[DRAFT]GNPy Configuration Guide

Based on GNPy v2.x Documentation

November 3, 2025

Abstract

This comprehensive guide provides documentation for configuring GNPy (Gaussian Noise model Python), an open-source optical network planning tool developed by the Telecom Infra Project.

Contents

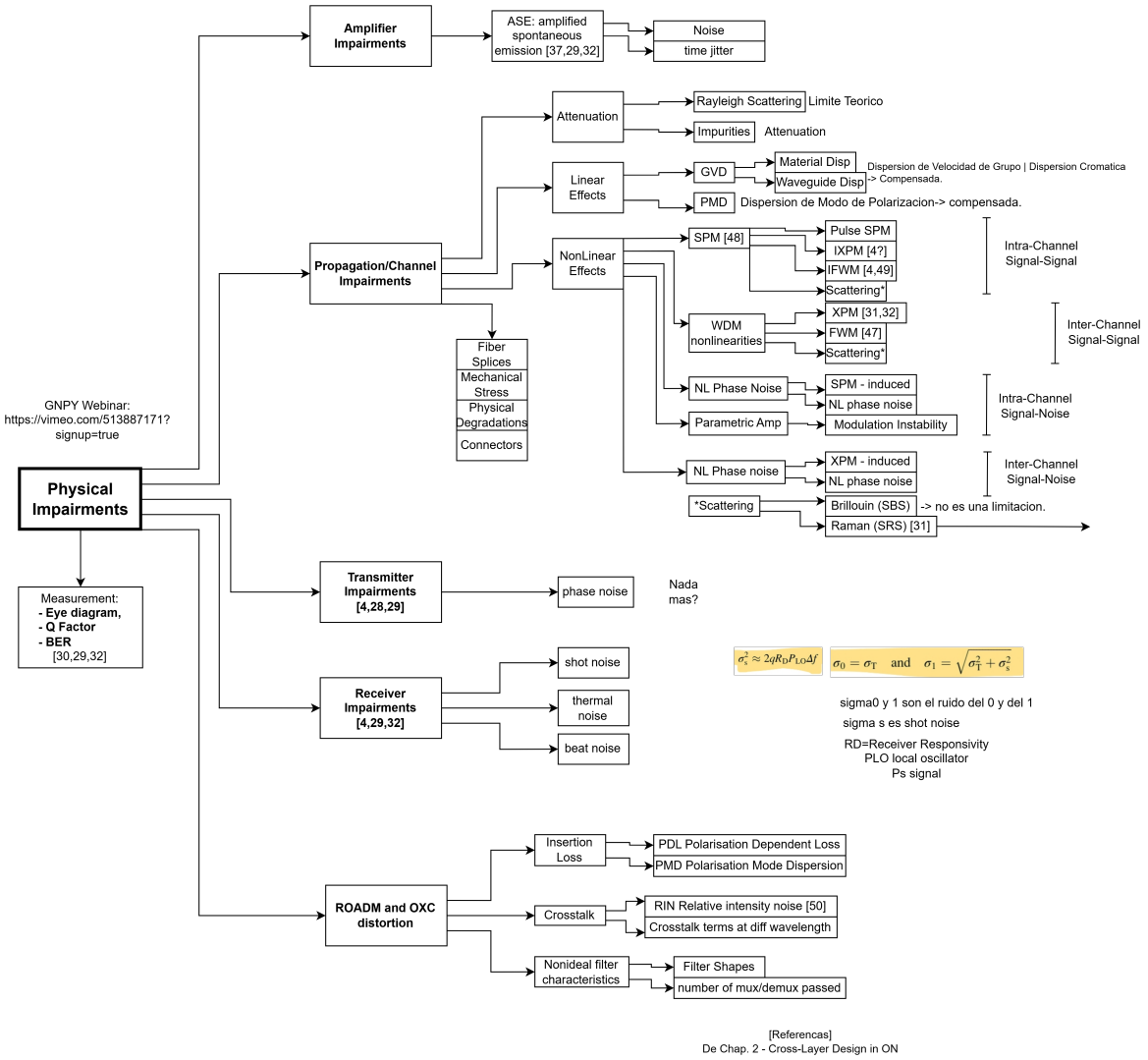

Figure 1: Enter Caption

1 Optical Impairments Overview

2 GNPy Impairments Model

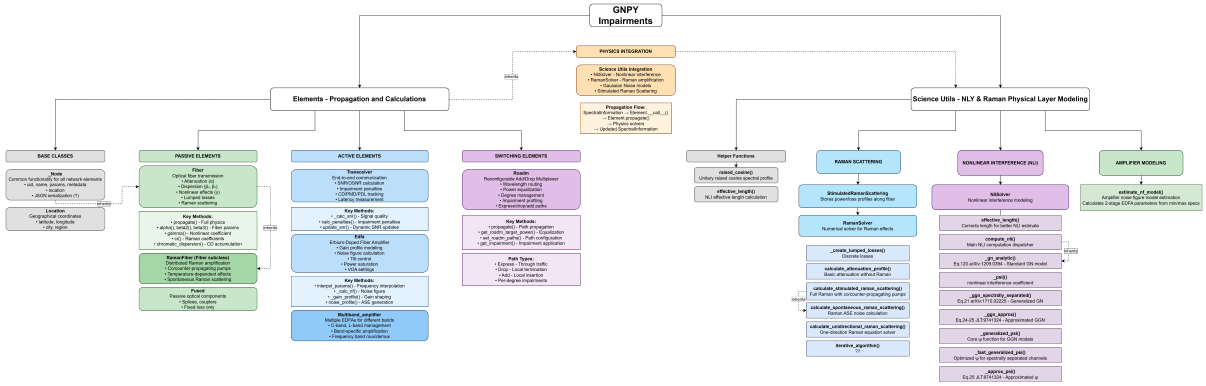

Figure 2: GNPy impairment impelementations.

Fuentes(COMPLETAR, CORREGIR) : Paper: $GNPy_model_of_the_physical_layer_for_open_and_disa$ jocn-12-6-c31 GNPY demo 2021(uso): https://www.youtube.com/watch?v=g1S7k52VkWQ GNPY Webinar (mas sobre teoria y utilizacion): https://vimeo.com/513887171?signup=true

2.1 Impairments implementation in GNPy

GNPy will create a structure that represents the network and will read the path requests and calculate the best path usting Dijstra algorythm. Once GNPy obtains the path to compute, it applies impariment effects on the signal in each element involved, depending on the specifications and the kind of element.

This affects the signal step by step, obtaining a final output with QOT estimations.

2.1.1 elements.py

who calls it

what it models

functions

2.1.2 science$_utils.py$

who calls it

what it models

functions

2.2 GNPy Class Diagram

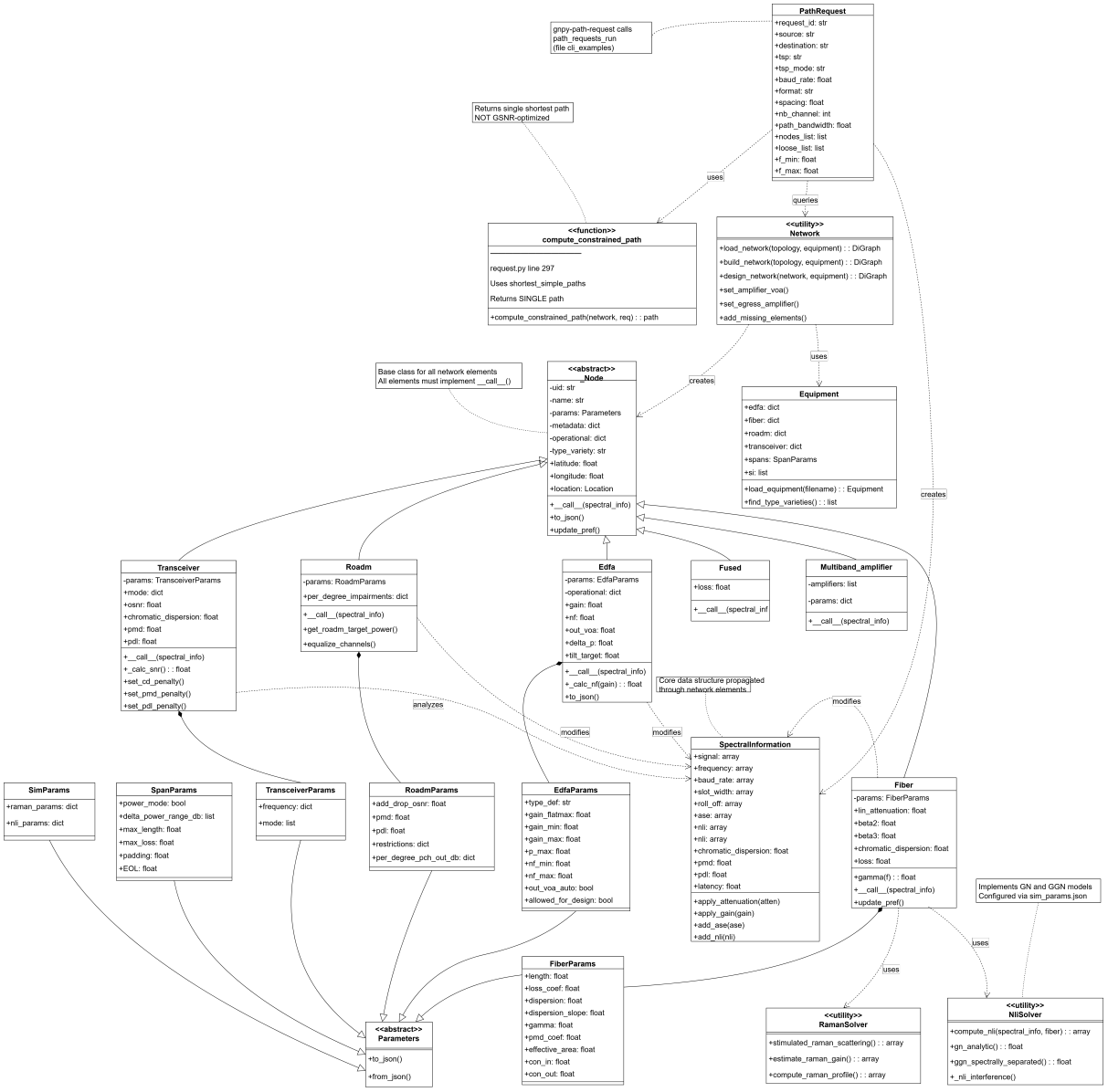

Figure 3: GNPy class diagram

3 GNPy Software Workflow

Diagram in Figure 4 shows the general GNPy workflow. It has different phases:

1. **Input Layer:** User prepares files that defines elements with specifications, network as connections, general parameters and path requests.
2. **Network Model Building:** GNPy prepares the network structure to get a usable model. Missing elements and specifications will be automatically calculated or choosen (HOW?), if possible.
3. **Execution Paths:**
   - `gnpy-path-request` computes every service request in `services.json`.
   - `gnpy-transmission-example` simulates all the paths in the network, this is not a throughout computation but an example (EXPLAIN WHAT DOES IT DO)
4. **Core Propagation Engine:** Actual funcions implemented by GNPy, computing path propagation, impairments and its effects in QOT.
5. **Output Layer:** results.

We will cover all the phases in this section.

3.1 Input Files

GNPy input files, their parameters, structure, and usage. Each file description includes complete parameter reference tables with OpenROADM model mappings where applicable.

3.1.1 Configuration File Summary

GNPy uses several JSON files to configure optical networks:

File (example name)	Purpose	Required?
`eqpt_config.json`	Equipment library	Yes
`topology.json`	Network topology (nodes and links)	Yes
`services.json`	Service requests (path demands)	Optional*
`sim_params.json`	Simulation parameters (NLI, Raman)	Optional*
`initial_spectrum.json`	Spectrum definition for transmission-example	Optional**
Advanced EDFA configs	Detailed amplifier models	Optional**

Table 1: GNPy Configuration Files

- * - Optional for `gnpy-transmission-example` but needed for `gnpy-path-request`.
- ** - additional specifications could be needed if advanced request is computed.

The configuration files are interconnected, with `eqpt_config.json` serving as the foundation:

```
eqpt_config.json
|-- Can reference advanced config files e.g.:
|   |-- Juniper-BoosterHG.json
|   |-- std_medium_gain_advanced_config.json
|
```

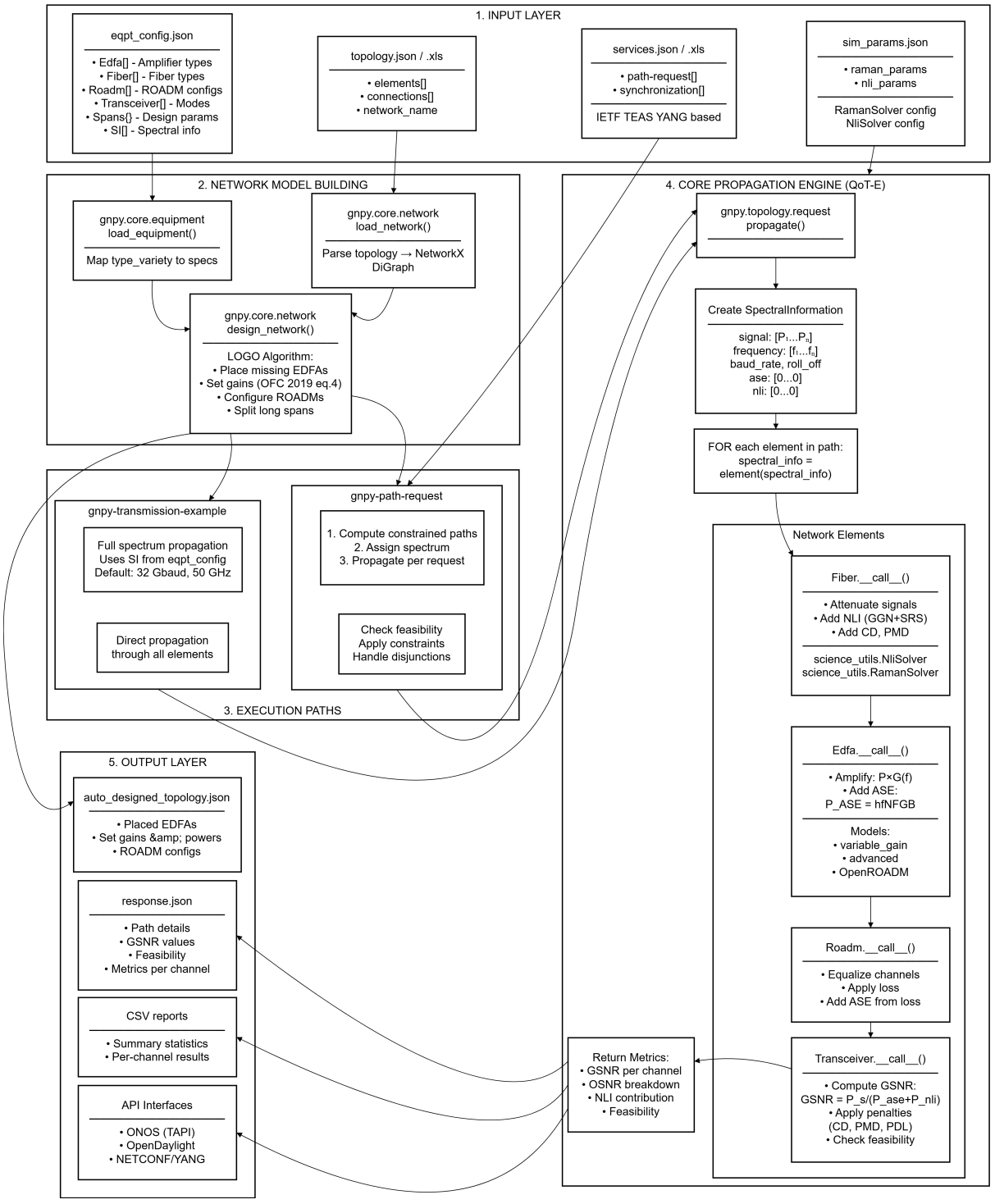

Figure 4: GNPy workflow.

```
topology.json
|-- Uses equipment types from eqpt_config.json e.g.:
|   |-- type_variety: "std_medium_gain" -> defined in eqpt_config e.g.:
|   |-- type_variety: "SSMF" -> defined in eqpt_config
|
services.json
|-- References nodes from topology.json
|   |-- source: "NodeA" -> must exist in topology
|   |-- destination: "NodeB" -> must exist in topology
|
sim_params.json
|-- Configures physical models (NLI, Raman)
```

3.1.2 `eqpt_config.json`

The equipment configuration file defines the library of optical components available for network design and simulation. It contains the physical and operational parameters for amplifiers, fibers, ROADMs, and transceivers. This file is mandatory.

Structure Overview

The equipment configuration has 7 main sections:

```
{
  "Edfa": [...],           // Optical amplifiers
  "Fiber": [...],          // Fiber types
  "RamanFiber": [...],     // Raman-enabled fibers
  "Span": {...},           // Span design rules
  "Roadm": [...],          // ROADM configurations
  "SI": [...],             // Spectral Information defaults
  "Transceiver": [...]     // Transceiver modes
}
```

COMPLETAR LOS PARAMETROS DE LAS TABLAS, YA ESCRITO

EDFA (Erbium-Doped Fiber Amplifier)

Parameter	Type	Required/Optional	Description
`type_variety`	string	Required	Unique identifier for this EDFA model
`type_def`	string	Required	Model type: `variable_gain`, `fixed_gain`, `openroadm`, `advanced_model`, `dual_stage`
`gain_flatmax`	number	Required	Maximum flat gain in dB
`gain_min`	number	Required	Minimum gain in dB
`p_max`	number	Required	Maximum output power in dBm
`nf_min`	number	Required	Minimum noise figure in dB
`nf_max`	number	Required	Maximum noise figure in dB
`out_voa_auto`	boolean	Optional	Enable automatic output VOA adjustment. Default: `false`

Parameter	Type	Required/Optional	Description
`allowed_for_design`	boolean	Optional	Allow auto-design to use this amplifier. Default: `false`
`nf_model`	object	Optional	Noise figure model parameters
`advanced_config_from_json`	string	Optional	Path to advanced configuration file

Table 2: EDFA Configuration Parameters

Fiber

Parameter	Type	Required/Optional	Description
`type_variety`	string	Required	Unique identifier for this fiber type
`dispersion`	number	Required	Chromatic dispersion at 1550 nm (ps/(nm·km))
`gamma`	number	Required	Nonlinear coefficient (1/(W·km))
`pmd_coef`	number	Optional	PMD coefficient (ps/km). Default: 0
`loss_coef`	number	Optional	Attenuation coefficient (dB/km). Default: 0.2
`dispersion_slope`	number	Optional	Dispersion slope (ps/(nm^2·km))
`effective_area`	number	Optional	Effective area (m^2)

Table 3: Fiber Configuration Parameters

RamanFiber

Parameter	Type	Required/Optional	Description
`type_variety`	string	Required	Unique identifier for this Raman fiber type
`dispersion`	number	Required	Chromatic dispersion at 1550 nm (ps/(nm·km))
`gamma`	number	Required	Nonlinear coefficient (1/(W·km))
`raman_efficiency`	object	Required	Raman efficiency profile with frequency offsets and coefficients

Table 4: RamanFiber Configuration Parameters

Span

Parameter	Type	Required/Optional	Description
power_mode	boolean	Optional	Use power mode for span design. Default: true
delta_power_range_db	array	Optional	Power deviation range [min, max] in dB
max_fiber_lineic_loss_for_raman	number	Optional	Maximum fiber loss for Raman pumps (dB/km)
target_extended_gain	number	Optional	Target extended gain for auto-design (dB)
max_length	number	Optional	Maximum span length for splitting (km)
length_units	string	Optional	Length units: "km" or "m". Default: "km"

Table 5: Span Configuration Parameters

Roadm

Parameter	Type	Required/Optional	Description
type_variety	string	Required	Unique identifier for this ROADM type
target_pch_out_db	number	Optional*	Target per-channel output power (dBm)
target_psd_out_mWperGHz	number	Optional*	Target PSD output (mW/GHz)
target_out_mWperSlotWidth	number	Optional*	Target power per slot width (mW)
add_drop_osnr	number	Required	OSNR contribution from add/drop ports (dB)
pmd	number	Required	PMD (ps)
pdl	number	Required	PDL (dB)
restrictions	object	Required	Restrictions for this ROADM type

Table 6: ROADM Configuration Parameters (* only one equalization type required)

SI (Spectral Information)

Parameter	Type	Required/Optional	Description
f_min	number	Required	Minimum frequency (Hz)
f_max	number	Required	Maximum frequency (Hz)
baud_rate	number	Required	Symbol rate (Baud)
spacing	number	Required	Channel spacing (Hz)
power_dbm	number	Required	Per-channel power (dBm)
power_range_db	array	Optional	Power range [min, max] (dB)

Parameter	Type	Required/Optional	Description
roll_off	number	Optional	Roll-off factor. Default: 0.15
tx_osnr	number	Optional	Transmitter OSNR (dB). Default: 100

Table 7: SI Configuration Parameters

Transceiver

Parameter	Type	Required/Optional	Description
type_variety	string	Required	Unique identifier for transceiver type
frequency	object	Required	Operating frequency range (min_spacing, max_spacing)
mode	array	Required	List of operational modes for this transceiver

Table 8: Transceiver Configuration Parameters

Transceiver Mode Each mode within a transceiver contains:

Parameter	Type	Required/Optional	Description
format	string	Required	Modulation format
baud_rate	number	Required	Symbol rate (Baud)
OSNR	number	Required	Required OSNR (dB)
bit_rate	number	Required	Bit rate (bit/s)
roll_off	number	Optional	Roll-off factor
tx_osnr	number	Optional	Transmitter OSNR (dB)
min_spacing	number	Optional	Minimum channel spacing (Hz)
cost	number	Optional	Relative cost metric
penalties	object	Optional	Impairment penalties

Table 9: Transceiver Mode Parameters

reescribir

Example

```
{
  "type_variety": "std_medium_gain",
  "type_def": "variable_gain",
  "gain_flatmax": 26,      // Maximum flat gain (dB)
  "gain_min": 15,          // Minimum gain (dB)
  "p_max": 23,             // Maximum output power (dBm)
  "nf_min": 6,             // Minimum noise figure (dB)
  "nf_max": 10,            // Maximum noise figure (dB)
  "out_voa_auto": false,   // Automatic VOA adjustment
  "allowed_for_design": true  // Can auto-design use this?
}
```

```
{
  "type_variety": "SSMF",
  "dispersion": 1.67e-05,   // 16.7 ps/(nm km) at 1550nm
  "gamma": 0.00127,         // 1.27e-3 (1/(W km))
  "pmd_coef": 0.1e-12,      // 0.1 ps/ km
  "loss_coef": 0.2,         // 0.2 dB/km
  "length_units": "km"
}
```

```
{
  "type_variety": "default",
  "target_pch_out_db": -20,  // Target per-channel power
  "add_drop_osnr": 38,       // Add/drop OSNR (dB)
  "pmd": 0,                  // PMD (ps)
  "pdl": 0,                  // PDL (dB)
  "restrictions": {
    "preamp_variety_list": [],
    "booster_variety_list": []
  }
}
```

```
{
  "type_variety": "default",
  "f_min": 191.35e12,        // 191.35 THz
  "f_max": 196.1e12,         // 196.1 THz
  "baud_rate": 32e9,         // 32 GBaud
  "spacing": 50e9,           // 50 GHz
  "power_dbm": 0,            // 0 dBm per channel
  "roll_off": 0.15,          // 15% roll-off
  "tx_osnr": 100             // 100 dB transmitter OSNR
}
```

```
{
  "type_variety": "vendorA_xcvr",
  "frequency": {
    "min": 191.35e12,
    "max": 196.1e12
  },
  "mode": [
    {
      "format": "DP-QPSK",
      "baud_rate": 32e9,
      "OSNR": 11,
      "bit_rate": 100e9,
      "roll_off": 0.15,
      "tx_osnr": 100,
      "min_spacing": 50e9,
      "cost": 1
    },
    {
```

```
      "format": "DP-16QAM",
      "baud_rate": 32e9,
      "OSNR": 18,
      "bit_rate": 200e9,
      "roll_off": 0.15,
      "tx_osnr": 100,
      "min_spacing": 50e9,
      "cost": 1
    }
  ]
}
```

Excel Alternative

While GNPy primarily uses JSON files, an Excel-based workflow is available using `gnpy-convert-xls`:

- **Limitations:**
  - Limited to basic equipment parameters
  - Cannot specify advanced EDFA configurations
  - Reduced flexibility compared to native JSON
  - Requires conversion step before use
- **Conversion Command:**

```
gnpy-convert-xls equipment.xlsx -o eqpt_config.json
```

3.1.3 topology.json / .xls

The topology file defines the physical network structure, including all network `elements` (nodes) and their `connections` (links). This file is mandatory.

The structure is as it follows:

```

{
  "elements": [
    {
      "uid": "NodeA",
      "type": "Transceiver",
      "metadata": {...}
    },
    {
      "uid": "Fiber1",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {...}
    },
    ...
  ],
  "connections": [
```

```
    {
      "from_node": "NodeA",
      "to_node": "Fiber1"
    },
    ...
  ]
}
```

Parameters

Element Types GNPy supports the following network element types:

Type	Description
`Transceiver`	Optical transmitter/receiver (network endpoints)
`Fiber`	Optical fiber span
`RamanFiber`	Raman-enabled fiber span
`Edfa`	Erbium-doped fiber amplifier
`Roadm`	Reconfigurable optical add-drop multiplexer
`Fused`	Passive optical component (splitter/coupler)

Table 10: Network Element Types

Common Element Parameters

Parameter	Type	Required/Optional	Description
`uid`	string	Required	Unique identifier for this element
`type`	string	Required	Element type (see Table 10)
`type_variety`	string	Optional	Reference to equipment library entry
`params`	object	Optional	Element-specific parameters
`metadata`	object	Optional	Additional information (location, etc.)
`operational`	object	Optional	Operational settings

Table 11: Common Element Parameters

Fiber-Specific Parameters

Parameter	Type	Required/Optional	Description
`length`	number	Required	Fiber length
`length_units`	string	Optional	Units: `"km"` or `"m"`. Default: `"km"`
`loss_coef`	number	Optional	Overrides library value (dB/km)

Parameter	Type	Required/Optional	Description
con_in	number	Optional	Input connector loss (dB). Default: 0
con_out	number	Optional	Output connector loss (dB). Default: 0
att_in	number	Optional	Input attenuator (dB). Default: 0

Table 12: Fiber-Specific Parameters

EDFA-Specific Parameters (Operational)

Parameter	Type	Required/Optional	Description
gain_target	number	Optional	Target gain (dB)
delta_p	number	Optional	Power adjustment (dB). Default: 0
tilt_target	number	Optional	Gain tilt (dB). Default: 0
out_voa	number	Optional	Output VOA attenuation (dB). Default: 0

Table 13: EDFA Operational Parameters

ROADM-Specific Parameters

Parameter	Type	Required/Optional	Description
target_pch_out_db	number	Optional*	Per-channel output power target (dBm)
target_psd_out_mWperGHz	number	Optional*	PSD output target (mW/GHz)
target_out_mWperSlotWidth	number	Optional*	Power per slot width target (mW)
add_drop_osnr	number	Optional	Overrides library value (dB)

Table 14: ROADM-Specific Parameters (* only one equalization type allowed)

Fused Element Parameters

Parameter	Type	Required/Optional	Description
loss	number	Required	Passive loss (dB)

Table 15: Fused Element Parameters

Connection Parameters

Parameter	Type	Required/Optional	Description
`from_node`	string	Required	Source element UID
`to_node`	string	Required	Destination element UID

Table 16: Connection Parameters

Usage

The topology file is used by:

- **`load_network()`** function to build the NetworkX DiGraph
- **`design_network()`** function for automatic amplifier placement
- All simulation executables (`gnpy-path-request`, `gnpy-transmission-example`)

Importance: The topology file defines the physical constraints and structure that GNPy uses to:

- Calculate path losses and propagation effects
- Place amplifiers automatically if not specified
- Compute feasible paths between transceivers
- Validate service requests against physical topology

Examples

An example of a complete Point-to-Point Topology

```
{
  "elements": [
    {
      "uid": "Paris",
      "type": "Transceiver",
      "metadata": {
        "location": {
          "city": "Paris",
          "region": "IDF",
          "latitude": 48.8566,
          "longitude": 2.3522
        }
      }
    },
    {
      "uid": "Fiber_Paris_Lyon",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {
        "length": 80.0,
        "length_units": "km",
        "con_in": 0.5,
        "con_out": 0.5
      }
    },
    {
      "uid": "Amp_preROADM_Lyon",
      "type": "Edfa",
      "type_variety": "std_medium_gain",
      "operational": {
        "gain_target": 20,
        "tilt_target": 0
      }
```

```
    },
    {
      "uid": "ROADM_Lyon",
      "type": "Roadm",
      "params": {
        "target_pch_out_db": -20
      }
    },
    {
      "uid": "Amp_postROADM_Lyon",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {
      "uid": "Fiber_Lyon_Marseille",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {
        "length": 120.0,
        "length_units": "km"
      }
    },
    {
      "uid": "Amp_Marseille",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {
      "uid": "Marseille",
      "type": "Transceiver"
    }
  ],
  "connections": [
    {"from_node": "Paris", "to_node": "Fiber_Paris_Lyon"},
    {"from_node": "Fiber_Paris_Lyon", "to_node": "Amp_preROADM_Lyon"},
    {"from_node": "Amp_preROADM_Lyon", "to_node": "ROADM_Lyon"},
    {"from_node": "ROADM_Lyon", "to_node": "Amp_postROADM_Lyon"},
    {"from_node": "Amp_postROADM_Lyon", "to_node": "Fiber_Lyon_Marseille"},
    {"from_node": "Fiber_Lyon_Marseille", "to_node": "Amp_Marseille"},
    {"from_node": "Amp_Marseille", "to_node": "Marseille"}
  ]
}
```

Network Design Rules

- Each element must have a unique `uid`
- `type_variety` must reference a valid entry in `eqpt_config.json`
- Transceivers mark the start and end of optical paths (they can be created by GNPy if a ROADM doesnt have a defined trx in the path request)
- The network must form a directed graph
- Elements can have multiple inputs and outputs
- Connection order defines signal flow direction

Excel Alternative

Excel-based topology definition is supported:

- **Format:** Nodes sheet and Links sheet
- **Limitations:**

  - Limited parameter specification
  - No metadata support
  - Requires conversion before use

- **Conversion Command:**

```
gnpy-convert-xls network_topology.xlsx -o topology.json
```

3.1.4 services.json / .xls

The services file defines path computation requests between network endpoints. It specifies source-destination pairs, routing constraints, and transmission parameters for each service. This file is optional - if not provided, GNPy can still perform full-spectrum analysis with `gnpy-transmission-example`, if provided, GNPy will compute it with `gnpy-path-request`

It's structure is as it follows:

```
{
  "path-request": [
    {
      "request-id": "service-1",
      "source": "NodeA",
      "destination": "NodeB",
      "src-tp-id": "NodeA",
      "dst-tp-id": "NodeB",
      "bidirectional": false,
      "path-constraints": {...}
    },
    ...
  ],
  "synchronization": [...]  // Optional
}
```

Path Request Parameters

Parameter	Type	Required/Optional	Description
`request-id`	string	Required	Unique identifier for this request
`source`	string	Required	Source transceiver UID
`destination`	string	Required	Destination transceiver UID
`src-tp-id`	string	Required	Source termination point (usually same as source)
`dst-tp-id`	string	Required	Destination termination point (usually same as destination)
`bidirectional`	boolean	Optional	Request bidirectional path. Default: `false`
`path-constraints`	object	Optional	Routing and transmission constraints

Table 17: Path Request Parameters

Path Constraints

Parameter	Type	Required/Optional	Description
`te-bandwidth`	object	Optional	Transmission parameters
`explicit-route-objects`	array	Optional	Include/exclude routing constraints
`path-bandwidth`	number	Optional	Requested path bandwidth (Hz)

Table 18: Path Constraint Parameters

TE Bandwidth

Parameter	Type	Required/Optional	Description
`technology`	string	Required	Transmission technology (e.g., `"flexi-grid"`)
`trx_type`	string	Optional	Transceiver type from equipment library
`trx_mode`	string	Optional	Specific transceiver mode
`spacing`	number	Optional	Channel spacing (Hz)
`effective-freq-slot`	array	Optional	Frequency slot assignment [N, M]

Table 19: TE Bandwidth Parameters

Explicit Route Objects

Parameter	Type	Required/Optional	Description
`index`	number	Required	Order in route
`num-unnum-hop`	object	Optional	Node to include/exclude
`type`	string	Required	Constraint type: `"include"` or `"exclude"`
`loose`	boolean	Optional	Loose constraint (allow other nodes between). Default: `true`

Table 20: Explicit Route Object Parameters

Usage

The services file is used by:

- **`gnpy-path-request`** for constrained path computation
- **`gnpy-transmission-example`** when `-r` flag is provided
- Path computation algorithms to determine feasible routes
- Spectrum assignment algorithms

Importance: Service requests define:

- Which paths to compute and analyze
- Transmission parameters (mode, power, spacing)
- Routing constraints (must-pass, must-avoid nodes)
- Disjunction requirements for path diversity

Excel Alternative Services can be defined in Excel format:

- **Sheet Name:** `Service`
- **Required Columns:**
  - `route id`: Request identifier
  - `Source`: Source node name
  - `Destination`: Destination node name
  - `TRX type`: Transceiver type
  - `Mode`: Transceiver mode
- **Optional Columns:**
  - `System: spacing`: Channel spacing (GHz)
  - `System: input power (dBm)`: Per-channel power
  - `System: nb of channels`: Number of channels
  - `routing: path`: Explicit path nodes
  - `routing: is loose?`: Loose/strict routing
  - `routing: disjoint from`: Disjunction requirements
- **Conversion Command:**

```
gnpy-convert-xls services.xlsx -o services.json
```

3.1.5 `sim_params.json`

The simulation parameters file configures the physical models and numerical methods used for propagation simulation. This file is optional - if not provided, GNPy uses default values optimized for typical C-band systems.

It's structure is as it follows:

```
{
  "nli_params": {
    "method": "gn_model_analytic",
    "dispersion_tolerance": 1,
    "phase_shift_tolerance": 0.1,
    "computed_channels": null,
    "computed_number_of_channels": null
  },
  "raman_params": {
    "flag": false,
    "method": "perturbative",
    "order": 1,
    "result_spatial_resolution": 10000,
```

```
    "solver_spatial_resolution": 50
  }
}
```

Parameters

NLI Solver Parameters

Parameter	Type	Required/Optional	Description
`method`	string	Optional	NLI calculation method. Default: `"gn_model_analytic"`
`dispersion_tolerance`	number	Optional	Integration step control for GGN. Default: 1
`phase_shift_tolerance`	number	Optional	Phase rotation threshold. Default: 0.1
`computed_channels`	array	Optional	Specific channel indices for NLI evaluation
`computed_number_of_channels`	number	Optional	Number of channels for NLI evaluation

Table 21: NLI Solver Parameters

Available NLI Methods:

Method	Description
`gn_model_analytic`	Fast analytical GN model (default)
`ggn_spectrally_separated`	GGN model with spectral separation (more accurate)
`gn_model_numerical`	Numerical integration of GN model

Table 22: NLI Calculation Methods

Raman Solver Parameters

Parameter	Type	Required/Optional	Description
`flag`	boolean	Optional	Enable Raman simulation. Default: `false`
`method`	string	Optional	Raman solver method. Default: `"perturbative"`
`order`	number	Optional	Perturbation order (1-4). Default: 1
`result_spatial_resolution`	number	Optional	Output resolution (m). Default: 10000
`solver_spatial_resolution`	number	Optional	Solver step size (m). Default: 50

Table 23: Raman Solver Parameters

Available Raman Methods:

Method	Description
`perturbative`	Perturbative solution (fast, up to 4th order)
`numerical`	Numerical ODE solution (slower, more accurate)

Table 24: Raman Solver Methods

Usage

The simulation parameters file is used by:

- **NLI Solver** for nonlinear interference calculation
- **Raman Solver** for stimulated Raman scattering effects
- All propagation functions in the core engine

Importance: Proper configuration affects:

- Simulation accuracy vs. computation time trade-off
- Ability to model Raman amplification
- NLI estimation precision
- Support for ultra-wideband systems

examples

Default Configuration (C-Band Systems)

```
{
  "nli_params": {
    "method": "gn_model_analytic",
    "dispersion_tolerance": 1,
    "phase_shift_tolerance": 0.1
  },
  "raman_params": {
    "flag": false
  }
}
```

High-Accuracy Configuration with Raman

```
{
  "nli_params": {
    "method": "ggn_spectrally_separated",
    "dispersion_tolerance": 0.5,
    "phase_shift_tolerance": 0.05,
    "computed_channels": [0, 47, 95]
  },
  "raman_params": {
    "flag": true,
    "method": "perturbative",
    "order": 2,
    "result_spatial_resolution": 5000,
    "solver_spatial_resolution": 25
  }
}
```

Ultra-Wideband Configuration

```
{
  "nli_params": {
    "method": "ggn_spectrally_separated",
    "dispersion_tolerance": 1,
    "phase_shift_tolerance": 0.1,
    "computed_number_of_channels": 10
  },
  "raman_params": {
    "flag": true,
    "method": "perturbative",
    "order": 3,
    "result_spatial_resolution": 10000,
    "solver_spatial_resolution": 50
  }
}
```

Fast Simulation (Planning Studies)

```
{
  "nli_params": {
    "method": "gn_model_analytic",
    "dispersion_tolerance": 2,
    "phase_shift_tolerance": 0.2,
    "computed_number_of_channels": 3
  },
  "raman_params": {
    "flag": false
  }
}
```

Configuration Trade-offs

Configuration	Accuracy	Speed	Use Case
Default (GN, no Raman)	Medium	Fast	Initial planning
GGN, no Raman	High	Medium	Detailed C-band design
GN with Raman	Medium	Medium	Raman systems
GGN with Raman	Very High	Slow	Ultra-wideband validation

Table 25: Simulation Configuration Trade-offs

3.2 Network Model Building

3.2.1 Overview

The network model building phase transforms input configuration files into operational data structures that GNPy uses for simulation. This process involves three core functions that work sequentially to create a complete, validated, and designed network model.

