; TRX type ; Mode ; System: spacing ; System: input␣
↪power (dBm) ; System: nb of channels ;  routing: disjoint from ; routing: path ;␣
↪routing: is loose?
```

- **route id** is mandatory. It must be unique. It is the identifier of the request. It can be an integer or a string (do not use blank or dash or coma)
- **Source** is mandatory. It is the name of the source node (as listed in Nodes sheet). Source MUST be a ROADM node. (TODO: relax this and accept trx entries)
- **Destination** is mandatory. It is the name of the destination node (as listed in Nodes sheet). Source MUST be a ROADM node. (TODO: relax this and accept trx entries)
- **TRX type** is mandatory. They are the variety type and selected mode of the transceiver to be used for the propagation simulation. These modes MUST be defined in the equipment library. The format of the mode is used as the name of the mode. (TODO: maybe add another mode id on Transceiver library ?). In particular the mode selection defines the channel baudrate to be used for the propagation simulation.
- **mode** is optional. If not specified, the program will search for the mode of the defined transponder with the highest baudrate fitting within the spacing value.
- **System: spacing** is mandatory. Spacing is the channel spacing defined in GHz difined for the feasibility propagation simulation, assuming system full load.
- **System: input power (dBm) ; System: nb of channels** are optional input defining the system parameters for the propagation simulation.
  - input power is the channel optical input power in dBm

  - nb of channels is the number of channels to be used for the simulation.

- **routing: disjoint from ; routing: path ; routing: is loose?** are optional.
  - disjoint from: identifies the requests from which this request must be disjoint. If filled it must contain request ids separated by ' | '
  - path: is the set of ROADM nodes that must be used by the path. It must contain the list of ROADM names that the path must cross. TODO : only ROADM nodes are accepted in this release. Relax this with any type of nodes. If filled it must contain ROADM ids separated by ' | '. Exact names are required.
  - is loose? 'no' value means that the list of nodes should be strictly followed, while any other value means that the constraint may be relaxed if the node is not reachable.
- **path bandwidth** is mandatory. It is the amount of capacity required between source and destination in Gbit/s. Value should be positive (non zero). It is used to compute the amount of required spectrum for the service.

CHAPTER

FOUR

PHYSICAL MODEL USED IN GNPY

4.1 QoT-E including ASE noise and NLI accumulation

The operations of PSE simulative framework are based on the capability to estimate the QoT of one or more channels operating lightpaths over a given network route. For backbone transport networks, we can suppose that transceivers are operating polarization-division-multiplexed multilevel modulation formats with DSP-based coherent receivers, including equalization. For the optical links, we focus on state-of-the-art amplified and uncompensated fiber links, connecting network nodes including ROADMs, where add and drop operations on data traffic are performed. In such a transmission scenario, it is well accepted [VRS+16][BSR+12][CCB+05][ME06][SF11][JK04][DFMS04][SB11][SFP12][PBC+02][DFMS16][PCC+06][Sav05][BBS13][JA01] to assume that transmission performances are limited by the amplified spontaneous emission (ASE) noise generated by optical amplifiers and and by nonlinear propagation effects: accumulation of a Gaussian disturbance defined as nonlinear interference (NLI) and generation of phase noise. State-of-the-art DSP in commercial transceivers are typically able to compensate for most of the phase noise through carrier-phase estimator (CPE) algorithms, for modulation formats with cardinality up to 16, per polarization state [PJ01][SLEF+15][FME+16]. So, for backbone networks covering medium-to-wide geographical areas, we can suppose that propagation is limited by the accumulation of two Gaussian disturbances: the ASE noise and the NLI. Additional impairments such as filtering effects introduced by ROADMs can be considered as additional equivalent power penalties depending on the ratio between the channel bandwidth and the ROADMs filters and the number of traversed ROADMs (hops) of the route under analysis. Modeling the two major sources of impairments as Gaussian disturbances, and being the receivers *coherent*, the unique QoT parameter determining the bit error rate (BER) for the considered transmission scenario is the generalized signal-to-noise ratio (SNR) defined as

$$\mathrm{SNR} = L_F \frac{P_{\mathrm{ch}}}{P_{\mathrm{ASE}} + P_{\mathrm{NLI}}} = L_F \left(\frac{1}{\mathrm{SNR}_{\mathrm{LIN}}} + \frac{1}{\mathrm{SNR}_{\mathrm{NL}}} \right)^{-1}$$

where P_{ch} is the channel power, P_{ASE} and P_{NLI} are the power levels of the disturbances in the channel bandwidth for ASE noise and NLI, respectively. L_F is a parameter assuming values smaller or equal than one that summarizes the equivalent power penalty loss such as filtering effects. Note that for state-of-the art equipment, filtering effects can be typically neglected over routes with few hops [RNR+01][FCBS06].

To properly estimate P_{ch} and P_{ASE} the transmitted power at the beginning of the considered route must be known, and losses and amplifiers gain and noise figure, including their variation with frequency, must be characterized. So, the evaluation of $\mathrm{SNR}_{\mathrm{LIN}}$ *just* requires an accurate knowledge of equipment, which is not a trivial aspect, but it is not related to physical-model issues. For the evaluation of the NLI, several models have been proposed and validated in the technical literature [VRS+16][BSR+12][CCB+05][ME06][SF11][JK04][DFMS04][SB11][SFP12][PBC+02][DFMS16][PCC+06][Sav05][BBS13][JA01]. The decision about which model to test within the PSE activities was driven by requirements of the entire PSE framework:

i. the model must be *local*, i.e., related individually to each network element (i.e. fiber span) generating NLI, independently of preceding and subsequent elements; and ii. the related computational time must be compatible with interactive operations.

So, the choice fell on the Gaussian Noise (GN) model with incoherent accumulation of NLI over fiber spans [PBC+02]. We implemented both the exact GN-model evaluation of NLI based on a double integral (Eq. (11) of [PBC+02]) and its analytical approximation (Eq. (120-121) of [PCC+06]). We performed several validation analyses comparing results of the two implementations with split-step simulations over wide bandwidths [PCCC07], and results clearly showed that for fiber types with chromatic dispersion roughly larger than 4 ps/nm/km, the analytical approximation ensures an excellent accuracy with a computational time compatible with real-time operations.

4.2 The Gaussian Noise Model to evaluate the NLI

As previously stated, fiber propagation of multilevel modulation formats relying on the polarization-division-multiplexing generates impairments that can be summarized as a disturbance called nonlinear interference (NLI), when exploiting a DSP-based coherent receiver, as in all state-of-the-art equipment. From a practical point of view, the NLI can be modeled as an additive Gaussian random process added by each fiber span, and whose strength depends on the cube of the input power spectral density and on the fiber-span parameters.

Since the introduction in the market in 2007 of the first transponder based on such a transmission technique, the scientific community has intensively worked to define the propagation behavior of such a trasnmission technique. First, the role of in-line chromatic dispersion compensation has been investigated, deducing that besides being not essential, it is indeed detrimental for performances [CPCF09]. Then, it has been observed that the fiber propagation impairments are practically summarized by the sole NLI, being all the other phenomena compensated for by the blind equalizer implemented in the receiver DSP [CBC+09]. Once these assessments have been accepted by the community, several prestigious research groups have started to work on deriving analytical models able to estimating the NLI accumulation, and consequentially the generalized SNR that sets the BER, according to the transponder BER vs. SNR performance. Many models delivering different levels of accuracy have been developed and validated. As previously clarified, for the purposes of the PSE framework, the GN-model with incoherent accumulation of NLI over fiber spans has been selected as adequate. The reason for such a choice is first such a model being a “local” model, so related to each fiber spans, independently of the preceding and succeeding network elements. The other model characteristic driving the choice is the availability of a closed form for the model, so permitting a real-time evaluation, as required by the PSE framework. For a detailed derivation of the model, please refer to [PCC+06], while a qualitative description can be summarized as in the following. The GN-model assumes that the channel comb propagating in the fiber is well approximated by unpolarized spectrally shaped Gaussian noise. In such a scenario, supposing to rely - as in state-of-the-art equipment - on a receiver entirely compensating for linear propagation effects, propagation in the fiber only excites the four-wave mixing (FWM) process among the continuity of the tones occupying the bandwidth. Such a FWM generates an unpolarized complex Gaussian disturbance in each spectral slot that can be easily evaluated extending the FWM theory from a set of discrete tones - the standard FWM theory introduced back in the 90s by Inoue [Ino92]- to a continuity of tones, possibly spectrally shaped. Signals propagating in the fiber are not equivalent to Gaussian noise, but thanks to the absence of in-line compensation for choromatic dispersion, the become so, over short distances. So, the Gaussian noise model with incoherent accumulation of NLI has estensively proved to be a quick yet accurate and conservative tool to estimate propagation impairments of fiber propagation. Note that the GN-model has not been derived with the aim of an *exact* performance estimation, but to pursue a conservative performance prediction. So, considering these characteristics, and the fact that the NLI is always a secondary effect with respect to the ASE noise accumulation, and - most importantly - that typically linear propagation parameters (losses, gains and noise figures) are known within a variation range, a QoT estimator based on the GN model is adequate to deliver performance predictions in terms of a reasonable SNR range, rather than an exact value. As final remark, it must be clarified that the GN-model is adequate to be used when relying on a relatively narrow bandwidth up to few THz. When exceeding such a bandwidth occupation, the GN-model must be generalized introducing the interaction with the Stimulated Raman Scattering in order to give a proper estimation for all channels [CAC18]. This will be the main upgrade required within the PSE framework.