The network building process follows this sequence:

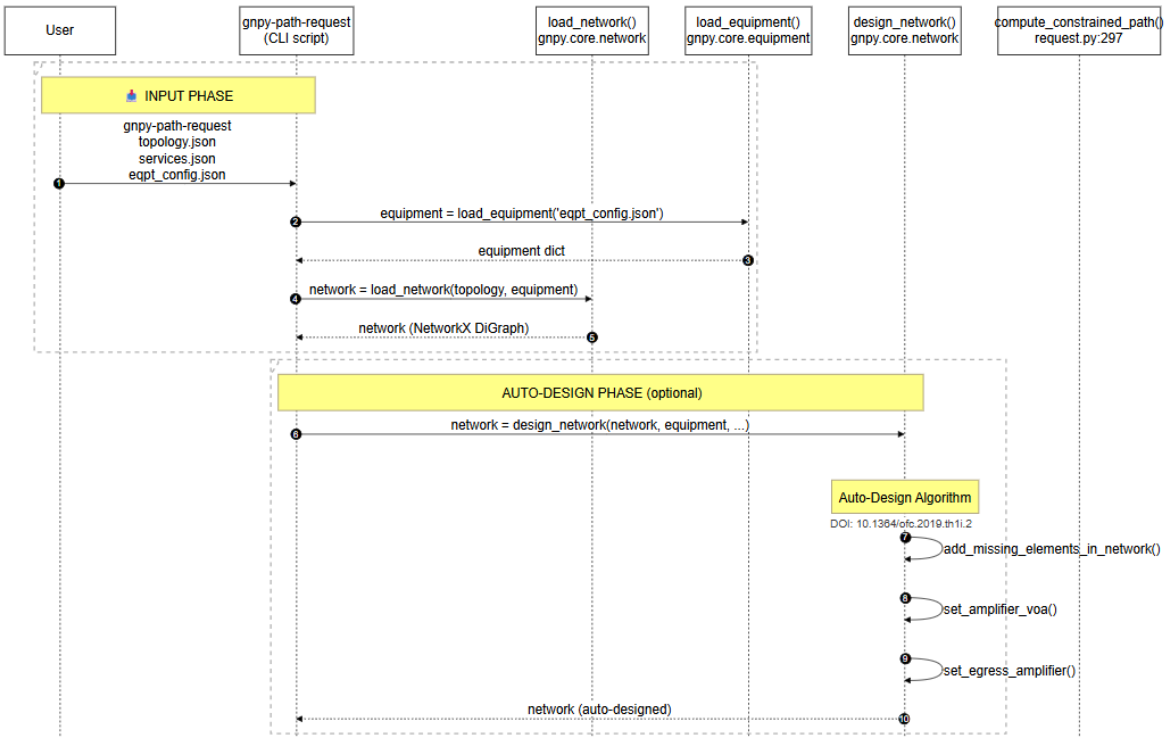

Figure 5: Input-Network Building Flow

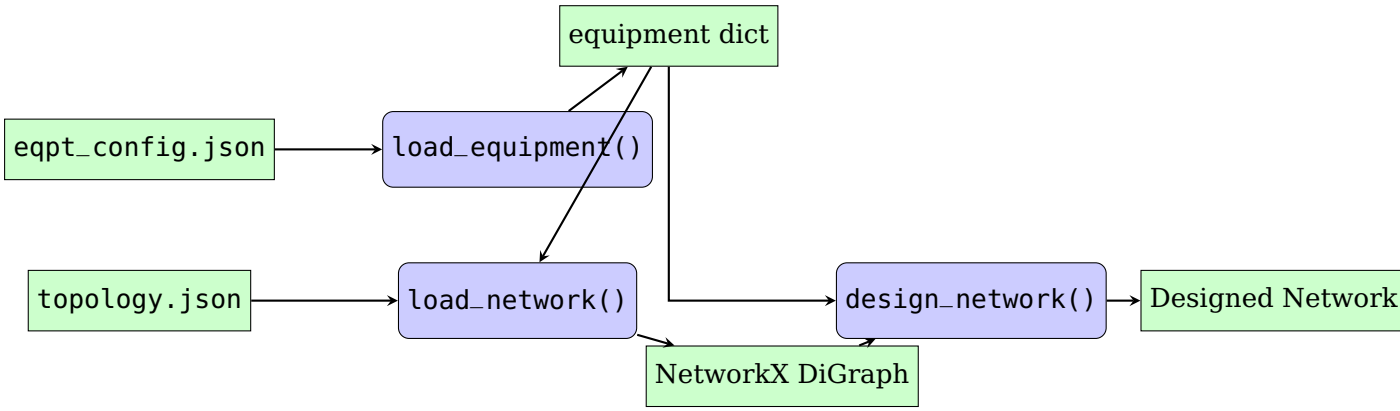

Figure 6: Network Model Building Data Flow

Relationship Between Functions

1. **`load_equipment()`**: Parses equipment configuration and creates Python objects for each equipment type
2. **`load_network()`**: Uses equipment dictionary to instantiate network elements and build graph structure
3. **`design_network()`**: Auto-places amplifiers, sets gains, and optimizes power levels

Data Flow Summary

`JSON files` → `equipment dict` → `NetworkX DiGraph` → `Designed Network`

OpenROADM Workflow Alignment This GNPy workflow aligns with OpenROADM's network discovery and planning:

GNPy Stage	OpenROADM Equivalent
`load_equipment()`	Operational Mode Catalog loading
`load_network()`	Network Model topology discovery
`design_network()`	Planning tools + auto-provisioning

Table 26: GNPy to OpenROADM Workflow Mapping

3.2.2 gnpy.core.equipment.load_equipment()

Loads equipment library from JSON configuration and creates Python objects for all equipment types. This function validates parameters, handles advanced EDFA configurations, and builds the equipment dictionary that serves as the foundation for network instantiation.

Inputs

Parameter	Type	Description
`filename`	Path	Path to `eqpt_config.json`
`extra_configs`	Dict[str, Dict]	Advanced EDFA configuration files (optional)

Table 27: load_equipment() Input Parameters

Process The function follows this workflow:

```
def load_equipment(filename, extra_configs):
    # 1. Load JSON file
    json_data = load_gnpy_json(filename)

    # 2. Build equipment dictionary
    equipment = _equipment_from_json(json_data, extra_configs)

    # 3. Validate and update
    #    - Check fiber vs. RamanFiber consistency
    #    - Update dual-stage amplifiers
    #    - Update multi-band amplifiers
    #    - Validate ROADM restrictions
    #    - Ensure 'default' SI exists

    return equipment
```

Listing 1: load_equipment() Workflow

Detailed Steps:

1. **Parse JSON Structure**

- Read `eqpt_config.json`
- Validate JSON syntax
- Extract main sections: Edfa, Fiber, RamanFiber, Span, Roadm, SI, Transceiver

2. **Create Equipment Objects**
   - For each section and each `type_variety`:
     - `Edfa` → `Amp.from_json()` object
     - `Fiber` → `Fiber()` object
     - `RamanFiber` → `RamanFiber()` object
     - `Span` → `Span()` object
     - `Roadm` → `Roadm()` object
     - `SI` → `SI()` object
     - `Transceiver` → `Transceiver()` object
   - Load advanced EDFA configurations if specified
   - Apply NF models, gain ripple, dynamic gain tilt
3. **Validation and Sanity Checks**
   - `_check_fiber_vs_raman_fiber()`: Ensure no conflicts
   - `_update_dual_stage()`: Link preamp/booster for dual-stage amplifiers
   - `_update_band()`: Configure multi-band amplifiers
   - `_roadm_restrictions_sanity_check()`: Validate ROADM restrictions
   - `_si_sanity_check()`: Ensure 'default' SI entry exists

Outputs Equipment Dictionary Structure:

```
equipment = {
    'Edfa': {
        'std_medium_gain': Amp object,
        'std_low_gain': Amp object,
        ...
    },
    'Fiber': {
        'SSMF': Fiber object,
        'LEAF': Fiber object,
        ...
    },
    'RamanFiber': {
        'SSMF': RamanFiber object,
        ...
    },
    'Span': {
        'default': Span object
    },
    'Roadm': {
        'default': Roadm object,
        'custom_roadm': Roadm object,
```

```
        ...
    },
    'SI': {
        'default': SI object,  # MUST exist
        ...
    },
    'Transceiver': {
        'vendorA_xcvr': Transceiver object,
        ...
    }
}
```

Listing 2: Equipment Dictionary Structure

Key Data Structures Type Mapping (variety_list):

The `type_variety` field in JSON maps to equipment dictionary keys:

```
// In eqpt_config.json
{
  "type_variety": "std_medium_gain",  // This becomes dict key
  "type_def": "variable_gain",
  "gain_flatmax": 26,
  ...
}

// Access in Python
amp = equipment['Edfa']['std_medium_gain']
```

Usage in GNPy Called by:

- `gnpy-path-request` (CLI)
- `gnpy-transmission-example` (CLI)
- `load_eqpt_topo_from_json()` (helper function)
- All API-based workflows

Used by:

- `load_network()`: To instantiate network elements
- `design_network()`: To select amplifiers for auto-placement
- Element constructors: To access equipment specifications

OpenROADM Mapping The equipment dictionary corresponds to OpenROADM's Operational Mode Catalog:

GNPy Equipment	OpenROADM Equivalent
`Edfa[type_variety]`	ILA/preamp/booster operational modes
`Fiber[type_variety]`	Span characteristics + ITU fiber types
`Transceiver[type_variety]`	`specific-operational-modes` catalog
`Roadm[type_variety]`	ROADM operational mode impairments
`SI['default']`	Default transmission parameters

Table 28: Equipment Dictionary to OpenROADM Mapping

3.2.3 `gnpy.core.equipment load_equipment()`

Parses network topology and creates a NetworkX directed graph (`DiGraph`) with fully instantiated network elements. This function uses the equipment dictionary to create proper element objects with correct parameters.

Inputs

Parameter	Type	Description
`filename`	Path	Path to `topology.json` or `.xls`
`equipment`	Dict	Equipment dictionary from `load_equipment()`

Table 29: load_network() Input Parameters

Process The function workflow:

```python
def load_network(filename, equipment):
    # 1. Load topology file
    if filename.suffix in ('.xls', '.xlsx'):
        json_data = xls_to_json_data(filename)
    else:
        json_data = load_gnpy_json(filename)

    # 2. Build network graph
    network = network_from_json(json_data, equipment)

    return network
```

Listing 3: load_network() Workflow

Detailed Steps (network_from_json):

1. **Initialize NetworkX DiGraph**

```
network = DiGraph()

```

2. **Create Network Elements**

   For each element in `topology['elements']`:

   - Extract `uid`, `type`, `type_variety`, `params`, `metadata`
   - Lookup `type_variety` in equipment dictionary
   - Instantiate appropriate element class:

```
if type == 'Edfa':
    node = elements.Edfa(uid=uid,
                         params=equipment['Edfa'][type_variety],
                         operational=operational)
elif type == 'Fiber':
    node = elements.Fiber(uid=uid,
                          params=equipment['Fiber'][type_variety],
                          **params)
# ... similar for other types

```

   - Add node to graph: `network.add_node(node)`

3. **Create Connections**

   For each connection in `topology['connections']`:

```
from_node = find_node_by_uid(network, connection['from_node'])
to_node = find_node_by_uid(network, connection['to_node'])
network.add_edge(from_node, to_node)

```

4. **Validate Graph**

   - Check all connections reference existing nodes
   - Verify graph is connected
   - Ensure transceivers are present (endpoints)

Outputs NetworkX DiGraph:

The output is a directed graph where:

- **Nodes**: Network element objects (Transceiver, Fiber, Edfa, Roadm, etc.)
- **Edges**: Directed connections between elements
- **Node Attributes**: All element properties (`params`, `operational`, `metadata`)

```
# Example graph structure
network = DiGraph()

# Nodes are element objects
for node in network.nodes():
    print(f"UID: {node.uid}")
    print(f"Type: {type(node).__name__}")
    print(f"Params: {node.params}")

# Edges define signal flow
for u, v in network.edges():
    print(f"{u.uid} -> {v.uid}")
```

Listing 4: NetworkX DiGraph Structure

Key Data Structures Node Attributes:

Each node in the graph has:

Attribute	Type	Description
`uid`	string	Unique identifier
`params`	object	Equipment-specific parameters
`operational`	object	Operational settings (gain, power, etc.)
`metadata`	dict	Location, coordinates, custom data
`type_variety`	string	Reference to equipment library

Table 30: Network Node Attributes

Edge Attributes:

Currently, edges store minimal information (just connectivity). Future versions may add:

- Fiber patch panel connections
- Physical routing information
- Wavelength assignments

Usage in GNPy Called by:

- `gnpy-path-request` (after `load_equipment()`)
- `gnpy-transmission-example` (after `load_equipment()`)
- `load_eqpt_topo_from_json()` (combined loader)

Provides to:

- `design_network()`: Raw network for auto-design
- `compute_constrained_path()`: Graph for path computation
- `propagate()`: Network elements for signal propagation

OpenROADM Mapping The NetworkX DiGraph corresponds to OpenROADM's Network Model:

GNPy Structure	OpenROADM Equivalent
DiGraph nodes	Network Model devices (ROADM, ILA, Xponder)
DiGraph edges	Express/Add/Drop links between degrees
Node `uid`	Node ID in network-topology
Node `params`	Device operational parameters
Node `metadata`	CLLI codes, location data
Graph connectivity	Logical link topology

Table 31: NetworkX DiGraph to OpenROADM Network Model

Key Differences:

- GNPy: Single node per network element
- OpenROADM: Hierarchical (device → shelf → circuit-pack → port)
- GNPy: Logical network only
- OpenROADM: Physical + logical with internal links

3.2.4 gnpy.core.network.load_network()

Parses network topology and creates a NetworkX directed graph (`DiGraph`) with fully instantiated network elements. This function uses the equipment dictionary to create proper element objects with correct parameters.

Inputs

Parameter	Type	Description
`filename`	Path	Path to `topology.json` or `.xls`
`equipment`	Dict	Equipment dictionary from `load_equipment()`

Table 32: load_network() Input Parameters

Process The function workflow:

```
def load_network(filename, equipment):
    # 1. Load topology file
    if filename.suffix in ('.xls', '.xlsx'):
        json_data = xls_to_json_data(filename)
    else:
        json_data = load_gnpy_json(filename)

    # 2. Build network graph
    network = network_from_json(json_data, equipment)

    return network
```

Listing 5: load_network() Workflow

Detailed Steps (network_from_json):

1. **Initialize NetworkX DiGraph**

```
network = DiGraph()
```

2. **Create Network Elements**

   For each element in `topology['elements']`:

   - Extract `uid`, `type`, `type_variety`, `params`, `metadata`
   - Lookup `type_variety` in equipment dictionary
   - Instantiate appropriate element class:

```
if type == 'Edfa':
    node = elements.Edfa(uid=uid,
                         params=equipment['Edfa'][type_variety],
                         operational=operational)
elif type == 'Fiber':
    node = elements.Fiber(uid=uid,
                          params=equipment['Fiber'][type_variety],
                          **params)
# ... similar for other types

```

- Add node to graph: `network.add_node(node)`

3. **Create Connections**

   For each connection in `topology['connections']`:

```
from_node = find_node_by_uid(network, connection['from_node'])
to_node = find_node_by_uid(network, connection['to_node'])
network.add_edge(from_node, to_node)

```

4. **Validate Graph**

   - Check all connections reference existing nodes
   - Verify graph is connected
   - Ensure transceivers are present (endpoints)

Outputs NetworkX DiGraph:

The output is a directed graph where:

- **Nodes**: Network element objects (Transceiver, Fiber, Edfa, Roadm, etc.)
- **Edges**: Directed connections between elements
- **Node Attributes**: All element properties (`params`, `operational`, `metadata`)

```
# Example graph structure
network = DiGraph()

# Nodes are element objects
for node in network.nodes():
    print(f"UID: {node.uid}")
    print(f"Type: {type(node).__name__}")
    print(f"Params: {node.params}")

# Edges define signal flow
for u, v in network.edges():
    print(f"{u.uid} -> {v.uid}")
```

Listing 6: NetworkX DiGraph Structure

Key Data Structures Node Attributes:

Each node in the graph has:

Attribute	Type	Description
`uid`	string	Unique identifier
`params`	object	Equipment-specific parameters
`operational`	object	Operational settings (gain, power, etc.)
`metadata`	dict	Location, coordinates, custom data
`type_variety`	string	Reference to equipment library

Table 33: Network Node Attributes

Edge Attributes:

Currently, edges store minimal information (just connectivity). Future versions may add:

- Fiber patch panel connections
- Physical routing information
- Wavelength assignments

Usage in GNPy Called by:

- `gnpy-path-request` (after `load_equipment()`)
- `gnpy-transmission-example` (after `load_equipment()`)
- `load_eqpt_topo_from_json()` (combined loader)

Provides to:

- `design_network()`: Raw network for auto-design
- `compute_constrained_path()`: Graph for path computation
- `propagate()`: Network elements for signal propagation

OpenROADM Mapping The NetworkX DiGraph corresponds to OpenROADM's Network Model:

GNPy Structure	OpenROADM Equivalent
DiGraph nodes	Network Model devices (ROADM, ILA, Xponder)
DiGraph edges	Express/Add/Drop links between degrees
Node `uid`	Node ID in network-topology
Node `params`	Device operational parameters
Node `metadata`	CLLI codes, location data
Graph connectivity	Logical link topology

Table 34: NetworkX DiGraph to OpenROADM Network Model

Key Differences:

- GNPy: Single node per network element

- OpenROADM: Hierarchical (device → shelf → circuit-pack → port)
- GNPy: Logical network only
- OpenROADM: Physical + logical with internal links

3.2.5 gnpy.core.network.design_network()

Automatically designs the network by placing missing amplifiers, setting amplifier gains and power levels, and optimizing for target performance. This function implements the LOGO (Local Optimization, Global Optimization) algorithm referenced in TIP/GNPy publications.

Inputs

Parameter	Type	Description
`reference_channel`	SpectralInformation	Reference channel for design (number of channels, power)
`network`	DiGraph	Network graph from `load_network()`
`equipment`	Dict	Equipment dictionary
`set_connector_losses`	bool	Whether to set fiber connector losses (default: True)
`verbose`	bool	Enable warning messages (default: True)

Table 35: design_network() Input Parameters

Process The function implements a comprehensive auto-design workflow:

```
def design_network(reference_channel, network, equipment):
    # 1. Add missing elements
    add_missing_elements_in_network(network, equipment)

    # 2. Set fiber attributes
    add_missing_fiber_attributes(network, equipment)

    # 3. Configure network
    build_network(network, equipment, reference_channel)
```

Listing 7: design_network() High-Level Workflow

Detailed Steps:

1. **Add Missing Elements (`add_missing_elements_in_network`)**

   [label=)]**Split Long Fibers**

   (a) • For each fiber > `max_length`:
   - Calculate optimal split points
   - Create multiple fiber segments
   - Insert amplifiers between segments

- Target length: 50-90 km (typical)
- Minimum length: `padding / 0.2 × 1000` meters

(b) **Add ROADM Preamplifiers**

- For each ROADM:
  - Check if preamplifier exists on ingress
  - If missing, insert appropriate amplifier type
  - Consider ROADM restrictions

(c) **Add ROADM Boosters**

- For each ROADM:
  - Check if booster exists on egress
  - If missing, insert appropriate amplifier type
  - Consider ROADM restrictions and design bands

(d) **Add Inline Amplifiers**

- For each fiber span:
  - Estimate span loss
  - If loss > amplifier capability, insert ILA
  - Select amplifier from allowed types

2. **Set Fiber Attributes (`add_missing_fiber_attributes`)**

[label=)]**Add Connector Losses**

(a)
- Set `con_in` and `con_out` from Span configuration
- Add EOL (End-of-Life) margin
- Default values from `equipment['Span']['default']`

(b) **Add Fiber Padding**

- For spans < `padding` loss:
  - Add attenuation at input (`att_in`)
  - Ensure minimum span loss for amplifier operation
  - Record as `design_span_loss`

3. **Build Network Configuration (`build_network`)**

[label=)]**Set ROADM Equalization Targets**

(a)
- For each ROADM:
  - Set reference carrier power
  - Configure per-degree targets
  - Set design bands (C, L, C+L)

- Apply `target_pch_out_db`, `target_psd_out_mWperGHz`, or `target_out_mWperSlotWidth`

(b) **Set Amplifier Gains (LOGO Algorithm)**

For each OMS (Optical Multiplex Section) from ROADM/Transceiver:

- **Estimate Span Loss:**

$$L_{\text{span}} = L_{\text{fiber}} + L_{\text{con\_in}} + L_{\text{con\_out}} + L_{\text{padding}}$$

- **Set Target Gain:**

$$G_{\text{target}} = L_{\text{span}}$$

- **Calculate Delta Power (if power_mode = True):**

$$\Delta P = P_{\text{target}} - P_{\text{ref}}$$

where P_{ref} is from Span configuration

- **Set Amplifier Parameters:**
  - `effective_gain` = G_{target}
  - `delta_p` = ΔP (if power mode)
  - `tilt_target` = 0 (default, can be customized)
- **Optimize Output VOA:**
  - If `out_voa_auto` = True:
    - Maximize gain within `p_max` limit
    - Set VOA to attenuate excess
    - Apply `voa_margin` and `voa_step`
- **Handle Raman Fibers:**
  - Estimate Raman gain contribution
  - Adjust EDFA gain accordingly
  - Record as `estimated_gain`

(c) **Set ROADM Input Powers**

- Calculate expected power at ROADM input
- Set for feasibility checking
- Configure internal paths (express, add, drop)

(d) **Set Fiber Input Powers**

- Propagate power from amplifiers
- Record for NLI calculation
- Validate against limits

Outputs The function modifies the network DiGraph in-place, adding:

Addition	Description
New amplifier nodes	Auto-placed EDFAs at required locations
Amplifier `effective_gain`	Computed gain for each amplifier
Amplifier `delta_p`	Power adjustment (if power mode enabled)
Amplifier `out_voa`	Output VOA setting
Amplifier `target_pch_out_dbm`	Target output power per channel
Fiber `design_span_loss`	Computed span loss including padding
ROADM `per_degree_design_bands`	Frequency bands per degree
ROADM `ref_carrier_power`	Reference power for equalization

Table 36: design_network() Output Modifications

LOGO Algorithm Details The LOGO (Local Optimization, Global Optimization) algorithm is detailed in the OFC 2019 paper:

Augé, J., Curri, V., Le Rouzic, E., "Open Design for Multi-Vendor Optical Networks," OFC 2019

Key Steps:

1. **Placement Strategy**
   - **Target:** Maintain span loss within amplifier gain range
   - **Method:** Insert ILA when span loss > max_loss
   - **Split:** Divide long spans at optical limits
   - **Constraints:** Consider equipment restrictions and design bands
2. **Set Gains (Equation 4 from paper)**

$$G_i = L_{\text{span},i} + \Delta P_i \tag{1}$$

   where:
   - G_i = gain of amplifier i
   - $L_{\text{span},i}$ = loss of span i
   - ΔP_i = power adjustment for optimization
3. **Fiber Loss Estimation**

$$L_{\text{fiber}} = \alpha \cdot L + L_{\text{conn,in}} + L_{\text{conn,out}} + L_{\text{padding}} \tag{2}$$

   where:
   - α = attenuation coefficient (dB/km)
   - L = fiber length (km)
   - L_{conn} = connector losses (dB)
   - L_{padding} = minimum loss padding (dB)

4. **Split Long Spans**
    - If $L > L_{\max}$:
        - Calculate $n = \lceil L/L_{\text{target}} \rceil$ segments
        - Create n fiber nodes of length L/n
        - Insert $n - 1$ inline amplifiers

Usage in GNPy Called by:

- `gnpy-transmission-example` (always)
- `gnpy-path-request` (optional, with `-design` flag)
- `designed_network()` wrapper function

Requires:

- Valid NetworkX DiGraph from `load_network()`
- Equipment dictionary with Span parameters
- Reference channel (number of channels, power)

Produces:

- Fully designed network ready for propagation
- `auto_designed_topology.json` (if saved)

OpenROADM Alignment The auto-design process aligns with OpenROADM provisioning workflows:

GNPy Auto-Design	OpenROADM Equivalent
Amplifier placement	Planning tool recommendations → `initially-planned-nodes`
Set gains	Computed gains → `initially-planned-gain`
Set target powers	Power targets → `egress-average-channel-power`
Set tilts	Tilt settings → `initially-planned-tilt`
Output VOA	VOA attenuation → `out-voa-att`
Design span loss	Reference loss → `span-loss-transmit/receive`

Table 37: Auto-Design to OpenROADM Provisioning Mapping

Integration Workflow:

1. **GNPy Design Phase:**
    - Run `design_network()` offline
    - Validate amplifier placements
    - Optimize power levels

2. **Export to OpenROADM:**
   - Convert amplifier gains → `initially-planned-gain`
   - Convert power targets → `egress-average-channel-power`
   - Map to OpenROADM node IDs
3. **Provision via OpenROADM:**
   - SDN controller reads planned values
   - Provisions devices
   - Monitors telemetry
4. **Validate:**
   - Compare measured values vs. GNPy predictions
   - Adjust if needed
   - Update equipment library

Power Mode vs. Fixed Power The design behavior depends on `power_mode` setting in Span configuration:

Setting	power_mode = True	power_mode = False
Amplifier gain	$G = L_{\text{span}} + \Delta P$	$G = L_{\text{span}}$
Delta power	Optimized per span	Set to None
Target power	Variable	Fixed
VOA usage	Active optimization	Fixed or None
Use case	Long-haul optimization	Metro/fixed design

Table 38: Power Mode Comparison

Design Validation After `design_network()`, verify:

- All amplifier gains within `[gain_min, gain_flatmax]`
- All amplifier output powers < `p_max`
- No warnings about out-of-range parameters
- Fiber input powers appropriate for NLI calculation
- ROADM equalization targets achievable

3.2.6 Summary: Complete Workflow

The three functions work together to transform configuration files into a simulation-ready network:

```
1. load_equipment('eqpt_config.json')
   → equipment = {
       'Edfa': {'std_medium_gain': Amp, ...},
       'Fiber': {'SSMF': Fiber, ...},
       ...
     }

2. load_network('topology.json', equipment)
   → network = DiGraph with nodes:
       [Transceiver('Paris'),
        Fiber('Fiber1', params=equipment['Fiber']['SSMF']),
        Edfa('Amp1', params=equipment['Edfa']['std_medium_gain']),
        ...]

3. design_network(ref_channel, network, equipment)
   → network (modified):
       - Added missing amplifiers
       - Set all gains and powers
       - Configured ROADMs
       - Ready for propagation
```

Figure 7: Complete Network Building Workflow

Next Step: The designed network is used by execution paths (`gnpy-path-request`, `gnpy-transmission-e` for path computation and signal propagation.

3.3 GNPy Execution Paths

3.3.1 Overview

GNPy provides two primary execution paths for network simulation and analysis:

Execution Path	Primary Function	Use Case
`gnpy-transmission-example`	Full spectrum propagation	Physical layer validation
`gnpy-path-request`	Path computation & feasibility	Service provisioning

Table 39: GNPy Execution Paths

Both execution paths share the same core propagation engine but differ fundamentally in their approach:

- **gnpy-transmission-example**: Propagates a full spectrum (default 96 channels) through the entire network, providing comprehensive physical layer characterization
- **gnpy-path-request**: Computes specific paths for service requests, assigns spectrum, and validates per-service feasibility

---

3.3.2 gnpy-transmission-example

Introduction The `gnpy-transmission-example` tool performs full-spectrum propagation simulation through an entire optical network. Unlike `gnpy-path-request` which focuses

on individual service paths, `gnpy-transmission-example` simulates the transmission of a complete C-band spectrum (or custom spectrum definition) through every network element, providing a comprehensive view of physical layer performance across the entire topology.

This tool is essential for:

- Network-wide physical layer characterization
- Amplifier and ROADM design validation
- Performance prediction for deployed networks
- Baseline establishment for planning studies
- Equipment library validation against real measurements

`gnpy-transmission-example` serves as the foundation for understanding network behavior before individual services are provisioned, ensuring that the physical infrastructure can support the expected traffic loads.

Purpose `gnpy-transmission-example` provides comprehensive network characterization through:

1. **Full Spectrum Simulation**
   - Propagate complete C-band spectrum (default: 96 channels)
   - Model inter-channel nonlinear interference (NLI)
   - Capture realistic multi-channel loading effects
   - Support custom spectrum definitions via `initial_spectrum.json`
2. **Network-Wide Propagation**
   - Simulate all network elements simultaneously
   - Track signal degradation through every path
   - Validate amplifier cascade performance
   - Assess ROADM equalization effectiveness
3. **Physical Layer Validation**
   - Verify OSNR budgets across all links
   - Identify limiting spans and amplifiers
   - Validate power level management
   - Check chromatic dispersion accumulation
4. **Design Verification**
   - Confirm auto-design results meet targets
   - Validate manual amplifier placement
   - Assess margin distribution across network
   - Identify potential bottlenecks

Inputs Required `gnpy-transmission-example` requires the following input files:

File	Required?	Purpose	Notes
`eqpt_config.json`	Yes	Equipment library	Contains SI defaults for spectrum
`topology.json`	Yes	Network topology	Complete network definition
`initial_spectrum.json`	Optional	Custom spectrum	Overrides SI defaults
`sim_params.json`	Optional	Simulation parameters	NLI and Raman solver settings

Table 40: gnpy-transmission-example Required Inputs

Spectral Information (SI) Configuration The spectrum to be propagated is defined by the `SI` section in `eqpt_config.json` or by `initial_spectrum.json`:

Parameter	Type	Default	Description
`f_min`	number	191.35 THz	Minimum frequency (C-band start)
`f_max`	number	196.10 THz	Maximum frequency (C-band end)
`baud_rate`	number	32 GBaud	Symbol rate per channel
`spacing`	number	50 GHz	Channel spacing
`power_dbm`	number	0 dBm	Per-channel power
`roll_off`	number	0.15	Nyquist pulse shaping factor
`tx_osnr`	number	100 dB	Transmitter OSNR

Table 41: Spectral Information Parameters

Default Spectrum Configuration With default settings, `gnpy-transmission-example` simulates:

- **Frequency Range**: 191.35 to 196.10 THz (C-band)
- **Number of Channels**: 96 channels at 50 GHz spacing
- **Channel Parameters**: 32 GBaud, 0 dBm per channel
- **Total Capacity**: Approximately 3 THz of optical spectrum
- **Channel Format**: Nyquist-shaped with 15% roll-off

Custom Spectrum Definition For non-standard configurations, use `initial_spectrum.json`:

```
{
  "spectrum": [
    {
      "f_min": 191350000000000.0,
      "f_max": 196100000000000.0,
      "baud_rate": 64000000000.0,
      "spacing": 75000000000.0,
      "power_dbm": -1.0,
      "roll_off": 0.15,
```

```
      "tx_osnr": 100.0,
      "delta_pdb": 0.0
    }
  ]
}
```

Listing 8: initial_spectrum.json Example

This example configures:

- 64 GBaud channels (higher capacity)
- 75 GHz spacing (wider guard bands)
- -1 dBm per channel power (lower total power)

Process Overview agregar foco en propagation CORE (explicada luego) `gnpy-transmission-example` follows this execution flow:

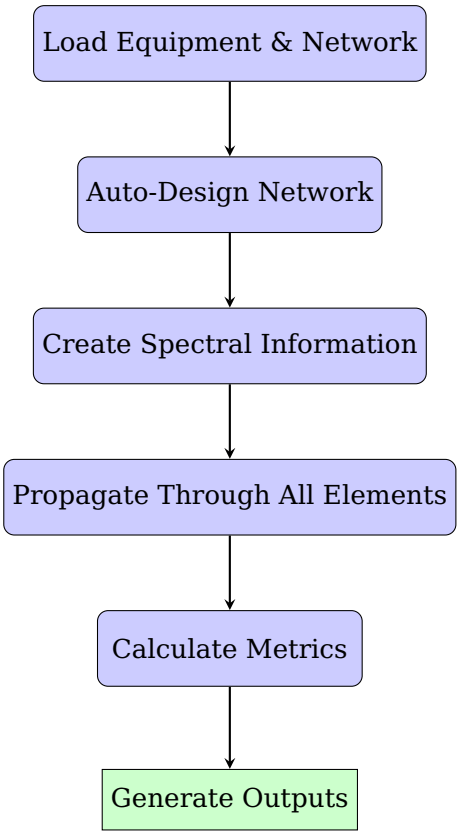

Figure 8: gnpy-transmission-example Workflow

```
# Load equipment library
equipment = load_equipment('eqpt_config.json')

# Load network topology
network = load_network('topology.json', equipment)

# Result: NetworkX DiGraph with all nodes and links
```

Listing 9: Network Loading

```
# Get spectral information for design reference
spectral_info = equipment['SI']['default']

# Design network (always performed by transmission-example)
network = design_network(
    network=network,
    equipment=equipment,
    pref_ch_db=spectral_info.power_dbm,
    pref_total_db=calculate_total_power(spectral_info)
)

# Result: Network with placed amplifiers and set gains
```

Listing 10: Network Auto-Design

The auto-design phase:

- Places inline amplifiers where needed
- Sets all amplifier gains based on span losses
- Configures ROADM equalization targets
- Optimizes power levels across network

```
# Load spectrum definition
if initial_spectrum_file:
    spectrum_config = load_json(initial_spectrum_file)
else:
    spectrum_config = equipment['SI']['default']

# Create spectral information object
spectral_info = create_arbitrary_spectral_information(
    f_min=spectrum_config.f_min,
    f_max=spectrum_config.f_max,
    baud_rate=spectrum_config.baud_rate,
    spacing=spectrum_config.spacing,
    power_dbm=spectrum_config.power_dbm,
    roll_off=spectrum_config.roll_off,
    tx_osnr=spectrum_config.tx_osnr
)

# Result: SpectralInformation with 96 channels (default)
# Each channel has:
#   - Center frequency
#   - Power (W)
#   - Baud rate
#   - OSNR (linear)
#   - NLI (initially 0)
```

Listing 11: Spectrum Creation

4. Full Spectrum Propagation This is the core operation that differentiates `gnpy-transmission-examp` from `gnpy-path-request`:

```
# Propagate through entire network
# For each path from every transceiver to every other transceiver
for source_trx in transceivers:
    # Initialize signal at source
    current_spectral_info = spectral_info.copy()
    current_spectral_info = source_trx.propagate(current_spectral_info)

    # Propagate through network using breadth-first traversal
    for path in all_paths_from_source(network, source_trx):
        temp_spectral_info = current_spectral_info.copy()

        for element in path:
            if isinstance(element, Fiber):
                temp_spectral_info = element.propagate(
                    temp_spectral_info,
                    degree=get_degree(element, path),
                    equipment=equipment
                )
                # Fiber adds: attenuation, NLI, CD, PMD

            elif isinstance(element, Edfa):
                temp_spectral_info = element.propagate(
                    temp_spectral_info,
                    degree=get_degree(element, path)
                )
                # EDFA adds: gain, ASE noise

            elif isinstance(element, Roadm):
                temp_spectral_info = element.propagate(
                    temp_spectral_info,
                    degree=get_degree(element, path)
                )
                # ROADM adds: loss, equalization, ASE

            elif isinstance(element, Transceiver):
                temp_spectral_info = element.propagate(
                    temp_spectral_info
                )
                # Destination transceiver

        # Store results for this path
        store_path_results(source_trx, path[-1], temp_spectral_info)
```

Listing 12: Full Network Propagation

Key Characteristics of Full Spectrum Propagation:

- **All Channels Simulated**: 96 channels (default) propagated simultaneously
- **NLI Interaction**: Nonlinear effects calculated considering all channels
- **Network-Wide**: Every possible path through topology is evaluated
- **Realistic Loading**: Captures multi-channel effects on amplifiers and fibers

5. Calculate Metrics After propagation, comprehensive metrics are computed:

```
for path_result in all_path_results:
    for channel_idx in range(num_channels):
        # Extract channel data
        signal_power = path_result.signal[channel_idx].power
```

```
        ase_noise = path_result.ase[channel_idx].power
        nli_noise = path_result.nli[channel_idx].power

        # Calculate OSNR (signal vs. ASE only)
        osnr = signal_power / ase_noise
        osnr_db = 10 * log10(osnr)

        # Calculate GSNR (signal vs. ASE + NLI)
        gsnr = signal_power / (ase_noise + nli_noise)
        gsnr_db = 10 * log10(gsnr)

        # Reference bandwidth for OSNR (12.5 GHz standard)
        osnr_0_1nm = convert_to_01nm_bandwidth(osnr_db, baud_rate)

        # Accumulate linear impairments
        total_cd = sum([elem.cd for elem in path if hasattr(elem, 'cd')])
        total_pmd = sqrt(sum([elem.pmd**2 for elem in path if hasattr(elem, 'pmd')]))
        total_pdl = sum([elem.pdl for elem in path if hasattr(elem, 'pdl')])

        # Store per-channel metrics
        metrics[channel_idx] = {
            'frequency': channel_frequency[channel_idx],
            'power_dbm': watts_to_dbm(signal_power),
            'osnr_db': osnr_db,
            'gsnr_db': gsnr_db,
            'osnr_0.1nm': osnr_0_1nm,
            'cd_ps_nm': total_cd,
            'pmd_ps': total_pmd,
            'pdl_db': total_pdl,
            'latency_ms': calculate_latency(path)
        }
```

Listing 13: Metrics Calculation

6. Generate Outputs Multiple output formats are created:

- Console summary with key statistics
- CSV files with per-channel results
- Optional JSON output with complete propagation data
- Optional auto-designed topology export

Outputs Generated `gnpy-transmission-example` produces comprehensive output data:

Output	Format	Contents
Console Summary	Text	Network-wide statistics and warnings
`propagated_results.csv`	CSV	Per-channel metrics for all paths
`network_transmission.json`	JSON	Complete propagation data (optional)
`auto_designed_topology.json`	JSON	Designed network with placed amplifiers

Table 42: gnpy-transmission-example Outputs

```
Network summary:
  Number of nodes: 15
```

```
  Number of links: 22
  Number of transceivers: 15
  Total fiber length: 2,450 km
  Number of amplifiers: 48 (32 auto-placed)

Propagation statistics (96 channels, 191.35-196.10 THz):
  Average OSNR: 22.3 dB (range: 18.5 - 26.1 dB)
  Average GSNR: 17.8 dB (range: 14.2 - 21.5 dB)
  Worst path: Paris -> Marseille (GSNR = 14.2 dB)
  Best path: Lyon -> Grenoble (GSNR = 21.5 dB)

Warnings:
  - Link Paris-Lyon: High NLI contribution (3.2 dB OSNR penalty)
  - Amplifier Amp_15: Operating near p_max limit
  - Path Paris-Marseille: GSNR margin < 2 dB for some channels

Design summary:
  Auto-placed amplifiers: 32
  Power mode: enabled
  Average gain: 18.5 dB (range: 15.2 - 23.1 dB)
  Average output power: 19.5 dBm
```

Listing 14: Console Summary

CSV Output Structure The `propagated_results.csv` contains detailed per-channel, per-path data:

```
source,destination,channel_number,frequency_THz,power_dbm,osnr_db,gsnr_db,osnr_0.1nm,cd_ps_nm
    ,pmd_ps,pdl_db,path_length_km
Paris,Lyon,1,191.35,-2.1,23.5,18.2,25.8,8450,12.3,2.1,450.2
Paris,Lyon,2,191.40,-2.0,23.6,18.3,25.9,8525,12.4,2.1,450.2
Paris,Lyon,3,191.45,-1.9,23.7,18.4,26.0,8600,12.5,2.1,450.2
...
Paris,Marseille,1,191.35,-3.5,19.2,14.2,21.5,14520,18.7,3.2,780.5
...
```

Listing 15: propagated_results.csv Format

Each row contains:

- Path endpoints (source, destination)
- Channel identification (number, frequency)
- Signal metrics (power, OSNR, GSNR)
- Impairments (CD, PMD, PDL)
- Path characteristics (length, latency)

JSON Output Structure The optional `network_transmission.json` provides complete data:

```
{
  "network_info": {
    "topology_file": "topology.json",
    "num_nodes": 15,
    "num_links": 22,
```

```
    "total_fiber_length_km": 2450.3
  },
  "spectrum_info": {
    "f_min_THz": 191.35,
    "f_max_THz": 196.10,
    "num_channels": 96,
    "spacing_GHz": 50,
    "baud_rate_GBaud": 32,
    "power_per_channel_dbm": 0
  },
  "paths": [
    {
      "source": "Paris",
      "destination": "Lyon",
      "route": ["Paris", "Orleans", "Tours", "Lyon"],
      "length_km": 450.2,
      "num_amplifiers": 8,
      "channels": [
        {
          "channel_number": 1,
          "frequency_THz": 191.35,
          "power_dbm": -2.1,
          "osnr_db": 23.5,
          "gsnr_db": 18.2,
          "cd_ps_nm": 8450,
          "pmd_ps": 12.3,
          "element_by_element": [
            {
              "element": "Paris_TX",
              "type": "Transceiver",
              "power_out_dbm": 0.0,
              "osnr_db": 100.0
            },
            {
              "element": "Fiber_1",
              "type": "Fiber",
              "length_km": 80.5,
              "loss_db": 16.1,
              "power_out_dbm": -16.1,
              "cd_ps_nm": 1367,
              "nli_added_dB": 0.3
            },
            ...
          ]
        },
        ...
      ]
    },
    ...
  ]
}
```

Listing 16: network_transmission.json Structure

Use Cases and Scenarios

Use Case 1: Network Commissioning Validation **Scenario:** Newly deployed network needs physical layer validation before accepting services.