CHAPTER

FIVE

API REFERENCE DOCUMENTATION

5.1 gnpy package

GNPy is an open-source, community-developed library for building route planning and optimization tools in real-world mesh optical networks. It is based on the Gaussian Noise Model.

Signal propagation is implemented in *core*. Path finding and spectrum assignment is in *topology*. Various tools and auxiliary code, including the JSON I/O handling, is in *tools*.

5.1.1 gnpy.core

Simulation of signal propagation in the DWDM network

Optical signals, as defined via *info.SpectralInformation*, enter *elements* which compute how these signals are affected as they travel through the *network*. The simulation is controlled via *parameters* and implemented mainly via *science_utils*.

5.1.1.1 gnpy.core.ansi_escapes

A random subset of ANSI terminal escape codes for colored messages

5.1.1.2 gnpy.core.elements

Standard network elements which propagate optical spectrum

A network element is a Python callable. It takes a *info.SpectralInformation* object and returns a copy with appropriate fields affected. This structure represents spectral information that is "propogated" by this network element. Network elements must have only a local "view" of the network and propogate *info.SpectralInformation* using only this information. They should be independent and self-contained.

Network elements MUST implement two attributes `uid` and `name` representing a unique identifier and a printable name, and provide the `__call__()` method taking a `SpectralInformation` as an input and returning another `SpectralInformation` instance as a result.

class gnpy.core.elements.**Edfa**(**args*, *params=None*, *operational=None*, ***kwargs*)

Bases: *gnpy.core.elements._Node*

_calc_nf(*avg=False*)

nf calculation based on 2 models: self.params.nf_model.enabled from json import: True => 2 stages amp modelling based on precalculated nf1, nf2 and delta_p in build_OA_json False => polynomial fit based on self.params.nf_fit_coeff

_gain_profile(*pin*, *err_tolerance=1e-11*, *simple_opt=True*)
Pin : input power / channel in W

Parameters

- **gain_ripple** (*numpy.ndarray*) – design flat gain
- **dgt** (*numpy.ndarray*) – design gain tilt
- **Pin** (*numpy.ndarray*) – total input power in W
- **gp** (*float*) – Average gain setpoint in dB units (provisioned gain)
- **gtp** (*float*) – gain tilt setting (provisioned tilt)

Returns gain profile in dBm, per channel or spectral slice

Return type numpy.ndarray

Checking of output power clamping is implemented in interpol_params().

Based on:

R. di Muro, "The Er3+ fiber gain coefficient derived from a dynamic gain tilt technique", Journal of Lightwave Technology, Vol. 18, Iss. 3, Pp. 343-347, 2000.

Ported from Matlab version written by David Boerges at Ciena.

_nf(*type_def*, *nf_model*, *nf_fit_coeff*, *gain_min*, *gain_flatmax*, *gain_target*)

carriers(*loc*, *attr*)
retrieve carriers information

Parameters

- **loc** – (in, out) of the class element
- **attr** – (ase, nli, signal, total) power information

interpol_params(*frequencies*, *pin*, *baud_rates*, *pref*)
interpolate SI channel frequencies with the edfa dgt and gain_ripple frquencies from JSON

noise_profile(*bw) computes amplifier ASE (W) in signal bandwidth (Hz*)
Noise is calculated at amplifier input

Bw signal bandwidth = baud rate in Hz

Returns the asepower in W in the signal bandwidth bw for 96 channels

Return type numpy array of float

ASE power using per channel gain profile inputs:

NF_dB - Noise figure in dB, vector of length number of channels or spectral slices

G_dB - Actual gain calculated for the EDFA, vector of length number of channels or spectral slices

ffs - Center frequency grid of the channels or spectral slices in THz, vector of length number of channels or spectral slices

dF - width of each channel or spectral slice in THz, vector of length number of channels or spectral slices

OUTPUT:

ase_dBm - ase in dBm per channel or spectral slice

NOTE:

The output is the total ASE in the channel or spectral slice. For 50GHz channels the ASE BW is effectively 0.4nm. To get to noise power in 0.1nm, subtract 6dB.

ONSR is usually quoted as channel power divided by the ASE power in 0.1nm RBW, regardless of the width of the actual channel. This is a historical convention from the days when optical signals were much smaller (155Mbps, 2.5Gbps, ... 10Gbps) than the resolution of the OSAs that were used to measure spectral power which were set to 0.1nm resolution for convenience. Moving forward into flexible grid and high baud rate signals, it may be convenient to begin quoting power spectral density in the same BW for both signal and ASE, e.g. 12.5GHz.

propagate(*pref*, **carriers*)
add ASE noise to the propagating carriers of *info.SpectralInformation*

property to_json

update_pref(*pref*)

class gnpy.core.elements.**EdfaOperational**(***operational*)
Bases: object

default_values = {'delta_p': None, 'gain_target': None, 'out_voa': None, 'tilt_targ

update_attr(*kwargs*)

class gnpy.core.elements.**EdfaParams**(***params*)
Bases: object

update_params(*kwargs*)

class gnpy.core.elements.**Fiber**(**args*, *params=None*, ***kwargs*)
Bases: *gnpy.core.elements._Node*

_gn_analytic(*carrier*, **carriers*)
Computes the nonlinear interference power on a single carrier. The method uses eq. 120 from arXiv:1209.0394.

Parameters

- **carrier** – the signal under analysis
- **carriers** – the full WDM comb

Returns carrier_nli: the amount of nonlinear interference in W on the under analysis

alpha(*frequencies*)
It returns the values of the series expansion of attenuation coefficient alpha(f) for all f in frequencies

Parameters **frequencies** – frequencies of series expansion [Hz]

Returns alpha: power attenuation coefficient for f in frequencies [Neper/m]

alpha0(*f_ref=193500000000000.0*)
It returns the zero element of the series expansion of attenuation coefficient alpha(f) in the reference frequency f_ref

Parameters **f_ref** – reference frequency of series expansion [Hz]

Returns alpha0: power attenuation coefficient in f_ref [Neper/m]

carriers(*loc*, *attr*)
retrieve carriers information

Parameters

- **loc** – (in, out) of the class element

- **attr** – (ase, nli, signal, total) power information

chromatic_dispersion(*freq=193500000000000.0*)
Returns accumulated chromatic dispersion (CD).

Parameters **freq** – the frequency at which the chromatic dispersion is computed

Returns chromatic dispersion: the accumulated dispersion [s/m]

property fiber_loss
Fiber loss in dB, not including padding attenuator

property loss
total loss including padding att_in: useful for polymorphism with roadm loss

property passive

property pmd
differential group delay (PMD) [s]

propagate(**carriers*)
Generator that computes the fiber propagation: attenuation, non-linear interference generation, CD accumulation and PMD accumulation.

Param *carriers: the channels at the input of the fiber

Yield carrier: the next channel at the output of the fiber

property to_json

update_pref(*pref*)

class gnpy.core.elements.**Fused**(**args*, *params=None*, ***kwargs*)
Bases: *gnpy.core.elements._Node*

propagate(**carriers*)

property to_json

update_pref(*pref*)

class gnpy.core.elements.**FusedParams**(*loss*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('loss',)

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new FusedParams object from a sequence or iterable

_replace(***kwds*)
Return a new FusedParams object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property loss
Alias for field number 0

class gnpy.core.elements.**Location**
Bases: *gnpy.core.elements.Location*

class gnpy.core.elements.**RamanFiber**(**args*, *params=None*, ***kwargs*)
Bases: *gnpy.core.elements.Fiber*

propagate(**carriers*)
Generator that computes the fiber propagation: attenuation, non-linear interference generation, CD accumulation and PMD accumulation.

Param *carriers: the channels at the input of the fiber

Yield carrier: the next channel at the output of the fiber

update_pref(*pref*, **carriers*)

class gnpy.core.elements.**Roadm**(**args*, *params*, ***kwargs*)
Bases: *gnpy.core.elements._Node*

propagate(*pref*, **carriers*)

property to_json

update_pref(*pref*)

class gnpy.core.elements.**RoadmParams**(*target_pch_out_db*, *add_drop_osnr*, *pmd*, *restrictions*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('target_pch_out_db', 'add_drop_osnr', 'pmd', 'restrictions')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new RoadmParams object from a sequence or iterable

_replace(***kwds*)
Return a new RoadmParams object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property add_drop_osnr
Alias for field number 1

property pmd
Alias for field number 2

property restrictions
Alias for field number 3

property target_pch_out_db
Alias for field number 0

class gnpy.core.elements.**Transceiver**(**args*, ***kwargs*)
Bases: *gnpy.core.elements._Node*

_calc_cd(*spectral_info*)
Updates the Transceiver property with the CD of the received channels. CD in ps/nm.

_calc_pmd(*spectral_info*)
Updates the Transceiver property with the PMD of the received channels. PMD in ps.

_calc_snr(*spectral_info*)

property to_json

update_snr(**args*)
snr_added in 0.1nm compute SNR penalties such as transponder Tx_osnr or Roadm add_drop_osnr only applied in request.py / propagate on the last Trasceiver node of the path all penalties are added in a single call because to avoid uncontrolled cumul

class gnpy.core.elements.**_Node**(*uid*, *name=None*, *params=None*, *metadata=None*, *operational=None*, *type_variety=None*)

Bases: object

Convenience class for providing common functionality of all network elements

This class is just an internal implementation detail; do **not** assume that all network elements inherit from *_Node*.

property coords

property lat

property latitude

property lng

property loc

property location

property longitude

5.1.1.3 gnpy.core.equipment

This module contains functionality for specifying equipment.

gnpy.core.equipment.**trx_mode_params**(*equipment*, *trx_type_variety=''*, *trx_mode=''*, *error_message=False*)

return the trx and SI parameters from eqpt_config for a given type_variety and mode (ie format)

5.1.1.4 gnpy.core.exceptions

Exceptions thrown by other gnpy modules

exception gnpy.core.exceptions.**ConfigurationError**

Bases: Exception

User-provided configuration contains an error

exception gnpy.core.exceptions.**DisjunctionError**

Bases: *gnpy.core.exceptions.ServiceError*

Disjunction of user-provided request can not be satisfied

exception gnpy.core.exceptions.**EquipmentConfigError**

Bases: *gnpy.core.exceptions.ConfigurationError*

Incomplete or wrong configuration within the equipment library

exception gnpy.core.exceptions.**NetworkTopologyError**

Bases: *gnpy.core.exceptions.ConfigurationError*

Topology of user-provided network is wrong

exception gnpy.core.exceptions.**ParametersError**

Bases: *gnpy.core.exceptions.ConfigurationError*