Workflow:

1. Create GNPy model from as-built documentation
2. Run `gnpy-transmission-example` with default full spectrum
3. Compare predicted OSNR vs. measured OSNR from test equipment
4. Identify discrepancies and refine equipment models
5. Establish baseline performance metrics

Expected Output Analysis:

- All paths have OSNR > 20 dB
- No amplifiers operating outside normal range
- ROADM equalization functioning correctly
- Prediction accuracy within ±1 dB of measurements

Use Case 2: Amplifier Cascade Design Scenario: Design amplifier placement and gains for new long-haul route.

Workflow:

1. Create topology with fibers only (no amplifiers)
2. Run `gnpy-transmission-example` with `-design` flag
3. Review auto-placed amplifier locations and gains
4. Export designed topology for implementation
5. Validate design meets OSNR targets across full spectrum

Design Validation Criteria:

- All amplifier gains within [gain_min, gain_flatmax]
- Output powers below p_max
- OSNR margin > 3 dB for worst channel
- Span losses matched to amplifier capabilities

Use Case 3: Equipment Library Calibration Scenario: Calibrate GNPy equipment models against lab or field measurements.

Workflow:

1. Measure network performance with full spectrum loading
2. Model identical configuration in GNPy
3. Run `gnpy-transmission-example`
4. Compare per-channel OSNR predictions vs. measurements
5. Adjust equipment parameters (NF, gamma, dispersion) to match
6. Iterate until prediction accuracy < 0.5 dB

Calibration Parameters:

- EDFA noise figure (NF)
- Fiber nonlinearity coefficient (gamma)
- Fiber dispersion
- ROADM insertion loss and OSNR penalty

Use Case 4: Network Upgrade Planning Scenario: Evaluate impact of adding new equipment or changing fiber types.

Workflow:

1. Baseline: Run transmission-example on current network
2. Modification: Update topology with proposed changes
3. Re-run: Simulate with new configuration
4. Compare: Analyze performance delta
5. Decision: Proceed if improvement justifies cost

Example Upgrade Scenarios:

- Replace SSMF with LEAF fiber: Expect +1-2 dB GSNR improvement
- Add Raman amplification: Expect +3-5 dB OSNR improvement
- Upgrade EDFAs to lower NF models: Expect +1 dB OSNR improvement
- Replace 50 GHz ROADMs with 75 GHz: Expect reduced channel count

Use Case 5: Spectrum Planning Scenario: Determine optimal channel spacing and power for network capacity.

Workflow:

1. Create multiple `initial_spectrum.json` configurations:
    - Config A: 50 GHz spacing, 32 GBaud (96 channels)
    - Config B: 75 GHz spacing, 64 GBaud (64 channels)
    - Config C: 37.5 GHz spacing, 32 GBaud (128 channels)
2. Run `gnpy-transmission-example` for each configuration
3. Compare total capacity vs. worst-case GSNR
4. Select configuration with best capacity-reach trade-off

Comparison Matrix:

Config	Channels	Capacity (Gb/s)	Worst GSNR (dB)	Feasible?
A (50 GHz, 32G)	96	9,600	17.8	Yes
B (75 GHz, 64G)	64	12,800	19.2	Yes
C (37.5 GHz, 32G)	128	12,800	15.2	Marginal

Table 43: Spectrum Configuration Comparison

Detailed Workflow

```
def gnpy_transmission_example(topology_file, args):
    """Main execution flow for transmission example."""

    # Stage 1: Load Network Model
    print("Loading network model...")
    equipment = load_equipment(args.equipment)
    network = load_network(topology_file, equipment)

    print(f"Network loaded: {len(network.nodes)} nodes, "
          f"{len(network.edges)} links")

    # Stage 2: Auto-Design Network
    print("Auto-designing network...")

    # Get spectral information for design
    if args.initial_spectrum:
        spectrum_config = load_json(args.initial_spectrum)['spectrum'][0]
    else:
        spectrum_config = equipment['SI']['default']

    # Calculate reference power
    num_channels = calculate_num_channels(
        spectrum_config['f_min'],
        spectrum_config['f_max'],
        spectrum_config['spacing']
    )
    pref_total_db = (spectrum_config['power_dbm'] +
                     10 * log10(num_channels))

    # Design network
    network = design_network(
        network=network,
        equipment=equipment,
        pref_ch_db=spectrum_config['power_dbm'],
        pref_total_db=pref_total_db
    )

    # Optional: Save designed topology
    if args.save_design:
        save_network(network, 'auto_designed_topology.json')

    print(f"Design complete: {count_amplifiers(network)} amplifiers placed")

    # Stage 3: Create Spectral Information
    print("Creating spectral information...")
    spectral_info = create_arbitrary_spectral_information(
        f_min=spectrum_config['f_min'],
        f_max=spectrum_config['f_max'],
```

```
        baud_rate=spectrum_config['baud_rate'],
        spacing=spectrum_config['spacing'],
        power_dbm=spectrum_config['power_dbm'],
        roll_off=spectrum_config.get('roll_off', 0.15),
        tx_osnr=spectrum_config.get('tx_osnr', 100.0)
    )

    print(f"Spectrum: {len(spectral_info.frequency)} channels, "
          f"{spectral_info.frequency[0]/1e12:.2f}-"
          f"{spectral_info.frequency[-1]/1e12:.2f} THz")

    # Stage 4: Propagate Through Network
    print("Propagating through network...")

    # Get all transceiver pairs
    transceivers = [n for n in network.nodes()
                    if isinstance(n, Transceiver)]

    all_results = []
    total_paths = len(transceivers) * (len(transceivers) - 1)

    for idx, source in enumerate(transceivers):
        for dest in transceivers:
            if source == dest:
                continue

            # Find path from source to dest
            try:
                path = nx.shortest_path(network, source, dest)
            except nx.NetworkXNoPath:
                print(f"No path from {source.uid} to {dest.uid}")
                continue

            # Propagate spectral info through path
            temp_si = spectral_info.copy()

            for element in path:
                temp_si = element.propagate(
                    temp_si,
                    degree=get_degree(element, path),
                    equipment=equipment
                )

            # Calculate and store metrics
            metrics = calculate_path_metrics(temp_si, path)
            all_results.append({
                'source': source.uid,
                'destination': dest.uid,
                'path': [e.uid for e in path],
                'metrics': metrics
            })

            # Progress indication
            if (idx * len(transceivers) + idx) % 10 == 0:
                progress = (len(all_results) / total_paths) * 100
                print(f"Progress: {progress:.1f}%")

    print(f"Propagation complete: {len(all_results)} paths computed")

    # Stage 5: Calculate Statistics
```

```
    print("Calculating network statistics...")

    stats = {
        'num_paths': len(all_results),
        'num_channels': len(spectral_info.frequency),
        'avg_osnr_db': 0,
        'avg_gsnr_db': 0,
        'worst_path': None,
        'worst_gsnr_db': float('inf'),
        'best_path': None,
        'best_gsnr_db': -float('inf')
    }

    osnr_sum = 0
    gsnr_sum = 0

    for result in all_results:
        # Average across all channels
        path_osnr = mean([m['osnr_db'] for m in result['metrics']])
        path_gsnr = mean([m['gsnr_db'] for m in result['metrics']])

        osnr_sum += path_osnr
        gsnr_sum += path_gsnr

        if path_gsnr < stats['worst_gsnr_db']:
            stats['worst_gsnr_db'] = path_gsnr
            stats['worst_path'] = f"{result['source']}->{result['destination']}"

        if path_gsnr > stats['best_gsnr_db']:
            stats['best_gsnr_db'] = path_gsnr
            stats['best_path'] = f"{result['source']}->{result['destination']}"

    stats['avg_osnr_db'] = osnr_sum / len(all_results)
    stats['avg_gsnr_db'] = gsnr_sum / len(all_results)

    # Stage 6: Generate Outputs
    print("Generating outputs...")

    # Console summary
    print_summary(network, stats, spectral_info)

    # CSV output
    write_csv_results(all_results, 'propagated_results.csv')

    # Optional JSON output
    if args.json_output:
        write_json_results(all_results, network, spectrum_config,
                           'network_transmission.json')

    print("Complete!")
    return all_results, stats
```

Listing 17: Complete gnpy-transmission-example Flow

Element-by-Element Propagation Detailed view of propagation through network elements:

```
def propagate_through_path(path, spectral_info, equipment):
```

```
    """Detailed propagation through each element."""

    current_si = spectral_info.copy()
    element_results = []

    for elem in path:
        # Record state before element
        power_in = [si.power for si in current_si.signal]
        osnr_in = calculate_osnr(current_si)

        # Element-specific propagation
        if isinstance(elem, Transceiver):
            # Source transceiver: set initial OSNR
            if elem == path[0]:
                current_si = initialize_transceiver(current_si, elem)
            # Destination transceiver: compute final metrics
            else:
                current_si = finalize_transceiver(current_si, elem)

        elif isinstance(elem, Fiber):
            # Fiber propagation includes:
            # 1. Attenuation
            # 2. Chromatic dispersion
            # 3. PMD
            # 4. Nonlinear effects (NLI via GN/GGN model)
            current_si = elem.propagate(
                current_si,
                degree=get_degree(elem, path),
                equipment=equipment
            )

            # Fiber-specific metrics
            cd_accumulated = elem.params.dispersion * elem.params.length
            pmd_accumulated = (elem.params.pmd_coef *
                               sqrt(elem.params.length))

        elif isinstance(elem, Edfa):
            # EDFA propagation includes:
            # 1. Amplification (gain)
            # 2. ASE noise addition
            # 3. Gain ripple (if modeled)
            # 4. Dynamic gain tilt (if modeled)
            current_si = elem.propagate(
                current_si,
                degree=get_degree(elem, path)
            )

            # EDFA-specific metrics
            gain_actual = elem.effective_gain
            nf_actual = elem.nf
            ase_added = calculate_ase_power(gain_actual, nf_actual)

        elif isinstance(elem, Roadm):
            # ROADM propagation includes:
            # 1. Channel equalization
            # 2. Insertion loss
            # 3. PDL
            # 4. ASE from ROADM impairments
            current_si = elem.propagate(
                current_si,
```

```
                degree=get_degree(elem, path)
            )

            # ROADM-specific metrics
            loss_applied = elem.loss
            pdl_added = elem.params.pdl

        # Record state after element
        power_out = [si.power for si in current_si.signal]
        osnr_out = calculate_osnr(current_si)

        element_results.append({
            'element': elem.uid,
            'type': type(elem).__name__,
            'power_in_dbm': [watts_to_dbm(p) for p in power_in],
            'power_out_dbm': [watts_to_dbm(p) for p in power_out],
            'osnr_in_db': osnr_in,
            'osnr_out_db': osnr_out,
            'osnr_contribution_db': osnr_in - osnr_out
        })

    return current_si, element_results
```

Listing 18: Per-Element Propagation Logic

Command Line Usage Basic Syntax:

```
gnpy-transmission-example [-h] [-e EQUIPMENT] [--sim-params SIMPARAMS]
                          [-pl POWER-MODE] [-o OUTPUT]
                          [--no-insert-edfas] [-v]
                          filename
```

Command Line Options:

Option	Description
`filename`	Network topology JSON file
`-e, -equipment`	Equipment library file (default: eqpt_config.json)
`-sim-params`	Simulation parameters file for NLI/Raman
`-pl, -power-mode`	Enable/disable power mode optimization
`-o, -output`	Output file for propagated results
`-no-insert-edfas`	Skip auto-design (use existing amplifiers only)
`-v, -verbose`	Enable detailed logging (-v for warning -v -v for debug)
`-save-design`	Export auto-designed topology to JSON

Table 44: gnpy-transmission-example Command Line Options

Example Commands:

```
# Simple full spectrum propagation
gnpy-transmission-example topology.json

# With custom equipment library
gnpy-transmission-example -e custom_eqpt.json topology.json

# With simulation parameters for high-accuracy NLI
```

```
gnpy-transmission-example --sim-params sim_params.json topology.json

# Skip auto-design (use existing amplifiers)
gnpy-transmission-example --no-insert-edfas topology.json

# Save designed topology
gnpy-transmission-example --save-design topology.json

# Full verbose output
gnpy-transmission-example -v \
                          -e eqpt_config.json \
                          --sim-params sim_params.json \
                          -o results.json \
                          topology.json
```

Listing 19: Basic Transmission Simulation

Aspect	gnpy-transmission-example	gnpy-path-request
Purpose	Network characterization	Service feasibility
Spectrum	Full C-band (96 channels default)	Per-request assignment
Propagation	All paths in network	Only requested paths
NLI Calculation	Full spectrum interaction	Per-service channel
Path Selection	All source-dest combinations	Constrained paths only
Inputs	Topology + equipment	+ service requests
Outputs	Network-wide metrics	Per-service feasibility
Auto-Design	Always performed	Optional (–design flag)
Use Case	Physical layer validation	Provisioning decisions
Execution Time	$O(N^2)$ paths	O(R) requests
SDN Integration	Offline planning	Real-time path computation

Table 45: gnpy-transmission-example vs. gnpy-path-request

Comparison with gnpy-path-request When to Use Each Tool:

- **Use gnpy-transmission-example** when:
  - Commissioning new network
  - Validating amplifier design
  - Characterizing network performance
  - Calibrating equipment models
  - Planning upgrades
- **Use gnpy-path-request** when:
  - Evaluating service requests
  - Checking path feasibility
  - Assigning spectrum to services
  - Enforcing disjunction constraints
  - Integrating with SDN controller

3.3.3 gnpy-path-request

Introduction The `gnpy-path-request` tool is GNPy's primary interface for path computation and service feasibility analysis. Unlike `gnpy-transmission-example` which propagates a full spectrum through the entire network, `gnpy-path-request` processes individual service requests, computes viable paths, assigns spectrum, and validates physical layer feasibility on a per-service basis.

This tool is designed for integration with SDN controllers and orchestration platforms, providing the optical planning capabilities needed for dynamic service provisioning. It supports advanced features including:

- Path constraint enforcement (explicit routes, diversity requirements)
- Spectrum assignment with collision avoidance
- Multi-path service requests with disjunction constraints
- Transceiver mode selection based on reach requirements
- Detailed per-service feasibility reporting

Purpose `gnpy-path-request` serves as the bridge between network planning and operational provisioning. Its core functions include:

1. **Path Computation**: Calculate viable paths from source to destination considering:
    - Network topology and link availability
    - Explicit routing constraints (strict/loose)
    - Path diversity requirements (link/node disjoint)
    - Wavelength continuity constraints
2. **Spectrum Assignment**: Assign frequency slots to each request:
    - Respect channel spacing requirements
    - Avoid spectrum collisions across shared links
    - Support flexible grid allocation
    - Handle pre-assigned spectrum (if specified)
3. **Physical Layer Validation**: Verify service feasibility:
    - Propagate signals through computed paths
    - Calculate accumulated impairments (OSNR, NLI, CD, PMD)
    - Validate against transceiver requirements
    - Identify limiting factors (bottleneck elements)
4. **Mode Selection**: Choose appropriate transceiver modes:
    - Select mode based on path length and impairments
    - Optimize for capacity vs. reach trade-offs
    - Respect equipment availability constraints

Inputs Required `gnpy-path-request` requires the following input files:

File	Required?	Purpose	Notes
`eqpt_config.json`	Yes	Equipment library	Contains transceiver modes, amplifiers, fibers
`topology.json`	Yes	Network topology	Defines nodes, links, and connectivity
`services.json`	Yes	Service requests	Path demands with source/destination
`sim_params.json`	Optional	Simulation parameters	NLI and Raman solver settings

Table 46: gnpy-path-request Required Inputs

Service Request Format The `services.json` file defines the service requests to be computed. Each request contains:

Parameter	Type	Required/Optional	Description
`path-request`	array	Required	List of service requests
`request-id`	string	Required	Unique identifier for this request
`source`	string	Required	Source node name (must exist in topology)
`destination`	string	Required	Destination node name
`src-tp-id`	string	Optional	Source transceiver ID
`dst-tp-id`	string	Optional	Destination transceiver ID
`bidirectional`	boolean	Optional	Bidirectional service (default: true)
`path-constraints`	object	Optional	Routing and spectrum constraints

Table 47: Service Request Parameters

Path Constraints The `path-constraints` object supports:

Parameter	Type	Required/Optional	Description
`te-bandwidth`	object	Optional	Requested bandwidth specification
`explicit-route-objects`	array	Optional	Explicit path through specific nodes
`loose-hop`	boolean	Optional	Allow intermediate hops (default: true)
`disjointness`	string	Optional	Disjunction type: `link`, `node`, `srlg`
`disjoint-from`	array	Optional	List of request IDs to be disjoint from

Parameter	Type	Required/Optional	Description
`path-metric-type`	string	Optional	Routing metric: `hop`, `distance`, `OSNR`

Table 48: Path Constraint Parameters

```
{
  "path-request": [
    {
      "request-id": "Service_001",
      "source": "Paris",
      "destination": "Lyon",
      "src-tp-id": "Paris_TX1",
      "dst-tp-id": "Lyon_RX1",
      "bidirectional": true,
      "path-constraints": {
        "te-bandwidth": {
          "technology": "flexi-grid",
          "trx_type": "Voyager",
          "trx_mode": "mode 1",
          "spacing": 50000000000.0,
          "max-nb-of-channel": 80
        },
        "path-metric-type": "OSNR"
      }
    },
    {
      "request-id": "Service_002",
      "source": "Paris",
      "destination": "Marseille",
      "bidirectional": true,
      "path-constraints": {
        "te-bandwidth": {
          "technology": "flexi-grid",
          "trx_type": "Voyager",
          "trx_mode": null,
          "spacing": 75000000000.0
        },
        "disjointness": "link",
        "disjoint-from": ["Service_001"]
      }
    }
  ]
}
```

Listing 20: services.json Example

Process Overview The `gnpy-path-request` execution follows a three-stage workflow focused on service-specific path computation and feasibility validation:

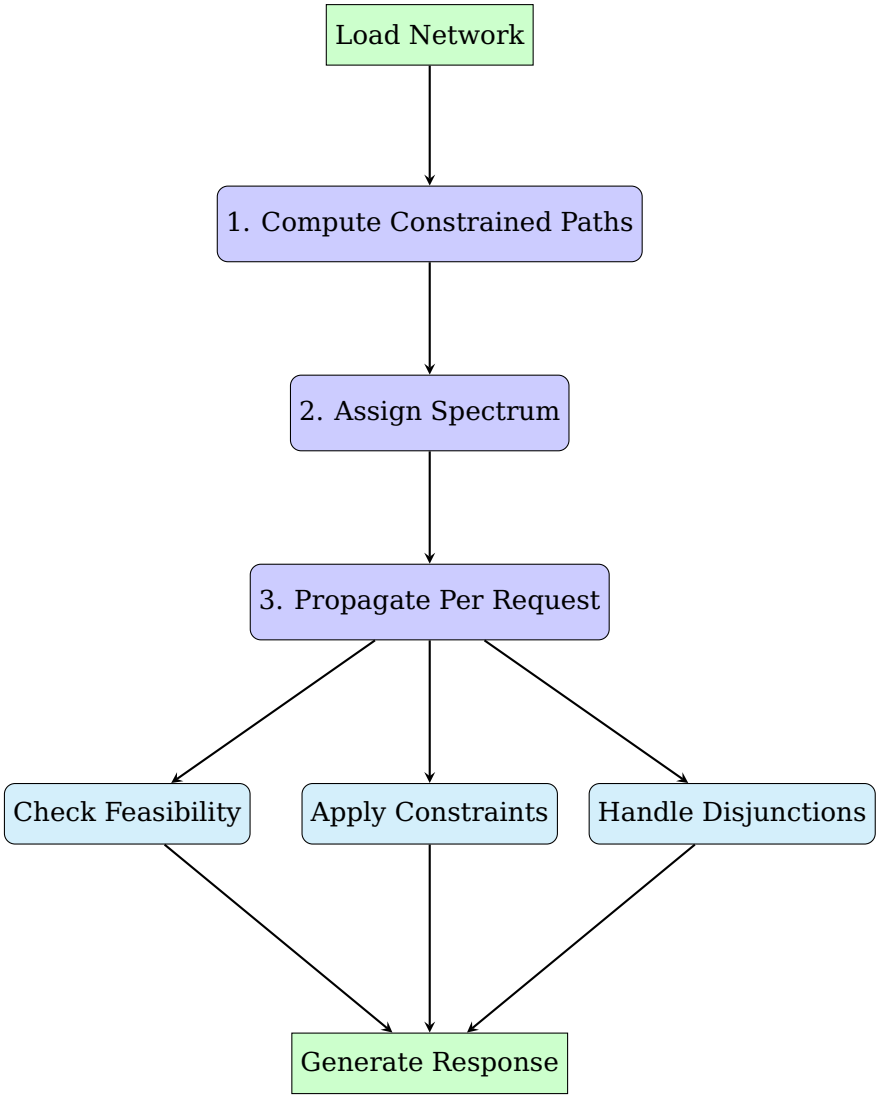

Figure 9: gnpy-path-request Workflow

Stage 1: Compute Constrained Paths

- Parse service request constraints (explicit routes, disjunctions, metrics)
- Apply routing algorithm (Dijkstra, A*, or constrained shortest path)
- Ensure path meets all hard constraints (explicit hops, diversity)
- Validate path connectivity and node/link availability

Stage 2: Assign Spectrum

- Build spectrum usage map across network
- Find available frequency slots for each request
- Allocate spectrum avoiding collisions on shared links
- Respect channel spacing and guard band requirements
- Update spectrum map with allocations

Stage 3: Propagate Per Request

- Create SpectralInformation for assigned spectrum
- Propagate through computed path (not full network)
- **Check Feasibility**: Validate OSNR, CD, PMD against transceiver requirements
- **Apply Constraints**: Ensure power levels, ROADM limits, amplifier ranges
- **Handle Disjunctions**: Verify diversity requirements satisfied
- Select transceiver mode if not specified
- Report feasibility with detailed metrics or infeasibility reason

Stage 1: Network Loading and Optional Design

1. Load equipment library using `load_equipment()`
2. Load network topology using `load_network()`
3. Optionally run `design_network()` with `-design` flag
    - Auto-place amplifiers
    - Set gains and power levels
    - Prepare network for propagation

Stage 2: Compute Constrained Paths For each service request, compute a viable path considering all constraints:

1. **Parse Constraints**
    - Extract source, destination, routing constraints
    - Identify explicit route requirements
    - Parse disjunction constraints
2. **Compute Path**
    - Use Dijkstra or A* algorithm based on metric type
    - Respect explicit route objects (strict/loose hops)
    - Ensure disjointness from specified services
    - Validate path connectivity
3. **Path Validation**
    - Verify all nodes exist in topology
    - Check link availability
    - Validate transceiver endpoints

Path Computation Algorithm:

```
def compute_path_with_constraints(network, request):
    source = request['source']
    destination = request['destination']
    constraints = request['path-constraints']

```

```
    # Handle explicit route
    if 'explicit-route-objects' in constraints:
        path = compute_explicit_path(network,
                                     constraints['explicit-route-objects'],
                                     constraints.get('loose-hop', True))
    else:
        # Compute shortest path based on metric
        metric = constraints.get('path-metric-type', 'hop')
        path = dijkstra_with_metric(network, source, destination, metric)

    # Apply disjointness constraints
    if 'disjoint-from' in constraints:
        path = ensure_disjointness(path,
                                   constraints['disjoint-from'],
                                   constraints.get('disjointness', 'link'))

    return path
```

Listing 21: Path Computation Pseudocode

Stage 3: Assign Spectrum `gnpy-path-request` assigns spectrum to avoid collisions:

1. **Build Spectrum Usage Map**
   - Track occupied frequencies per link
   - Respect channel spacing requirements
   - Consider guard bands
2. **Assign Frequencies**
   - For each request in order:
     - Find available spectrum slots
     - Allocate based on transceiver spacing
     - Update spectrum map
   - Handle pre-assigned spectrum (if specified in request)
3. **Validate Assignment**
   - Ensure no collisions on shared links
   - Verify wavelength continuity
   - Check against equipment frequency ranges

Spectrum Assignment Algorithm:

```
def assign_spectrum(path_requests, spectrum_map):
    for request in path_requests:
        path = request['computed_path']
        trx_mode = request['trx_mode']
        spacing = trx_mode['spacing']

        # Find first available slot across entire path
        f_center = find_first_fit(spectrum_map, path, spacing)

        if f_center is None:
            raise SpectrumExhaustedError(request)
```

```
        # Reserve spectrum on all links in path
        for link in path:
            spectrum_map[link].add_allocation(f_center, spacing)

        request['assigned_frequency'] = f_center

    return path_requests
```

Listing 22: Spectrum Assignment Pseudocode

Stage 4: Propagate Per Request For each request with assigned spectrum, perform detailed propagation and validation:

1. **Create Spectral Information**
   - Build `SpectralInformation` object
   - Set center frequency from spectrum assignment
   - Configure baud rate, power, roll-off from transceiver mode
2. **Propagate Through Computed Path**
   - Initialize signal at source transceiver
   - Call `propagate()` for each element in path:
     - `Transceiver.propagate()`: Set initial OSNR
     - `Roadm.propagate()`: Apply loss, add ASE
     - `Fiber.propagate()`: Add attenuation, NLI, CD, PMD
     - `Edfa.propagate()`: Amplify, add ASE noise
   - Accumulate impairments along path
3. **Check Feasibility**
   - Calculate final OSNR (signal vs. ASE noise)
   - Calculate GSNR (signal vs. ASE + NLI)
   - Determine CD and PMD accumulation
   - Compare GSNR against transceiver requirements
   - Check if CD/PMD within compensation range
   - Determine limiting factor if infeasible
4. **Apply Constraints**
   - Validate power levels at all ROADMs
   - Ensure amplifier output powers within limits
   - Verify fiber input powers appropriate for NLI
   - Check against path-specific constraints
5. **Handle Disjunctions**

- Verify path disjointness requirements satisfied
- Check link disjunction (no shared fibers)
- Check node disjunction (no shared nodes except endpoints)
- Validate against all disjoint-from service IDs

Key Differences from transmission-example:

Aspect	gnpy-path-request	gnpy-transmission-example
Spectrum	Per-request assignment	Full C-band (default 96 channels)
Propagation	Only computed path	All elements in network
Focus	Service feasibility	Physical layer characterization
NLI Calculation	Per-request channel	Full spectrum interaction
Output	Per-service metrics	Network-wide propagation

Table 49: Path-Request vs. Transmission-Example

Transceiver Mode Selection (if infeasible) If initial propagation fails feasibility, GNPy can automatically select an alternative mode:

1. **Analyze Failure Reason**
   - OSNR insufficient
   - CD/PMD exceeds limits
   - Power constraints violated
2. **Select Alternative Mode**
   - Choose lower bit rate mode (higher OSNR margin)
   - Select mode with better CD tolerance
   - Consider different modulation format
3. **Re-propagate**
   - Update transceiver mode parameters
   - Re-run propagation with new mode
   - Check feasibility again
4. **Report Results**
   - If feasible: Report selected mode
   - If all modes fail: Report as infeasible with reason

Outputs Generated `gnpy-path-request` produces multiple output formats:

Output File	Format	Contents
`response.json`	JSON	Detailed per-service results
`path_result.csv`	CSV	Summary table of all services
`-output`	JSON	Custom output file (if specified)

Output File	Format	Contents
Console	Text	Summary with feasibility status

Table 50: gnpy-path-request Output Files

Response JSON Structure The primary output is `response.json` with this structure:

```
{
  "response": [
    {
      "response-id": "Service_001",
      "path-properties": {
        "path-route-objects": [
          {"path-route-object": {"num-unnum-hop": {"node-id": "Paris"}}},
          {"path-route-object": {"num-unnum-hop": {"node-id": "Orleans"}}},
          {"path-route-object": {"num-unnum-hop": {"node-id": "Lyon"}}}
        ],
        "z-a-path-route-objects": [...],
        "path-metric": [
          {
            "metric-type": "SNR-bandwidth",
            "accumulative-value": 18.5
          },
          {
            "metric-type": "OSNR-bandwidth",
            "accumulative-value": 23.2
          },
          {
            "metric-type": "reference_power",
            "accumulative-value": 0.001
          }
        ],
        "path-channel-attributes": [
          {
            "frequency": 193100000000000.0,
            "channel-power": -3.0,
            "chromatic-dispersion": 12450.0,
            "pmd": 8.5,
            "pdl": 2.1,
            "latency": 0.025
          }
        ]
      },
      "feasibility": true,
      "no-path": false
    },
    {
      "response-id": "Service_002",
      "no-path": false,
      "feasibility": false,
      "path-properties": {...},
      "infeasibility-reason": "OSNR below transceiver requirement (15.2 dB measured, 18.0 dB
    required)"
    }
  ]
}
```

Listing 23: response.json Structure

Output Metrics Each service response includes:

Metric	Description
response-id	Matches input request-id
feasibility	Boolean: true if service meets requirements
no-path	Boolean: true if no route found
path-route-objects	Ordered list of nodes in computed path
SNR-bandwidth	Generalized SNR (dB) including NLI
OSNR-bandwidth	Optical SNR (dB) excluding NLI
reference_power	Channel power (W)
chromatic-dispersion	Accumulated CD (ps/nm)
pmd	Accumulated PMD (ps)
pdl	Accumulated PDL (dB)
latency	Propagation delay (ms)
selected-mode	Chosen transceiver mode
infeasibility-reason	Explanation if feasibility = false

Table 51: Service Response Metrics

CSV Output Format The path_result.csv provides a summary table:

```
request_id,source,destination,path_length_km,osnr_db,gsnr_db,feasible,mode,
    infeasibility_reason
Service_001,Paris,Lyon,450.2,23.2,18.5,True,mode 1,
Service_002,Paris,Marseille,780.5,19.1,15.2,False,mode 1,OSNR below requirement
Service_003,Lyon,Nice,320.8,25.6,21.3,True,mode 2,
```

Listing 24: path_result.csv Example

Command Line Usage Basic Syntax:

```
gnpy-path-request [-h] [-e EQUIPMENT] [-o OUTPUT] [--design DESIGN]
                  [--no-design] [-sim SIMPARAMS] [-v]
                  network_file service_file
```

Common Commands:

Command	Description
-e, -equipment	Equipment library file (default: eqpt_config.json)
-o, -output	Output file for results (default: response.json)
-design	Auto-design network before path computation
-no-design	Skip auto-design (use existing amplifier placement)
-sim, -sim-params	Simulation parameters file
-v, -verbose	Enable detailed logging

Table 52: gnpy-path-request Command Line Options

Example Commands:

```
# Simple path computation
gnpy-path-request topology.json services.json
```

```
# With auto-design
gnpy-path-request --design topology.json services.json

# Custom equipment and output
gnpy-path-request -e custom_eqpt.json \
                  -o my_results.json \
                  topology.json services.json

# With simulation parameters and verbose logging
gnpy-path-request -e eqpt_config.json \
                  -sim sim_params.json \
                  -v \
                  topology.json services.json
```

Listing 25: Basic Path Request

Use Cases and Scenarios

Use Case 1: Service Provisioning Validation **Scenario:** SDN controller receives new service request and needs to verify feasibility before provisioning.

Workflow:

1. Controller exports current network state to GNPy format
2. Creates service request JSON with required parameters
3. Calls `gnpy-path-request` via API or CLI
4. Receives feasibility response with computed path
5. Provisions service if feasible, rejects if infeasible

Example Service Request:

```
{
  "path-request": [{
    "request-id": "CustomerA-ServiceX",
    "source": "DataCenter_Paris",
    "destination": "DataCenter_Frankfurt",
    "bidirectional": true,
    "path-constraints": {
      "te-bandwidth": {
        "technology": "flexi-grid",
        "trx_type": "400G-CFP2-DCO",
        "trx_mode": "16QAM_400G"
      },
      "path-metric-type": "OSNR"
    }
  }]
}
```

Use Case 2: Network Capacity Planning **Scenario:** Planning team needs to assess how many additional services can be supported.

Workflow:

1. Create batch of hypothetical service requests

2. Run `gnpy-path-request` with spectrum assignment
3. Analyze feasibility results and spectrum utilization
4. Identify bottleneck links and amplifiers
5. Plan upgrades based on infeasible services

Analysis Output:

- Percentage of feasible services: 85% (17/20)
- Bottleneck link: Paris-Lyon (spectrum exhausted)
- Infeasibility reasons: 2 OSNR limited, 1 spectrum unavailable
- Recommendation: Add parallel fiber on Paris-Lyon

Use Case 3: Multi-Path Service with Diversity Scenario: Customer requires protected service with link-disjoint primary and backup paths.

Workflow:

1. Create two requests with disjunction constraint
2. Specify primary path as normal service
3. Add backup with `disjoint-from: ["primary_id"]`
4. Verify both paths are feasible
5. Confirm link disjointness in results

Example Request:

```
{
  "path-request": [
    {
      "request-id": "Primary_Path",
      "source": "SiteA",
      "destination": "SiteB",
      "path-constraints": {
        "path-metric-type": "OSNR"
      }
    },
    {
      "request-id": "Backup_Path",
      "source": "SiteA",
      "destination": "SiteB",
      "path-constraints": {
        "disjointness": "link",
        "disjoint-from": ["Primary_Path"]
      }
    }
  ]
}
```

Use Case 4: Explicit Route Validation Scenario: Operator specifies preferred path for regulatory or business reasons.

Workflow:

1. Define explicit route through specific nodes
2. Set `loose-hop` to allow/disallow intermediate nodes
3. Run path computation with constraint enforcement
4. Verify physical layer feasibility of specified route
5. Report if explicit route is not viable

Example Request:

```
{
  "path-request": [{
    "request-id": "Regulated_Path",
    "source": "Paris",
    "destination": "Berlin",
    "path-constraints": {
      "explicit-route-objects": [
        {"num-unnum-hop": {"node-id": "Paris"}},
        {"num-unnum-hop": {"node-id": "Brussels"}},
        {"num-unnum-hop": {"node-id": "Amsterdam"}},
        {"num-unnum-hop": {"node-id": "Berlin"}}
      ],
      "loose-hop": false
    }
  }]
}
```

Use Case 5: Mode Selection Optimization **Scenario:** Network operator wants highest capacity mode that meets reach requirements.

Workflow:

1. Specify transceiver type but leave mode unspecified (`"trx_mode": null`)
2. GNPy tries modes in order from highest to lowest capacity
3. First feasible mode is selected
4. Results include selected mode and margins
5. Operator can override if needed

Mode Selection Logic:

```
# GNPy tries modes in order of decreasing bit rate
modes_sorted = sorted(transceiver.modes,
                      key=lambda m: m.bit_rate,
                      reverse=True)

for mode in modes_sorted:
    spectral_info = create_spectral_info(mode)
    propagate_result = propagate(path, spectral_info)

    if is_feasible(propagate_result, mode):
        return mode  # First feasible mode

return None  # No feasible mode found
```

Detailed Workflow

Complete Execution Flow The following pseudocode shows the complete `gnpy-path-request` workflow:

```
def gnpy_path_request(network_file, service_file, args):
    # Stage 1: Load Network
    equipment = load_equipment(args.equipment)
    network = load_network(network_file, equipment)

    if args.design:
        network = design_network(network, equipment)

    # Stage 2: Load Service Requests
    services = load_services(service_file)

    # Stage 3: Initialize Spectrum Management
    spectrum_map = SpectrumMap(network)

    # Stage 4: Process Each Request
    responses = []
    for request in services['path-request']:
        try:
            # Compute path
            path = compute_path_with_constraints(
                network, request, spectrum_map
            )

            # Assign spectrum
            frequency = assign_spectrum_for_request(
                request, path, spectrum_map
            )

            # Select transceiver mode
            if request['trx_mode'] is None:
                modes = get_available_modes(request['trx_type'])
            else:
                modes = [get_specific_mode(request['trx_mode'])]

            # Try modes until feasible
            selected_mode = None
            for mode in modes:
                # Create spectral information
                spectral_info = create_spectral_information(
                    frequency, mode, equipment['SI']['default']
                )

                # Propagate through path
                propagated = propagate_path(path, spectral_info)

                # Check feasibility
                is_feas, reason = check_feasibility(
                    propagated, mode
                )

                if is_feas:
                    selected_mode = mode
                    break

            # Build response
            response = build_response(
                request, path, propagated,
                selected_mode, is_feas, reason
```

```
            )

            # Update spectrum map if feasible
            if is_feas:
                spectrum_map.commit_allocation(path, frequency)

        except NoPathFoundError as e:
            response = build_no_path_response(request, str(e))
        except SpectrumExhaustedError as e:
            response = build_spectrum_exhausted_response(request)

        responses.append(response)

    # Stage 5: Generate Outputs
    write_response_json(responses, args.output)
    write_summary_csv(responses, 'path_result.csv')
    print_console_summary(responses)

    return responses
```

Listing 26: Complete gnpy-path-request Workflow

Propagation Details The propagation for each request follows this detailed flow:

```
def propagate_path(path, spectral_info):
    """Propagate spectral information through computed path."""

    # Initialize at source transceiver
    current_spectral_info = spectral_info.copy()
    source_trx = path[0]
    current_spectral_info = source_trx.propagate(current_spectral_info)

    # Track metrics
    metrics = {
        'osnr_db': [],
        'gsnr_db': [],
        'power_dbm': [],
        'cd_ps_nm': 0,
        'pmd_ps': 0,
        'pdl_db': 0
    }

    # Propagate through each element
    for elem in path[1:]:
        # Call element-specific propagation
        if isinstance(elem, Fiber):
            current_spectral_info = elem.propagate(
                current_spectral_info,
                degree=get_degree(elem, path),
                equipment=equipment
            )
            # Accumulate linear impairments
            metrics['cd_ps_nm'] += elem.chromatic_dispersion
            metrics['pmd_ps'] += elem.pmd

        elif isinstance(elem, Edfa):
            current_spectral_info = elem.propagate(
                current_spectral_info,
                degree=get_degree(elem, path)
            )
```

```
        elif isinstance(elem, Roadm):
            current_spectral_info = elem.propagate(
                current_spectral_info,
                degree=get_degree(elem, path)
            )
            metrics['pdl_db'] += elem.pdl

        elif isinstance(elem, Transceiver):
            # Destination transceiver
            current_spectral_info = elem.propagate(
                current_spectral_info
            )

        # Record metrics after each element
        osnr = calculate_osnr(current_spectral_info)
        gsnr = calculate_gsnr(current_spectral_info)
        power = current_spectral_info.signal[0].power

        metrics['osnr_db'].append(osnr)
        metrics['gsnr_db'].append(gsnr)
        metrics['power_dbm'].append(watts_to_dbm(power))

    # Final calculations
    final_osnr = metrics['osnr_db'][-1]
    final_gsnr = metrics['gsnr_db'][-1]

    return {
        'spectral_info': current_spectral_info,
        'metrics': metrics,
        'final_osnr': final_osnr,
        'final_gsnr': final_gsnr
    }
```

Listing 27: Per-Request Propagation

Feasibility Check Details The feasibility determination considers multiple factors:

```
def check_feasibility(propagated, mode):
    """Check if propagated signal meets transceiver requirements."""

    final_gsnr = propagated['final_gsnr']
    metrics = propagated['metrics']

    # 1. Check GSNR requirement
    required_osnr = mode.OSNR  # in dB
    implementation_margin = 2.0  # dB

    if final_gsnr < (required_osnr + implementation_margin):
        reason = (f"GSNR below requirement: {final_gsnr:.1f} dB "
                  f"measured, {required_osnr + implementation_margin:.1f} dB "
                  f"required (mode OSNR + {implementation_margin} dB margin)")
        return False, reason

    # 2. Check chromatic dispersion
    cd_limit = getattr(mode, 'cd_tolerance', 50000)  # ps/nm
    if abs(metrics['cd_ps_nm']) > cd_limit:
        reason = (f"CD exceeds tolerance: {metrics['cd_ps_nm']:.0f} ps/nm "
                  f"accumulated, {cd_limit:.0f} ps/nm limit")
        return False, reason
```

```
    # 3. Check PMD
    pmd_limit = getattr(mode, 'pmd_tolerance', 30)  # ps
    if metrics['pmd_ps'] > pmd_limit:
        reason = (f"PMD exceeds tolerance: {metrics['pmd_ps']:.1f} ps "
                  f"accumulated, {pmd_limit:.1f} ps limit")
        return False, reason

    # 4. Check PDL
    pdl_limit = getattr(mode, 'pdl_tolerance', 5.0)  # dB
    if metrics['pdl_db'] > pdl_limit:
        reason = (f"PDL exceeds tolerance: {metrics['pdl_db']:.1f} dB "
                  f"accumulated, {pdl_limit:.1f} dB limit")
        return False, reason

    # 5. Check power levels at ROADMs
    for i, power_dbm in enumerate(metrics['power_dbm']):
        if power_dbm < -30 or power_dbm > 10:
            reason = (f"Power out of range at element {i}: "
                      f"{power_dbm:.1f} dBm")
            return False, reason

    # All checks passed
    gsnr_margin = final_gsnr - (required_osnr + implementation_margin)
    reason = f"Service feasible with {gsnr_margin:.1f} dB GSNR margin"
    return True, reason
```

Listing 28: Feasibility Check Logic

Integration with OpenROADM `gnpy-path-request` aligns with OpenROADM Service Model:

GNPy Element	OpenROADM Equivalent
`request-id`	`service-name`
`source/destination`	`service-a-end/service-z-end`
`bidirectional`	Service type (point-to-point vs. unidirectional)
`path-route-objects`	`a-to-z-direction/path-description`
`explicit-route-objects`	`hard-constraints/include/ordered-hops`
`disjointness`	`diversity/diversity-type`
`feasibility`	Service validation result
`SNR-bandwidth`	Computed GSNR from planning tool
`selected-mode`	`operational-mode-id`

Table 53: GNPy to OpenROADM Service Model Mapping

OpenROADM Integration Workflow

1. **Service Request Translation**
    - SDN controller receives OpenROADM service-create RPC
    - Translate to GNPy service request format
    - Map endpoints to network nodes
    - Convert constraints to GNPy format

2. **GNPy Computation**
    - Call `gnpy-path-request` via API
    - Receive feasibility response
    - Extract computed path and metrics
3. **Response Translation**
    - Map path-route-objects to `path-description`
    - Convert GSNR to OpenROADM link metrics
    - Translate infeasibility reasons to response codes
4. **Service Provisioning**
    - If feasible: Provision service via OpenROADM
    - If infeasible: Return error with reason
    - Monitor service against predicted metrics

```python
from gnpy.core.equipment import load_equipment
from gnpy.core.network import load_network
from gnpy.core.request import compute_path_with_disjunction
from gnpy.tools.json_io import load_requests, write_output

# Load network model
equipment = load_equipment('eqpt_config.json')
network = load_network('topology.json', equipment)

# Load service requests
service_requests = load_requests('services.json')

# Compute paths
responses = []
for request in service_requests:
    path = compute_path_with_disjunction(network, request)
    propagated = propagate_and_evaluate(path, request, equipment)
    response = build_response(request, path, propagated)
    responses.append(response)

# Write results
write_output(responses, 'response.json')
```

Listing 29: Python API Usage

Performance Considerations

Computation Time Typical execution times for `gnpy-path-request`:

Network Size	Services	Time (s)	Time per Service
Small (10 nodes)	5	0.5	0.10 s
Medium (50 nodes)	20	3.2	0.16 s
Large (100 nodes)	50	12.5	0.25 s
Very Large (200 nodes)	100	45.0	0.45 s

Table 54: gnpy-path-request Performance

Performance Factors:

- Network size (nodes and links)
- Number of service requests
- Path constraints complexity (disjunctions)
- NLI calculation method (GN vs. GGN)
- Number of transceiver modes to evaluate

Optimization Strategies To improve performance for large-scale computations:

1. **Batch Processing**
   - Group related service requests
   - Parallelize independent path computations
   - Cache spectrum maps for incremental updates
2. **Network Simplification**
   - Pre-aggregate metro networks
   - Use representative fiber models
   - Limit NLI computation to critical channels
3. **Mode Selection Heuristics**
   - Pre-filter modes by path length
   - Start with most likely feasible mode
   - Skip modes with insufficient reach
4. **Caching**
   - Cache path computations for common endpoints
   - Reuse propagation results for similar requests
   - Store spectrum availability maps

Troubleshooting

Common Issues

Issue	Cause	Solution
No path found	Missing link or node in topology	Verify network connectivity
Spectrum exhausted	Too many services on link	Reduce channel count or add capacity
All modes infeasible	Path too long or noisy	Check amplifier gains, consider regeneration
Disjunction not satisfied	Insufficient alternate paths	Relax constraints or add links
Wrong node names	Mismatch between topology and services	Verify node naming consistency

Table 55: Common gnpy-path-request Issues

Debugging Workflow

1. **Enable Verbose Logging**

```
gnpy-path-request -v topology.json services.json
```

2. **Check Network Validity**

```
# Verify topology loads correctly
gnpy-transmission-example topology.json
```

3. **Validate Service Requests**
   - Ensure source/destination exist in topology
   - Check transceiver types match equipment library
   - Verify constraint syntax
4. **Analyze Individual Service**
   - Extract single service to separate file
   - Run path-request on isolated service
   - Examine detailed propagation results
5. **Review Infeasibility Reasons**
   - Check `infeasibility-reason` in response
   - Identify limiting factor (OSNR, CD, spectrum)
   - Consider network modifications or relaxed requirements

3.4 Core Propagation Engine (QoT-E)

3.4.1 Introduction

The Core Propagation Engine, also referred to as the Quality of Transmission Estimator (QoT-E), is the computational heart of GNPy. It implements the physics-based models that

simulate optical signal propagation through network elements, calculating the accumulation of impairments and predicting end-to-end quality of transmission metrics.

Purpose and Role in GNPy The propagation engine serves multiple critical functions:

1. **Physical Layer Simulation**
   - Models signal propagation through optical fibers
   - Simulates amplifier gain and noise characteristics
   - Accounts for ROADM filtering and equalization effects
   - Predicts transceiver performance at signal endpoints
2. **Impairment Calculation**
   - Linear impairments: Chromatic dispersion (CD), polarization mode dispersion (PMD), polarization dependent loss (PDL)
   - Amplifier noise: Amplified spontaneous emission (ASE)
   - Nonlinear impairments: Four-wave mixing (FWM), cross-phase modulation (XPM), self-phase modulation (SPM) via Gaussian Noise (GN) models
   - Stimulated Raman scattering (SRS) effects
3. **Quality Metrics Computation**
   - Optical signal-to-noise ratio (OSNR)
   - Generalized signal-to-noise ratio (GSNR) including nonlinear effects
   - Per-channel power levels throughout the network
   - Accumulated linear impairments (CD, PMD, PDL)
4. **Feasibility Determination**
   - Compare predicted GSNR against transceiver requirements
   - Validate power levels within equipment operating ranges
   - Assess impact of accumulated impairments
   - Identify limiting factors for infeasible paths

When and How the Engine is Invoked The propagation engine is invoked by both execution paths:

Execution Path	Engine Invocation
`gnpy-transmission-example`	Invoked once for full spectrum through entire network topology
`gnpy-path-request`	Invoked per service request for computed path with assigned spectrum

Table 56: Propagation Engine Invocation Context

Invocation Workflow:

```
# Both execution paths follow this pattern:

# 1. Create spectral information
spectral_info = create_arbitrary_spectral_information(
    f_min, f_max, baud_rate, spacing, power_dbm, ...
)

# 2. Get path elements (entire network or specific path)
path = get_propagation_path(source, destination)

# 3. Invoke propagation engine
result = propagate(spectral_info, path, equipment)

# 4. Extract metrics
gsnr = calculate_gsnr(result)
feasibility = check_feasibility(result, transceiver_requirements)
```

Listing 30: Engine Invocation Pattern

The engine operates element-by-element, sequentially updating the spectral information object as it passes through each network component.

Relationship to OpenROADM Models The GNPy propagation engine implements the physical layer models referenced in OpenROADM specifications:

OpenROADM Concept	GNPy Implementation
Media channel	Spectral information with frequency slots
Operational modes	Transceiver modes with OSNR requirements
Link budget calculation	Fiber and amplifier propagation models
Nonlinear penalties	GN/GGN model for NLI calculation
OSNR estimation	ASE and NLI accumulation through path
Power spectral density	Per-channel power tracking

Table 57: OpenROADM to GNPy Propagation Mapping

3.4.2 Input: Spectral Information Object

The `SpectralInformation` object is the fundamental data structure that flows through the propagation engine. It encapsulates all information about the optical spectrum being propagated, including signal powers, frequencies, and accumulated impairments.

Structure and Components The `SpectralInformation` class contains the following key attributes:

Detailed Attribute Descriptions

Signal Power Array The `signal` array contains the optical power of each channel in watts:

$$\text{signal}[i] = P_i \text{ (W)} \tag{3}$$

- **Initialization**: Set from equipment SI configuration or service request
- **Updates**: Modified by fiber attenuation and amplifier gain
- **Usage**: Primary input for OSNR and GSNR calculations

Example Values:

```
# For 0 dBm per channel (1 mW)
signal = [0.001, 0.001, 0.001, ...]  # Watts

# After 20 dB fiber attenuation
signal = [0.00001, 0.00001, 0.00001, ...]  # 0.01 mW = -20 dBm
```

Frequency Array The `frequency` array defines the center frequency of each channel:

$$\text{frequency}[i] = f_i = f_{\min} + i \cdot \Delta f \text{ (Hz)} \tag{4}$$

where Δf is the channel spacing.

- **Standard C-band**: 191.35 THz to 196.10 THz
- **Typical spacing**: 50 GHz (0.4 nm) or 37.5 GHz for dense systems
- **Remains constant**: Frequency does not change during propagation

Example:

```
# 96 channels, 50 GHz spacing
f_min = 191.35e12  # Hz (191.35 THz)
spacing = 50e9     # Hz (50 GHz)
num_channels = 96

frequency = [f_min + i * spacing for i in range(num_channels)]
# [191.35e12, 191.40e12, 191.45e12, ..., 196.10e12]
```

Baud Rate and Roll-off

- **baud_rate**: Symbol rate in Hz (e.g., 32 GBaud = 32e9 Hz)
- **roll_off**: Nyquist pulse shaping factor (typically 0.15 to 0.35)

These parameters determine the occupied bandwidth:

$$\text{Occupied BW} = \text{baud\_rate} \times (1 + \text{roll\_off}) \tag{5}$$

Example:

```
baud_rate = 32e9  # 32 GBaud
roll_off = 0.15   # 15% excess bandwidth

occupied_bw = baud_rate * (1 + roll_off)
# = 32e9 * 1.15 = 36.8 GHz
```

Nonlinear Interference (NLI) Array The `nli` array accumulates nonlinear interference power:

$$\text{nli}[i] = N_{\text{NLI},i} = N_{\text{SPM},i} + N_{\text{XPM},i} + N_{\text{FWM},i} \text{ (W)} \tag{6}$$

- **Initialization**: Set to 0 at source transceiver
- **Updates**: Calculated and added by fiber elements using GN or GGN models
- **Accumulation**: Incoherently summed across spans
- **Usage**: Included in GSNR calculation as noise term

GN Model Equation (simplified):

$$N_{\text{NLI}} = P_{\text{ch}}^3 \cdot \gamma^2 \cdot L_{\text{eff}}^2 \cdot \eta_{\text{NLI}}(\Delta f, B_s, \beta_2) \tag{7}$$

where:

- P_{ch}: Channel power
- γ: Fiber nonlinearity coefficient
- L_{eff}: Effective length
- η_{NLI}: NLI efficiency factor

ASE Noise Array The `ase` array accumulates amplified spontaneous emission noise:

$$\text{ase}[i] = P_{\text{ASE},i} \text{ (W)} \tag{8}$$

- **Initialization**: Set to 0 at source transceiver
- **Updates**: Added by EDFA and ROADM elements
- **Amplification**: Scaled by gain in subsequent amplifiers
- **Reference bandwidth**: Typically 12.5 GHz (0.1 nm at 1550 nm)

ASE Power Calculation:

$$P_{\text{ASE}} = n_{\text{sp}} h\nu (G - 1) B_{\text{ref}} \tag{9}$$

where:

- n_{sp}: Spontaneous emission factor (related to NF)
- $h\nu$: Photon energy
- G: Amplifier gain (linear)
- B_{ref}: Reference bandwidth

Linear Impairments The linear impairment arrays accumulate deterministically:

1. **Chromatic Dispersion**:

$$\text{CD\_total} = \sum_{\text{fibers}} D_i \cdot L_i \text{ (ps/nm)} \tag{10}$$

2. **Polarization Mode Dispersion**:

$$\text{PMD\_total} = \sqrt{\sum_{\text{fibers}} \text{PMD}_i^2} \text{ (ps)} \tag{11}$$

3. **Polarization Dependent Loss**:

$$\text{PDL\_total} = \sum_{\text{elements}} \text{PDL}_i \text{ (dB)} \tag{12}$$

SpectralInformation Creation The spectral information object is typically created using the `create_arbitrary_spectral_information()` function:

```
from gnpy.core.info import SpectralInformation, create_arbitrary_spectral_information

# Create from SI configuration
spectral_info = create_arbitrary_spectral_information(
    f_min=191.35e12,        # Hz
    f_max=196.10e12,        # Hz
    baud_rate=32e9,         # Hz
    spacing=50e9,           # Hz
    power_dbm=0.0,          # dBm per channel
    roll_off=0.15,          # Nyquist roll-off
    tx_osnr=100.0,          # dB (effectively infinite)
    delta_pdb=None          # Power deviation (None = uniform)
)

# Result: SpectralInformation object with 96 channels
# - signal: [0.001, 0.001, ..., 0.001] W (0 dBm each)
# - frequency: [191.35e12, 191.40e12, ..., 196.10e12] Hz
# - baud_rate: [32e9, 32e9, ..., 32e9] Hz
# - nli: [0, 0, ..., 0] W (initialized to zero)
# - ase: [0, 0, ..., 0] W (initialized to zero)
# - chromatic_dispersion: [0, 0, ..., 0] ps/nm
# - pmd: [0, 0, ..., 0] ps
# - pdl: [0, 0, ..., 0] dB
```

Listing 31: Creating SpectralInformation

SpectralInformation Updates During Propagation As the spectral information flows through elements, it is updated:

```
# Initial state at source transceiver
spectral_info.signal = [1e-3, 1e-3, ...]  # 0 dBm
spectral_info.ase = [0, 0, ...]
spectral_info.nli = [0, 0, ...]

# After first fiber span (80 km SSMF)
fiber.propagate(spectral_info)
# - signal attenuated: [1e-5, 1e-5, ...]  # -20 dBm
# - nli added: [1e-10, 1e-10, ...]
# - cd accumulated: [1360, 1360, ...] ps/nm

# After first EDFA (20 dB gain)
edfa.propagate(spectral_info)
# - signal amplified: [1e-3, 1e-3, ...]  # back to 0 dBm
# - ase added: [1e-7, 1e-7, ...]
# - nli amplified: [1e-8, 1e-8, ...]
```

```

# After ROADM
roadm.propagate(spectral_info)
# - signal attenuated: [7.9e-4, 7.9e-4, ...]  # -1 dBm
# - ase from ROADM loss: [5e-8, 5e-8, ...]
# - pdl added: [0.5, 0.5, ...] dB
```

Listing 32: Propagation Update Pattern

```python
@dataclass
class SpectralInformation:
    """Container for spectral information during propagation."""

    # Power arrays (per channel, in Watts)
    signal: np.ndarray      # Signal power [W]
    nli: np.ndarray         # Nonlinear interference [W]
    ase: np.ndarray         # ASE noise [W]

    # Frequency definition (per channel, in Hz)
    frequency: np.ndarray   # Center frequencies [Hz]
    baud_rate: np.ndarray   # Symbol rates [Hz]
    slot_width: np.ndarray  # Channel slot widths [Hz]
    roll_off: np.ndarray    # Roll-off factors [dimensionless]

    # Accumulated impairments (per channel)
    chromatic_dispersion: np.ndarray  # CD [ps/nm]
    pmd: np.ndarray                   # PMD [ps]
    pdl: np.ndarray                   # PDL [dB]
    latency: np.ndarray               # Latency [s]

    # Power management
    delta_pdb_per_channel: np.ndarray  # Power deviation [dB]

    def copy(self):
        """Deep copy of spectral information."""
        return copy.deepcopy(self)

    @property
    def num_channels(self):
        """Number of channels in spectrum."""
        return len(self.frequency)

    @property
    def total_power(self):
        """Total signal power across all channels [W]."""
        return np.sum(self.signal)
```

Listing 33: SpectralInformation Class Structure

3.4.3 Propagation Workflow

The propagation workflow orchestrates the sequential update of spectral information through network elements. This section details the core propagation function and its execution flow.

Core Propagation Function The primary entry point is `gnpy.topology.request.propagate()`:

```
def propagate(path, req, equipment):
    """
    Propagate spectral information through a path.

    Parameters
    ----------
    path : list of network elements
        Ordered sequence of elements to traverse
    req : ServiceRequest or dict
        Service request with spectrum specification
    equipment : dict
        Equipment library from load_equipment()

    Returns
    -------
    spectral_info : SpectralInformation
        Updated spectral information after propagation
    path : list
        Path elements (may be modified if propagation fails)
    """

    # 1. Extract spectral parameters from request
    spectral_params = extract_spectral_params(req, equipment)

    # 2. Create initial spectral information
    spectral_info = create_arbitrary_spectral_information(
        f_min=spectral_params['f_min'],
        f_max=spectral_params['f_max'],
        baud_rate=spectral_params['baud_rate'],
        spacing=spectral_params['spacing'],
        power_dbm=spectral_params['power_dbm'],
        roll_off=spectral_params['roll_off'],
        tx_osnr=spectral_params['tx_osnr']
    )

    # 3. Propagate through each element in path
    for element in path:
        # Call element-specific propagation method
        spectral_info = element(spectral_info)

        # Check for propagation failure
        if spectral_info is None:
            raise PropagationError(f"Propagation failed at {element.uid}")

    # 4. Return final spectral information
    return spectral_info, path
```

Listing 34: gnpy.topology.request.propagate() Function

Propagation Process Steps

Step 1: Create SpectralInformation Before propagation begins, the spectral information object is initialized with parameters from the service request or default SI configuration:

```
# From service request
if 'path-constraints' in request:
```

```
    te_bw = request['path-constraints']['te-bandwidth']
    trx_mode = te_bw['trx_mode']

    # Get transceiver mode parameters
    mode_params = equipment['Transceiver'][te_bw['trx_type']].mode[trx_mode]

    f_center = te_bw.get('frequency', 193.1e12)  # Default to C-band center
    baud_rate = mode_params['baud_rate']
    roll_off = mode_params['roll_off']
    spacing = te_bw.get('spacing', baud_rate * (1 + roll_off))

# From default SI configuration
else:
    si_default = equipment['SI']['default']
    f_min = si_default.f_min
    f_max = si_default.f_max
    baud_rate = si_default.baud_rate
    spacing = si_default.spacing
    power_dbm = si_default.power_dbm
    roll_off = si_default.roll_off

# Create spectral information
spectral_info = create_arbitrary_spectral_information(
    f_min=f_min,
    f_max=f_max,
    baud_rate=baud_rate,
    spacing=spacing,
    power_dbm=power_dbm,
    roll_off=roll_off,
    tx_osnr=100.0
)
```

Listing 35: SpectralInformation Initialization

Step 2: Element-by-Element Propagation Loop The core propagation loop iterates through path elements:

```
# Path is an ordered list of network elements
# Example: [Transceiver('Paris'), Fiber('F1'), Edfa('A1'), ...]

for element in path:
    # Each element implements __call__() method
    # which takes spectral_info and returns updated spectral_info

    # Get degree information (for multi-degree elements)
    degree = get_degree(element, path)

    # Call element-specific propagation
    if isinstance(element, (Fiber, Edfa)):
        # Fiber and EDFA need equipment library
        spectral_info = element(spectral_info, degree=degree, equipment=equipment)
    elif isinstance(element, Roadm):
        # ROADM needs degree info
        spectral_info = element(spectral_info, degree=degree)
    else:
        # Transceiver
        spectral_info = element(spectral_info)

    # Optional: Store intermediate results
    if store_intermediate:
```

```
        intermediate_results[element.uid] = spectral_info.copy()
```

Listing 36: Element-by-Element Propagation

Element Call Signature Each network element implements a `__call__()` method with specific signature:

```
class Transceiver:
    def __call__(self, spectral_info):
        """Propagate through transceiver."""
        # Source: Initialize OSNR
        # Destination: Compute final metrics
        return updated_spectral_info

class Fiber:
    def __call__(self, spectral_info, degree=None, equipment=None):
        """Propagate through fiber."""
        # Apply attenuation
        # Add NLI using GN/GGN model
        # Accumulate CD, PMD
        return updated_spectral_info

class Edfa:
    def __call__(self, spectral_info, degree=None):
        """Propagate through EDFA."""
        # Apply gain
        # Add ASE noise
        return updated_spectral_info

class Roadm:
    def __call__(self, spectral_info, degree=None):
        """Propagate through ROADM."""
        # Equalize channels
        # Apply loss
        # Add ASE from loss
        return updated_spectral_info
```

Listing 37: Element $_{c}all_{()Signatures}$

Degree Information For multi-degree elements (ROADMs with multiple input/output ports), the degree parameter specifies the signal path:

```
def get_degree(element, path):
    """
    Determine degree (input/output port) for element in path.

    For ROADM with 3 degrees:
    - Degree 1: West port
    - Degree 2: East port
    - Degree 3: North port

    Returns degree based on previous and next elements in path.
    """
    element_idx = path.index(element)

    if element_idx == 0:
        # First element (source transceiver)
        return None

```

```
    if element_idx == len(path) - 1:
        # Last element (destination transceiver)
        return None

    # Get previous and next elements
    prev_element = path[element_idx - 1]
    next_element = path[element_idx + 1]

    # For ROADM, determine degree from connected fibers
    if isinstance(element, Roadm):
        # Check which input/output ports are used
        input_degree = element.get_degree_for_neighbor(prev_element)
        output_degree = element.get_degree_for_neighbor(next_element)
        return (input_degree, output_degree)

    return None
```

Listing 38: Degree Information

```
# Example path: Paris -> Fiber1 -> Amp1 -> Fiber2 -> Lyon
path = [
    Transceiver('Paris'),
    Fiber('Fiber1'),
    Edfa('Amp1'),
    Fiber('Fiber2'),
    Transceiver('Lyon')
]

# Create initial spectral information
spectral_info = create_arbitrary_spectral_information(
    f_min=193.0e12,
    f_max=193.1e12,  # Just 2 channels for example
    baud_rate=32e9,
    spacing=50e9,
    power_dbm=0.0,
    roll_off=0.15,
    tx_osnr=100.0
)

print("Initial state:")
print(f"Signal power: {spectral_info.signal} W")
print(f"ASE: {spectral_info.ase} W")
print(f"NLI: {spectral_info.nli} W")

# Element 1: Source transceiver
spectral_info = path[0](spectral_info)
print("\nAfter source transceiver:")
print(f"Signal power: {spectral_info.signal} W")
# TX OSNR set to 100 dB (essentially infinite)

# Element 2: First fiber (80 km)
spectral_info = path[1](spectral_info, equipment=equipment)
print("\nAfter Fiber1 (80 km):")
print(f"Signal power: {spectral_info.signal} W")  # Attenuated
print(f"NLI: {spectral_info.nli} W")  # NLI added
print(f"CD: {spectral_info.chromatic_dispersion} ps/nm")  # CD accumulated

# Element 3: Amplifier (20 dB gain)
spectral_info = path[2](spectral_info)
print("\nAfter Amp1 (20 dB gain):")
```

```
print(f"Signal power: {spectral_info.signal} W")  # Amplified
print(f"ASE: {spectral_info.ase} W")  # ASE added

# Element 4: Second fiber (80 km)
spectral_info = path[3](spectral_info, equipment=equipment)
print("\nAfter Fiber2 (80 km):")
print(f"Signal power: {spectral_info.signal} W")  # Attenuated again
print(f"NLI: {spectral_info.nli} W")  # More NLI added
print(f"CD: {spectral_info.chromatic_dispersion} ps/nm")  # More CD

# Element 5: Destination transceiver
spectral_info = path[4](spectral_info)
print("\nAfter destination transceiver:")

# Calculate final GSNR
osnr = spectral_info.signal / spectral_info.ase
gsnr = spectral_info.signal / (spectral_info.ase + spectral_info.nli)
print(f"OSNR: {10*np.log10(osnr)} dB")
print(f"GSNR: {10*np.log10(gsnr)} dB")
```

Listing 39: Complete Propagation Through Simple Path

3.4.4 Network Element Computations

This section provides detailed documentation of the propagation computations performed by each network element type. Each element implements specific physical models to update the spectral information object.

Fiber.__call__() The fiber element models signal propagation through optical fiber, including attenuation, nonlinear effects, and linear impairments.

Purpose Fiber propagation accounts for:

- Signal attenuation due to fiber loss
- Nonlinear interference (NLI) from Kerr effect
- Chromatic dispersion accumulation
- Polarization mode dispersion accumulation
- Optional: Stimulated Raman scattering (SRS) effects

```
def __call__(self, spectral_info, degree=None, equipment=None):
    """
    Propagate spectral information through fiber.

    Steps:
    1. Attenuate signal
    2. Calculate and add NLI
    3. Accumulate chromatic dispersion
    4. Accumulate PMD
    5. Update latency
```

```
    """

    # 1. Signal Attenuation
    # P_out = P_in    exp(-      L)
    # In dB: Loss =  _dB /km    L_km

    loss_linear = 10 ** (-self.params.loss_coef * self.params.length / 10)
    spectral_info.signal *= loss_linear
    spectral_info.ase *= loss_linear
    spectral_info.nli *= loss_linear

    # 2. Calculate Nonlinear Interference
    if equipment and 'sim_params' in equipment:
        nli_params = equipment['sim_params']['nli_params']
    else:
        nli_params = default_nli_params

    # Create NLI solver
    nli_solver = NliSolver(
        fiber_params=self.params,
        nli_params=nli_params,
        spectral_info=spectral_info
    )

    # Compute NLI for each channel
    nli_power = nli_solver.compute_nli()

    # Add NLI power to existing NLI
    spectral_info.nli += nli_power

    # 3. Accumulate Chromatic Dispersion
    # CD = D    L (ps/nm)
    cd_span = self.params.dispersion * self.params.length
    spectral_info.chromatic_dispersion += cd_span

    # 4. Accumulate PMD
    # PMD grows as sqrt of length
    pmd_span = self.params.pmd_coef * np.sqrt(self.params.length)
    spectral_info.pmd = np.sqrt(spectral_info.pmd**2 + pmd_span**2)

    # 5. Update Latency
    # t = L / (c/n) where n is refractive index
    speed_of_light = 299792458  # m/s
    refractive_index = 1.468    # for silica fiber
    latency_span = (self.params.length * 1000) / (speed_of_light / refractive_index)
    spectral_info.latency += latency_span

    return spectral_info
```

Listing 40: Fiber.$_{call_{()Implementation}}$

Attenuation Model Signal attenuation follows Beer-Lambert law:

$$P_{\text{out}}(z) = P_{\text{in}} \cdot e^{-\alpha z} \tag{13}$$

In decibels:

$$L_{\text{dB}} = \alpha_{\text{dB/km}} \cdot L_{\text{km}} \tag{14}$$

Physical Origins of Attenuation:

Fiber attenuation arises from two primary mechanisms:

1. **Rayleigh Scattering**: Dominant loss mechanism in the 1550 nm window
   - Caused by microscopic density fluctuations in glass
   - Wavelength dependent: $\alpha_{\text{Rayleigh}} \propto \lambda^{-4}$
   - Contributes approximately 0.14-0.16 dB/km at 1550 nm
   - Fundamental limit - cannot be eliminated
2. **Impurities and Absorption**:
   - OH ion absorption peaks (residual water in fiber)
   - Intrinsic material absorption (infrared, UV tails)
   - Contributes approximately 0.04-0.06 dB/km at 1550 nm
   - Reduced through manufacturing process improvements

Total attenuation:

$$\alpha_{\text{total}} = \alpha_{\text{Rayleigh}} + \alpha_{\text{impurities}} + \alpha_{\text{connector}} \tag{15}$$

Typical fiber loss coefficients:

- SSMF (G.652): 0.2 dB/km at 1550 nm
- LEAF (G.655): 0.2 dB/km
- Pure silica core: 0.17 dB/km (ultra-low loss)
- Connector/splice loss: 0.05-0.3 dB per connection

Implementation:

```
# Fiber parameters
loss_coef = 0.2  # dB/km
length = 80.0    # km

# Calculate loss
loss_db = loss_coef * length  # 16 dB

# Convert to linear
loss_linear = 10 ** (-loss_db / 10)  # 0.0251

# Apply to signal
signal_out = signal_in * loss_linear
```

Nonlinear Interference Calculation GNPy implements multiple NLI models through `science_utils.NliSolver`:

Understanding Nonlinear Interference:

Nonlinear interference (NLI) in optical fibers arises from the Kerr effect, where the refractive index depends on optical intensity. This manifests as three distinct phenomena:

1. **Self-Phase Modulation (SPM)** [Intra-Channel Signal-Signal]:
    - A channel modulates its own phase
    - Intensity variations cause phase variations
    - Leads to spectral broadening
    - Dominant in single-channel systems
2. **Cross-Phase Modulation (XPM)** [Inter-Channel Signal-Signal]:
    - One channel modulates the phase of another
    - Intensity variations in channel i affect phase of channel j
    - Walk-off between channels reduces efficiency
    - Significant in WDM systems
3. **Four-Wave Mixing (FWM)** [Inter-Channel Signal-Signal]:
    - Three waves interact to generate a fourth wave
    - Frequencies: $f_4 = f_i + f_j - f_k$
    - Phase-matching conditions: most efficient at low dispersion
    - Can cause interference on existing channels

Signal-Noise Interactions:

In addition to signal-signal interactions, nonlinear effects also involve:

- **Intra-Channel Signal-Noise**: SPM-induced phase noise, nonlinear phase noise
- **Inter-Channel Signal-Noise**: XPM-induced phase noise from adjacent channels

These are typically modeled as additional noise contributions in advanced NLI calculations.

GN Model Aggregation:

The Gaussian Noise (GN) model treats all these nonlinear effects collectively as additive Gaussian noise:

$$P_{\text{NLI,total}} = P_{\text{SPM}} + P_{\text{XPM}} + P_{\text{FWM}} + P_{\text{signal-noise}} \tag{16}$$

GNPy's GN and GGN models compute this aggregate NLI power efficiently without separating individual contributions, which is valid when many channels interact.