Incomplete or wrong configurations within parameters json

exception gnpy.core.exceptions.**ServiceError**

Bases: Exception

Service of user-provided request is wrong

exception gnpy.core.exceptions.**SpectrumError**
Bases: Exception

Spectrum errors of the program

5.1.1.5 gnpy.core.info

This module contains classes for modelling *SpectralInformation*.

class gnpy.core.info.**Channel**
Bases: *gnpy.core.info.Channel*

Class containing the parameters of a WDM signal.

Parameters

- **channel_number** – channel number in the WDM grid
- **frequency** – central frequency of the signal (Hz)
- **baud_rate** – the symbol rate of the signal (Baud)
- **roll_off** – the roll off of the signal. It is a pure number between 0 and 1
- **(gnpy.core.info.Power)** (*power*) – power of signal, ASE noise and NLI (W)
- **chromatic_dispersion** – chromatic dispersion (s/m)
- **pmd** – polarization mode dispersion (s)

class gnpy.core.info.**Power**
Bases: *gnpy.core.info.Power*

carriers power in W

class gnpy.core.info.**Pref**
Bases: *gnpy.core.info.Pref*

noiseless reference power in dBm: p_span0: inital target carrier power p_spani: carrier power after element i neq_ch: equivalent channel count in dB

class gnpy.core.info.**SpectralInformation**
Bases: *gnpy.core.info.SpectralInformation*

gnpy.core.info.**create_input_spectral_information**(*f_min*, *f_max*, *roll_off*, *baud_rate*, *power*, *spacing*)

5.1.1.6 gnpy.core.network

Working with networks which consist of network elements

gnpy.core.network.**add_connector_loss**(*network*, *fibers*, *default_con_in*, *default_con_out*, *EOL*)

gnpy.core.network.**add_egress_amplifier**(*network*, *node*)

gnpy.core.network.**add_fiber_padding**(*network*, *fibers*, *padding*)
last_fibers = (fiber for n in network.nodes() if not (isinstance(n, elements.Fiber) or isinstance(n, elements.Fused)) for fiber in network.predecessors(n) if isinstance(fiber, elements.Fiber))

gnpy.core.network.**build_network**(*network*, *equipment*, *pref_ch_db*, *pref_total_db*)

gnpy.core.network.**calculate_new_length**(*fiber_length*, *bounds*, *target_length*)

gnpy.core.network.**edfa_nf**(*gain_target*, *variety_type*, *equipment*)

gnpy.core.network.**find_first_node**(*network*, *node*)
Fused node interest: returns the 1st node at the origin of a succession of fused nodes (aka no amp in between)

gnpy.core.network.**find_last_node**(*network*, *node*)
Fused node interest: returns the last node in a succession of fused nodes (aka no amp in between)

gnpy.core.network.**next_node_generator**(*network*, *node*)
fused spans interest: iterate over all successors while they are Fused or Fiber type

gnpy.core.network.**prev_node_generator**(*network*, *node*)
fused spans interest: iterate over all predecessors while they are Fused or Fiber type

gnpy.core.network.**select_edfa**(*raman_allowed*, *gain_target*, *power_target*, *equipment*, *uid*, *restrictions=None*)
amplifer selection algorithm @Orange Jean-Luc Augé

gnpy.core.network.**set_amplifier_voa**(*amp*, *power_target*, *power_mode*)

gnpy.core.network.**set_egress_amplifier**(*network*, *roadm*, *equipment*, *pref_total_db*)

gnpy.core.network.**span_loss**(*network*, *node*)
Fused span interest: return the total span loss of all the fibers spliced by a Fused node

gnpy.core.network.**split_fiber**(*network*, *fiber*, *bounds*, *target_length*, *equipment*)

gnpy.core.network.**target_power**(*network*, *node*, *equipment*)

5.1.1.7 gnpy.core.parameters

This module contains all parameters to configure standard network elements.

class gnpy.core.parameters.**FiberParams**(***kwargs*)
Bases: *gnpy.core.parameters.Parameters*

asdict()

property asymptotic_length

property att_in

property beta2

property beta3

property con_in

property con_out

property dispersion

property dispersion_slope

property effective_length

property f_loss_ref

property gamma

property length

property lin_attenuation

property lin_loss_exp

property loss_coef

property pmd_coef

property pumps_loss_coef

property raman_efficiency

property ref_frequency

property ref_wavelength

class gnpy.core.parameters.**NLIParams**(***kwargs*)
Bases: *gnpy.core.parameters.Parameters*

property computed_channels

property dispersion_tolerance

property f_cut_resolution

property f_pump_resolution

property nli_method_name

property phase_shift_tolerance

property wdm_grid_size

class gnpy.core.parameters.**Parameters**
Bases: object

asdict()

class gnpy.core.parameters.**PumpParams**(*power*, *frequency*, *propagation_direction*)
Bases: *gnpy.core.parameters.Parameters*

property frequency

property power

property propagation_direction

class gnpy.core.parameters.**RamanParams**(***kwargs*)
Bases: *gnpy.core.parameters.Parameters*

property flag_raman

property space_resolution

property tolerance

class gnpy.core.parameters.**SimParams**(***kwargs*)
Bases: *gnpy.core.parameters.Parameters*

property nli_params

property raman_params

class gnpy.core.science_utils.**NliSolver**(*fiber=None*)
Bases: object

This class implements the NLI models. Model and method can be specified in *sim_params.nli_params.method*. List of implemented methods: 'gn_model_analytic': brute force triple integral solution 'ggn_spectrally_separated_xpm_spm': XPM plus SPM

static _carrier_nli_from_eta_matrix(*eta_matrix*, *carrier*, **carriers*)

_compute_eta_matrix(*carrier_cut*, **carriers*)

_fast_generalized_psi(*carrier_cut*, *pump_carrier*, *f_eval*, *f_cut_resolution*)
It computes the generalized psi function similarly to the one used in the GN model :return: generalized_psi

_frequency_offset_threshold(*symbol_rate*)

_generalized_psi(*carrier_cut*, *pump_carrier*, *f_eval*, *f_cut_resolution*, *f_pump_resolution*)
It computes the generalized psi function similarly to the one used in the GN model :return: generalized_psi

static _generalized_rho_nli(*delta_beta*, *rho_norm_pump*, *z*, *alpha0*)

_generalized_spectrally_separated_spm(*carrier*)

_generalized_spectrally_separated_xpm(*carrier_cut*, *pump_carrier*)

_gn_analytic(*carrier*, **carriers*)
Computes the nonlinear interference power on a single carrier. The method uses eq. 120 from arXiv:1209.0394. :param carrier: the signal under analysis :param carriers: the full WDM comb :return: carrier_nli: the amount of nonlinear interference in W on the carrier under analysis

compute_nli(*carrier*, **carriers*)
Compute NLI power generated by the WDM comb **carriers* on the channel under test *carrier* at the end of the fiber span.

property fiber

property stimulated_raman_scattering

class gnpy.core.science_utils.**RamanSolver**(*fiber=None*)
Bases: object

static _compute_power_spectrum(*carriers*, *raman_pumps=None*)
Rearrangement of spectral and Raman pump information to make them compatible with Raman solver :param carriers: a tuple of namedtuples describing the transmitted channels :param raman_pumps: a namedtuple describing the Raman pumps :return:

_initial_guess_stimulated_raman(*z*, *power_spectrum*, *alphap_fiber*, *prop_direct*)
Computes the initial guess knowing the boundary conditions :param z: patial axis [m]. numpy array :param power_spectrum: power in each frequency slice [W]. Frequency axis is defined by freq_array. numpy array :param alphap_fiber: frequency dependent fiber attenuation of signal power [1/m]. Frequency defined by freq_array. numpy array :param prop_direct: indicates the propagation direction of each power slice in power_spectrum: +1 for forward propagation and -1 for backward propagation. Frequency defined by freq_array. numpy array :return: power_guess: guess on the initial conditions [W]. The first ndarray index identifies the frequency slice, the second ndarray index identifies the step in z. ndarray

_int_spontaneous_raman(*z_array*, *raman_matrix*, *alphap_fiber*, *freq_array*, *cr_raman_matrix*, *freq_diff*, *ase_bc*, *bn_array*, *temperature*)

_ode_stimulated_raman(*z*, *power_spectrum*, *alphap_fiber*, *freq_array*, *cr_raman_matrix*, *prop_direct*)
Aim of ode_raman is to implement the set of ordinary differential equations (ODEs) describing the Raman effect. :param z: spatial axis (unused). :param power_spectrum: power in each frequency slice [W]. Frequency axis is defined by freq_array. numpy array. Size n :param alphap_fiber: frequency dependent fiber attenuation of signal power [1/m]. Frequency defined by freq_array. numpy array. Size n :param freq_array: reference frequency axis [Hz]. numpy array. Size n :param cr_raman: Cr(f) Raman gain efficiency variation in frequency [1/W/m]. Frequency defined by freq_array. numpy ndarray. Size nxn :param prop_direct: indicates the propagation direction of each power slice in power_spectrum: +1 for forward propagation and -1 for backward propagation. Frequency defined by freq_array. numpy array. Size n :return: dP/dz: the power variation in dz [W/m]. numpy array. Size n

_residuals_stimulated_raman(*ya*, *yb*, *power_spectrum*, *prop_direct*)

calculate_spontaneous_raman_scattering(*carriers*, *raman_pumps*)

calculate_stimulated_raman_scattering(*carriers*, *raman_pumps*)
Returns stimulated Raman scattering solution including fiber gain/loss profile. :return: None

property carriers

property fiber

property raman_pumps

property spontaneous_raman_scattering

property stimulated_raman_scattering

class gnpy.core.science_utils.**Simulation**
Bases: object

_shared_dict = {}

classmethod get_simulation()

classmethod set_params(*sim_params*)

property sim_params

class gnpy.core.science_utils.**SpontaneousRamanScattering**(*frequency*, *z*, *power*)
Bases: object

class gnpy.core.science_utils.**StimulatedRamanScattering**(*frequency*, *z*, *rho*, *power*)
Bases: object

gnpy.core.science_utils.**_psi**(*carrier*, *interfering_carrier*, *beta2*, *asymptotic_length*)
Calculates eq. 123 from arXiv:1209.0394

gnpy.core.science_utils.**estimate_nf_model**(*type_variety*, *gain_min*, *gain_max*, *nf_min*, *nf_max*)

gnpy.core.science_utils.**frequency_resolution**(*carrier*, *carriers*, *sim_params*, *fiber*)

gnpy.core.science_utils.**propagate_raman_fiber**(*fiber*, **carriers*)

gnpy.core.science_utils.**raised_cosine_comb**(*f*, **carriers*)
Returns an array storing the PSD of a WDM comb of raised cosine shaped channels at the input frequencies defined in array f :param f: numpy array of frequencies in Hz :param carriers: namedtuple describing the WDM comb :return: PSD of the WDM comb evaluated over f

5.1.1.8 gnpy.core.utils

This module contains utility functions that are used with gnpy.

gnpy.core.utils.**arrange_frequencies**(*length*, *start*, *stop*)
Create an array of frequencies

Parameters

- **length** (*integer*) – number of elements
- **start** (*float*) – Start frequency in THz
- **stop** (*float*) – Stop frequency in THz

Returns an array of frequencies determined by the spacing parameter

Return type numpy.ndarray

gnpy.core.utils.**automatic_fmax**(*f_min*, *spacing*, *nch*)
Find the high-frequenecy boundary of a spectrum

:param f_min Start of the spectrum (lowest frequency edge) [Hz] :param spacing Grid/channel spacing [Hz] :param nch Number of channels :return End of the spectrum (highest frequency) [Hz]

```
>>> automatic_fmax(191.325e12, 50e9, 96)
196125000000000.0
```

gnpy.core.utils.**automatic_nch**(*f_min*, *f_max*, *spacing*)
How many channels are available in the spectrum

:param f_min Lowest frequenecy [Hz] :param f_max Highest frequency [Hz] :param spacing Channel width [Hz] :return Number of uniform channels

```
>>> automatic_nch(191.325e12, 196.125e12, 50e9)
96
>>> automatic_nch(193.475e12, 193.525e12, 50e9)
1
```

gnpy.core.utils.**convert_length**(*value*, *units*)
Convert length into basic SI units

```
>>> convert_length(1, 'km')
1000.0
>>> convert_length(2.0, 'km')
2000.0
>>> convert_length(123, 'm')
123.0
>>> convert_length(123.0, 'm')
123.0
>>> convert_length(42.1, 'km')
42100.0
>>> convert_length(666, 'yards')
Traceback (most recent call last):
    ...
gnpy.core.exceptions.ConfigurationError: Cannot convert length in "yards" into␣
↪meters
```

gnpy.core.utils.**db2lin**(*value*)
Convert logarithimic units to linear

```
>>> round(db2lin(10.0), 2)
10.0
>>> round(db2lin(20.0), 2)
100.0
>>> round(db2lin(1.0), 2)
1.26
>>> round(db2lin(0.0), 2)
1.0
>>> round(db2lin(-10.0), 2)
0.1
```

gnpy.core.utils.**deltaf2deltawl**(*delta_f*, *frequency*)

deltawl2deltaf(delta_f, frequency): converts delta frequency to delta wavelength units for delta_wl and wavelength must be same

Parameters

- **delta_f** (*float or numpy.ndarray*) – delta frequency in same units as frequency
- **frequency** (*float*) – frequency BW is relevant for

Returns The BW in wavelength units

Return type float or ndarray

gnpy.core.utils.**deltawl2deltaf**(*delta_wl*, *wavelength*)
deltawl2deltaf(delta_wl, wavelength): delta_wl is BW in wavelength units wavelength is the center wl units for delta_wl and wavelength must be same

Parameters

- **delta_wl** (*float or numpy.ndarray*) – delta wavelength BW in same units as wavelength
- **wavelength** (*float*) – wavelength BW is relevant for

Returns The BW in frequency units

Return type float or ndarray

gnpy.core.utils.**freq2wavelength**(*value*)
Converts frequency units to wavelength units.