1. GN Model (Gaussian Noise Model):

The analytical GN model calculates NLI power as:

$$P_{\text{NLI},i} = \frac{16}{27}\gamma^2 P_{\text{ch}}^3 L_{\text{eff}}^2 B_{\text{ref}} \eta_{\text{GN}} \tag{17}$$

where:

- γ: Nonlinear coefficient (1/(W·km))
- P_{ch}: Channel power (W)

- L_{eff}: Effective length = $\frac{1-e^{-\alpha L}}{\alpha}$
- B_{ref}: Reference bandwidth (typically 12.5 GHz)
- η_{GN}: GN efficiency factor (depends on dispersion, channel count)

2. GGN Model (Generalized Gaussian Noise):

The GGN model accounts for spectral separation and provides higher accuracy:

$$P_{\text{NLI},i} = \sum_{j,k} \int \int \mu_{ijk}(f_1, f_2) \cdot \rho(f_i, f_j, f_k, f_1, f_2)\, df_1 df_2 \tag{18}$$

This requires numerical integration but captures:

- Inter-channel interference more accurately
- Channel spacing effects
- Different modulation formats per channel

NLI Solver Usage:

```
from gnpy.core.science_utils import NliSolver

# Create NLI solver with fiber parameters
nli_solver = NliSolver(
    fiber_params={
        'length': 80.0,          # km
        'loss_coef': 0.2,        # dB/km
        'dispersion': 17.0,      # ps/(nm km)
        'gamma': 1.3e-3,         # 1/(W km)
        'effective_area': 80.0,  #   m
    },
    nli_params={
        'method': 'gn_model_analytic',  # or 'ggn_spectrally_separated'
        'dispersion_tolerance': 1.0,
        'phase_shift_tolerance': 0.1
    },
    spectral_info=spectral_info
)

# Compute NLI for all channels
nli_power = nli_solver.compute_nli()

# nli_power is array of NLI power per channel in Watts
# Example: [2.5e-9, 2.6e-9, 2.5e-9, ...] W
```

Listing 41: NliSolver Implementation

Chromatic Dispersion Accumulation Chromatic dispersion accumulates linearly with fiber length:

$$CD_{\text{total}} = \sum_{i=1}^{N} D_i \cdot L_i \quad \text{(ps/nm)} \tag{19}$$

where:

- D_i: Dispersion parameter of fiber i (ps/(nm·km))
- L_i: Length of fiber i (km)

Physical Origins of Chromatic Dispersion:

Chromatic dispersion, also known as Group Velocity Dispersion (GVD), arises from two components:

1. **Material Dispersion**:
   - Caused by wavelength-dependent refractive index of silica: $n(\lambda)$
   - Different wavelengths travel at different velocities
   - Dominant contribution in standard fibers
   - Zero-dispersion wavelength for bulk silica: ∼1270 nm
2. **Waveguide Dispersion**:
   - Results from wavelength-dependent mode confinement in fiber core
   - Light distribution between core and cladding varies with wavelength
   - Can be engineered through fiber design (core size, index profile)
   - Used to shift zero-dispersion wavelength to 1550 nm window

Total dispersion parameter:

$$D(\lambda) = D_{\text{material}}(\lambda) + D_{\text{waveguide}}(\lambda) \tag{20}$$

In standard single-mode fiber (SSMF):

- Material dispersion dominates: ∼+20 ps/(nm·km) at 1550 nm
- Waveguide dispersion contribution: ∼-3 ps/(nm·km)
- Net result: $D \approx$ +17 ps/(nm·km) at 1550 nm

Typical dispersion values:

- SSMF (G.652): 17 ps/(nm·km) at 1550 nm
- NZDSF (G.655): 4-8 ps/(nm·km) (engineered waveguide dispersion)
- DCF (Dispersion Compensating): -100 to -170 ps/(nm·km)
- DSF (Dispersion Shifted): Near-zero at 1550 nm (suffers from high FWM)

Example:

```
# Fiber 1: 80 km SSMF
D1 = 17.0  # ps/(nm km)
L1 = 80.0  # km
CD1 = D1 * L1  # 1360 ps/nm

# Fiber 2: 80 km SSMF
D2 = 17.0
L2 = 80.0
CD2 = D2 * L2  # 1360 ps/nm

# Total CD
```

```
CD_total = CD1 + CD2  # 2720 ps/nm

# Impact on signal:
# Pulse broadening     |D|    L
# For 100 GHz channel BW (0.8 nm): ~2.2 ns broadening
```

PMD Accumulation PMD accumulates as the square root of sum of squares:

$$\text{PMD}_{\text{total}} = \sqrt{\sum_{i=1}^{N} \text{PMD}_i^2} \tag{21}$$

where each span contributes:

$$\text{PMD}_i = D_{\text{PMD}} \sqrt{L_i} \tag{22}$$

- D_{PMD}: PMD coefficient (ps/km)
- Typical values: 0.05-0.5 ps/km

Example:

```
# Modern fiber: D_PMD = 0.1 ps/ km
D_PMD = 0.1  # ps/ km

# After 80 km
PMD_1 = D_PMD * np.sqrt(80)  # 0.89 ps

# After another 80 km
PMD_2 = D_PMD * np.sqrt(80)  # 0.89 ps

# Total PMD (RMS sum)
PMD_total = np.sqrt(PMD_1**2 + PMD_2**2)  # 1.26 ps
```

Raman Effects For Raman-enabled fibers, SRS effects are calculated using `science_utils.RamanSolve`

```
from gnpy.core.science_utils import RamanSolver

if isinstance(self, RamanFiber) and raman_params['flag']:
    # Create Raman solver
    raman_solver = RamanSolver(
        fiber_params=self.params,
        raman_params=raman_params,
        spectral_info=spectral_info
    )

    # Solve Raman equations
    #  - Pump power distribution along fiber
    #  - Signal power evolution
    #  - SRS tilt (power variation across channels)
    spectral_info = raman_solver.propagate(spectral_info)

    # Update signal powers with Raman tilt
    spectral_info.signal *= raman_solver.get_raman_tilt()
```

Listing 42: Raman Solver Integration

Additional Fiber Impairments Beyond the primary propagation effects modeled above, real fibers experience additional impairments:

Mechanical and Environmental:

- **Fiber Splices**: Connection points with 0.01-0.3 dB loss each
- **Mechanical Stress**: Bending, tension causing micro-bending loss
- **Physical Degradation**: Aging, hydrogen darkening (rare in modern fibers)
- **Connector Loss**: 0.3-0.5 dB per connector pair

These are typically modeled in GNPy as:

- Lumped losses added to fiber attenuation coefficient
- Explicit `Fused` elements for major connection points
- Padding factor in span loss calculations

WDM-Specific Nonlinearities:

The nonlinear effects (SPM, XPM, FWM) discussed earlier are particularly significant in wavelength-division multiplexing (WDM) systems:

- **High Channel Count**: 80-96 channels increase XPM and FWM
- **Channel Spacing**: Closer spacing increases nonlinear crosstalk
- **Total Power**: Higher aggregate power increases all nonlinear effects
- **Dispersion Management**: Low dispersion enhances FWM phase matching

GNPy's GN/GGN models specifically account for these WDM scenarios by considering the full spectrum of interfering channels when calculating NLI for each channel.

Key Internal Methods Fiber elements use several internal methods during propagation:

Method	Purpose
`_calc_attenuation()`	Calculate fiber loss based on length and loss coefficient
`_calc_lumped_losses()`	Add connector/splice losses to total attenuation
`interpol_params()`	Interpolate fiber parameters for specific wavelengths
`_calc_lin_attenuation()`	Calculate linear attenuation profile along fiber
`_fiber_att()`	Apply fiber attenuation to spectral information
`_nli_computation()`	Invoke NliSolver for nonlinear interference

Table 59: Fiber Internal Methods

Example: Lumped Losses Calculation

```
def _calc_lumped_losses(self):
    """
    Calculate additional losses from connectors and splices.

    Returns total loss in dB.
    """
    # Base fiber loss
```

```
    fiber_loss_db = self.params.loss_coef * self.params.length

    # Add connector losses
    con_in_db = self.params.con_in  # Input connector
    con_out_db = self.params.con_out  # Output connector

    # Add splice losses (if any)
    num_splices = int(self.params.length / 80)  # Assume splice every 80 km
    splice_loss_db = num_splices * 0.05  # 0.05 dB per splice

    # Add padding/margin
    padding_db = self.params.padding  # Additional loss margin

    # Total loss
    total_loss_db = (fiber_loss_db +
                     con_in_db +
                     con_out_db +
                     splice_loss_db +
                     padding_db)

    return total_loss_db
```

Listing 43: Lumped Losses in Fiber

Fused.__call__() The Fused element models passive optical couplers, splitters, and fusion splices - components with fixed insertion loss and no active functionality.

Purpose Fused element accounts for:

- Fixed insertion loss from passive coupling
- Optical power splitting (for splitters/couplers)
- Splice loss at fusion points
- No wavelength-dependent effects (flat across spectrum)

```
def __call__(self, spectral_info):
    """
    Propagate spectral information through fused passive component.

    Steps:
    1. Apply fixed insertion loss
    2. No noise addition (passive component)
    3. No impairment accumulation
    """

    # Get fixed loss parameter
    loss_db = self.params.loss  # e.g., 3 dB for 50/50 splitter

    # Convert to linear
    loss_linear = 10 ** (-loss_db / 10)

    # Apply to all power components equally
    spectral_info.signal *= loss_linear
    spectral_info.ase *= loss_linear
    spectral_info.nli *= loss_linear

    # No additional impairments (CD, PMD, PDL remain unchanged)
```

```
    # No ASE addition (passive component)

    return spectral_info
```

Listing 44: Fused.$_{call_{()Implementation}}$

Loss Model Fused components have wavelength-independent insertion loss:

$$P_{\text{out}} = P_{\text{in}} \cdot 10^{-L_{\text{dB}}/10} \tag{23}$$

Typical Loss Values:

Component Type	Insertion Loss
Fusion splice (quality)	0.01-0.05 dB
Fusion splice (typical)	0.1-0.3 dB
50/50 Coupler	3.0 dB
90/10 Coupler (through port)	0.5 dB
1×2 Splitter	3.0 dB
1×4 Splitter	6.0 dB
1×8 Splitter	9.0 dB

Table 60: Fused Component Insertion Losses

Usage in Network Models Fused elements are used to represent:

- **Optical splitters**: Distribution of signal to multiple paths
- **Optical couplers**: Combining/splitting wavelengths
- **Fusion splices**: Connection points between fiber segments
- **Fixed attenuators**: Deliberate power reduction points

Example in Topology:

```
{
  "uid": "Coupler_Site_A",
  "type": "Fused",
  "type_variety": "fused",
  "params": {
    "loss": 3.0  // dB
  }
}
```

Element	Key Differences from Fused
Fiber	Has length-dependent attenuation, adds NLI, accumulates CD/PMD
ROADM	Active equalization, wavelength-selective, adds ASE noise
Edfa	Provides gain, adds ASE noise
Fused	Purely passive, fixed loss, no wavelength dependence

Table 61: Fused vs. Other Elements

Distinction from Other Elements

Edfa.__call__() The EDFA element models erbium-doped fiber amplifier operation, including gain and noise figure characteristics. GNPy supports multiple EDFA variants for different modeling fidelities.

EDFA Variants GNPy implements three main EDFA types:

1. **Standard EDFA** (`type_def: "variable_gain"` or `"fixed_gain"`):
   - Simplified gain and NF models
   - Flat or polynomial wavelength dependence
   - Fast computation, suitable for network-scale planning
   - Gain set by `design_network()` or fixed value
2. **OpenROADM EDFA** (`type_def: "openroadm"`):
   - Follows OpenROADM operational mode specifications
   - Band-specific characterization (C-band, L-band)
   - Predefined gain and NF profiles per band
   - Compatible with OpenROADM service model
3. **Doped Fiber Amplifier** (`type_def: "advanced_model"`):
   - High-fidelity distributed amplification model
   - Includes distributed Raman amplification effects
   - Pump power evolution along doped fiber
   - Signal-pump interaction modeling
   - Requires advanced configuration JSON

Purpose EDFA propagation accounts for:

- Signal amplification across optical spectrum
- Amplified spontaneous emission (ASE) noise addition
- Gain spectral shape (flatness, ripple, tilt)

- Noise figure wavelength dependence
- Operating point constraints (gain range, output power limits)

```python
def __call__(self, spectral_info, degree=None):
    """
    Propagate spectral information through EDFA.

    Steps:
    1. Calculate gain for each channel
    2. Amplify signal, ASE, and NLI
    3. Calculate and add new ASE noise
    4. Apply gain spectral shape (if modeled)
    """

    # 1. Get amplifier operating point
    # This was set during design_network()
    gain_db = self.effective_gain  # dB
    gain_linear = 10 ** (gain_db / 10)

    # 2. Apply gain to all power components
    spectral_info.signal *= gain_linear
    spectral_info.ase *= gain_linear
    spectral_info.nli *= gain_linear

    # 3. Calculate wavelength-dependent noise figure
    # NF may vary with wavelength and operating point
    nf_db = self.get_nf_per_channel(spectral_info.frequency, gain_db)
    nf_linear = 10 ** (nf_db / 10)

    # 4. Calculate ASE power per channel
    # P_ASE = n_sp    h          (G - 1)    B_ref
    # where n_sp = (NF    G - 1) / (2(G - 1))     NF/2 for high gain

    h = 6.62607015e-34  # Planck constant (J s)
    B_ref = 12.5e9      # Reference bandwidth (Hz) - 0.1 nm at 1550 nm

    for i, freq in enumerate(spectral_info.frequency):
        photon_energy = h * freq
        nsp = (nf_linear[i] * gain_linear - 1) / (2 * (gain_linear - 1))

        ase_added = nsp * photon_energy * (gain_linear - 1) * B_ref
        spectral_info.ase[i] += ase_added

    # 5. Optional: Apply gain ripple and dynamic tilt
    if self.params.has_gain_ripple:
        ripple = self.get_gain_ripple(spectral_info.frequency)
        spectral_info.signal *= (1 + ripple)

    if self.params.has_dynamic_tilt:
        tilt = self.get_dynamic_tilt(spectral_info.frequency, gain_db)
        spectral_info.signal *= tilt

    return spectral_info
```

Listing 45: Edfa.$_{call()Implementation}$

Amplification Model The core amplification equation:

$$P_{\text{out}} = G \cdot P_{\text{in}} \tag{24}$$

where gain G is determined by EDFA model type:

1. Variable Gain Model:

- Gain set to compensate span loss + power adjustment
- $G_{\text{dB}} = L_{\text{span,dB}} + \Delta P_{\text{dB}}$
- Constrained by [gain_min, gain_flatmax]

2. Fixed Gain Model:

- Gain is constant regardless of input power
- Output power varies with input

3. Advanced Model:

- Gain depends on input power, wavelength, pump power
- Loaded from advanced configuration JSON

ASE Noise Calculation ASE noise power per channel is given by:

$$P_{\text{ASE}} = n_{\text{sp}} h\nu (G-1) B_{\text{ref}} \tag{25}$$

where the spontaneous emission factor:

$$n_{\text{sp}} = \frac{F \cdot G - 1}{2(G-1)} \approx \frac{F}{2} \quad \text{for } G \gg 1 \tag{26}$$

and F is the noise figure (linear).

Example Calculation:

```
# EDFA parameters
gain_db = 20.0       # dB
nf_db = 5.5          # dB
frequency = 193.1e12 # Hz (1552.52 nm)

# Convert to linear
gain_linear = 10 ** (gain_db / 10)  # 100
nf_linear = 10 ** (nf_db / 10)      # 3.55

# Calculate n_sp
nsp = (nf_linear * gain_linear - 1) / (2 * (gain_linear - 1))
# = (3.55    100 - 1) / (2    99) = 1.785

# Calculate ASE power
h = 6.626e-34      # J s
B_ref = 12.5e9     # Hz
photon_energy = h * frequency  # 1.281e-19 J

P_ASE = nsp * photon_energy * (gain_linear - 1) * B_ref
# = 1.785    1.281e-19    99    12.5e9
# = 2.84e-7 W (-35.5 dBm)
```

Noise Figure Models GNPy supports multiple NF models:

1. Flat NF (Simple Model):

```
# Constant NF across spectrum
nf_db = np.full(num_channels, self.params.nf_min)
```

2. Polynomial NF Model:

$$\text{NF}(\lambda) = a_0 + a_1\lambda + a_2\lambda^2 + a_3\lambda^3 \tag{27}$$

3. Gain-Dependent NF Model:

$$\text{NF}(G) = \text{NF}_{\text{min}} + \alpha(G - G_{\text{nom}})^2 \tag{28}$$

4. Advanced NF Model:

- Loaded from JSON with measured data
- Interpolated for operating conditions
- Accounts for pump power, input power, wavelength

Gain Spectral Shape Real amplifiers have non-flat gain:

Gain Ripple:

$$G(\lambda) = G_{\text{flat}} \cdot (1 + r(\lambda)) \tag{29}$$

where $r(\lambda)$ is the ripple function (typically ±0.5 dB peak-to-peak).

Dynamic Gain Tilt:

$$\text{Tilt}_{\text{dB}} = k \cdot (P_{\text{in}} - P_{\text{ref}}) \tag{30}$$

where k is the tilt coefficient (dB/dB of input power change).

Output Power Limiting EDFA output power is constrained:

```
# Calculate total output power
P_out_total = np.sum(spectral_info.signal)
P_out_total_dbm = 10 * np.log10(P_out_total * 1000)

# Check against p_max
if P_out_total_dbm > self.params.p_max:
    # Option 1: Reduce gain
    gain_reduction = P_out_total_dbm - self.params.p_max
    gain_db -= gain_reduction

    # Option 2: Use output VOA
    if self.params.out_voa_auto:
        voa_attenuation = gain_reduction
        apply_voa(spectral_info, voa_attenuation)

    warnings.warn(f"EDFA {self.uid} output power limited to {self.params.p_max} dBm")
```

Key Internal Methods EDFA elements use several internal methods during propagation:

Method	Purpose
`interpol_params()`	Interpolate gain and NF for given frequency/power
`nf_model()`	Calculate wavelength-dependent noise figure
`delta_p()`	Compute power deviation for optimization
`target_gain()`	Determine target gain based on span loss
`calc_nf()`	Calculate operating-point-dependent NF
`get_gain_ripple()`	Retrieve gain spectral ripple
`get_dynamic_tilt()`	Calculate gain tilt from power variation

Table 62: EDFA Internal Methods

Example: interpol_params() Usage

```python
# For advanced EDFA models with measured data
def interpol_params(frequency, input_power, gain_target):
    """
    Interpolate amplifier parameters for operating conditions.

    Returns gain and NF from lookup tables or polynomial fits.
    """
    # Load advanced configuration data
    config = self.advanced_config

    # Interpolate gain vs. frequency
    gain_profile = np.interp(
        frequency,
        config['freq_table'],
        config['gain_table']
    )

    # Interpolate NF vs. frequency and gain
    nf_profile = interpolate_2d(
        frequency,
        gain_target,
        config['freq_table'],
        config['gain_table'],
        config['nf_table']
    )

    return gain_profile, nf_profile
```

Listing 46: Gain and NF Interpolation

Roadm.__call__() The ROADM element models reconfigurable optical add-drop multiplexer operation, including channel equalization, insertion loss, crosstalk, and various noise contributions.

Purpose ROADM propagation accounts for:

- Channel-by-channel equalization to target power
- Insertion loss from WSS and passive components
- PDL from WSS and fiber connectors
- ASE noise equivalent to insertion loss

- Crosstalk between channels
- Relative intensity noise (RIN) from add/drop operations
- Filter shape effects from cascaded WSS
- Per-degree configurations (multi-degree ROADMs)

```
def __call__(self, spectral_info, degree=None):
    """
    Propagate spectral information through ROADM.

    Steps:
    1. Apply per-channel equalization
    2. Apply insertion loss
    3. Add crosstalk contributions
    4. Add RIN (Relative Intensity Noise)
    5. Apply filter shape effects
    6. Add PDL
    7. Add ASE noise from loss
    """

    # 1. Get equalization target
    # Target is set during design_network() or from restrictions
    target_power = self.get_target_power_per_channel(degree)

    # 2. Calculate required attenuation per channel
    for i in range(spectral_info.num_channels):
        current_power = spectral_info.signal[i]
        current_power_dbm = 10 * np.log10(current_power * 1000)

        # Attenuation needed to reach target
        attenuation_db = current_power_dbm - target_power

        # Apply limits from ROADM restrictions
        attenuation_db = np.clip(
            attenuation_db,
            self.params.restrictions['min_attenuation'],
            self.params.restrictions['max_attenuation']
        )

        attenuation_linear = 10 ** (-attenuation_db / 10)

        # Apply to signal
        spectral_info.signal[i] *= attenuation_linear

    # 3. Apply insertion loss (affects all components)
    loss_db = self.params.insertion_loss  # Typically 10-13 dB
    loss_linear = 10 ** (-loss_db / 10)

    spectral_info.signal *= loss_linear
    spectral_info.ase *= loss_linear
    spectral_info.nli *= loss_linear

    # 4. Add Crosstalk Effects
    # Crosstalk occurs when signal from one channel leaks into adjacent channels
    if self.params.crosstalk_enabled:
        crosstalk_power = self.calculate_crosstalk(spectral_info, degree)
        spectral_info.ase += crosstalk_power  # Treated as noise

    # 5. Add RIN (Relative Intensity Noise)
```

```
    # RIN primarily affects add/drop paths due to laser noise transfer
    path_type = self.get_path_type(degree)  # 'express', 'add', or 'drop'

    if path_type in ['add', 'drop']:
        rin_power = self.calculate_rin(spectral_info)
        spectral_info.ase += rin_power

    # 6. Apply Filter Shape Effects
    # Cascaded WSS cause spectral narrowing
    num_wss_passed = self.get_num_wss_passed()
    if num_wss_passed > 0:
        filter_penalty_db = self.calculate_filter_penalty(
            spectral_info.baud_rate,
            num_wss_passed
        )
        # Apply as effective OSNR degradation
        filter_penalty_linear = 10 ** (filter_penalty_db / 10)
        spectral_info.ase *= filter_penalty_linear

    # 7. Add PDL
    if path_type == 'express':
        pdl_contribution = self.params.restrictions['roadm_path_impairments'][0]['
    roadm_express_path']['roadm_pdl']
    elif path_type == 'add':
        pdl_contribution = self.params.restrictions['roadm_path_impairments'][0]['
    roadm_add_path']['roadm_pdl']
    else:  # drop
        pdl_contribution = self.params.restrictions['roadm_path_impairments'][0]['
    roadm_drop_path']['roadm_pdl']

    spectral_info.pdl += pdl_contribution

    # 8. Add ROADM-induced PMD (if specified)
    roadm_pmd = self.params.pmd
    spectral_info.pmd = np.sqrt(spectral_info.pmd**2 + roadm_pmd**2)

    # 9. Add ASE noise equivalent to loss
    # ROADM ASE     loss equivalent noise
    # NF_ROADM     Loss (in dB)

    h = 6.62607015e-34
    B_ref = 12.5e9

    for i, freq in enumerate(spectral_info.frequency):
        photon_energy = h * freq

        # ASE from loss: equivalent to amplifier with NF = Loss and G = Loss^(-1)
        # P_ASE = h          (L - 1)    B_ref  where L = 10^(loss_dB/10)
        ase_added = photon_energy * (10 ** (loss_db / 10) - 1) * B_ref
        spectral_info.ase[i] += ase_added

    # 10. Additional ROADM OSNR penalty
    # Add from add/drop OSNR specification
    add_drop_osnr_db = self.params.add_drop_osnr
    add_drop_osnr_linear = 10 ** (add_drop_osnr_db / 10)

    # This is additional ASE equivalent
    for i in range(spectral_info.num_channels):
        # ASE equivalent to achieve specified OSNR
        ase_from_osnr = spectral_info.signal[i] / add_drop_osnr_linear
```

```
        spectral_info.ase[i] += ase_from_osnr

    return spectral_info
```

Listing 47: Roadm.$_{call()Implementation}$

Crosstalk Modeling Crosstalk in ROADMs occurs when signal energy from one channel leaks into adjacent channels through imperfect filtering:

$$P_{\text{crosstalk},i} = \sum_{j \neq i} P_{\text{signal},j} \cdot X_{ij} \tag{31}$$

where X_{ij} is the crosstalk coefficient from channel j to channel i.

Crosstalk Types:

1. **In-Band Crosstalk**:
   - From adjacent channels at different wavelengths
   - Wavelength-selective switch (WSS) isolation: 25-40 dB
   - Treated as incoherent noise
2. **Coherent Crosstalk**:
   - From same wavelength through different paths
   - More detrimental than incoherent (phase effects)
   - Typically modeled with higher penalty

```
def calculate_crosstalk(self, spectral_info, degree):
    """Calculate crosstalk power contributions."""

    crosstalk_power = np.zeros(spectral_info.num_channels)

    # Crosstalk isolation (dB) - typically 30-35 dB for adjacent channels
    isolation_db = self.params.crosstalk_isolation  # e.g., 30 dB
    isolation_linear = 10 ** (-isolation_db / 10)  # 0.001

    for i in range(spectral_info.num_channels):
        # Crosstalk from adjacent channels
        if i > 0:  # From lower frequency neighbor
            crosstalk_power[i] += spectral_info.signal[i-1] * isolation_linear

        if i < spectral_info.num_channels - 1:  # From higher frequency neighbor
            crosstalk_power[i] += spectral_info.signal[i+1] * isolation_linear

    return crosstalk_power
```

Listing 48: Crosstalk Calculation

RIN (Relative Intensity Noise) Modeling RIN arises from laser intensity fluctuations and multipath interference in add/drop paths:

$$P_{\text{RIN}} = \text{RIN}_{\text{dB/Hz}} \cdot P_{\text{signal}} \cdot B_{\text{ref}} \tag{32}$$

Typical RIN values:

- High-quality DFB lasers: -155 to -160 dB/Hz
- ROADM add/drop contribution: -140 to -150 dB/Hz
- External cavity lasers: -150 to -155 dB/Hz

```
def calculate_rin(self, spectral_info):
    """Calculate RIN contributions for add/drop paths."""

    rin_power = np.zeros(spectral_info.num_channels)

    # RIN specification in dB/Hz
    rin_db_hz = self.params.rin_db_hz  # e.g., -145 dB/Hz
    rin_linear = 10 ** (rin_db_hz / 10)

    B_ref = 12.5e9  # Reference bandwidth

    for i in range(spectral_info.num_channels):
        # RIN power proportional to signal power
        rin_power[i] = rin_linear * spectral_info.signal[i] * B_ref

    return rin_power
```

Listing 49: RIN Calculation

Filter Shape Effects Cascaded wavelength-selective switches cause spectral narrowing:

$$\text{Penalty}_{\text{filter}} = k \cdot N_{\text{WSS}} \cdot \left(\frac{B_{\text{signal}}}{B_{\text{filter}}} \right)^2 \tag{33}$$

where:

- N_{WSS}: Number of WSS passed
- B_{signal}: Signal bandwidth
- B_{filter}: Filter bandwidth
- k: Penalty coefficient (typically 0.1-0.5 dB per WSS)

```
def calculate_filter_penalty(self, baud_rate, num_wss):
    """Calculate penalty from cascaded filter narrowing."""

    # Filter bandwidth (assume slightly wider than signal)
    filter_bw_ghz = self.params.filter_bandwidth  # e.g., 50 GHz

    # Signal bandwidth
    signal_bw_ghz = baud_rate / 1e9  # Convert to GHz

    # Penalty per WSS
    penalty_per_wss = 0.3  # dB (typical value)

    # Filter narrowing factor
    narrowing_factor = (signal_bw_ghz / filter_bw_ghz) ** 2

```

```
    # Total penalty
    total_penalty_db = penalty_per_wss * num_wss * max(0, narrowing_factor - 0.5)

    return total_penalty_db
```

Listing 50: Filter Penalty Calculation

Key Internal Methods ROADM elements use several internal methods:

Method	Purpose
`get_roadm_target_power()`	Determine equalization target per channel
`set_roadm_paths()`	Configure internal express/add/drop paths
`calc_penalties()`	Calculate ROADM-specific impairment penalties
`get_path_type()`	Identify signal path type (express/add/drop)
`get_num_wss_passed()`	Count WSS traversals for filter penalty
`calculate_crosstalk()`	Compute inter-channel crosstalk
`calculate_rin()`	Compute RIN contributions
`calculate_filter_penalty()`	Compute cascaded filter narrowing penalty

Table 63: ROADM Internal Methods

Channel Equalization ROADMs can equalize channel powers using wavelength-selective switches (WSS):

$$P_{\text{out},i} = P_{\text{target}} \tag{34}$$

Three equalization strategies:

1. Per-Channel Power (`target_pch_out_db`):

```
target_power_dbm = self.params.target_pch_out_db  # e.g., -20 dBm
# All channels set to same power
```

2. Power Spectral Density (`target_psd_out_mWperGHz`):

```
target_psd = self.params.target_psd_out_mWperGHz  # e.g., 0.01 mW/GHz
baud_rate = spectral_info.baud_rate[i]
target_power = target_psd * (baud_rate / 1e9)  # Scale by channel bandwidth
```

3. Power per Slot Width (`target_out_mWperSlotWidth`):

```
target_power_per_slot = self.params.target_out_mWperSlotWidth
slot_width = spectral_info.slot_width[i]
target_power = target_power_per_slot * slot_width
```

ROADM Loss Components Total ROADM loss comprises:

$$L_{\text{ROADM}} = L_{\text{WSS}} + L_{\text{splitter/combiner}} + L_{\text{connectors}} \tag{35}$$

Typical values:

- WSS insertion loss: 6-8 dB

- Splitter/combiner: 3-6 dB
- Connectors: 0.5-1 dB each (×2-4)
- **Total**: 10-15 dB

ROADM Restrictions ROADMs have operational restrictions defined in equipment library:

```
{
  "restrictions": {
    "preamp_variety_list": [],
    "booster_variety_list": [],
    "max_per_degree_add_drop_ports": 20,
    "roadm_path_impairments": [
      {
        "roadm_express_path": {
          "roadm_osnr": 30,
          "roadm_pmd": 0,
          "roadm_cd": 0,
          "roadm_pdl": 0.5
        },
        "roadm_add_path": {
          "roadm_osnr": 33,
          "roadm_pmd": 0,
          "roadm_cd": 0,
          "roadm_pdl": 0.5
        },
        "roadm_drop_path": {
          "roadm_osnr": 33,
          "roadm_pmd": 0,
          "roadm_cd": 0,
          "roadm_pdl": 0.5
        }
      }
    ]
  }
}
```

Listing 51: ROADM Restrictions Example

These restrictions define:

- Express path OSNR: Through traffic (degree to degree)
- Add path OSNR: Traffic entering from transponder
- Drop path OSNR: Traffic exiting to transponder
- PDL per path

Transceiver.__call__() The transceiver element represents optical transponders at path endpoints, handling signal generation (source) and reception/evaluation (destination).

Purpose at Source Source transceiver initializes spectral information:

- Set initial signal power from transmitter
- Set transmitter OSNR (typically very high, 100 dB)

- Initialize all impairment counters to zero
- Account for transmitter phase noise

Transmitter Impairments:

Real transmitters introduce impairments that degrade signal quality:

1. **Phase Noise**:
   - Random phase fluctuations in laser output
   - Characterized by linewidth (Hz) or phase noise spectral density
   - Causes: spontaneous emission, thermal fluctuations
   - Impact: Increases symbol error rate, especially for high-order modulation
   - Typical values:
     - High-quality external cavity laser (ECL): 10-100 kHz linewidth
     - DFB laser: 1-10 MHz linewidth
     - Integrated tunable laser: 100 kHz - 1 MHz linewidth
2. **RIN (from laser)**:
   - Intensity noise from laser
   - Measured in dB/Hz (typically -155 to -160 dB/Hz)
   - Less significant in coherent systems
3. **Transmitter OSNR**:
   - Finite OSNR from transmitter (not infinite)
   - Typical values: 35-45 dB (electrical), 45-55 dB (optical)
   - GNPy default: 100 dB (effectively ideal transmitter)
   - Can be reduced to model non-ideal transmitters

GNPy Transmitter Modeling:

GNPy models transmitter quality through the `tx_osnr` parameter:

```
# Ideal transmitter (default)
tx_osnr_db = 100.0  # Effectively infinite

# Non-ideal transmitter
tx_osnr_db = 40.0  # Finite transmitter quality

# Convert to linear and initialize ASE
tx_osnr_linear = 10 ** (tx_osnr_db / 10)

# ASE from transmitter
for i in range(num_channels):
    spectral_info.ase[i] = spectral_info.signal[i] / tx_osnr_linear

# This initial ASE represents all transmitter imperfections:
# - Phase noise
# - Laser RIN
# - Modulator imperfections
```

```
# - DSP quantization noise
```

Listing 52: Transmitter OSNR Modeling

Phase Noise Impact:

Phase noise primarily affects:

- High-order modulation formats (64QAM, 256QAM)
- Systems with high symbol rates (> 32 GBaud)
- Long-haul systems where phase noise accumulates

For most planning purposes, TX OSNR of 100 dB is sufficient. Detailed transceiver modeling would reduce this to measured values (35-45 dB).

Purpose at Destination Destination transceiver computes final metrics and feasibility:

- Calculate GSNR including all accumulated noise
- Apply transceiver penalties (CD, PMD, PDL)
- Compare against transceiver mode requirements
- Determine feasibility

```
def __call__(self, spectral_info):
    """
    Propagate through transceiver.

    At source: Initialize signal
    At destination: Compute final metrics and feasibility
    """

    if self.is_source:
        # Source transceiver
        # Set TX OSNR (essentially infinite for ideal transmitter)
        tx_osnr_linear = 10 ** (self.params.tx_osnr / 10)

        # Initialize minimal ASE to represent TX OSNR
        for i in range(spectral_info.num_channels):
            spectral_info.ase[i] = spectral_info.signal[i] / tx_osnr_linear

        # All other impairments start at zero
        spectral_info.nli[:] = 0
        spectral_info.chromatic_dispersion[:] = 0
        spectral_info.pmd[:] = 0
        spectral_info.pdl[:] = 0

        return spectral_info

    else:
        # Destination transceiver
        # Compute final GSNR and check feasibility

        for i in range(spectral_info.num_channels):
            # 1. Calculate OSNR (signal vs. ASE only)
            osnr_linear = spectral_info.signal[i] / spectral_info.ase[i]
            osnr_db = 10 * np.log10(osnr_linear)

```

```
            # 2. Calculate GSNR (signal vs. ASE + NLI)
            total_noise = spectral_info.ase[i] + spectral_info.nli[i]
            gsnr_linear = spectral_info.signal[i] / total_noise
            gsnr_db = 10 * np.log10(gsnr_linear)

            # 3. Apply impairment penalties
            # These reduce effective GSNR

            # CD penalty
            cd_penalty_db = self.calculate_cd_penalty(
                spectral_info.chromatic_dispersion[i],
                spectral_info.baud_rate[i]
            )

            # PMD penalty
            pmd_penalty_db = self.calculate_pmd_penalty(
                spectral_info.pmd[i],
                spectral_info.baud_rate[i]
            )

            # PDL penalty
            pdl_penalty_db = self.calculate_pdl_penalty(
                spectral_info.pdl[i]
            )

            # Total penalty
            total_penalty_db = cd_penalty_db + pmd_penalty_db + pdl_penalty_db

            # Effective GSNR
            gsnr_eff_db = gsnr_db - total_penalty_db

            # 4. Check against transceiver mode requirement
            required_osnr_db = self.mode.OSNR
            implementation_margin = 2.0  # dB

            if gsnr_eff_db >= (required_osnr_db + implementation_margin):
                feasible = True
                margin_db = gsnr_eff_db - required_osnr_db
            else:
                feasible = False
                margin_db = gsnr_eff_db - required_osnr_db  # Negative

            # Store results
            self.results[i] = {
                'osnr_db': osnr_db,
                'gsnr_db': gsnr_db,
                'gsnr_eff_db': gsnr_eff_db,
                'cd_penalty_db': cd_penalty_db,
                'pmd_penalty_db': pmd_penalty_db,
                'pdl_penalty_db': pdl_penalty_db,
                'feasible': feasible,
                'margin_db': margin_db
            }

        return spectral_info
```

Listing 53: Transceiver.$_{call()Implementation}$

GSNR Calculation The generalized SNR accounts for all noise sources:

$$\text{GSNR} = \frac{P_{\text{signal}}}{P_{\text{ASE}} + P_{\text{NLI}}} \tag{36}$$

In dB:

$$\text{GSNR}_{\text{dB}} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{ASE}} + P_{\text{NLI}}} \right) \tag{37}$$

Example:

```
# At destination after propagation
P_signal = 1e-3      # 1 mW (0 dBm)
P_ASE = 1e-5         # 0.01 mW from amplifiers
P_NLI = 5e-6         # 0.005 mW from fiber nonlinearity

# OSNR (without NLI)
OSNR = P_signal / P_ASE
OSNR_dB = 10 * np.log10(OSNR)  # 20 dB

# GSNR (with NLI)
GSNR = P_signal / (P_ASE + P_NLI)
GSNR_dB = 10 * np.log10(GSNR)  # 17.2 dB

# NLI penalty = OSNR - GSNR = 2.8 dB
```

Impairment Penalties **1. Chromatic Dispersion Penalty:**

$$\text{Penalty}_{\text{CD}} = \alpha_{\text{CD}} \left(\frac{CD \cdot B_s^2}{1000} \right)^2 \tag{38}$$

where:

- CD: Accumulated dispersion (ps/nm)
- B_s: Symbol rate (GHz)
- α_{CD}: Modulation format factor

2. PMD Penalty:

$$\text{Penalty}_{\text{PMD}} = \begin{cases} 0 & \text{if PMD} < 0.1 T_s \\ \alpha_{\text{PMD}} \left(\frac{\text{PMD}}{T_s} \right)^2 & \text{otherwise} \end{cases} \tag{39}$$

where $T_s = 1/B_s$ is the symbol period.