```
>>> round(freq2wavelength(191.35e12) * 1e9, 3)
1566.723
>>> round(freq2wavelength(196.1e12) * 1e9, 3)
1528.773
```

gnpy.core.utils.**lin2db**(*value*)
Convert linear unit to logarithmic (dB)

```
>>> lin2db(0.001)
-30.0
>>> round(lin2db(1.0), 2)
0.0
>>> round(lin2db(1.26), 2)
1.0
>>> round(lin2db(10.0), 2)
10.0
>>> round(lin2db(100.0), 2)
20.0
```

gnpy.core.utils.**merge_amplifier_restrictions**(*dict1*, *dict2*)
Updates contents of dicts recursively

```
>>> d1 = {'params': {'restrictions': {'preamp_variety_list': [], 'booster_variety_
↪list': []}}}
>>> d2 = {'params': {'target_pch_out_db': -20}}
>>> merge_amplifier_restrictions(d1, d2)
{'params': {'restrictions': {'preamp_variety_list': [], 'booster_variety_list':␣
↪[]}, 'target_pch_out_db': -20}}
```

```
>>> d3 = {'params': {'restrictions': {'preamp_variety_list': ['foo'], 'booster_
↪variety_list': ['bar']}}}
>>> merge_amplifier_restrictions(d1, d3)
{'params': {'restrictions': {'preamp_variety_list': [], 'booster_variety_list':␣
↪[]}}}
```

gnpy.core.utils.**round2float**(*number*, *step*)

gnpy.core.utils.**rrc**(*ffs*, *baud_rate*, *alpha*)
rrc(ffs, baud_rate, alpha): computes the root-raised cosine filter function.

Parameters

- **ffs** (*numpy.ndarray*) – A numpy array of frequencies
- **baud_rate** (*float*) – The Baud Rate of the System
- **alpha** (*float*) – The roll-off factor of the filter

Returns hf a numpy array of the filter shape

Return type numpy.ndarray

gnpy.core.utils.**silent_remove**(*this_list*, *elem*)
Remove matching elements from a list without raising ValueError

```
>>> li = [0, 1]
>>> li = silent_remove(li, 1)
>>> li
[0]
>>> li = silent_remove(li, 1)
>>> li
[0]
```

gnpy.core.utils.**snr_sum**(*snr*, *bw*, *snr_added*, *bw_added=12500000000.0*)

gnpy.core.utils.**write_csv**(*obj*, *filename*)
Convert dictionary items to a CSV file the dictionary format:

```
{'result category 1':
                [
                # 1st line of results
                {'header 1' : value_xxx,
                 'header 2' : value_yyy},
                 # 2nd line of results: same headers, different results
                {'header 1' : value_www,
                 'header 2' : value_zzz}
                ],
'result_category 2':
                [
                {},{}
                ]
}
```

The generated csv file will be:

```
result_category 1
header 1    header 2
value_xxx   value_yyy
value_www   value_zzz
result_category 2
...
```

5.1.2 gnpy.topology

Tracking *request* for spectrum and their *spectrum_assignment*.

5.1.2.1 gnpy.topology.request

This module contains path request functionality.

This functionality allows the user to provide a JSON request file in accordance with a Yang model for requesting path computations and returns path results in terms of path and feasibility

See: draft-ietf-teas-yang-path-computation-01.txt

class gnpy.topology.request.**Disjunction**(**args*, ***params*)
Bases: object

the class that contains all attributes related to disjunction constraints

class gnpy.topology.request.**DisjunctionParams**(*disjunction_id*, *relaxable*, *link_diverse*, *node_diverse*, *disjunctions_req*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('disjunction_id', 'relaxable', 'link_diverse', 'node_diverse', 'disjunction

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new DisjunctionParams object from a sequence or iterable

_replace(***kwds*)
Return a new DisjunctionParams object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property disjunction_id
Alias for field number 0

property disjunctions_req
Alias for field number 4

property link_diverse
Alias for field number 2

property node_diverse
Alias for field number 3

property relaxable
Alias for field number 1

class gnpy.topology.request.**PathRequest**(**args*, ***params*)
Bases: object

the class that contains all attributes related to a request

class gnpy.topology.request.**RequestParams**(*request_id*, *source*, *destination*, *bidir*, *trx_type*, *trx_mode*, *nodes_list*, *loose_list*, *spacing*, *power*, *nb_channel*, *f_min*, *f_max*, *format*, *baud_rate*, *OSNR*, *bit_rate*, *roll_off*, *tx_osnr*, *min_spacing*, *cost*, *path_bandwidth*)
Bases: tuple

property OSNR
Alias for field number 15

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('request_id', 'source', 'destination', 'bidir', 'trx_type', 'trx_mode', 'no

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new RequestParams object from a sequence or iterable

_replace(***kwds*)
Return a new RequestParams object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property baud_rate
Alias for field number 14

property bidir
Alias for field number 3

property bit_rate
Alias for field number 16

property cost
Alias for field number 20

property destination
Alias for field number 2

property f_max
Alias for field number 12

property f_min
Alias for field number 11

property format
Alias for field number 13

property loose_list
Alias for field number 7

property min_spacing
Alias for field number 19

property nb_channel
Alias for field number 10

property nodes_list
Alias for field number 6

property path_bandwidth
Alias for field number 21

property power
Alias for field number 9

property request_id
Alias for field number 0

property roll_off
Alias for field number 17

property source
Alias for field number 1

property spacing
Alias for field number 8

property trx_mode
Alias for field number 5

property trx_type
Alias for field number 4

property tx_osnr
Alias for field number 18

class gnpy.topology.request.**ResultElement**(*path_request*, *computed_path*, *reversed_computed_path=None*)
Bases: object

property detailed_path_json
a function that builds path object for normal and blocking cases

property json

property path_properties
a function that returns the path properties (metrics, crossed elements) into a dict

property pathresult
create the result dictionnary (response for a request)

property uid

gnpy.topology.request.**compare_reqs**(*req1*, *req2*, *disjlist*)
compare two requests: returns True or False

gnpy.topology.request.**compute_constrained_path**(*network*, *req*)

gnpy.topology.request.**compute_path_dsjctn**(*network*, *equipment*, *pathreqlist*, *disjunctions_list*)

gnpy.topology.request.**compute_path_with_disjunction**(*network*, *equipment*, *pathreqlist*, *pathlist*)
use a list but a dictionnary might be helpful to find path based on request_id TODO change all these req, dsjct, res lists into dict !

gnpy.topology.request.**correct_json_route_list**(*network*, *pathreqlist*)
all names in list should be exact name in the network, and there is no ambiguity This function only checks that list is correct, warns user if the name is incorrect and suppresses the constraint it it is loose or raises an error if it is strict

gnpy.topology.request.**deduplicate_disjunctions**(*disjn*)
clean disjunctions to remove possible repetition

gnpy.topology.request.**find_reversed_path**(*pth*)
select of intermediate roadms and find the path between them note that this function may not give an exact result in case of multiple links between two adjacent nodes.

gnpy.topology.request.**isdisjoint**(*pth1*, *pth2*)
returns 0 if disjoint

gnpy.topology.request.**ispart**(*ptha*, *pthb*)
the functions takes two paths a and b and retrns True if all a elements are part of b and in the same order

gnpy.topology.request.**jsontocsv**(*json_data*, *equipment*, *fileout*)
reads json path result file in accordance with: Yang model for requesting Path Computation draft-ietf-teas-yang-path-computation-01.txt. and write results in an CSV file

gnpy.topology.request.**jsontoparams**(*my_p*, *tsp*, *mode*, *equipment*)
a function that derives optical params from transponder type and mode supports the no mode case

gnpy.topology.request.**jsontopath_metric**(*path_metric*)
a functions that reads resulting metric from json string

gnpy.topology.request.**propagate**(*path*, *req*, *equipment*)

gnpy.topology.request.**propagate2**(*path*, *req*, *equipment*)

gnpy.topology.request.**propagate_and_optimize_mode**(*path*, *req*, *equipment*)

gnpy.topology.request.**remove_candidate**(*candidates*, *allpaths*, *rqst*, *pth*)
filter duplicate candidates

gnpy.topology.request.**requests_aggregation**(*pathreqlist*, *disjlist*)
this function aggregates requests so that if several requests exist between same source and destination and with same transponder type

5.1.2.2 gnpy.topology.spectrum_assignment

This module contains the `Oms` and *`Bitmap`* classes and methods to select and assign spectrum. The *`spectrum_selection()`* function identifies the free slots and *`select_candidate()`* selects the candidate spectrum according to strategy: for example first fit oms records its elements, and elements are updated with an oms to have element/oms correspondace

class gnpy.topology.spectrum_assignment.**Bitmap**(*f_min*, *f_max*, *grid*, *guard-band=150000000000.0*, *bitmap=None*)
Bases: `object`

records the spectrum occupation

geti(*nvalue*)
converts the local index into n (itu grid)

getn(*i*)
converts the n (itu grid) into a local index

insert_left(*newbitmap*)
insert bitmap on the left to align oms bitmaps if their start frequencies are different

insert_right(*newbitmap*)
insert bitmap on the right to align oms bitmaps if their stop frequencies are different

class gnpy.topology.spectrum_assignment.**OMS**(**args*, ***params*)
Bases: `object`

OMS class is the logical container that represent a link between two adjacent ROADMs and records the crossed elements and the occupied spectrum

add_element(*elem*)
records oms elements

add_service(*service_id*, *nb_wl*)
record service and mark spectrum as occupied

assign_spectrum(*nvalue*, *mvalue*)
change oms spectrum to mark spectrum assigned

update_spectrum(*f_min*, *f_max*, *guardband=150000000000.0*, *existing_spectrum=None*, *grid=6250000000.0*)
frequencies expressed in Hz

class gnpy.topology.spectrum_assignment.**OMSParams**(*oms_id*, *el_id_list*, *el_list*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('oms_id', 'el_id_list', 'el_list')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new OMSParams object from a sequence or iterable

_replace(***kwds*)
Return a new OMSParams object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property el_id_list
Alias for field number 1

property el_list
Alias for field number 2

property oms_id
Alias for field number 0

gnpy.topology.spectrum_assignment.**align_grids**(*oms_list*)
used to apply same grid to all oms : same starting n, stop n and slot size out of grid slots are set to 0

gnpy.topology.spectrum_assignment.**bitmap_sum**(*band1*, *band2*)
mark occupied bitmap by 0 if the slot is occupied in band1 or in band2

gnpy.topology.spectrum_assignment.**build_oms_list**(*network*, *equipment*)
initialization of OMS list in the network an oms is build reading all intermediate nodes between two adjacent ROADMs each element within the list is being added an oms and oms_id to record the oms it belongs to. the function supports different spectrum width and supposes that the whole network works with the min range among OMSs

gnpy.topology.spectrum_assignment.**frequency_to_n**(*freq*, *grid=6250000000.0*)

converts frequency into the n value (ITU grid) reference to Recommendation G.694.1 (02/12), Figure I.3 https://www.itu.int/rec/T-REC-G.694.1-201202-I/en

```
>>> frequency_to_n(193.1375e12)
6
>>> frequency_to_n(193.225e12)
20
```

gnpy.topology.spectrum_assignment.**m_to_freq**(*nvalue*, *mvalue*, *grid=6250000000.0*)

converts m into frequency range spectrum(13,7) is (193137500000000.0, 193225000000000.0) reference to Recommendation G.694.1 (02/12), Figure I.3 https://www.itu.int/rec/T-REC-G.694.1-201202-I/en

```
>>> fstart, fstop = m_to_freq(13, 7)
>>> fstart
193137500000000.0
>>> fstop
193225000000000.0
```

gnpy.topology.spectrum_assignment.**mvalue_to_slots**(*nvalue*, *mvalue*)
convert center n an m into start and stop n

gnpy.topology.spectrum_assignment.**nvalue_to_frequency**(*nvalue*, *grid=6250000000.0*)

converts n value into a frequency reference to Recommendation G.694.1 (02/12), Table 1 https://www.itu.int/rec/T-REC-G.694.1-201202-I/en

```
>>> nvalue_to_frequency(6)
193137500000000.0
>>> nvalue_to_frequency(-1, 0.1e12)
193000000000000.0
```

gnpy.topology.spectrum_assignment.**pth_assign_spectrum**(*pths*, *rqs*, *oms_list*, *rpths*)
basic first fit assignment if reversed path are provided, means that occupation is bidir

gnpy.topology.spectrum_assignment.**reversed_oms**(*oms_list*)
identifies reversed OMS only applicable for non parallel OMS

gnpy.topology.spectrum_assignment.**select_candidate**(*candidates*, *policy*)
selects a candidate among all available spectrum

gnpy.topology.spectrum_assignment.**slots_to_m**(*startn*, *stopn*)

converts the start and stop n values to the center n and m value reference to Recommendation G.694.1 (02/12), Figure I.3 https://www.itu.int/rec/T-REC-G.694.1-201202-I/en