3. PDL Penalty:

$$\text{Penalty}_{\text{PDL}} = \frac{\text{PDL}_{\text{dB}}}{2} \tag{40}$$

(Conservative estimate for worst-case polarization alignment)

Feasibility Check Feasibility is determined by comparing effective GSNR to mode requirements:

```
# Mode requirement (from transceiver equipment library)
required_osnr_db = mode.OSNR  # e.g., 15.0 dB for 16QAM

# Implementation margin (safety factor)
margin_db = 2.0  # dB

# Threshold for feasibility
threshold_db = required_osnr_db + margin_db  # 17.0 dB

# Check
if gsnr_eff_db >= threshold_db:
    feasibility = True
    print(f"Feasible with {gsnr_eff_db - required_osnr_db:.1f} dB margin")
else:
    feasibility = False
    print(f"Infeasible: {gsnr_eff_db:.1f} dB < {threshold_db:.1f} dB required")
    print(f"Shortfall: {threshold_db - gsnr_eff_db:.1f} dB")
```

Listing 54: Feasibility Determination

Receiver Noise Components At the receiver, multiple noise sources affect signal detection:

1. **Shot Noise**:
   - Quantum noise from discrete photon detection
   - Fundamental noise floor in photodetection
   - Power spectral density: $S_{\text{shot}} = 2qI_{\text{photo}}$ (A²/Hz)
   - Becomes dominant at low signal powers
2. **Thermal Noise**:
   - Johnson-Nyquist noise from receiver electronics
   - Power: $P_{\text{thermal}} = kTB$ where k is Boltzmann constant
   - Dominant in direct detection systems
   - Reduced through transimpedance amplifier design
3. **Beat Noise**:
   - **Signal-ASE Beat Noise**: Mixing of signal with ASE in photodetector
   - **ASE-ASE Beat Noise**: Mixing of ASE with itself
   - Dominant noise in coherent detection
   - Proportional to ASE power

Receiver Noise Model:

In coherent receivers (which GNPy primarily models), the dominant noise is signal-ASE beat noise:

$$\text{SNR}_{\text{receiver}} = \frac{P_{\text{signal}}}{2P_{\text{ASE}} + P_{\text{shot}} + P_{\text{thermal}}} \tag{41}$$

For high OSNR systems (typical in optical networks):

$$\text{SNR}_{\text{receiver}} \approx \frac{P_{\text{signal}}}{2P_{\text{ASE}}} = \frac{\text{OSNR}}{2} \tag{42}$$

This is why OSNR is the primary metric - it directly relates to electrical SNR at the receiver.

GNPy Receiver Modeling:

GNPy models receiver performance through:

- GSNR threshold comparison (captures signal-ASE beat noise)
- Impairment penalties (CD, PMD, PDL)
- Transceiver OSNR requirement (includes receiver sensitivity)
- Shot and thermal noise are implicit in the mode OSNR requirement

```
# The transceiver mode OSNR requirement implicitly includes:
# - Photodetector quantum efficiency
# - Receiver thermal noise floor
# - Required BER (e.g., 1e-15 for hard-decision FEC)
# - Implementation margin

# Example mode definition:
mode = {
    'format': '16QAM',
    'baud_rate': 32e9,
    'OSNR': 15.0,  # dB - includes all receiver imperfections
    'bit_rate': 128e9
}

# GNPy checks: GSNR_eff >= mode.OSNR + margin
# This ensures receiver can achieve target BER
```

Listing 55: Receiver Noise in Context

Receiver Responsivity and PLO Additional receiver considerations (typically absorbed in OSNR requirement):

- **Photodetector Responsivity**: Conversion efficiency from optical to electrical power
- **PLO (Phase-Locked Oscillator)**: Local oscillator stability in coherent receivers
- **Ps signal**: Power of signal component in coherent mixing

These are accounted for in the transceiver mode characterization rather than modeled explicitly.

3.4.5 Science Utils Module Integration

The propagation engine relies heavily on the `gnpy.core.science_utils` module for physics-based calculations. This module provides the computational solvers for complex physical phenomena.

Module Architecture The `science_utils` module is organized into three main solver categories:

Solver Category	Components
NLI Solvers	GN model, GGN model, NLI computation functions
Raman Solvers	Raman gain computation, pump-signal interaction
Amplifier Models	EDFA noise figure, gain profiles, ASE calculation

Table 64: Science Utils Solver Categories

NLI Solver Functions Key Functions for Nonlinear Interference:

Function	Purpose
`NliSolver`	Main class for NLI computation
`_gn_analytic()`	Analytical GN model implementation
`_gn_generalized_psii()`	Generalized GN (GGN) model with spectral separation
`_psi()`	Phase-matching function for GN model
`compute_nli()`	Compute NLI for all channels

Table 65: NLI Solver Functions (from Diagram 2)

GN Model Implementation:

```python
def _gn_analytic(fiber, spectral_info, nli_params):
    """
    Analytical GN model implementation.

    Based on Poggiolini equations (Eq 4.30, 5.1 from references).

    Parameters
    ----------
    fiber : Fiber object
        Fiber with gamma, dispersion, loss parameters
    spectral_info : SpectralInformation
        Spectrum with channel powers, frequencies
    nli_params : dict
        NLI solver parameters

    Returns
    -------
    nli_power : array
        NLI power per channel [W]
    """

    # Extract fiber parameters
    gamma = fiber.params.gamma  # Nonlinear coefficient
    beta2 = fiber.params.beta2  # Dispersion parameter
    alpha = fiber.params.loss_coef  # Loss coefficient
    length = fiber.params.length
```

```

    # Calculate effective length
    L_eff = (1 - np.exp(-alpha * length)) / alpha

    # For each channel, compute NLI from all interfering channels
    nli_power = np.zeros(len(spectral_info.frequency))

    for i in range(len(spectral_info.frequency)):
        # GN model equation (simplified)
        nli_power[i] = compute_gn_integral(
            i,
            spectral_info,
            gamma,
            L_eff,
            beta2
        )

    return nli_power
```

Listing 56: GN Analytic Function

GGN Model Implementation:

```
def _gn_generalized_psii(fiber, spectral_info, nli_params):
    """
    Generalized GN model with spectral separation.

    More accurate than simple GN, accounts for:
    - Spectral separation between channels
    - Different modulation formats per channel
    - Improved phase-matching calculation

    Based on GGN equations with PSII formulation.
    """

    # Compute enhanced phase-matching efficiency
    # Accounts for walk-off between channels

    nli_power = np.zeros(len(spectral_info.frequency))

    for i in range(len(spectral_info.frequency)):
        # Sum contributions from all channel pairs
        for j in range(len(spectral_info.frequency)):
            for k in range(len(spectral_info.frequency)):
                # Triple integral over frequency
                nli_contribution = compute_ggn_triple_integral(
                    i, j, k,
                    spectral_info,
                    fiber,
                    nli_params
                )
                nli_power[i] += nli_contribution

    return nli_power
```

Listing 57: GGN Generalized PSII Function

Raman Solver Functions Key Functions for Raman Scattering:

Function	Purpose
RamanSolver	Main class for Raman computation
raman_solver()	Solve pump/loss profiles along fiber
calculate_attenuation_profile()	Fiber loss with Raman gain
Raman_gain_rho()	Raman gain coefficient
create_lumped_losses()	Apply Raman effect as effective loss reduction
calculate_spontaneous_raman_scattering()	Spontaneous Raman noise

Table 66: Raman Solver Functions (from Diagram 2)

Raman Solver Implementation:

```python
def raman_solver(fiber, spectral_info, pump_power, pump_frequency):
    """
    Solve Raman equations for pump and signal evolution.

    Solves coupled ODEs:
    dP_signal/dz = -  P_signal   + g_R P_pump P_signal
    dP_pump/dz = -  _pumpP_pump   -   (g_R P_signal_i P_pump)

    Parameters
    ----------
    fiber : RamanFiber
        Fiber with Raman efficiency profile
    spectral_info : SpectralInformation
        Signal spectrum
    pump_power : float
        Raman pump power [W]
    pump_frequency : float
        Pump frequency [Hz]

    Returns
    -------
    signal_profile : array
        Signal power vs. distance along fiber
    gain_profile : array
        Raman gain vs. distance
    """

    # Discretize fiber into segments
    num_segments = fiber.params.length * 1000 / 50  # 50m resolution
    z = np.linspace(0, fiber.params.length * 1000, int(num_segments))

    # Initialize power profiles
    P_signal = np.zeros((len(spectral_info.frequency), len(z)))
    P_pump = np.zeros(len(z))

    # Boundary conditions
    P_signal[:, 0] = spectral_info.signal
    P_pump[0] = pump_power

    # Solve ODEs using RK4 or similar
    for i in range(1, len(z)):
        # Update pump power
        P_pump[i] = update_pump_power(P_pump[i-1], P_signal[:, i-1], ...)

        # Update signal powers
```

```
        for ch in range(len(spectral_info.frequency)):
            P_signal[ch, i] = update_signal_power(
                P_signal[ch, i-1],
                P_pump[i],
                raman_gain_coefficient,
                ...
            )

    return P_signal, P_pump
```

Listing 58: Raman Solver Usage

Amplifier Modeling Functions **Key Functions for EDFA Modeling:**

Function	Purpose
`edfa_nf()`	Calculate EDFA noise figure
`nf_model()`	NF modeling with different methods
`nf_polynomial()`	Polynomial NF vs. wavelength
`nf_gain_dependent()`	NF vs. gain operating point
`effective_length()`	Calculate fiber effective length for gain
`nsp_to_nf()`	Convert spontaneous emission factor to NF
`nf_to_nsp()`	Convert NF to spontaneous emission factor

Table 67: Amplifier Modeling Functions

Noise Figure Calculation:

```
def edfa_nf(frequency, gain_db, edfa_params):
    """
    Calculate EDFA noise figure for given conditions.

    Supports multiple NF models:
    - Flat NF (constant across spectrum)
    - Polynomial NF vs. wavelength
    - Gain-dependent NF
    - Advanced NF from measured data

    Parameters
    ----------
    frequency : array
        Channel frequencies [Hz]
    gain_db : float
        Amplifier gain [dB]
    edfa_params : dict
        EDFA parameters including NF model

    Returns
    -------
    nf_db : array
        Noise figure per frequency [dB]
    """

    # Convert frequency to wavelength
    c = 299792458  # m/s
    wavelength_nm = (c / frequency) * 1e9

```

```
    if edfa_params['nf_model'] == 'flat':
        # Constant NF
        nf_db = np.full(len(frequency), edfa_params['nf_min'])

    elif edfa_params['nf_model'] == 'polynomial':
        # NF(  ) = a0 + a1*   + a2*     + a3*
        coeffs = edfa_params['nf_coef']
        nf_db = np.polyval(coeffs, wavelength_nm)

    elif edfa_params['nf_model'] == 'gain_dependent':
        # NF varies with gain setting
        nf_min = edfa_params['nf_min']
        nf_max = edfa_params['nf_max']
        gain_nom = edfa_params['gain_flatmax']

        # Quadratic dependence on gain deviation
        delta_g = gain_db - gain_nom
        nf_db = nf_min + (nf_max - nf_min) * (delta_g / 10) ** 2
        nf_db = np.clip(nf_db, nf_min, nf_max)
        nf_db = np.full(len(frequency), nf_db)

    elif edfa_params['nf_model'] == 'advanced':
        # Interpolate from measured data
        nf_db = interpolate_nf_from_table(
            frequency,
            gain_db,
            edfa_params['nf_table']
        )

    return nf_db
```

Listing 59: EDFA Noise Figure Calculation

Integration with Element Classes The science utils functions are called from element propagation methods:

```
# In Fiber.__call__()
from gnpy.core.science_utils import NliSolver

nli_solver = NliSolver(fiber_params, nli_params, spectral_info)
nli_power = nli_solver.compute_nli()

# In RamanFiber.__call__()
from gnpy.core.science_utils import RamanSolver

raman_solver = RamanSolver(fiber_params, raman_params, spectral_info)
spectral_info = raman_solver.propagate(spectral_info)

# In Edfa.__call__()
from gnpy.core.science_utils import edfa_nf

nf_db = edfa_nf(spectral_info.frequency, gain_db, edfa_params)
```

Listing 60: Science Utils Integration Pattern

This modular design separates:

- **Element classes**: Network topology and data flow
- **Science utils**: Physics-based computation solvers

- **Equipment library**: Physical parameters and configurations

---

3.4.6 Output Metrics

After propagation through all elements, the engine returns comprehensive per-channel metrics used for feasibility determination and performance reporting.

Returned Metrics per Channel

Metric	Unit	Description	Usage
`gsnr_db`	dB	Generalized SNR (includes NLI)	Primary feasibility metric
`osnr_db`	dB	Optical SNR (ASE only)	Legacy metric, troubleshooting
`gsnr_eff_db`	dB	GSNR after penalties	Final feasibility check
`power_dbm`	dBm	Signal power at destination	Power budget verification
`cd_ps_nm`	ps/nm	Accumulated chromatic dispersion	Transceiver compensation check
`pmd_ps`	ps	Accumulated PMD	Impairment budget check
`pdl_db`	dB	Accumulated PDL	Penalty calculation
`nli_power_w`	W	NLI power	Nonlinearity contribution
`ase_power_w`	W	ASE power	Linear noise contribution
`latency_ms`	ms	Propagation delay	Timing analysis
`feasibility`	boolean	Pass/fail determination	Service acceptance
`margin_db`	dB	GSNR margin over requirement	Quality indicator

Table 68: Propagation Engine Output Metrics

```python
def extract_metrics_from_spectral_info(spectral_info, transceiver_mode):
    """Extract and format metrics after propagation."""

    metrics = []

    for i in range(spectral_info.num_channels):
        # Basic power metrics
        signal_power = spectral_info.signal[i]
        ase_power = spectral_info.ase[i]
        nli_power = spectral_info.nli[i]

        # Calculate SNRs
        osnr_linear = signal_power / ase_power
        gsnr_linear = signal_power / (ase_power + nli_power)

        osnr_db = 10 * np.log10(osnr_linear)
        gsnr_db = 10 * np.log10(gsnr_linear)

        # Reference bandwidth conversion (to 0.1 nm)
        osnr_0_1nm = convert_to_01nm(osnr_db, spectral_info.baud_rate[i])
```

```
        # Linear impairments
        cd = spectral_info.chromatic_dispersion[i]
        pmd = spectral_info.pmd[i]
        pdl = spectral_info.pdl[i]
        latency = spectral_info.latency[i]

        # Penalties
        cd_penalty = calculate_cd_penalty(cd, spectral_info.baud_rate[i])
        pmd_penalty = calculate_pmd_penalty(pmd, spectral_info.baud_rate[i])
        pdl_penalty = calculate_pdl_penalty(pdl)

        total_penalty = cd_penalty + pmd_penalty + pdl_penalty
        gsnr_eff = gsnr_db - total_penalty

        # Feasibility check
        required_osnr = transceiver_mode.OSNR
        margin = gsnr_eff - required_osnr
        feasible = (margin >= 2.0)  # 2 dB implementation margin

        # Compile metrics for this channel
        channel_metrics = {
            'channel_number': i + 1,
            'frequency_thz': spectral_info.frequency[i] / 1e12,
            'power_dbm': 10 * np.log10(signal_power * 1000),
            'osnr_db': osnr_db,
            'osnr_0.1nm': osnr_0_1nm,
            'gsnr_db': gsnr_db,
            'gsnr_eff_db': gsnr_eff,
            'ase_power_w': ase_power,
            'nli_power_w': nli_power,
            'cd_ps_nm': cd,
            'pmd_ps': pmd,
            'pdl_db': pdl,
            'cd_penalty_db': cd_penalty,
            'pmd_penalty_db': pmd_penalty,
            'pdl_penalty_db': pdl_penalty,
            'total_penalty_db': total_penalty,
            'latency_ms': latency * 1000,
            'required_osnr_db': required_osnr,
            'margin_db': margin,
            'feasible': feasible
        }

        metrics.append(channel_metrics)

    return metrics
```

Listing 61: Extracting Metrics After Propagation

Metrics Aggregation For path-level reporting, channel metrics are aggregated:

```
def aggregate_path_metrics(channel_metrics):
    """Aggregate per-channel metrics to path level."""

    # Extract arrays
    gsnr_values = [m['gsnr_eff_db'] for m in channel_metrics]
    margin_values = [m['margin_db'] for m in channel_metrics]
```

```
    feasibility = [m['feasible'] for m in channel_metrics]

    path_metrics = {
        # Worst-case metrics
        'worst_gsnr_db': min(gsnr_values),
        'worst_margin_db': min(margin_values),
        'best_gsnr_db': max(gsnr_values),
        'best_margin_db': max(margin_values),

        # Average metrics
        'avg_gsnr_db': np.mean(gsnr_values),
        'avg_margin_db': np.mean(margin_values),

        # Feasibility
        'num_channels': len(channel_metrics),
        'num_feasible': sum(feasibility),
        'all_feasible': all(feasibility),

        # Summary
        'path_feasible': all(feasibility),
        'limiting_channel': channel_metrics[np.argmin(gsnr_values)]['channel_number']
    }

    return path_metrics
```

Listing 62: Path-Level Metrics Aggregation

OpenROADM Metrics Mapping GNPy metrics map to OpenROADM service validation parameters:

GNPy Metric	OpenROADM Equivalent
`gsnr_eff_db`	Computed GSNR for path feasibility
`power_dbm`	`media-channel-egress-power`
`cd_ps_nm`	`chromatic-dispersion`
`pmd_ps`	`pmd`
`pdl_db`	`pdl`
`feasibility`	Service validation result
`margin_db`	Margin over operational mode requirement

Table 69: GNPy to OpenROADM Metrics Mapping

Summary: Complete Propagation Flow The following diagram summarizes the complete data flow through the propagation engine:

```
INPUT: Service Request + Network Topology
  ↓
CREATE: SpectralInformation
  - Initialize signal, frequency, baud_rate
  - Set ase = 0, nli = 0, cd = 0, pmd = 0, pdl = 0
  ↓
FOR EACH ELEMENT IN PATH:
  ↓
  Transceiver (source) → Set TX OSNR
  ↓
  Fiber → Attenuate, Add NLI, Accumulate CD/PMD
  ↓
  Edfa → Amplify, Add ASE
  ↓
  Roadm → Equalize, Apply Loss, Add ASE
  ↓
  ... (repeat for all elements)
  ↓
  Transceiver (dest) → Calculate GSNR, Check Feasibility
  ↓
OUTPUT: Metrics per Channel
  - GSNR, OSNR, Power
  - CD, PMD, PDL
  - Feasibility, Margin
```

Figure 10: Complete Propagation Engine Flow

3.5 Output Layer

3.5.1 Overview

GNPy provides multiple output formats to accommodate different use cases and integration requirements. The output selection depends on the execution path used and command-line flags specified. All outputs contain the results of optical propagation simulation, but differ in format, level of detail, and intended audience.

Output Type	Use Case	How to Generate
`response.json`	Path computation results	Default output
`auto_designed_topology.json`	Network design verification	`-output` flag
CSV reports	Human-readable statistics	`-output file.csv`
API responses	System integration	ONOS/OpenROADM APIs

Table 70: GNPy Output Types

```python
# From cli_examples.py - Output generation logic
if args.output:
    if args.output.suffix.lower() == '.json':
        save_json(path_result_json, args.output)
        print(f'Saved JSON to {args.output}')
    elif args.output.suffix.lower() == '.csv':
        with open(args.output, "w", encoding='utf-8') as fcsv:
            jsontocsv(path_result_json, equipment, fcsv)
        print(f'Saved CSV to {args.output}')
```

3.5.2 response.json - Path Computation Results

Purpose The `response.json` file contains the complete path computation results in IETF TEAS YANG format (draft-ietf-teas-yang-path-computation). This is the primary output format for integration with network controllers and path computation engines (PCE).

Structure Overview The response follows the IETF standard for path computation responses:

```
{
  "response": [
    {
      "response-id": "service-1",
      "path-properties": {
        "path-metric": [
          {
            "metric-type": "OSNR-bandwidth",
            "accumulative-value": 18.5
          },
          {
            "metric-type": "OSNR-0.1nm",
            "accumulative-value": 21.3
          },
          {
            "metric-type": "SNR-bandwidth",
            "accumulative-value": 15.2
          }
        ],
        "path-route-objects": [...]
      },
      "response-type": "path-found"  // or "no-path"
    }
  ]
}
```

Content Details

Top-Level Structure

Field	Type	Description
`response`	Array	List of path computation responses, one per request
`response-id`	string	Matches `request-id` from service request
`response-type`	string	`"path-found"` or `"no-path"`
`no-path`	Object	Present only if path not found
`path-properties`	Object	Detailed path metrics and route

Table 71: response.json Top-Level Fields

Path Metrics GNPy computes comprehensive Quality of Transmission (QoT) metrics:

Metric Type	Units	Description
`OSNR-bandwidth`	dB	OSNR calculated at signal bandwidth
`OSNR-0.1nm`	dB	OSNR at reference 0.1nm (12.5 GHz)
`SNR-bandwidth`	dB	SNR including NLI at signal bandwidth
`GSNR-bandwidth`	dB	Generalized SNR (signal/(ASE+NLI))
`reference-power`	dBm	Reference per-channel power
`path-bandwidth`	Hz	Total path spectral occupancy
`CD`	ps/nm	Total chromatic dispersion
`PMD`	ps	Total polarization mode dispersion
`PDL`	dB	Total polarization dependent loss

Table 72: Path Metrics in response.json

Path Route Objects The `path-route-objects` array contains the complete traversed path:

```
"path-route-objects": [
  {
    "path-route-object": {
      "index": 0,
      "num-unnum-hop": {
        "node-id": "Paris",
        "link-tp-id": "Paris"
      }
    }
  },
  {
    "path-route-object": {
      "index": 1,
      "num-unnum-hop": {
        "node-id": "Fiber_Paris_Lyon",
        "link-tp-id": "Fiber_Paris_Lyon"
      }
    }
  },
  ...
]
```

Per-Channel Metrics Each response includes detailed per-channel results:

```
"path-properties": {
  "z-a-path-metric": [
    {
      "metric-type": "osnr-0.1nm",
      "accumulative-value": [
        {"value": 21.5, "frequency": 191.35e12},
        {"value": 21.3, "frequency": 191.40e12},
        ...
      ]
    },
    {
      "metric-type": "chromatic-dispersion",
      "accumulative-value": [
```

```
        {"value": 1340, "frequency": 191.35e12},
        {"value": 1345, "frequency": 191.40e12},
        ...
      ]
    }
  ]
}
```

Feasibility Status

```
{
  "response-id": "service-1",
  "response-type": "path-found",
  "path-properties": {
    "path-metric": [...],
    "path-route-objects": [...]
  }
}
```

No Path Response When a path cannot be found or fails feasibility checks:

```
{
  "response-id": "service-2",
  "response-type": "no-path",
  "no-path": {
    "no-path-cause": "NO_FEASIBLE_MODE",
    "no-path-message": "GSNR too low: computed 12.3 dB < required 15.0 dB"
  }
}
```

Blocking Reasons

Cause	Description
NO_FEASIBLE_MODE	No transceiver mode meets GSNR requirement
NO_PATH	No path exists between source and destination
NO_SPECTRUM	Insufficient spectrum available
MODE_NOT_FEASIBLE	Requested mode cannot achieve required QoT
PATH_CONSTRAINT_VIOLATION	Explicit path violates topology constraints

Table 73: Path Blocking Reasons

How to Generate

```
# Basic usage - response.json created by default
gnpy-path-request -e eqpt_config.json network.json services.json

# Specify output filename
gnpy-path-request -e eqpt_config.json network.json services.json \
  -o path_results.json
```

```python
from gnpy.tools.json_io import load_equipment, load_network, \
    requests_from_json, save_json
from gnpy.topology.request import compute_path_dsjctn, \
    compute_path_with_disjunction

# Load configuration
equipment = load_equipment('eqpt_config.json')
network = load_network('topology.json', equipment)

# Load and process requests
with open('services.json') as f:
    service_data = json.load(f)
rqs = requests_from_json(service_data, equipment)

# Compute paths
paths = compute_path_dsjctn(network, equipment, rqs, [])
propagated_paths, _, _ = compute_path_with_disjunction(
    network, equipment, rqs, paths)

# Generate response
results = []
for i, pth in enumerate(propagated_paths):
    results.append(ResultElement(rqs[i], pth, None))

response_json = _path_result_json(results)
save_json(response_json, 'response.json')
```

OpenROADM Alignment The response format aligns with OpenROADM Service Model for path computation results:

GNPy Field	OpenROADM Field	Mapping
`response-id`	`service-name`	Direct mapping
`GSNR-bandwidth`	`calculated-gsnr`	From path-computation-results
`OSNR-0.1nm`	`calculated-osnr`	From path-computation-results
`path-route-objects`	`a-to-z/network-topology`	Node sequence
`response-type`	`configuration-response-common`	Success/failure

Table 74: GNPy to OpenROADM Response Mapping

3.5.3 auto_designed_topology.json - Network Design Output

Purpose The `auto_designed_topology.json` file contains the complete network topology after GNPy's automatic amplifier placement and power optimization. This output is essential for:

- Verifying amplifier placement decisions
- Extracting gain and power settings for provisioning
- Comparing designed vs. input topology
- Debugging network design issues

Content

Placed EDFAs All amplifiers placed by `design_network()` include complete operational parameters:

```
{
  "uid": "Edfa_autodesign_Paris_Lyon",
  "type": "Edfa",
  "type_variety": "std_medium_gain",
  "operational": {
    "gain_target": 20.5,
    "delta_p": 0.0,
    "tilt_target": 0.0,
    "out_voa": 0.0
  },
  "metadata": {
    "location": {
      "latitude": 48.5,
      "longitude": 2.8,
      "city": "auto-placed",
      "region": "IDF"
    }
  },
  "params": {
    "f_min": 191.35e12,
    "f_max": 196.1e12,
    "gain_flatmax": 27.0,
    "gain_min": 15.0,
    "p_max": 21.0,
    "nf_model": "standard",
    "nf_fit_coeff": [...]
  }
}
```

Set Gains and Powers Each amplifier in the designed topology has computed operational parameters:

Parameter	Units	Description
`gain_target`	dB	Amplifier gain setting
`delta_p`	dB	Power adjustment from reference
`tilt_target`	dB	Gain tilt (slope) across band
`out_voa`	dB	Output variable optical attenuator
`effective_gain`	dB	Actual gain after VOA
`effective_pch`	dBm	Per-channel output power

Table 75: EDFA Operational Parameters

ROADM Configurations ROADMs include equalization targets and per-degree settings:

```
{
  "uid": "ROADM_Lyon",
  "type": "Roadm",
  "params": {
    "target_pch_out_db": -20,
    "per_degree_pch_out_db": {
```

```
      "degree_1": -20,
      "degree_2": -19,
      "degree_3": -21
    }
  }
}
```

Fiber Updated Parameters Fibers include updated loss and PMD accumulation:

```
{
  "uid": "Fiber_Paris_Lyon",
  "type": "Fiber",
  "type_variety": "SSMF",
  "params": {
    "length": 80.0,
    "length_units": "km",
    "loss_coef": 0.2,
    "total_loss": 16.5,
    "pmd": 8.94,
    "con_in": 0.5,
    "con_out": 0.5
  }
}
```

How to Generate

```
# Auto-designed topology is saved automatically
gnpy-transmission-example -e eqpt_config.json network.json \
  --output designed_network.json

# The file contains the modified topology with all placements
```

```
# Only generated if auto-design performed
gnpy-path-request -e eqpt_config.json network.json services.json \
  --output results.json

# Check for auto_designed_topology.json in output directory
# Created only if design_network() was called
```

```
from gnpy.tools.json_io import load_equipment, load_network, save_network
from gnpy.core.network import design_network
from gnpy.core.info import create_input_spectral_information

# Load and design
equipment = load_equipment('eqpt_config.json')
network = load_network('topology.json', equipment)

# Create reference channel for design
spectral_info = create_input_spectral_information(
    f_min=191.35e12,
    f_max=196.1e12,
    spacing=50e9,
    power=1e-3,  # 0 dBm
```

```
    equipment=equipment
)

# Design network
design_network(spectral_info.carriers[0], network, equipment)

# Save designed topology
save_network(network, 'auto_designed_topology.json')
```

Use Cases

Verification Workflow

1. Run simulation with auto-design
2. Review `auto_designed_topology.json`
3. Verify amplifier placements are reasonable
4. Check gains are within equipment limits
5. Validate power levels throughout network
6. Compare with manual design if applicable

```
# Extract amplifier settings for provisioning
import json

with open('auto_designed_topology.json') as f:
    designed = json.load(f)

for element in designed['elements']:
    if element['type'] == 'Edfa':
        print(f"Amplifier: {element['uid']}")
        print(f"  Gain: {element['operational']['gain_target']} dB")
        print(f"  Tilt: {element['operational']['tilt_target']} dB")
        print(f"  Type: {element['type_variety']}")
        print()
```

3.5.4 CSV Reports - Human-Readable Statistics

Purpose CSV output provides human-readable tabular data suitable for:

- Quick review in spreadsheet applications
- Statistical analysis and trending
- Report generation
- Comparison across multiple simulations

Content Structure

Summary Section The CSV begins with a summary table:

```
request_id,source,destination,GSNR (dB),OSNR (dB),OSNR+margin (dB),
  mode,path_bandwidth (GHz),nb_channels,spectrum (N,M)
service-1,Paris,Marseille,18.5,21.3,24.3,DP-QPSK,200,4,(96,100)
service-2,Lyon,Nice,15.2,18.1,21.1,DP-16QAM,400,8,(100,108)
```

Per-Channel Details Detailed metrics for each channel on each path:

```
request_id,channel_frequency (THz),channel_lambda (nm),GSNR (dB),
  OSNR (dB),CD (ps/nm),PMD (ps),PDL (dB),
  Carrier_power (dBm),ASE_noise (dBm),NLI_noise (dBm)
service-1,191.35,1566.72,18.5,21.3,1340,8.9,0.5,-20.0,-40.1,-41.5
service-1,191.40,1566.32,18.3,21.2,1345,8.9,0.5,-20.0,-40.2,-41.4
...
```

Column Descriptions

Column	Units	Description
`request_id`	-	Service identifier
`source`	-	Source transceiver node
`destination`	-	Destination transceiver node
`GSNR (dB)`	dB	Generalized SNR (mean across channels)
`OSNR (dB)`	dB	OSNR at 0.1nm (mean across channels)
`OSNR+margin (dB)`	dB	OSNR including system margins
`mode`	-	Selected transceiver mode
`path_bandwidth (GHz)`	GHz	Total spectral occupancy
`nb_channels`	-	Number of channels
`spectrum (N,M)`	-	Slot indices (start, end)
`channel_frequency (THz)`	THz	Channel center frequency
`channel_lambda (nm)`	nm	Channel wavelength
`CD (ps/nm)`	ps/nm	Accumulated chromatic dispersion
`PMD (ps)`	ps	Accumulated PMD
`PDL (dB)`	dB	Accumulated PDL
`Carrier_power (dBm)`	dBm	Signal power at receiver
`ASE_noise (dBm)`	dBm	ASE noise power
`NLI_noise (dBm)`	dBm	Nonlinear interference power

Table 76: CSV Column Definitions

Generation Methods

```
# Generate CSV directly from path-request
gnpy-path-request -e eqpt_config.json network.json services.json \
  -o results.csv

# CSV file created with summary and per-channel data
```

```
# Convert existing JSON response to CSV
python write_path_jsontocsv.py response.json output.csv eqpt_config.json

# Arguments:
#   response.json - Input path results (JSON)
#   output.csv - Output CSV file
#   eqpt_config.json - Equipment library (for mode details)
```

```
from gnpy.topology.request import jsontocsv
from gnpy.tools.json_io import load_equipment
import json

# Load results and equipment
with open('response.json') as f:
    results = json.load(f)
equipment = load_equipment('eqpt_config.json')

# Generate CSV
with open('output.csv', 'w', encoding='utf-8') as fcsv:
    jsontocsv(results, equipment, fcsv)
```

Example Analysis Workflow

Excel/LibreOffice Analysis

1. Generate CSV output
2. Open in spreadsheet application
3. Create pivot tables for summary statistics
4. Generate charts showing:
    - GSNR distribution across services
    - Per-channel OSNR variations
    - Chromatic dispersion accumulation
5. Export charts for documentation

```
import pandas as pd
import matplotlib.pyplot as plt

# Load CSV
df = pd.read_csv('results.csv')

# Summary statistics
print("GSNR Statistics:")
print(df.groupby('request_id')['GSNR (dB)'].describe())

# Plot OSNR distribution
df.boxplot(column='OSNR (dB)', by='request_id')
plt.title('OSNR Distribution by Service')
plt.suptitle('')  # Remove default title
```

```
plt.ylabel('OSNR (dB)')
plt.savefig('osnr_distribution.png')

# Identify margin violations
margin_threshold = 15.0
violations = df[df['GSNR (dB)'] < margin_threshold]
print(f"Channels below {margin_threshold} dB: {len(violations)}")
```

3.5.5 API Interfaces

Overview GNPy provides standardized APIs for integration with network management systems and controllers. These interfaces enable real-time path computation and QoT estimation within automated workflows.