```
>>> nval, mval = slots_to_m(6, 20)
>>> nval
13
>>> mval
7
```

gnpy.topology.spectrum_assignment.**spectrum_selection**(*pth*, *oms_list*, *requested_m*, *requested_n=None*)
Collects spectrum availability and call the select_candidate function

5.1.3 gnpy.tools

Processing of data via *json_io*. Utilities for Excel conversion in *convert* and *service_sheet*. Example code in *cli_examples* and *plots*.

5.1.3.1 gnpy.tools.cli_examples

Common code for CLI examples

gnpy.tools.cli_examples.**_add_common_options**(*parser: argparse.ArgumentParser*, *network_default: pathlib.Path*)

gnpy.tools.cli_examples.**_path_result_json**(*pathresult*)

gnpy.tools.cli_examples.**_setup_logging**(*args*)

gnpy.tools.cli_examples.**load_common_data**(*equipment_filename*, *topology_filename*, *simulation_filename*, *save_raw_network_filename*)
Load common configuration from JSON files

gnpy.tools.cli_examples.**path_requests_run**(*args=None*)

gnpy.tools.cli_examples.**show_example_data_dir**()

gnpy.tools.cli_examples.**transmission_main_example**(*args=None*)

5.1.3.2 gnpy.tools.convert

This module contains utilities for converting between XLS and JSON.

The input XLS file must contain sheets named "Nodes" and "Links". It may optionally contain a sheet named "Eqpt".

In the "Nodes" sheet, only the "City" column is mandatory. The column "Type" can be determined automatically given the topology (e.g., if degree 2, ILA; otherwise, ROADM.) Incorrectly specified types (e.g., ILA for node of degree 2) will be automatically corrected.

In the "Links" sheet, only the first three columns ("Node A", "Node Z" and "east Distance (km)") are mandatory. Missing "west" information is copied from the "east" information so that it is possible to input undirected data.

class gnpy.tools.convert.**Eqpt**(***kwargs*)
Bases: object

default_values = {'east_amp_dp': None, 'east_amp_gain': None, 'east_amp_type': '',

update_attr(*kwargs*)

class gnpy.tools.convert.**Link**(***kwargs*)
Bases: object

attribtes from west parse_ept_headers dict +node_a, node_z, west_fiber_con_in, east_fiber_con_in

default_values = {'east_cable': '', 'east_con_in': None, 'east_con_out': None, 'ea

update_attr(*kwargs*)

class gnpy.tools.convert.**Node**(***kwargs*)
Bases: object

default_values = {'booster_restriction': '', 'city': '', 'country': '', 'latitud

update_attr(*kwargs*)

gnpy.tools.convert.**_do_convert**()

gnpy.tools.convert.**all_rows**(*sh*, *start=0*)

gnpy.tools.convert.**connect_eqpt**(*from_*, *in_*, *to_*)

gnpy.tools.convert.**convert_file**(*input_filename*, *filter_region=[]*, *output_json_file_name=None*)

gnpy.tools.convert.**corresp_names**(*input_filename*, *network*)
a function that builds the correspondance between names given in the excel, and names used in the json, and created by the autodesign. All names are listed

gnpy.tools.convert.**corresp_next_node**(*network*, *corresp_ila*, *corresp_roadm*)
for each name in corresp dictionnaries find the next node in network and its name given by user in excel. for meshTopology_exampleV2.xls: user ILA name Stbrieuc covers the two direction. convert.py creates 2 different ILA with possible names (depending on the direction and if the eqpt was defined in eqpt sheet) - east edfa in Stbrieuc to Rennes_STA - west edfa in Stbrieuc to Rennes_STA - Edfa0_fiber (Lannion_CAS → Stbrieuc)-F056 - Edfa0_fiber (Rennes_STA → Stbrieuc)-F057 next_nodes finds the user defined name of next node to be able to map the path constraints - east edfa in Stbrieuc to Rennes_STA next node = Rennes_STA - west edfa in Stbrieuc to Rennes_STA next node Lannion_CAS

Edfa0_fiber (Lannion_CAS → Stbrieuc)-F056 and Edfa0_fiber (Rennes_STA → Stbrieuc)-F057 do not exist the function supports fiber splitting, fused nodes and shall only be called if excel format is used for both network and service

gnpy.tools.convert.**eqpt_connection_by_city**(*city_name*)

gnpy.tools.convert.**eqpt_in_city_to_city**(*in_city*, *to_city*, *direction='east'*)

gnpy.tools.convert.**fiber_dest_from_source**(*city_name*)

gnpy.tools.convert.**fiber_link**(*from_city*, *to_city*)

gnpy.tools.convert.**midpoint**(*city_a*, *city_b*)

gnpy.tools.convert.**parse_excel**(*input_filename*)

gnpy.tools.convert.**parse_headers**(*my_sheet*, *input_headers_dict*, *headers*, *start_line*, *slice_in*)
return a dict of header_slice key = column index value = header name

gnpy.tools.convert.**parse_row**(*row*, *headers*)

gnpy.tools.convert.**parse_sheet**(*my_sheet*, *input_headers_dict*, *header_line*, *start_line*, *column*)

gnpy.tools.convert.**read_header**(*my_sheet*, *line*, *slice_*)
return the list of headers !:= '' header_i = [(header, header_column_index), ...] in a {line, slice1_x, slice_y} range

gnpy.tools.convert.**read_slice**(*my_sheet*, *line*, *slice_*, *header*)
return the slice range of a given header in a defined range {line, slice_x, slice_y}

gnpy.tools.convert.**sanity_check**(*nodes*, *links*, *nodes_by_city*, *links_by_city*, *eqpts_by_city*)

gnpy.tools.convert.**xls_to_json_data**(*input_filename*, *filter_region=[]*)

5.1.3.3 gnpy.tools.json_io

Loading and saving data from JSON files in GNPy's internal data format

class gnpy.tools.json_io.**Amp**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'allowed_for_design': False, 'dgt': None, 'dual_stage_model': Non

classmethod from_json(*filename*, ***kwargs*)

class gnpy.tools.json_io.**Fiber**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'dispersion': None, 'gamma': 0, 'pmd_coef': 0, 'type_variety': '

class gnpy.tools.json_io.**Model_dual_stage**(*preamp_variety*, *booster_variety*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('preamp_variety', 'booster_variety')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Model_dual_stage object from a sequence or iterable

_replace(***kwds*)
Return a new Model_dual_stage object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property booster_variety
Alias for field number 1

property preamp_variety
Alias for field number 0

class gnpy.tools.json_io.**Model_fg**(*nf0*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('nf0',)

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Model_fg object from a sequence or iterable

_replace(***kwds*)
Return a new Model_fg object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property nf0
Alias for field number 0

class gnpy.tools.json_io.**Model_hybrid**(*nf_ram*, *gain_ram*, *edfa_variety*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('nf_ram', 'gain_ram', 'edfa_variety')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Model_hybrid object from a sequence or iterable

_replace(***kwds*)
Return a new Model_hybrid object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property edfa_variety
Alias for field number 2

property gain_ram
Alias for field number 1

property nf_ram
Alias for field number 0

class gnpy.tools.json_io.**Model_openroadm**(*nf_coef*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('nf_coef',)

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Model_openroadm object from a sequence or iterable

_replace(***kwds*)
Return a new Model_openroadm object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property nf_coef
Alias for field number 0

class gnpy.tools.json_io.**Model_vg**(*nf1*, *nf2*, *delta_p*)
Bases: tuple

_asdict()
Return a new OrderedDict which maps field names to their values.

_fields = ('nf1', 'nf2', 'delta_p')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Model_vg object from a sequence or iterable

_replace(***kwds*)
Return a new Model_vg object replacing specified fields with new values

_source = "from builtins import property as _property, tuple as _tuple\nfrom operator

property delta_p
Alias for field number 2

property nf1
Alias for field number 0

property nf2
Alias for field number 1

class gnpy.tools.json_io.**RamanFiber**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'dispersion': None, 'gamma': 0, 'pmd_coef': 0, 'raman_efficiency'

class gnpy.tools.json_io.**Roadm**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'add_drop_osnr': 100, 'pmd': 0, 'restrictions': {'booster_variety

class gnpy.tools.json_io.**SI**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'baud_rate': 32000000000.0, 'f_max': 196100000000000.0, 'f_min':

class gnpy.tools.json_io.**Span**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'EOL': 0, 'con_in': 0, 'con_out': 0, 'delta_power_range_db': None

class gnpy.tools.json_io.**Transceiver**(***kwargs*)
Bases: *gnpy.tools.json_io._JsonThing*

default_values = {'frequency': None, 'mode': {}, 'type_variety': None}

class gnpy.tools.json_io.**_JsonThing**
Bases: object

update_attr(*default_values*, *kwargs*, *name*)

gnpy.tools.json_io.**_automatic_spacing**(*baud_rate*)
return the min possible channel spacing for a given baud rate

gnpy.tools.json_io.**_check_one_request**(*params*, *f_max_from_si*)
Checks that the requested parameters are consistant (spacing vs nb channel vs transponder mode...)

gnpy.tools.json_io.**_cls_for**(*equipment_type*)

gnpy.tools.json_io.**_equipment_from_json**(*json_data*, *filename*)
build global dictionnary eqpt_library that stores all eqpt characteristics: edfa type type_variety, fiber type_variety from the eqpt_config.json (filename parameter) also read advanced_config_from_json file parameters for edfa if they are available: typically nf_ripple, dfg gain ripple, dgt and nf polynomial nf_fit_coeff if advanced_config_from_json file parameter is not present: use nf_model: requires nf_min and nf_max values boundaries of the edfa gain range

gnpy.tools.json_io.**_roadm_restrictions_sanity_check**(*equipment*)
verifies that booster and preamp restrictions specified in roadm equipment are listed in the edfa.

gnpy.tools.json_io.**_update_dual_stage**(*equipment*)

gnpy.tools.json_io.**_update_trx_osnr**(*equipment*)
add sys_margins to all Transceivers OSNR values

gnpy.tools.json_io.**convert_service_sheet**(*input_filename*, *eqpt*, *network*, *network_filename=None*, *output_filename=''*, *bidir=False*, *filter_region=None*)