ONOS Integration (QVOS/TAPI)

Purpose ONOS (Open Network Operating System) integration provides path computation services via:

- QVOS (Quality of Transmission Verification for Optical Systems)
- TAPI (Transport API) - ONF standard

```
{
  "tapi-path-computation:input": {
    "service-port": [
      {
        "local-id": "source-port",
        "service-interface-point": {"service-interface-point-uuid": "src-uuid"}
      },
      {
        "local-id": "dest-port",
        "service-interface-point": {"service-interface-point-uuid": "dst-uuid"}
      }
    ],
    "path-optimization-constraint": {
      "cost-name": ["osnr"],
      "objective-function": {
        "bandwidth-optimization": "minimize-cost"
      }
    }
  }
}
```

```
{
  "tapi-path-computation:output": {
    "service": {
      "path": [
        {
          "path-element": [
```

```
            {"port-id": "Paris", "path-element-direction": "unidirectional"},
            {"port-id": "Lyon", "path-element-direction": "unidirectional"}
          ],
          "routing-constraint": {
            "cost-characteristic": [
              {"cost-name": "osnr", "cost-value": "21.3"}
            ]
          }
        }
      ]
    }
  }
}
```

Integration Architecture

```
ONOS Controller
    â†"
[TAPI Northbound Interface]
    â†"
[GNPy Service]
    â†" (REST API)
[GNPy Engine]
    â†"
[Network Topology Database]
```

OpenROADM Path Computation

Service Interface OpenROADM defines standard RPCs for optical path computation:

```
// RPC: service-create
{
  "input": {
    "service-name": "service-1",
    "connection-type": "service",
    "service-a-end": {
      "service-format": "Ethernet",
      "service-rate": "100",
      "clli": "Paris",
      "node-id": "ROADM-Paris"
    },
    "service-z-end": {
      "service-format": "Ethernet",
      "service-rate": "100",
      "clli": "Marseille",
      "node-id": "ROADM-Marseille"
    },
    "hard-constraints": {
      "include": {
        "node-id": ["ROADM-Lyon"]
      }
    },
    "soft-constraints": {
      "exclude": {
        "node-id": ["ROADM-Nice"]
```

```
      }
    }
  }
}
```

```
{
  "output": {
    "configuration-response-common": {
      "ack-final-indicator": "Yes",
      "response-code": "200",
      "response-message": "Service created successfully"
    },
    "response-parameters": {
      "path-description": {
        "a-to-z-direction": {
          "rate": 100,
          "modulation-format": "dp-qpsk",
          "a-to-z": [
            {"id": "0", "resource": {"node-id": "ROADM-Paris"}},
            {"id": "1", "resource": {"node-id": "ROADM-Lyon"}},
            {"id": "2", "resource": {"node-id": "ROADM-Marseille"}}
          ]
        }
      },
      "path-computation-results": {
        "calculated-osnr": 21.3,
        "calculated-gsnr": 18.5,
        "tx-power": 0.0
      }
    }
  }
}
```

OpenROADM RPC	GNPy Function	Mapping
`service-create`	Path computation	Request â†’ response
`service-feasibility-check`	QoT estimation	Pre-validation
`temp-service-create`	Trial computation	Test paths
`optical-tunnel-create`	Alien wavelength	External transceivers

Table 77: OpenROADM to GNPy Mapping

Integration Points

NetworkX/Python Direct API

Purpose For Python-based tools and scripts, GNPy provides direct programmatic access without JSON serialization overhead.

```
from gnpy.core.equipment import load_equipment
from gnpy.topology.request import PathRequest, propagate
from gnpy.core.network import build_network
import networkx as nx
```

```

# Load equipment and build network
equipment = load_equipment('eqpt_config.json')
network = build_network(json_data, equipment)

# Create path request programmatically
request = PathRequest(
    request_id='api-request-1',
    source='Paris',
    destination='Marseille',
    tsp='vendorA_xcvr',
    tsp_mode='DP-QPSK',
    baud_rate=32e9,
    spacing=50e9,
    power=1e-3  # 0 dBm
)

# Compute path using NetworkX
path = nx.shortest_path(network, 'Paris', 'Marseille')

# Propagate and get results
propagated_path = propagate(path, request, equipment)

# Access results directly
for element in propagated_path:
    if hasattr(element, 'osnr'):
        print(f"{element.uid}: OSNR = {element.osnr} dB")
```

```
from gnpy.core.info import SpectralInformation, create_arbitrary_spectral_information
from gnpy.core.elements import Edfa, Fiber, Roadm

# Create custom spectral information
carriers = create_arbitrary_spectral_information(
    frequency=np.linspace(191.35e12, 196.1e12, 96),
    baud_rate=32e9,
    power=dbm2watt(0),
    spacing=50e9,
    roll_off=0.15
)

spectral_info = SpectralInformation(
    carriers=carriers,
    tx_osnr=100  # dB
)

# Propagate through individual elements
for element in path:
    spectral_info = element(spectral_info)

    # Access intermediate results
    if isinstance(element, Fiber):
        nli = spectral_info.nli
        print(f"NLI after {element.uid}: {watt2dbm(nli.signal)} dBm")
    elif isinstance(element, Edfa):
        ase = spectral_info.ase
        print(f"ASE after {element.uid}: {watt2dbm(ase.signal)} dBm")

# Final GSNR calculation
```

```
signal = spectral_info.signal
ase = spectral_info.ase
nli = spectral_info.nli
gsnr = signal / (ase + nli)
gsnr_db = lin2db(gsnr)
print(f"Final GSNR: {gsnr_db} dB")
```

```
from multiprocessing import Pool
from functools import partial

def simulate_request(request, equipment, network):
    """Simulate single request"""
    path = compute_constrained_path(network, request)
    propagated = propagate(path, request, equipment)
    return extract_metrics(propagated)

# Parallel processing
requests = load_bulk_requests('services.json')
simulate = partial(simulate_request,
                   equipment=equipment,
                   network=network)

with Pool(processes=8) as pool:
    results = pool.map(simulate, requests)

# Aggregate results
summary = pd.DataFrame(results)
summary.to_csv('bulk_results.csv')
```

3.5.6 Conversion Utilities

gnpy-convert-xls - Excel to JSON Conversion

Purpose The `gnpy-convert-xls` utility converts Excel-based network definitions to GNPy's native JSON format. This enables users familiar with spreadsheets to create network models without writing JSON directly.

```
# Convert equipment library
gnpy-convert-xls equipment.xlsx -o eqpt_config.json

# Convert network topology
gnpy-convert-xls network.xlsx -o topology.json

# Convert service requests
gnpy-convert-xls services.xlsx -o services.json

# Convert all in one command
gnpy-convert-xls master_file.xlsx --eqpt-sheet Equipment \
  --network-sheet Nodes --links-sheet Links \
  --service-sheet Services \
  -o output_dir/
```

Excel File Structure Equipment Sheet Requirements:

Column	Description
Equipment type	Edfa, Fiber, Roadm, Transceiver
Type variety	Unique identifier
...parameters...	Type-specific columns

Table 78: Equipment Sheet Structure

Nodes Sheet Requirements:

Column (Required)	Description
City	Node unique identifier
Column (Optional)	**Description**
Latitude	Geographic latitude
Longitude	Geographic longitude
Type	ROADM, ILA, FUSED (auto-detected if omitted)
Booster	Booster amplifier restrictions
Preamp	Preamplifier restrictions

Table 79: Nodes Sheet Structure

Links Sheet Requirements:

Column (Required)	Description
Node A	Source node name
Node Z	Destination node name
east Distance (km)	Link length
Column (Optional)	**Description**
west Distance (km)	Reverse direction length
Fiber type	Fiber variety (default: SSMF)
east Connector loss (dB)	Input connector loss
east Patchcord loss (dB)	Output connector loss

Table 80: Links Sheet Structure

Limitations vs Native JSON

Feature	JSON	Excel
Advanced EDFA models	Full support	Not supported
Metadata fields	Unlimited	Limited columns
RamanFiber parameters	Complete	Basic only
Multiband amplifiers	Supported	Not supported
Complex constraints	Full expressions	Basic only
Comments	JSON comments	Cell comments
Version control	Git-friendly	Binary format

Feature	JSON	Excel
Programmatic generation	Direct	Requires conversion

Table 81: JSON vs Excel Comparison

```
# Validate converted files
from gnpy.tools.json_io import load_equipment, load_network
from gnpy.core.exceptions import ConfigurationError

try:
    # Load equipment
    equipment = load_equipment('eqpt_config.json')
    print("    " Equipment configuration valid")

    # Load network
    network = load_network('topology.json', equipment)
    print(f"    " Network topology valid: {len(network.nodes())} nodes")

except ConfigurationError as e:
    print(f"        Validation error: {e}")
```

Best Practices

1. **Start with Excel for rapid prototyping:**
   - Quick network sketching
   - Easy data entry and modification
   - Familiar spreadsheet interface
2. **Convert to JSON for production:**
   - Version control with Git
   - Advanced parameter support
   - Automated processing
3. **Maintain both formats:**
   - Excel as source of truth for documentation
   - JSON for actual simulation
   - Scripted conversion pipeline

```
#!/bin/bash
# convert_and_validate.sh

# Convert Excel to JSON
gnpy-convert-xls network_design.xlsx -o json/

# Validate conversion
python -c "
from gnpy.tools.json_io import load_equipment, load_network
equipment = load_equipment('json/eqpt_config.json')
```

```
network = load_network('json/topology.json', equipment)
print(f'Conversion successful: {len(network.nodes())} nodes')
" || exit 1

# Run simulation
gnpy-path-request -e json/eqpt_config.json \
  json/topology.json json/services.json \
  -o results/response.json

echo "Pipeline complete!"
```

3.5.7 Best Practices

Input File Preparation

Equipment Library Management

1. **Maintain Equipment Versions:**

```
equipment/
          eqpt_config_v1.0.json
          eqpt_config_v1.1.json
          eqpt_config_current.json -> eqpt_config_v1.1.json
          vendor_configs/
              vendorA_booster.json
              vendorB_preamp.json
```

2. **Document Equipment Sources:**

```
{
  "Edfa": [
    {
      "type_variety": "vendorA_booster",
      "metadata": {
        "source": "Vendor datasheet Rev 3.2",
        "date": "2024-01-15",
        "measured": true,
        "notes": "Lab-validated NF model"
      },
      ...
    }
  ]
}
```

3. **Use Descriptive Names:**
   - Good: `vendorA_C_band_booster_23dBm`
   - Bad: `amp1`, `booster_v2`

Topology Best Practices

1. **Consistent Naming Convention:**

```
// Follow pattern: TYPE_Location1_Location2
{
  "uid": "Fiber_Paris_Lyon",
  "uid": "Amp_preROADM_Lyon",
  "uid": "ROADM_Lyon",
  "uid": "Amp_postROADM_Lyon"
}
```

2. **Include Geographic Metadata:**

```
{
  "uid": "ROADM_Paris",
  "type": "Roadm",
  "metadata": {
    "location": {
      "city": "Paris",
      "region": "IDF",
      "country": "France",
      "latitude": 48.8566,
      "longitude": 2.3522
    },
    "site_id": "FR-PAR-01",
    "owner": "OperatorA"
  }
}
```

3. **Validate Topology Connectivity:**

```
import networkx as nx

# Check for isolated nodes
G = load_network('topology.json', equipment)
if not nx.is_connected(G.to_undirected()):
    print("Warning: Network has isolated components")
    components = list(nx.connected_components(G.to_undirected()))
    print(f"Number of components: {len(components)}")
```

Network Design Considerations

Span Design Rules

1. **Maximum Span Loss:**

```
# Rule: L_span < G_flatmax - margin
max_span_loss = edfa_gain_flatmax - design_margin

# Typical values:
# - Standard design: margin = 3-5 dB
# - High reliability: margin = 8-10 dB
# - Submarine: margin = 1-2 dB

# Example:
edfa_gain_flatmax = 27.0  # dB
design_margin = 5.0  # dB
max_span_loss = 27.0 - 5.0  # = 22 dB
max_fiber_length = max_span_loss / fiber_loss_coef  # km
```

2. **Amplifier Spacing Strategy:**

   - Short spans (<50 km): Consider Raman

- Medium spans (50-80 km): Standard EDFAs
- Long spans (>80 km): High-gain EDFAs
- Very long spans (>120 km): Distributed Raman + EDFA

3. **ROADM Degree Planning:**

```
{
  "uid": "ROADM_Hub",
  "type": "Roadm",
  "params": {
    "per_degree_impairments": true,
    "target_pch_out_db": -20,
    // Higher loss for multi-degree ROADMs
    "add_drop_osnr": 35  // vs 38 for simple ROADM
  }
}
```

Power Budget Management

1. **Launch Power Selection:**

```
# Balance between OSNR and nonlinearity
# Rule of thumb for SSMF:
fiber_length_km = 80
if fiber_length_km < 50:
    launch_power_dbm = -3  # Low for short spans
elif fiber_length_km < 100:
    launch_power_dbm = 0   # Standard
else:
    launch_power_dbm = 2   # High for long spans

# Always check NLI impact
```

2. **Power Mode Selection:**

```
// In sim_params.json or equipment config
{
  "Span": {
    "default": {
      "power_mode": true,  // Variable power optimization
      "delta_power_range_db": [-5, 5],
      "max_fiber_lineic_loss_for_raman": 0.25
    }
  }
}
```

Performance Optimization

Simulation Speed

1. **NLI Calculation Optimization:**

```
{
  "nli_params": {
    "method": "gn_model_analytic",  // Fastest
    // vs "ggn_spectrally_separated" (more accurate, slower)

    "computed_channels": null,  // Compute all channels
```

```
    // vs [0, 23, 47, 71, 95]  // Compute only subset

    "dispersion_tolerance": 1,  // Larger = faster, less accurate
    "phase_shift_tolerance": 0.1
  }
}
```

2. **Raman Solver Tuning:**

```
{
  "raman_params": {
    "flag": true,
    "solver_spatial_resolution": 50,  // Larger = faster (m)
    // vs 10 for high accuracy
    "result_spatial_resolution": 10000  // Storage resolution
  }
}
```

3. **Batch Processing:**

```
# Process multiple services in parallel
from multiprocessing import Pool

def process_service(service_file):
    # Run gnpy-path-request
    result = subprocess.run([
        'gnpy-path-request',
        '-e', 'eqpt_config.json',
        'network.json',
        service_file
    ], capture_output=True)
    return result

with Pool(8) as pool:
    results = pool.map(process_service, service_files)
```

Memory Management

1. **Large Network Handling:**

```
# For networks with >1000 nodes
# Process requests in batches
batch_size = 100
for i in range(0, len(requests), batch_size):
    batch = requests[i:i+batch_size]
    results = process_batch(batch)
    save_results(results, f'batch_{i//batch_size}.json')
```

2. **Spectral Resolution:**

```
// Reduce spectral resolution for screening studies
{
  "SI": [{
    "f_min": 191.35e12,
    "f_max": 196.1e12,
    "spacing": 100e9,  // 100 GHz vs 50 GHz
    // Fewer channels = faster simulation
  }]
}
```

Validation Strategies

```python
#!/usr/bin/env python3
"""validate_inputs.py - Comprehensive input validation"""

from gnpy.tools.json_io import load_equipment, load_network, \
    requests_from_json
from gnpy.core.exceptions import ConfigurationError
import sys

def validate_equipment(filename):
    """Validate equipment library"""
    try:
        equipment = load_equipment(filename)

        # Check for required types
        required = ['Edfa', 'Fiber', 'Roadm', 'Transceiver', 'SI']
        for req in required:
            if req not in equipment:
                print(f"        Missing required type: {req}")
                return False

        print(f"    " Equipment valid: {len(equipment['Edfa'])} EDFAs, "
              f"{len(equipment['Fiber'])} fiber types")
        return True

    except ConfigurationError as e:
        print(f"        Equipment error: {e}")
        return False

def validate_topology(topo_file, eqpt_file):
    """Validate network topology"""
    try:
        equipment = load_equipment(eqpt_file)
        network = load_network(topo_file, equipment)

        # Check connectivity
        import networkx as nx
        if not nx.is_weakly_connected(network):
            print("      Warning: Network is not fully connected")

        # Check for transceivers
        transceivers = [n for n in network.nodes()
                        if network.nodes[n].get('type') == 'Transceiver']
        if len(transceivers) < 2:
            print("        Insufficient transceivers (need         2)")
            return False

        print(f"    " Topology valid: {len(network.nodes())} nodes, "
              f"{len(transceivers)} transceivers")
        return True

    except Exception as e:
        print(f"        Topology error: {e}")
        return False

def validate_services(service_file, topo_file, eqpt_file):
    """Validate service requests"""
    try:
        equipment = load_equipment(eqpt_file)
```

```
        network = load_network(topo_file, equipment)

        with open(service_file) as f:
            import json
            service_data = json.load(f)

        requests = requests_from_json(service_data, equipment)

        # Check source/destination exist
        nodes = set(network.nodes())
        for req in requests:
            if req.source not in nodes:
                print(f"        Unknown source: {req.source}")
                return False
            if req.destination not in nodes:
                print(f"        Unknown destination: {req.destination}")
                return False

        print(f"    " Services valid: {len(requests)} requests")
        return True

    except Exception as e:
        print(f"        Services error: {e}")
        return False

if __name__ == '__main__':
    all_valid = True
    all_valid &= validate_equipment('eqpt_config.json')
    all_valid &= validate_topology('topology.json', 'eqpt_config.json')
    all_valid &= validate_services('services.json', 'topology.json',
                                   'eqpt_config.json')

    if all_valid:
        print("\ n    " All inputs valid - ready for simulation")
        sys.exit(0)
    else:
        print("\ n       Validation failed - fix errors before simulation")
        sys.exit(1)
```

```
def validate_results(response_file):
    """Validate simulation results"""
    with open(response_file) as f:
        results = json.load(f)

    issues = []

    for response in results.get('response', []):
        req_id = response['response-id']

        # Check for path found
        if response['response-type'] == 'no-path':
            no_path = response.get('no-path', {})
            issues.append(f"{req_id}: {no_path.get('no-path-cause')}")
            continue

        # Check GSNR margins
        metrics = response['path-properties']['path-metric']
        gsnr_metric = next((m for m in metrics
```

```
                        if m['metric-type'] == 'GSNR-bandwidth'), None)

        if gsnr_metric:
            gsnr = gsnr_metric['accumulative-value']
            if gsnr < 15.0:
                issues.append(f"{req_id}: Low GSNR ({gsnr:.1f} dB)")

    if issues:
        print("Validation issues:")
        for issue in issues:
            print(f"        {issue}")
    else:
        print("    " All results valid")

    return len(issues) == 0
```

3.5.8 Troubleshooting

Common Errors and Solutions

Configuration Errors

Error Message	Cause	Solution
`Configuration error in equipment library`	Invalid JSON syntax or missing required fields	Validate JSON syntax; check required parameters
`Unknown type_variety 'XYZ'`	Referenced equipment type not in library	Add equipment type or fix reference
`Edfa gain_target exceeds gain_flatmax`	Span loss too high for amplifier	Use higher-gain amplifier or split span
`RamanFiber requires simulation params`	Missing sim_params.json with Raman config	Provide sim_params.json or remove RamanFiber

Table 82: Equipment Configuration Errors

Topology Errors

Error Message	Cause	Solution
`Invalid network definition`	Malformed topology JSON or cyclic dependencies	Check JSON syntax and element ordering
`Node 'X' not found in network`	Service references non-existent node	Fix node name or add node to topology
`Network is not connected`	Isolated sub-graphs	Add missing connections

Error Message	Cause	Solution
`Fiber length must be positive`	Negative or zero fiber length	Correct fiber parameters

Table 83: Topology Errors

Service Request Errors

Error Message	Cause	Solution
`NO_FEASIBLE_MODE`	GSNR too low for any transceiver mode	Improve network design or use higher-performance transceiver
`NO_PATH`	No route exists between endpoints	Check topology connectivity
`NO_SPECTRUM`	Insufficient spectrum available	Reduce number of channels or use different spectrum assignment
`PATH_CONSTRAINT_VIOLATION`	Explicit path invalid	Fix path constraints or use loose routing

Table 84: Service Request Errors

Validation Failures

Debugging Network Design Issues

1. **Enable Verbose Logging:**

```
# Get detailed execution trace
gnpy-path-request -vv -e eqpt_config.json network.json services.json

# Output shows:
# - Equipment loading details
# - Network building steps
# - Amplifier placement decisions
# - Propagation element-by-element
# - Feasibility check details
```

2. **Export Auto-Designed Topology:**

```
# Review amplifier placements
gnpy-transmission-example -e eqpt_config.json network.json \
  --output designed_network.json

# Check:
# - Are amplifiers placed reasonably?
# - Are gains within limits?
# - Are power levels appropriate?
```

3. **Visualize Network:**

```python
import networkx as nx
import matplotlib.pyplot as plt
from gnpy.tools.json_io import load_equipment, load_network

equipment = load_equipment('eqpt_config.json')
network = load_network('topology.json', equipment)

# Plot network graph
pos = nx.spring_layout(network)
nx.draw(network, pos, with_labels=True, node_color='lightblue',
        node_size=500, font_size=8)
plt.savefig('network_topology.png', dpi=300)

# Identify issues
degrees = dict(network.degree())
isolated = [n for n, d in degrees.items() if d == 0]
if isolated:
    print(f"Isolated nodes: {isolated}")
```

```python
def analyze_gsnr_breakdown(propagated_path):
    """Analyze GSNR contributors along path"""
    print("GSNR Breakdown Analysis:")
    print("-" * 60)

    for i, element in enumerate(propagated_path):
        if hasattr(element, 'osnr'):
            signal = element.carriers[0].power.signal
            ase = element.carriers[0].power.ase
            nli = element.carriers[0].power.nli

            osnr = signal / ase
            snr = signal / (ase + nli)
            gsnr = signal / (ase + nli)

            print(f"{i}: {element.uid:30s}")
            print(f"    Signal: {watt2dbm(signal):6.2f} dBm")
            print(f"    ASE:    {watt2dbm(ase):6.2f} dBm")
            print(f"    NLI:    {watt2dbm(nli):6.2f} dBm")
            print(f"    OSNR:   {lin2db(osnr):6.2f} dB")
            print(f"    GSNR:   {lin2db(gsnr):6.2f} dB")

            # Identify dominant impairment
            if ase > 10 * nli:
                print(f"         ' ASE-limited")
            elif nli > 10 * ase:
                print(f"         ' NLI-limited")
            else:
                print(f"         ' Mixed impairments")
            print()

# Usage:
analyze_gsnr_breakdown(propagated_path)
```

Debug Techniques

```python
from gnpy.core.info import create_input_spectral_information
```

```

# Create test signal
spectral_info = create_input_spectral_information(
    f_min=191.35e12,
    f_max=196.1e12,
    spacing=50e9,
    power=1e-3,
    equipment=equipment
)

# Propagate step-by-step
print("Propagation trace:")
for element in path:
    spectral_info_before = spectral_info
    spectral_info = element(spectral_info)

    # Log changes
    print(f"\n{element.uid} ({element.__class__.__name__}):")

    if hasattr(element, 'effective_gain'):
        print(f"  Gain: {element.effective_gain:.2f} dB")

    if hasattr(spectral_info, 'signal'):
        power_in = spectral_info_before.signal[0]
        power_out = spectral_info.signal[0]
        delta = watt2dbm(power_out) - watt2dbm(power_in)
        print(f"  Power change: {delta:+.2f} dB")
        print(f"  Output power: {watt2dbm(power_out):.2f} dBm")
```

```
def sensitivity_analysis(base_params, param_name, param_range):
    """Test impact of parameter variation"""
    results = []

    for value in param_range:
        # Modify parameter
        params = base_params.copy()
        params[param_name] = value

        # Run simulation
        gsnr = run_simulation(params)
        results.append((value, gsnr))

    # Plot results
    import matplotlib.pyplot as plt
    values, gsnrs = zip(*results)
    plt.plot(values, gsnrs)
    plt.xlabel(param_name)
    plt.ylabel('GSNR (dB)')
    plt.title(f'Sensitivity to {param_name}')
    plt.grid(True)
    plt.savefig(f'sensitivity_{param_name}.png')

# Example: Test launch power impact
sensitivity_analysis(
    base_params={'launch_power_dbm': 0},
    param_name='launch_power_dbm',
    param_range=[-5, -3, -1, 0, 1, 2, 3]
)
```

```python
def compare_with_measurements(simulation_results, measurements):
    """Compare simulated vs measured OSNR"""
    import pandas as pd

    # Load data
    sim_df = pd.DataFrame(simulation_results)
    meas_df = pd.DataFrame(measurements)

    # Merge on common fields
    comparison = pd.merge(sim_df, meas_df,
                          on=['path_id', 'frequency'],
                          suffixes=('_sim', '_meas'))

    # Calculate errors
    comparison['osnr_error'] = (comparison['osnr_sim'] -
                                comparison['osnr_meas'])

    # Statistics
    print("Simulation vs Measurement:")
    print(f"  Mean error: {comparison['osnr_error'].mean():.2f} dB")
    print(f"  Std error:  {comparison['osnr_error'].std():.2f} dB")
    print(f"  Max error:  {comparison['osnr_error'].abs().max():.2f} dB")

    # Plot
    plt.figure(figsize=(10, 6))
    plt.scatter(comparison['osnr_meas'], comparison['osnr_sim'])
    plt.plot([10, 30], [10, 30], 'r--', label='Perfect agreement')
    plt.xlabel('Measured OSNR (dB)')
    plt.ylabel('Simulated OSNR (dB)')
    plt.legend()
    plt.grid(True)
    plt.savefig('simulation_validation.png')

    return comparison

# Usage:
comparison = compare_with_measurements(
    simulation_results=gnpy_results,
    measurements=field_measurements
)
```

3.5.9 Summary

GNPy provides flexible output options to support different workflows:

- **response.json**: Standard IETF YANG format for system integration
- **auto_designed_topology.json**: Complete network design for verification
- **CSV reports**: Human-readable tables for analysis
- **API interfaces**: Real-time integration with controllers (ONOS, OpenROADM)

Key Takeaways:

1. Use appropriate output format for your use case
2. Validate inputs before simulation
3. Enable verbose logging for debugging
4. Compare simulation results with measurements when available
5. Maintain version control of configuration files
6. Document equipment sources and assumptions

The combination of these outputs enables comprehensive analysis from initial design through production deployment and ongoing optimization.

4 GNPy and OpenROADM Device Model

4.1 Device Types

4.1.1 GNPy Representable Devices

GNPy models the following network element types (from `gnpy.core.elements`):

1. **Transceiver** - Optical transceivers (transmit/receive)
2. **Roadm** - Reconfigurable Optical Add-Drop Multiplexer dual-stage)
3. **Fused** - Passive optical components (couplers, splitters)
4. **Fiber** - Optical fiber spans with physical impairments
5. **RamanFiber** - Fiber with Raman amplification
6. **Edfa** - Erbium-Doped Fiber Amplifiers (single-band) or Multiband_amplifier - Multi-band amplifier support (C+L band)

4.1.2 OpenROADM Device Model Hierarchy

OpenROADM Device Model defines a multi-level equipment hierarchy:

1. **Device** (Top-level container)
   - **Shelves** - Physical racks/bays (with FIC identifiers)
   - **Circuit Packs** - Modules/cards within shelves
     - **Ports** - Physical connection points
   - **Internal Links** - Connections between circuit packs
2. **Network Elements** (Network Model perspective):
   - **ROADM** - With degrees (directions) and SRGs (add/drop groups)
   - **Transponder/Muxponder** (Xponder)
   - **Regenerator**
   - **Amplifier** (ILA - In-Line Amplifier)
   - **External Pluggable** - Modules in packet boxes

4.2 One-to-One Device Attribute Comparison

4.2.1 OPTICAL AMPLIFIER (EDFA)

EDFA means Erbium Doped Fiber Amplifier, in GNPy it also represents OpenROADM devices like ILA, preamps, and boosters. The way of managing it is using an attribute called `type_def` with options {`variable_gain`, `fixed_gain`, `openroadm`, `openroadm_preamp`, `openroadm_booster`, `advanced_model`}, defining a different noise figure for each.

1. `'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. `'fixed_gain'` is a fixed gain model. `NF == Cte == nf0` if `gain_min < gain < gain_flatmax`
3. `'openroadm'` models the incremental OSNR contribution as a function of input power. It is suitable for inline amplifiers that conform to the OpenROADM specification. The input parameters are coefficients of the third-degree polynomial.

4. `'openroadm_preamp'` and `openroadm_booster` approximate the preamp and booster within an OpenROADM network. No extra parameters specific to the NF model are accepted.
5. `'advanced_model'` is an advanced model. A detailed JSON configuration file is required (by default https://github.com/Telecominfraproject/oopt-gnpy/blob/master/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.
6. `'multi_band'` defines an amplifier type corresponding to an amplification site composed of multiple amplifier elements, where each amplifies a different band of the spectrum. The `amplifiers` list contains the list of single-band amplifier type varieties that can compose such multiband amplifiers. Several options can be listed for the same spectrum band. Only one can be selected for the actual Multiband_amplifier element.

For all single band 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.)
`f_min` and `f_max`	(number)	Optional. In Hz. Minimum and maximum frequency range for the amplifier. Signal must fit entirely within this range (center frequency and spectrum width). Default is 191.275e-12 Hz and 196.125e-12.

Table 85: Single band amplifier model parameters

Default values are defined for the frequency range for:

- noise figure ripple
- gain ripple
- dynamic gain tilt

Users can introduce custom values using `default_config_from_json` which should be populated with a file name containing the desired parameters.

field	type	description
type_variety	(string)	A unique name to ID the amplifier in the JSON template topology input file.
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.)

Table 86: Multi-band amplifier model parameters

GNPy: `Edfa` Class Core Attributes

GNPy Attribute	Type	Description	Source
uid	str	Unique identifier	User topology
type_variety	str	Equipment type reference	User topology
params	EdfaParams	Amplifier specifications	Equipment library
operational	dict	Operational settings	User topology/auto-design
gain_target	float	Target gain (dB)	Computed/specified
effective_gain	float	Actual operational gain (dB)	Computed
tilt_target	float	Spectral tilt target (dB)	Computed/specified
delta_p	float	Delta power per channel (dB)	Computed
out_voa	float	Output VOA attenuation (dB)	Computed
in_voa	float	Input VOA attenuation (dB)	Optional
nf	float	Noise figure (dB)	Computed from model
pin_db	float	Input power (dBm)	Measured during propagation
pout_db	float	Output power (dBm)	Computed

EdfaParams (from equipment library):

Parameter	Type	Description
type_def	str	Model type: ‘variable_gain’, ‘fixed_gain’, ‘openroadm_preamp’, ‘openroadm_booster’, ‘advanced_model’, ‘dual_stage’

Parameter	Type	Description
gain_min	float	Minimum gain (dB)
gain_flatmax	float	Maximum flat gain (dB)
gain_max	float	Extended maximum gain (dB)
p_max	float	Maximum output power (dBm)
nf_model	NfModel	Noise figure model
nf_fit_coeff	array	NF polynomial coefficients
dgt	array	Dynamic gain tilt
gain_ripple	array	Gain ripple vs frequency
nf_ripple	array	NF ripple vs frequency
out_voa_auto	bool	Auto VOA adjustment
allowed_for_design	bool	Can be auto-selected
raman	bool	Raman capability flag

Examples of amplifier definitions are shown below. Note that advanced_model depends on another file, which defines a specific noise figure (also shown in the examples).

```
"Edfa": [
  {
    "type_variety": "openroadm_ila_low_noise",
    "type_def": "openroadm",
    "gain_flatmax": 27,
    "gain_min": 0,
    "p_max": 22,
    "nf_coef": [
      -8.104e-4,
      -6.221e-2,
      -5.889e-1,
      37.62
    ],
    "allowed_for_design": false
  }

{
    "type_variety": "openroadm_mw_mw_preamp",
    "type_def": "openroadm_preamp",
    "gain_flatmax": 27,
    "gain_min": 0,
    "p_max": 22,
    "allowed_for_design": false
  }

{
    "type_variety": "openroadm_mw_mw_booster",
    "type_def": "openroadm_booster",
    "gain_flatmax": 32,
    "gain_min": 0,
    "p_max": 22,
    "allowed_for_design": false
  }

{
    "type_variety": "std_fixed_gain",
    "type_def": "fixed_gain",
    "gain_flatmax": 21,
    "gain_min": 20,
    "p_max": 21,
    "nf0": 5.5,
    "allowed_for_design": false
```

```
    }

    {
      "type_variety": "operator_model_example",
      "type_def": "variable_gain",
      "gain_flatmax": 26,
      "gain_min": 15,
      "p_max": 23,
      "nf_min": 6,
      "nf_max": 10,
      "out_voa_auto": false,
      "allowed_for_design": false
    }

    {
      "type_variety": "hybrid_4pumps_lowgain",
      "type_def": "dual_stage",
      "raman": true,
      "gain_min": 25,
      "preamp_variety": "4pumps_raman",
      "booster_variety": "std_low_gain",
      "allowed_for_design": true
    }

    {
      "type_variety": "high_detail_model_example",
      "type_def": "advanced_model",
      "gain_flatmax": 25,
      "gain_min": 15,
      "p_max": 21,
      "advanced_config_from_json": "std_medium_gain_advanced_config.json", *
      "out_voa_auto": false,
      "allowed_for_design": false
    },
    ],
* - in std_medium_gain_advanced_config.json:
{
  "nf_fit_coeff": [
    0.000168241,
    0.0469961,
    0.0359549,
    5.82851
  ],
  "f_min": 191.275e12,
  "f_max": 196.125e12,
  "nf_ripple": [
    0.4372876328262819,
    0.4372876328262819,
    ...
    -0.3110761646066259
  ],
  "dgt": [
    1.0,
    1.017807767853702,
    ...
    2.714526681131686
  ],
  "gain_ripple": [
    0.07704745697916238,
    0.06479749697916048,
    ...
    0.1359703369791596
  ]
}
```

OpenROADM: Amplifier Device Model Device-Level Attributes (from Device Model Whitepaper):

OpenROADM Attribute	Type	Description	Source
Physical Hierarchy			
`shelf-name`	string	Shelf identifier (e.g., Bay FIC)	Planning/Inventory
`slot-name`	string	Slot within shelf	Planning/Inventory
`circuit-pack-name`	string	Circuit pack identifier	Device discovery
`circuit-pack-type`	string	Hardware type	Device model
OTS Interface Container			
`amplifier-type`	enum	‘EDFA’, ‘Raman’, ‘Hybrid’	Device capability
`amplifier-gain-range`	uint8	Gain range 1-4	Device capability

Network-Level Attributes (from Network Model Whitepaper - Table 9):

OpenROADM Attribute	Type	Description	Source
`amp-type`	string	Amplifier type	From device
`amp-gain-range`	integer	Gain range (1-4)	From device
`ingress-span-loss-aging-margin`	dB	Margin for aging/repair	Planning tool
`gain`	ratio-dB	Overall signal gain (excluding ASE, including VOA)	PM/operational
`initially-planned-gain`	ratio-dB	Reference design gain	Planning tool
`tilt`	ratio-dB	Smart EDFA tilt	PM/operational
`initially-planned-tilt`	ratio-dB	Reference design tilt	Planning tool
`out-voa-att`	ratio-dB	Output VOA attenuation	Operational
`eol-max-load-pIn`	power-dBm	EOL max input power for amp/VOA setting	Planning tool
`egress-average-channel-power`	power-dBm	Max power across 4.8 THz passband	SDN controller
`type-variety`	string	Equipment type reference	Planning

Operational Mode Catalog (Service Model - Table for Amplifiers):

Attribute	Type	Description
`min-gain`	ratio-dB	Minimum amplifier gain

Attribute	Type	Description
`max-gain`	ratio-dB	Maximum gain respecting guaranteed tilt
`max-extended-gain`	ratio-dB	Maximum extended gain (tilt not guaranteed)
`noise-mask`	list	OSNR vs Pin characteristics (polynomial fit)
`gain-ripple`	dB	Gain ripple specification
`mask-gain-ripple-vs-tilt`	list	Ripple limits vs tilt (C, D parameters)

Comparison Analysis: Similarities: - Both models support gain, tilt, and VOA parameters - Both distinguish between planned/design values and operational values - Both support noise figure/OSNR modeling - Both reference equipment types via type identifiers

Key Differences:

Aspect	GNPy	OpenROADM
Abstraction Level	High-level physics model	Detailed device hierarchy
Noise Modeling	Multiple models: variable_gain, advanced_model, openroadm_preamp/booster	Operational mode catalog with OSNR masks
Equipment Hierarchy	Single element	Shelf → Circuit Pack → Port
Inventory Tracking	Not modeled	Bay/FIC, shelf, slot locations
Raman Support	Dedicated RamanFiber element + flag	Hybrid amplifier type
Multi-band	Separate Multiband_amplifier class	Implicit in device capabilities
Auto-Design	Automatic amplifier placement algorithm	Planning tool responsibility

---

4.2.2 OPTICAL FIBER

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

GNPy: `Fiber` Class Core Attributes:

GNPy Attribute	Type	Description
`uid`	str	Unique identifier
`type_variety`	str	Fiber type reference
`params`	FiberParams	Fiber specifications
`length`	float	Fiber length (m)

FiberParams (from equipment library):

Parameter	Type	Description	Unit
`length`	float	Fiber length	m or km
`loss_coef`	float	Attenuation coefficient	dB/km
`length_units`	str	'km' or 'm'	-
`att_in`	float	Additional input loss	dB
`con_in`	float	Connector loss input	dB
`con_out`	float	Connector loss output	dB
`dispersion`	float	Chromatic dispersion	ps/(nm·km)
`gamma`	float or callable	Nonlinear coefficient	1/(W·km)
`pmd_coef`	float	PMD coefficient	ps/$\sqrt{km}$
`beta2`	float	GVD parameter	s^2/m
`beta3`	float	Dispersion slope param	s^3/m

Raman Fiber Additional Parameters:

The RamanFiber can be used to simulate Raman amplification through dedicated Raman pumps. The Raman pumps must be listed in the key `raman_pumps` within the RamanFiber `operational` dictionary. The description of each Raman pump must contain is in Table 96.

field	**type**	**description**
`power`	(number)	Total pump power in W considering a depolarized pump
`frequency`	(number)	Pump central frequency in Hz
`propagation_direction`	(number)	The pumps can propagate in the same or opposite direction with respect the signal. Valid choices are `coprop` and `counterprop`, respectively

Table 96: Raman pump parameters

Beside the list of Raman pumps, the RamanFiber `operational` dictionary must include the `temperature` that affects the amplified spontaneous emission noise generated by the Raman amplification. As the loss coefficient significantly varies outside the C-band, where the Raman pumps are usually placed, it is suggested to include an estimation of the loss coefficient for the Raman pump central frequencies within a dictionary-like definition of the `RamanFiber.params.loss_coef` (e.g. `loss_coef = {"value": [0.18, 0.18, 0.20, 0.20], "frequency": [191e12, 196e12, 200e12, 210e12]}`).

Parameter	Type	Description
`raman_efficiency`	array	Raman gain efficiency profile
`cr`	float	Raman coefficient
`frequency_offset`	array	Frequency offsets for Raman

Examples of fiber and raman fiber element definitions are as it follows:

```
   "Fiber": {
     "type_variety": "SSMF",
     "dispersion": 1.67e-05,
     "effective_area": 83e-12,
     "pmd_coef": 1.265e-15
   },
   {
     "type_variety": "NZDF",
     "dispersion": 0.5e-05,
     "effective_area": 72e-12,
     "pmd_coef": 1.265e-15
   },
   {
     "type_variety": "LOF",
     "dispersion": 2.2e-05,
     "effective_area": 125e-12,
     "pmd_coef": 1.265e-15
   }
 "RamanFiber": [
   {
     "type_variety": "SSMF",
     "dispersion": 1.67e-05,
     "effective_area": 83e-12,
     "pmd_coef": 1.265e-15
   }
   ],
```

OpenROADM: Fiber/Span Model Network Model - Span Attributes (Table 9):

OpenROADM Attribute	Type	Description	Source
Non-Amplified Link			
`auto-spanloss`	boolean	Enable auto span loss measurement	Planning
`spanloss-current`	ratio-dB	Current measured span loss	SDN controller
`spanloss-last-measured`	timestamp	Last measurement time	SDN controller
`engineered-spanloss`	ratio-dB	Reference span loss for design	External system
Link Concatenation (SRLG)			
`SRLG-Id`	identifier	Shared Risk Link Group ID	Planning
`fiber-type`	string	Underlying fiber type	Planning
`SRLG-length`	meters	Fiber length	Planning
`pmd`	ps/$\sqrt{km}$	PMD coefficient	External system

Device Model - OMS Interface:

Attribute	Type	Description
`fiber-type`	identityref	Standard fiber (SSMF, LEAF, TWC, etc.)
`span-length`	km	Physical length
`span-loss-transmit/receive`	dB	Measured losses

Operational Mode Catalog (implicit fiber params):

- Fiber types reference ITU-T G.652, G.653, G.655, G.656 standards
- Dispersion and attenuation derived from fiber type
- PMD specified per link

Comparison Analysis:

Aspect	GNPy	OpenROADM
Physical Modeling	Detailed physics (dispersion, nonlinearity, PMD)	Reference to fiber types + measured loss
Nonlinearity	Explicit gamma parameter + NLI solvers	Not explicitly modeled (implicitly handled)
Raman	Dedicated RamanFiber class with efficiency profile	Hybrid amplifier type in device
Loss Tracking	Computed from coefficient + connectors	Measured span loss (auto-spanloss)
Length Units	Flexible (m or km)	Meters (standardized)
Chromatic Dispersion	ps/(nm·km) or beta2 (s^2/m)	Implicit in fiber type
SRLG	Not modeled	Explicit SRLG-Id and risk groups

4.2.3 RECONFIGURABLE OPTICAL ADD-DROP MULTIPLEXER (ROADM)

The user can only modify the value of existing parameters:

Table 101: ROADM parameters

field	type	description
`type_variety`	(string)	Optional. Default: `default`. A unique name to ID the ROADM variety in the JSON template topology input file.
`target_pch_out_db` or `target_psd_out_mWperGHz` or `target_out_mWperSlotWidth` (mutually exclusive)	(number)	Default equalization strategy for this ROADM type. Auto-design sets the ROADM egress channel power. This reflects typical control loop algorithms that adjust ROADM losses to equalize channels (e.g., coming from different ingress direction or add ports). These values are used as defaults when no overrides are set per each `Roadm` element in the network topology.
`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.
`roadm-path-impairments`	(list of dict)	Optional. List of ROADM path category impairments.

In addition to these general impairment, the user may define detailed set of impairments for add, drop and express path within the ROADM. The impairment description is inspired from the IETF CCAMP optical impairment topology (details here: ROADM attributes IETF).

The `roadm-path-impairments` list allows the definition of the list of impairments by internal path category (add, drop or express). Several additional paths can be defined – add-path, drop-path or express-path. They are indexed and the related impairments are defined per band.

Each item should contain:

ROADM path impairment parameters

field	type	description
`roadm-path-impairments-id`	(number)	A unique number to ID the impairments.
`roadm-express-path` or `roadm-add-path` or `roadm-drop-path` (mutually exclusive)	(list)	List of the impairments defined per frequency range. The impairments are detailed in the following table.