gnpy.tools.json_io.**disjunctions_from_json**(*json_data*)
reads the disjunction requests from the json dict and create the list of requested disjunctions for this set of requests

gnpy.tools.json_io.**load_equipment**(*filename*)

gnpy.tools.json_io.**load_json**(*filename*)

gnpy.tools.json_io.**load_network**(*filename*, *equipment*)

gnpy.tools.json_io.**load_requests**(*filename*, *eqpt*, *bidir*, *network*, *network_filename*)
loads the requests from a json or an excel file into a data string

gnpy.tools.json_io.**network_from_json**(*json_data*, *equipment*)

gnpy.tools.json_io.**network_to_json**(*network*)

gnpy.tools.json_io.**requests_from_json**(*json_data*, *equipment*)
Extract list of requests from data parsed from JSON

gnpy.tools.json_io.**save_json**(*obj*, *filename*)

gnpy.tools.json_io.**save_network**(*network: networkx.classes.digraph.DiGraph*, *filename: str*)
Dump the network into a JSON file

Parameters

- **network** – network to work on
- **filename** – file to write to

5.1.3.4 gnpy.tools.plots

Graphs and plots usable form a CLI application

gnpy.tools.plots.**plot_baseline**(*network*)

gnpy.tools.plots.**plot_results**(*network*, *path*, *source*, *destination*, *infos*)

5.1.3.5 gnpy.tools.service_sheet

XLS parser that can be called to create a JSON request file in accordance with Yang model for requesting path computation.

See: draft-ietf-teas-yang-path-computation-01.txt

class gnpy.tools.service_sheet.**Element**
Bases: object

class gnpy.tools.service_sheet.**Request**
Bases: *gnpy.tools.service_sheet.Request*

class gnpy.tools.service_sheet.**Request_element**(*Request*, *equipment*, *bidir*)
Bases: *gnpy.tools.service_sheet.Element*

property json

property pathrequest

property pathsync

property uid

gnpy.tools.service_sheet.**all_rows**(*sheet*, *start=0*)

gnpy.tools.service_sheet.**correct_xlrd_int_to_str_reading**(*v*)

gnpy.tools.service_sheet.**correct_xls_route_list**(*network_filename*, *network*, *pathrequest_list*)
prepares the format of route list of nodes to be consistant with nodes names: remove wrong names, find correct names for ila, roadm and fused if the entry was xls. if it was not xls, all names in list should be exact name in the network.

gnpy.tools.service_sheet.**parse_excel**(*input_filename*)

gnpy.tools.service_sheet.**parse_row**(*row*, *fieldnames*)

gnpy.tools.service_sheet.**parse_service_sheet**(*service_sheet*)
reads each column according to authorized fieldnames. order is not important.

gnpy.tools.service_sheet.**read_service_sheet**(*input_filename*, *eqpt*, *network*, *network_filename=None*, *bidir=False*, *filter_region=None*)
converts a service sheet into a json structure

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[BSR+12] A. Bononi, P. Serena, N. Rossi, E. Grellier, and F. Vacondio. Modeling nonlinearity in coherent transmissions with dominant intrachannel-four-wave-mixing. *Optics Express*, 20(7):7777, 2012. URL: https://www.osapublishing.org/oe/abstract.cfm?uri=oe-20-7-7777, doi:10.1364/OE.20.007777.

[BBS13] Alberto Bononi, Ottmar Beucher, and Paolo Serena. Single- and cross-channel nonlinear interference in the Gaussian Noise model with rectangular spectra. *Optics Express*, 21(26):32254, 2013. URL: https://www.osapublishing.org/oe/abstract.cfm?uri=oe-21-26-32254, doi:10.1364/OE.21.032254.

[CAC18] Mattia Cantono, Jean Luc Auge, and Vittorio Curri. Modelling the impact of SRS on NLI generation in commercial equipment: an experimental investigation. In *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2018*. 2018. doi:10.1364/OFC.2018.M1D.2.

[CBC+09] A. Carena, G. Bosco, V. Curri, P. Poggiolini, M. Tapia Taiba, and F. Forghieri. Statistical characterization of PM-QPSK signals after propagation in uncompensated fiber links. In *European Conference on Optical Communications, 2010*, 1–3. IEEE, 2010-09. URL: http://ieeexplore.ieee.org/document/5621509/, doi:10.1109/ECOC.2010.5621509.

[CCB+05] A. Carena, V. Curri, G. Bosco, P. Poggiolini, and F. Forghieri. Modeling of the Impact of Nonlinear Propagation Effects in Uncompensated Optical Coherent Transmission Links. *Journal of Lightwave Technology*, 30(10):1524–1539, 2012-05. URL: http://ieeexplore.ieee.org/document/6158564/, doi:10.1109/JLT.2012.2189198.

[CPCF09] V. Curri, P. Poggiolini, A. Carena, and F. Forghieri. Dispersion Compensation and Mitigation of Nonlinear Effects in 111-Gb/s WDM Coherent PM-QPSK Systems. *IEEE Photonics Technology Letters*, 20(17):1473–1475, 2008-09. URL: http://ieeexplore.ieee.org/document/4589011/, doi:10.1109/LPT.2008.927906.

[DFMS04] Ronen Dar, Meir Feder, Antonio Mecozzi, and Mark Shtaif. Properties of nonlinear noise in long, dispersion-uncompensated fiber links. *Optics Express*, 21(22):25685, 2013-11-04. URL: https://www.osapublishing.org/oe/abstract.cfm?uri=oe-21-22-25685, doi:10.1364/OE.21.025685.

[DFMS16] Ronen Dar, Meir Feder, Antonio Mecozzi, and Mark Shtaif. Accumulation of nonlinear interference noise in fiber-optic systems. *Optics Express*, 22(12):14199, 2014-06-16. URL: https://www.osapublishing.org/oe/abstract.cfm?uri=oe-22-12-14199, doi:10.1364/OE.22.014199.

[FME+16] T. Fehenberger, M. Mazur, T. A. Eriksson, M. Karlsson, and N. Hanik. Experimental analysis of correlations in the nonlinear phase noise in optical fiber systems. In *ECOC 2016; 42nd European Conference on Optical Communication*, volume, 1–3. Sept 2016. doi:.

[FCBS06] Tommaso Foggi, Giulio Colavolpe, Alberto Bononi, and Paolo Serena. Overcoming filtering penalties in flexi-grid long-haul optical systems. In *International Conference on Communications*, 5168–5173. IEEE, 2015-06. URL: http://ieeexplore.ieee.org/document/7249144/, doi:10.1109/ICC.2015.7249144.

[Ino92] K. Inoue. Four-wave mixing in an optical fiber in the zero-dispersion wavelength region. *Journal of Lightwave Technology*, 10(11):1553–1561, Nov 1992. doi:10.1109/50.184893.

[JA01] Pontus Johannisson and Erik Agrell. Modeling of Nonlinear Signal Distortion in Fiber-Optic Networks. *Journal of Lightwave Technology*, 32(23):4544–4552, 2014-12-01. URL: http://ieeexplore.ieee.org/document/6915838/, doi:10.1109/JLT.2014.2361357.

[JK04] Pontus Johannisson and Magnus Karlsson. Perturbation Analysis of Nonlinear Propagation in a Strongly Dispersive Optical Communication System. *Journal of Lightwave Technology*, 31(8):1273–1282, 2013-04. URL: http://ieeexplore.ieee.org/document/6459512/, doi:10.1109/JLT.2013.2246543.

[ME06] Antonio Mecozzi and René-Jean Essiambre. Nonlinear Shannon Limit in Pseudolinear Coherent Systems. *Journal of Lightwave Technology*, 30(12):2011–2024, 2012-06. URL: http://ieeexplore.ieee.org/document/6175093/, doi:10.1109/JLT.2012.2190582.

[PCCC07] Dario Pilori, Mattia Cantono, Andrea Carena, and Vittorio Curri. FFSS: The fast fiber simulator software. In *International Conference on Transparent Optical Networks*, 1–4. IEEE, 2017-07. URL: http://ieeexplore.ieee.org/document/8025002/, doi:10.1109/ICTON.2017.8025002.

[PCC+06] P Poggiolini, A Carena, V Curri, G Bosco, and F Forghieri. Analytical Modeling of Nonlinear Propagation in Uncompensated Optical Transmission Links. *IEEE Photonics Technology Letters*, 23(11):742–744, 2011-06. URL: http://ieeexplore.ieee.org/document/5735190/, doi:10.1109/LPT.2011.2131125.

[PBC+02] P. Poggiolini, G. Bosco, A. Carena, V. Curri, Y. Jiang, and F. Forghieri. The GN-Model of Fiber Non-Linear Propagation and its Applications. *Journal of Lightwave Technology*, 32(4):694–721, 2014-02. URL: http://ieeexplore.ieee.org/document/6685826/, doi:10.1109/JLT.2013.2295208.

[PJ01] P. Poggiolini and Y. Jiang. Recent Advances in the Modeling of the Impact of Nonlinear Fiber Propagation Effects on Uncompensated Coherent Transmission Systems. *Journal of Lightwave Technology*, 35(3):458–480, 2017-02-01. URL: http://ieeexplore.ieee.org/document/7577767/, doi:10.1109/JLT.2016.2613893.

[RNR+01] Talha Rahman, Antonio Napoli, Danish Rafique, Bernhard Spinnler, Maxim Kuschnerov, Iveth Lobato, Benoit Clouet, Marc Bohn, Chigo Okonkwo, and Huug de Waardt. On the Mitigation of Optical Filtering Penalties Originating From ROADM Cascade. *IEEE Photonics Technology Letters*, 26(2):154–157, 2014-01. URL: http://ieeexplore.ieee.org/document/6662421/, doi:10.1109/LPT.2013.2290745.

[Sav05] Seb J. Savory. Approximations for the Nonlinear Self-Channel Interference of Channels With Rectangular Spectra. *IEEE Photonics Technology Letters*, 25(10):961–964, 2013-05. URL: http://ieeexplore.ieee.org/document/6491442/, doi:10.1109/LPT.2013.2255869.

[SLEF+15] C. Schmidt-Langhorst, R. Elschner, F. Frey, R. Emmerich, and C. Schubert. Experimental analysis of nonlinear interference noise in heterogeneous flex-grid wdm transmission. In *2015 European Conference on Optical Communication (ECOC)*, volume, 1–3. Sept 2015. doi:10.1109/ECOC.2015.7341918.

[SF11] M. Secondini and E. Forestieri. Analytical Fiber-Optic Channel Model in the Presence of Cross-Phase Modulation. *IEEE Photonics Technology Letters*, 24(22):2016–2019, 2012-11. URL: http://ieeexplore.ieee.org/document/6297443/, doi:10.1109/LPT.2012.2217952.

[SFP12] Marco Secondini, Enrico Forestieri, and Giancarlo Prati. Achievable Information Rate in Nonlinear WDM Fiber-Optic Systems With Arbitrary Modulation Formats and Dispersion Maps. *Journal of Lightwave Technology*, 31(23):3839–3852, 2013-12. URL: http://ieeexplore.ieee.org/document/6655896/, doi:10.1109/JLT.2013.2288677.

[SB11] Paolo Serena and Alberto Bononi. An Alternative Approach to the Gaussian Noise Model and its System Implications. *Journal of Lightwave Technology*, 31(22):3489–3499, 2013-11. URL: http://ieeexplore.ieee.org/document/6621015/, doi:10.1109/JLT.2013.2284499.

[VRS+16] Francesco Vacondio, Olivier Rival, Christian Simonneau, Edouard Grellier, Alberto Bononi, Laurence Lorcy, Jean-Christophe Antona, and Sébastien Bigo. On nonlinear distortions of highly dispersive optical coherent systems. *Optics Express*, 20(2):1022, 2012-01-16. URL: https://www.osapublishing.org/oe/abstract.cfm?uri=oe-20-2-1022, doi:10.1364/OE.20.001022.

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

W

X