Here are the parameters for each path category and the implementation status:

Table 103: ROADM path category parameters and implementation status

field	Type	Description	Drop path	Add path	Express path
`frequency-range`	(list)	List containing `lower-frequency` and `upper-frequency` in Hz.			
`roadm-maxloss`	(num)	In dB. Default: 0 dB. Maximum expected path loss on this roadm-path assuming no additional path loss is added = minimum loss applied to channels when crossing the ROADM (worst case expected loss due to the ROADM).	Imptd.	Imptd.	Imptd.
`roadm-minloss`		The net loss from the ROADM input, to the output of the drop block (best case expected loss).	Not yet Imptd.	N.A.	N.A.
`roadm-typloss`		The net loss from the ROADM input, to the output of the drop block (typical).	Not yet Imptd.	N.A.	N.A.
`roadm-pmin`		Minimum power levels per carrier expected at the output of the drop block.	Not yet Imptd.	N.A.	N.A.

Continued on next page

Table 103: ROADM path category parameters and implementation status (continued)

field	Type	Description	Drop path	Add path	Express path
roadm-pmax		(Add) Maximum (per carrier) power level permitted at the add block input ports. (Drop) Best case per carrier power levels expected at the output of the drop block.	Not yet Imptd.	Not yet Imptd.	N.A.
roadm-ptyp		Typical case per carrier power levels expected at the output of the drop block.	Not yet Imptd.	N.A.	N.A.
roadm-noise -figure		If the add (drop) path contains an amplifier, this is the noise figure of that amplifier inferred to the add (drop) port.	Not yet Imptd.	Not yet Imptd.	N.A.
roadm-osnr	(num)	(Add) Optical Signal-to-Noise Ratio (OSNR). If the add path contains the ability to adjust the carrier power levels into an add path amplifier (if present) to a target value, this reflects the OSNR contribution of the add amplifier assuming this target value is obtained. (Drop) Expected OSNR contribution of the drop path amplifier(if present) for the case of additional drop path loss (before this amplifier) in order to hit a target power level (per carrier).	Imptd.	Imptd.	N.A.
roadm-pmd	(num)	PMD contribution of the specific roadm path.	Imptd.	Imptd.	Imptd.
roadm-cd			Not yet Imptd.	Not yet Imptd.	Not yet Imptd.
roadm-pdl	(num)	PDL contribution of the specific roadm path.	Imptd.	Imptd.	Imptd.

Continued on next page

Table 103: ROADM path category parameters and implementation status (continued)

field	Type	Description	Drop path	Add path	Express path
roadm-inband-crosstalk			Not yet Imptd.	Not yet Imptd.	Not yet Imptd.

Here is a ROADM example with two add-path possible impairments:

```
"roadm-path-impairments": [
  {
      "roadm-path-impairments-id": 0,
      "roadm-express-path": [{
          "frequency-range": {
              "lower-frequency": 191.3e12,
              "upper-frequency": 196.1e12
              },
          "roadm-maxloss": 16.5
          }]
  }, {
      "roadm-path-impairments-id": 1,
      "roadm-add-path": [{
          "frequency-range": {
              "lower-frequency": 191.3e12,
              "upper-frequency": 196.1e12
          },
          "roadm-maxloss": 11.5,
          "roadm-osnr": 41
      }]
  }, {
      "roadm-path-impairments-id": 2,
      "roadm-drop-path": [{
          "frequency-range": {
              "lower-frequency": 191.3e12,
              "upper-frequency": 196.1e12
              },
          "roadm-pmd": 0,
          "roadm-cd": 0,
          "roadm-pdl": 0,
          "roadm-maxloss": 11.5,
          "roadm-osnr": 41
      }]
  }, {
      "roadm-path-impairments-id": 3,
      "roadm-add-path": [{
          "frequency-range": {
              "lower-frequency": 191.3e12,
              "upper-frequency": 196.1e12
          },
          "roadm-pmd": 0,
          "roadm-cd": 0,
          "roadm-pdl": 0,
          "roadm-maxloss": 11.5,
          "roadm-osnr": 20
      }]
  }]
```

Listing 63: ROADM path impairments example

On this example, the express channel has at least 16.5 dB loss when crossing the ROADM express path with the corresponding impairment id.

`roadm-path-impairments` is optional. If present, its values are considered instead of the ROADM general parameters. For example, if add-path specifies 0.5 dB PDL and the general PDL parameter states 1.0 dB, then 0.5 dB is applied for this roadm-path only. If present in add and/or drop path, `roadm-osnr` replaces the portion of `add-drop-osnr` defined for the whole ROADM, assuming that add and drop contribution aggregated in `add-drop-osnr` are identical:

$$add\_drop\_osnr = -10\log_{10}\left(\frac{1}{add_{osnr}} + \frac{1}{drop_{osnr}}\right)$$

when:

$$add_{osnr} = drop_{osnr}$$

$$add_{osnr} = drop_{osnr} = add\_drop\_osnr + 10\log_{10}(2)$$

The user can specify the roadm `type_variety` in the json topology ROADM instance. If no variety is defined, `default` ID is used. The user can define the impairment type for each roadm-path using the degrees ingress/egress immediate neighbor elements and the `roadm-path-impairment-id` defined in the library for the corresponding type-variety. Here is an example:

```
{
  "uid": "roadm SITE1",
  "type": "Roadm",
  "type_variety": "detailed_impairments",
  "params": {
    "per_degree_impairments": [
    {
      "from_degree": "trx SITE1",
      "to_degree": "east edfa in SITE1 to ILA1",
      "impairment_id": 1
    }]
  }
}
```

Listing 64: ROADM per-degree impairments example

It is not permitted to use a `roadm-path-impairment-id` for the wrong roadm path type (add impairment only for add path). If nothing is stated for impairments on roadm-paths, the program identifies the paths implicitly and assigns the first impairment_id that matches the type: if a transceiver is present on one degree, then it is an add/drop degree.

On the previous example, all «implicit» express roadm-path are assigned `roadm-path-impairment-id = 0`.

GNPy: `Roadm` Class Core Attributes

GNPy Attribute	Type	Description
`uid`	str	Unique identifier
`type_variety`	str	Equipment type reference
`params`	RoadmParams	ROADM specifications
`per_degree_impairments`	dict	Loss per degree/direction

RoadmParams (from equipment library):

Parameter	Type	Description
`add_drop_osnr`	float	OSNR for add/drop paths (dB)
`pmd`	float	Polarization mode dispersion (ps)
`pdl`	float	Polarization dependent loss (dB)
`restrictions`	dict	Frequency restrictions
`roadm_path_impairments`	dict	Impairments per path type
`target_pch_out_db`	float	Target per-channel output power
`per_degree_pch_out_db`	dict	Per-degree power targets
`design_bands`	list	Supported frequency bands

Path-Specific Impairments:

Path Type	Attributes
`express`	Loss, OSNR degradation
`add`	Loss, OSNR
`drop`	Loss

An example of the element ROADM definition is the following:

```
    "Roadm": [
  {
    "target_pch_out_db": -20,
    "add_drop_osnr": 38,
    "pmd": 0,
    "pdl": 0,
    "restrictions": {
      "preamp_variety_list": [],
      "booster_variety_list": []
    }
  },
  {
    "type_variety": "roadm_type_1",
    "target_pch_out_db": -18,
    "add_drop_osnr": 35,
    "pmd": 0,
    "pdl": 0,
    "restrictions": {
      "preamp_variety_list": [],
      "booster_variety_list": []
    },
    "roadm-path-impairments": []
  },
  {
    "type_variety": "detailed_impairments",
    "target_pch_out_db": -20,
    "add_drop_osnr": 38,
```

```
      "pmd": 0,
      "pdl": 0,
      "restrictions": {
        "preamp_variety_list": [],
        "booster_variety_list": []
      },
      "roadm-path-impairments": [
        {
          "roadm-path-impairments-id": 0,
          "roadm-express-path": [
            {
              "frequency-range": {
                "lower-frequency": 191.3e12,
                "upper-frequency": 196.1e12
              },
              "roadm-pmd": 0,
              "roadm-cd": 0,
              "roadm-pdl": 0,
              "roadm-inband-crosstalk": 0,
              "roadm-maxloss": 16.5
            }
          ]
        },
        {
            "roadm-path-impairments-id": 1,
            ...
        }
        {
          "roadm-path-impairments-id": 2,
          "roadm-drop-path": [
            {
              "frequency-range": {
                "lower-frequency": 191.3e12,
                "upper-frequency": 196.1e12
              },
              "roadm-pmd": 0,
              "roadm-cd": 0,
              "roadm-pdl": 0,
              "roadm-inband-crosstalk": 0,
              "roadm-maxloss": 11.5,
              "roadm-minloss": 7.5,
              "roadm-typloss": 10,
              "roadm-pmin": -13.5,
              "roadm-pmax": -9.5,
              "roadm-ptyp": -12,
              "roadm-osnr": 41,
              "roadm-noise-figure": 15
            }
          ]
        }
      ]
    }
  ],
```

OpenROADM: ROADM Device Model Network Model - ROADM Degree Attributes (Table 6):

OpenROADM Attribute	Type	Description	Source
`degree-number`	identifier	Direction/degree identifier	Planning
`max-wavelengths` (N)	integer	Max wavelengths in fixed grid	Device

OpenROADM Attribute	Type	Description	Source
`ingress-span-loss-margin`	dB	Span-loss margin from OTS interface	Device
`eol-max-load-pIn`	power-dBm	EOL total input power at max load	Device
`egress-average-channel-power`	power-dBm	Total max power across 4.8 THz	SDN controller

ROADM SRG (Add/Drop Group) Attributes (Table 7):

Attribute	Type	Description	Source
`srg-number`	identifier	SRG identifier	Planning
`max-pp` (Npp)	integer	Max add/drop port pairs in SRG	Device Model
`current-provisioned-pp`	integer	Currently provisioned ports	Device
`Wavelength-Duplication`	enum	1=per SRG, 2=per Degree	Planning
`avail-freq-maps`	list	Available frequency maps	Device/SDN

Device Model Hierarchy:

```
ROADM Device
|-- Degrees (directions)
|   |-- CTP-Tx (Transmit Connection Point)
|   |-- CTP-Rx (Receive Connection Point)
|   |-- TTP-Tx (Transmit Tributary Port - to line)
|   --- TTP-Rx (Receive Tributary Port - from line)
|-- SRGs (Add/Drop Groups)
|   |-- CP-Tx (Client Transmit)
|   |-- CP-Rx (Client Receive)
|   --- Port Pairs (pp's)
--- Internal/Physical Links
    |-- Express Links (degree-to-degree)
    |-- Add Links (srg-to-degree)
    --- Drop Links (degree-to-srg)
```

Operational Mode Catalog (Service Model - ROADM Specs):

Path Type	Attributes
Express Path	`pmd`, `cd`, `pdl`, `insertion-loss`, `osnr-penalty`
Add Path	`insertion-loss`, `osnr-penalty`
Drop Path	`insertion-loss`
Per-Degree	`mux-demux loss`, `add/drop filtering`

Comparison Analysis:

Aspect	GNPy	OpenROADM
Degree Modeling	Implicit in network topology	Explicit degree-number per direction
Add/Drop Groups	Not explicitly modeled	SRG with port pair management
Path Impairments	express/add/drop impairment sets	Similar, plus per-degree variations
Spectrum Management	Channel power equalization	Frequency maps + wavelength duplication modes
Physical Connectivity	Logical only	CTPs, TTPs, CPs, internal links
Equipment Hierarchy	Single element	Degrees + SRGs + circuit packs
Design Bands	Frequency bands list	Per-degree design bands support

4.2.4 TRANSCEIVER (Transponder/Muxponder)

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 `gnpy-path-request` script.

The modes are defined as follows:

Table 112: Transceiver mode parameters

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
`min_spacing`	(number)	in Hz. Min required slot size for this mode.
`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.
`equalization_offset_db`	(number)	In dB. Deviation from the per channel equalization target in ROADM for this type of transceiver.
`penalties`	(list)	list of impairments as described in impairment table.
`cost`	(number)	Arbitrary unit

Penalties are linearly interpolated between given points and set to 'inf' outside interval. The accumulated penalties are subtracted to the path GSNR before comparing with the min required OSNR. The penalties per impairment type are defined as a list of dict (impairment type - penalty values) as follows:

For example:

```
"penalties": [{
        "chromatic_dispersion": 360000,
        "penalty_value": 0.5
    }, {
        "pmd": 110,
        "penalty_value": 0.5
    }
]
```

Listing 65: Example penalties configuration

GNPy: `Transceiver` Class Core Attributes:

GNPy Attribute	Type	Description
`uid`	str	Unique identifier
`type_variety`	str	Transceiver type reference
`mode`	dict	Selected operational mode
`params`	TransceiverParams	Transceiver specs

Transceiver Mode Parameters (from equipment library):

Parameter	Type	Description
`format`	str	Modulation format (e.g., 'DP-16QAM')
`baud_rate`	float	Symbol rate (Hz)
`OSNR`	float	Required OSNR (dB in 0.1nm)
`bit_rate`	float	Line bit rate (bits/s)
`roll_off`	float	Pulse shaping roll-off factor
`tx_osnr`	float	Transmitter OSNR (dB)
`min_spacing`	float	Minimum channel spacing (Hz)
`cost`	int	Relative cost
`penalties`	dict	Impairment penalties (CD, PMD, PDL)
`equalization_offset_db`	float	Equalization capability
`f_min`, `f_max`	float	Frequency range (Hz)

TransceiverParams:

Parameter	Type	Description
`mode`	list	List of available modes
`frequency`	dict	min/max frequency support

Examples of transceivers with different modes are as it follos:

```
  "Transceiver": [
  {
    "type_variety": "vendorA_trx-type1",
    "frequency": {
      "min": 191.35e12,
      "max": 196.1e12
    },
    "mode": [
      {
        "format": "mode 1",
        "baud_rate": 32e9,
        "OSNR": 11,
        "bit_rate": 100e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
        "min_spacing": 37.5e9,
        "cost": 1
      },
      {
        "format": "mode 2",
        "baud_rate": 66e9,
        "OSNR": 15,
        "bit_rate": 200e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
```

```
        "min_spacing": 75e9,
        "cost": 1
      }
    ]
  },
  {
    "type_variety": "Voyager",
    "frequency": {
      "min": 191.35e12,
      "max": 196.1e12
    },
    "mode": [
      {
        "format": "mode 1",
        "baud_rate": 32e9,
        "OSNR": 12,
        "bit_rate": 100e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
        "min_spacing": 37.5e9,
        "cost": 1
      },
      {
        "format": "mode 3",
        "baud_rate": 44e9,
        "OSNR": 18,
        "bit_rate": 300e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
        "min_spacing": 62.5e9,
        "cost": 1
      },
      {
        "format": "mode 2",
        "baud_rate": 66e9,
        "OSNR": 21,
        "bit_rate": 400e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
        "min_spacing": 75e9,
        "cost": 1
      },
      {
        "format": "mode 4",
        "baud_rate": 66e9,
        "OSNR": 16,
        "bit_rate": 200e9,
        "roll_off": 0.15,
        "tx_osnr": 40,
        "min_spacing": 75e9,
        "cost": 1
      }
    ]
  }
]
```

Keep in mind that if the mode is not specified the software will choose one at the `auto-design` phase (**??**), evaluating the optimal for the case.

OpenROADM: Transponder Device & Operational Mode Device Model Hierarchy:

```
Transponder Device
|-- Shelf
|-- Circuit Pack
|   |-- Client Ports (Ethernet, OTN)
|   --- Line Ports (optical)
```

```
--- Internal Links (client <-> line)
```

Network Model - Xponder Attributes:

OpenROADM Attribute	Type	Description	Source
`xpdr-number`	uint16	Xponder identifier	Device
`xpdr-type`	enum	Transponder, Muxponder, Switchponder, Regen	Device capability
`customer-code`	string	Customer identifier	Device

Node Capabilities:

Attribute	Type	Description
`supported-xpdr-list`	list	Supported xponder types
`xpdr-type`	identityref	Transponder type
`recolor`	boolean	Recoloring capability
`supported-operational-modes`	leaf-list	Operational mode IDs
`equipment-capacity`	integer	Available capacity
`equipment-available`	integer	Available equipment

Service Model - Operational Mode Catalog (Table - specific-operational-modes):

Parameter	Mandatory	Description
`operational-mode-id`	Yes	Unique mode identifier
`min-central-frequency`	Yes	Lower spectrum boundary (THz)
`max-central-frequency`	Yes	Upper spectrum boundary (THz)
`central-frequency-granularity`	Yes	Spectrum granularity (GHz)
`min-spacing`	Yes	Min channel spacing (GHz)
`baud-rate`	No	Symbol rate (Gbaud)
`line-rate`	Yes	Line rate (Gbps)
`modulation-format`	Yes	Modulation format (enum)
`channel-width`	No	-20 dB channel width (GHz)
`min-TX-osnr`	Yes	Min TX OSNR (dB @ 0.1nm, 193.6 THz)
`min-RX-osnr-tolerance`	Yes	Min RX OSNR required (dB @ 0.1nm)
`min-input-power-at-RX-osnr`	Yes	Min RX input power (dBm)
`max-input-power`	Yes	Max RX input power (dBm)
`min-output-power`	Yes	Min TX output power (dBm)
`max-output-power`	Yes	Max TX output power (dBm)
`fec-type`	No	FEC type
`min-roll-off`	No	Min roll-off factor (dB/decade)
`max-roll-off`	No	Max roll-off factor (dB/decade)
`penalties`	-	Impairment penalties list
`TX-OOB-osnr`	-	Out-of-band noise contributions
`Configurable-output-power`	Yes	Power tunability (boolean)

Penalty Structure:

Parameter	Description
`parameters-and-unit`	Impairment type (CD, PMD, PDL, etc.)
`up-to-boundary`	Upper limit for penalty validity
`penalty-value`	Penalty in dB

Comparison Analysis:

Aspect	GNPy	OpenROADM
Mode Selection	Format string (e.g., 'DP-16QAM')	operational-mode-id
OSNR Requirement	Single value per mode	TX and RX OSNR separately
Output Power	Implicit in network design	Explicit min/max range + configurability
Penalties	Dict of penalty functions (CD, PMD, PDL)	List with boundaries and values
Equipment Hierarchy	Single element	Shelf + Circuit Pack + Ports
Client Interfaces	Not modeled	Explicit client port types (Ethernet, OTN)
Spectrum Granularity	Implicit in spacing	Explicit central-frequency-granularity
OOB Noise	Part of tx_osnr	Explicit TX-OOB-osnr list per SRG config
Cost	Relative integer	Not modeled

4.2.5 FUSED / PASSIVE ELEMENTS

Fused elements represent discrete losses, as a fusion splice in a fiber. "params" is optional, if not used default loss value of 1dB is used.

GNPy: `Fused` Class Attributes:

GNPy Attribute	Type	Description
`uid`	str	Unique identifier
`params`	FusedParams	Loss specification
`loss`	float	Insertion loss (dB)

FusedParams:

Parameter	Type	Description
`loss`	float	Insertion loss (dB)

As an example, a node representing a passive or fused element is shown:

```
  {
     "uid": "ingress fused spans in Site_B",
     "metadata": {
       "location": {
         "city": "Site_B",
         "region": "",
         "latitude": 0,
         "longitude": 0
       }
     },
     "type": "Fused",
     "params": {
       "loss": 0.5
     }
   },
```

OpenROADM: Passive Elements OpenROADM does not have a dedicated passive element type in the Network Model. Passive losses are represented as:

1. **Connector losses** - Part of fiber span loss
2. **Internal ROADM losses** - Part of per-degree impairments
3. **Patch panel/LGX losses** - Included in degree/SRG loss budgets

Implicit Representation: - Coupler losses → Part of ROADM add/drop path loss - Splitter losses → Part of ROADM SRG port pair loss - Patch panels → Part of `ingress-span-loss-aging-margin`

Comparison Analysis:

Aspect	GNPy	OpenROADM
Explicit Modeling	Dedicated Fused element	Implicit in other elements
Loss Specification	Direct loss value (dB)	Bundled into span/ROADM losses
Flexibility	Can model arbitrary passive elements	Limited to standard configurations
Use Cases	Couplers, splitters, combiners, custom passives	Connectors, internal ROADM paths

4.3 Attribute Mapping Summary Tables

4.3.1 Common Attributes Across All Elements

Concept	GNPy	OpenROADM Device	OpenROADM Network
Unique Identifier	`uid` (string)	`device-id` + `circuit-pack` + `port-name`	`node-id` + topology refs
Equipment Type	`type_variety` (string)	`circuit-pack-type`	`type-variety` (optional)
Location	`metadata.location` (lat/lon/city/region)	`shelf.location`	`CLLI` code

Concept	GNPy	OpenROADM Device	OpenROADM Network
Operational State	Not modeled	`operational-state` (enum)	`equipment-state` (Table 14)
Administrative State	Not modeled	`administrative-state` (enum)	Not in network model
Parameters	`params` (from equipment lib)	Device-specific containers	Operational mode catalog

4.3.2 Amplifier Parameters Cross-Reference

Parameter	GNPy	OpenROADM Device (OTS Interface)	OpenROADM Network (ILA)
Gain	`effective_gain` (dB)	`amplifier-gain`	`gain` (ratio-dB)
Target Gain	`gain_target` (dB)	-	`initially-planned-gain`
Tilt	`tilt_target` (dB)	`tilt`	`tilt`, `initially-planned-tilt`
Output VOA	`out_voa` (dB)	`out-voa`	`out-voa-att`
Input VOA	`in_voa` (dB)	`in-voa` (if present)	Not typically modeled
Noise Figure	`nf` (dB, computed)	Not directly exposed	Via operational-mode OSNR mask
NF Model	`nf_model` (type selector)	-	-
NF Min/Max	`nf_min`, `nf_max` (dB)	-	Via OSNR mask
NF Coefficient	`nf_coef` (polynomial)	-	-
NF Ripple	`nf_ripple` (array, dB)	-	-
Output Power	`pout_db` (dBm)	`output-power`	`egress-average-channel-power`
Max Output Power	`p_max` (dBm)	-	-
Delta Power	`delta_p` (dB per channel)	Not directly modeled	Implicit in power settings
Gain Range	`gain_min`, `gain_max` (dB)	`amplifier-gain-range` (1-4)	`amp-gain-range`
Gain Flatmax	`gain_flatmax` (dB)	-	-
Min Gain	-	-	`min-gain` (Service Model)
Max Gain	-	-	`max-gain` (Service Model)
Max Extended Gain	-	-	`max-extended-gain`
Gain Ripple	`gain_ripple` (array, dB)	-	-

Parameter	GNPy	OpenROADM Device (OTS Interface)	OpenROADM Network (ILA)
Tilt Ripple	`tilt_ripple` (array, dB)	-	-
DGT	`dgt` (array, dynamic gain tilt)	-	-
Ripple Reference Freq	`f_ripple_ref` (Hz array)	-	-
Gain Adjust (Power)	-	-	`gain-adjust-power-range` (C/D params)
Ripple Mask (Tilt)	-	-	`mask-gain-ripple-vs-tilt` (C/D params)
Amplifier Type	`type_def` (variable_gain, openroadm, etc.)	`amplifier-type` (EDFA/Raman/Hybrid)	`amp-type`
Type Variety	`type_variety` (string ID)	-	`type-variety`
Frequency Range	`f_min`, `f_max` (Hz)	-	-
Bandwidth	`bandwidth` (Hz, computed)	-	-
PMD	`pmd` (ps)	-	-
PDL	`pdl` (dB)	-	-
Raman Flag	`raman` (boolean)	-	-
Aging Margin	`EOL` in Span config	-	`ingress-span-loss-aging-margin`
EOL Max Input Power	-	-	`eol-max-load-pIn`
Auto Output VOA	`out_voa_auto` (boolean)	-	-
Allowed for Design	`allowed_for_design` (boolean)	-	-
Advanced Config File	`advance_configuration`	`ns_from_json`	-
TX OOB OSNR	-	-	`TX-OOB-osnr` (Service Model)
Dual Stage Model	`dual_stage_model`	-	-
Preamp Variety	`preamp_variety`	-	-
Preamp Type Def	`preamp_type_def`	-	-
Preamp NF Model	`preamp_nf_model`	-	-
Preamp Gain Min	`preamp_gain_min`	-	-
Preamp Gain Flatmax	`preamp_gain_flatmax`	-	-
Booster Variety	`booster_variety`	-	-
Booster Type Def	`booster_type_def`	-	-
Booster NF Model	`booster_nf_model`	-	-
Booster Gain Min	`booster_gain_min`	-	-
Booster Gain Flatmax	`booster_gain_flatmax`	-	-

Parameter	GNPy	OpenROADM Device (OTS Interface)	OpenROADM Network (ILA)
Multi-Band Flag	`multi_band` (boolean)	-	-
Bands	`bands` (array of band defs)	-	-

4.3.3 ROADM Parameters Cross-Reference

Parameter	GNPy	OpenROADM Network Model
Degree Identifier	Implicit in network edges	`degree-number`
Add/Drop Group	Not explicitly modeled	`srg-number`
Max Port Pairs	-	`max-pp` (Npp)
Current Port Pairs	-	`current-provisioned-pp`
Wavelength Duplication	-	Enumerated value (1=per SRG, 2=per Degree)
Express Path Loss	`roadm_path_impairments['express']`	Per-degree express loss
Add Path Loss/OSNR	`roadm_path_impairments['add']`	Add path impairments
Drop Path Loss	`roadm_path_impairments['drop']`	Drop path impairments
Impairment ID	`impairment_id` (reference)	`roadm-path-impairments-id`
ROADM PMD	`roadm-pmd` (in impairment)	`roadm-pmd` (ps)
ROADM CD	`roadm-cd` (in impairment)	`roadm-cd`
ROADM PDL	`roadm-pdl` (in impairment)	`roadm-pdl` (dB)
In-band Crosstalk	`roadm-inband-crosstalk`	`roadm-inband-crosstalk`
ROADM Max Loss	`roadm-maxloss` (in impairment)	`roadm-maxloss` (dB)
ROADM OSNR	`roadm-osnr` (in impairment)	`roadm-osnr` (dB)
ROADM Noise Figure	`roadm-noise-figure`	`roadm-noise-figure` (dB)
Min Loss	`minloss` (in impairment)	`minloss`
Typical Loss	`typloss` (in impairment)	`typloss`
Min Power	`pmin` (in impairment)	`pmin`
Typical Power	`ptyp` (in impairment)	`ptyp`
Target Output Power	`target_pch_out_db`	`egress-average-channel-power`
Target PSD Output	`target_psd_out_mWperGHz`	Not in OpenROADM
Target Output per Slot	`target_out_mWperSlotWidth`	Not in OpenROADM
Per-Degree Power	`per_degree_pch_out_db`	Per-degree power settings
Per-Degree PSD	`per_degree_psd_out_mWperGHz`	Not in OpenROADM
Per-Degree PSW	`per_degree_psd_out_mWperSlotWidth`	Not in OpenROADM
Per-Degree Impairments	`per_degree_impairments` (list)	Not in OpenROADM
PMD	`pmd` (ps)	`pmd` (ps)
PDL	`pdl` (dB)	`pdl` (dB)
Add/Drop OSNR	`add_drop_osnr` (dB)	Via operational-mode penalties
Max Wavelengths	Not explicitly limited	`max-wavelengths` (N)
Available Spectrum Maps	Managed by spectrum assignment	`avail-freq-maps`

Parameter	GNPy	OpenROADM Network Model
Freq Map Name	-	`map-name`
Freq Map Start Edge	-	`start-edge-freq` (THz)
Freq Map Granularity	-	`freq-map-granularity`
Freq Map Effective Bits	-	`effective-bits`
Freq Map	-	`freq-map` (bitmap)
Restrictions	`restrictions` (dict)	Not in OpenROADM
Design Bands	`design_bands` (list)	Not in OpenROADM
Per-Degree Design Bands	`per_degree_design_bands` (dict)	Not in OpenROADM

4.3.4 Fiber Parameters Cross-Reference

Parameter	GNPy	OpenROADM Network (Span)
Length	`length` (m or km)	`SRLG-length` (meters)
Length Units	`length_units`	-
Loss Coefficient	`loss_coef` (dB/km)	Implicit in `fiber-type`
Loss Coef (Freq Dep)	`loss_coef` (dict: value, frequency arrays)	Not in OpenROADM
Total Loss	Computed: `length * loss_coef + con_in + con_out`	`spanloss-current` (measured)
Engineered Span Loss	-	`engineered-spanloss`
Span Loss Last Measured	-	`spanloss-last-measured`
Auto Span Loss	-	`auto-spanloss` (boolean flag)
Fiber Type	`type_variety`	`fiber-type` (SSMF, LEAF, etc.)
Chromatic Dispersion	`dispersion` (ps/(nm·km)) or `beta2`	Implicit in fiber type
Dispersion (Freq Dep)	`dispersion_per_frequency` (dict: value, frequency)	Not in OpenROADM
Dispersion Slope	`dispersion_slope` (ps/(nm^2·km))	Not in OpenROADM
PMD Coefficient	`pmd_coef` (ps/$\sqrt{km}$)	`pmd` (ps/$\sqrt{km}$)
PMD Coef Defined	`pmd_coef_defined` (boolean)	Not in OpenROADM
Effective Area	`effective_area` (m^2)	Not directly modeled
Nonlinear Coefficient	`gamma` (1/(W·km))	Not modeled
Core Radius	`core_radius` (m)	Not in OpenROADM
n1 (Refractive Index)	`n1`	Not in OpenROADM
n2 (Nonlinear Index)	`n2` (m^2/W)	Not in OpenROADM
Reference Wavelength	`ref_wavelength` (m)	Not in OpenROADM
Reference Frequency	`ref_frequency` (Hz)	Not in OpenROADM
Raman Coefficient	`raman_coefficient` (full profile)	Not in OpenROADM

Parameter	GNPy	OpenROADM Network (Span)
Raman Reference Freq	`raman_reference_frequency` (Hz)	Not in OpenROADM
Raman g0	`g0` (array, 1/(W·m))	Not in OpenROADM
Raman Frequency Offset	`frequency_offset` (Hz array)	Not in OpenROADM
Input Attenuator	`att_in` (dB)	Not in OpenROADM
Connector Loss (in)	`con_in` (dB)	Part of span loss
Connector Loss (out)	`con_out` (dB)	Part of span loss
Lumped Losses	`lumped_losses` (array)	Not in OpenROADM
Latency	`latency` (s, computed)	Not in OpenROADM
SRLG	Not modeled	`SRLG-Id`, `link-concatenation`
Future SRLGs	-	`future-SRLGs` with `state-date`

4.3.5 Transceiver/Operational Mode Cross-Reference

Parameter	GNPy (Mode)	OpenROADM (Operational Mode Catalog)
Mode Identifier	`format` (string)	`operational-mode-id` (string)
Baud Rate	`baud_rate` (Hz)	`baud-rate` (Gbaud)
Bit Rate	`bit_rate` (bps)	`line-rate` (Gbps)
Modulation	`format` (e.g., 'DP-16QAM')	`modulation-format` (enum)
Required OSNR	`OSNR` (dB @ 0.1nm)	`min-RX-osnr-tolerance` (dB @ 0.1nm)
TX OSNR	`tx_osnr` (dB)	`min-TX-osnr` (dB @ 0.1nm @ 193.6 THz)
Min Channel Spacing	`min_spacing` (Hz)	`min-spacing` (GHz)
Channel Spacing	`spacing` (Hz)	Not in OpenROADM
Roll-Off	`roll_off`	`min-roll-off`, `max-roll-off` (dB/decade)
Frequency Range	`f_min`, `f_max` (Hz)	`min-central-frequency`, `max-central-frequency` (THz)
Central Freq Granularity	-	`central-frequency-granularity` (GHz)
Channel Width	-	`channel-width` (-20 dB width, GHz)
CD Penalty	`penalties['chromatic_dispersion']`	`penalties` with `parameters-and-unit='CD'`
PMD Penalty	`penalties['pmd']`	`penalties` with `parameters-and-unit='PMD'`
PDL Penalty	`penalties['pdl']`	`penalties` with `parameters-and-unit='PDL'`

Parameter	GNPy (Mode)	OpenROADM (Operational Mode Catalog)
Output Power Range	Not directly specified	`min-output-power`, `max-output-power`
Input Power Range	Not directly specified	`min-input-power-at-RX-osnr`, `max-input-power`
FEC	Not modeled	`fec-type`
Equalization	`equalization_offset_db` (dB)	Not explicitly modeled
Cost	`cost` (integer)	Not modeled
Originator	-	`originator` (vendor string)
Sponsor	-	`sponsor` (service provider)
TX OOB OSNR	-	`TX-OOB-osnr` (per SRG architecture)
Transceiver Design Bands	`design_bands` (list)	Not in OpenROADM
Per-Degree Design Bands	`per_degree_design_bands` (dict)	Not in OpenROADM

4.3.6 Simulation Parameters (GNPy Only)

Parameter	Description
NLI Solver Parameters (NLIParams)	
`method`	NLI calculation method (gn_model_analytic, ggn, etc.)
`dispersion_tolerance`	Tuning parameter for GGN model
`phase_shift_tolerance`	Tuning parameter for GGN model
`computed_channels`	Specific channels for NLI evaluation
`computed_number_of_channels`	Number of channels for NLI evaluation
Raman Solver Parameters (RamanParams)	
`flag`	Enable/disable Raman evaluation (boolean)
`method`	Raman solver method (perturbative, etc.)
`order`	Solution order for perturbative method
`result_spatial_resolution`	Spatial resolution of Raman profile (m)
`solver_spatial_resolution`	Spatial step for ODE solution (m)
Pump Parameters (PumpParams)	
`power`	Pump power
`frequency`	Pump frequency (Hz)
`propagation_direction`	Forward or backward

4.3.7 OTN Parameters (OpenROADM Only)

Parameter Category	Key Parameters
OTN Topology Node	`tp-bandwidth-sharing`, `switching-pools`, `xpdr-attributes`
Switching Pools	`switching-pool-number`, `switching-pool-type` (blocking/non-blocking)

Parameter Category	Key Parameters
Non-Blocking List	`interconnect-bandwidth-unit`, `capable-interconnect-bandwidth`
OTN Termination Point	`tp-supported-interfaces`, `tail-equipment`, `supported-client-services`
Tributary Parameters	`odtu-tpn-pool`, `tpn-pool`, `ts-pool`
ODU Parameters	`rate`, `Oducn-n-rate`, `Oduflex-cbr-service`, `Oduflex-gfp-number-ts`
OTU Parameters	`rate`, `Otucn-N-rate`, `Otucn-M-subrate`
OTSi Parameters	`Otsi-rate`, `OTSi-group` (Group-rate, Group-id)
OTN Links	`available-bandwidth`, `used-bandwidth`, `ODU-protected`

Note: OpenROADM OTN parameters are for multi-layer network management. GNPy focuses on optical layer QoT analysis only.

4.4 Key Architectural Differences

4.4.1 Abstraction Philosophy

GNPy: - **Physics-first abstraction**: Focuses on optical impairment modeling for path feasibility - **Network planning tool**: Optimized for "what-if" analysis and design - **Simplified equipment model**: Single element per network component - **Automated design**: Built-in algorithms for amplifier placement and configuration

OpenROADM: - **Device management model**: Focuses on real-world equipment control and inventory - **SDN controller interface**: Designed for service provisioning and lifecycle management - **Detailed equipment hierarchy**: Multi-level (device → shelf → circuit pack → port) - **Planning tool integration**: Expects external planning tools, provides configuration interface

4.4.2 Scope Comparison

Aspect	GNPy	OpenROADM
Primary Use Case	Network planning & path computation (offline)	Device configuration & service provisioning (online)
Time Domain	Design-time and what-if analysis	Operational runtime
Equipment Inventory	Type varieties in library	Full device hierarchy with bay/shelf/slot
Physical Connectivity	Logical network topology only	Physical internal links + LGX tracking
Impairment Modeling	Detailed physics (NLI, Raman, CD, PMD, PDL)	Simplified OSNR penalties + measured values
Multi-Vendor	Vendor-agnostic equipment library	Vendor-specific device models + standard interface
Spectrum Management	Channel assignment algorithms	Frequency maps + wavelength duplication tracking

Aspect	GNPy	OpenROADM
Fault Management	Not modeled	Operational/administrative states + alarms

4.4.3 Data Flow Comparison

GNPy Workflow:

```
Equipment Library (JSON) → Network Topology (JSON) → Simulation Engine → Impairment Resul
                              ↓                          ↓                  ↓
                    Service Requests (JSON)    Auto-Design Algorithm   Feasibility Asses
```

OpenROADM Workflow:

```
Device Model (YANG) → Controller Inventory → Network Model (YANG) → Service Model (YANG)
       ↓                     ↓                      ↓                      ↓
Physical Equipment     Device Discovery     Topology Abstraction   Path Computation (PCE)
```

4.4.4 Attribute Source Mapping

Information Type	GNPy Source	OpenROADM Source
Equipment Specifications	Equipment library JSON	Device model YANG + vendor specs
Topology	User-defined JSON/XLS	Network discovery + planning input
Physical Locations	Metadata (lat/lon)	CLLI codes + bay/shelf/slot
Operational Parameters	Computed or user-defined	Device telemetry (PM) + SDN controller
Design Values	Auto-design algorithm	External planning tools
Measurements	Not collected (simulation only)	Device telemetry + OTS interface
Impairment Budget	Physics-based computation	Operational mode catalog + penalties

4.5 Coverage Analysis

4.5.1 What GNPy Models

- Detailed nonlinear impairment modeling (GN/GGN models)
- Raman amplification (dedicated solver)
- Chromatic dispersion accumulation
- Multi-band amplifier support (C+L)
- Automatic amplifier placement and configuration
- Spectrum assignment algorithms
- Path computation with physical constraints
- Equipment library extensibility

4.5.2 What GNPy Does NOT Model

- Physical equipment hierarchy (shelves, circuit packs, ports)
- Equipment inventory tracking (bay/FIC, slot locations)
- Internal device connectivity (physical links inside ROADM)
- Operational and administrative states
- Fault management and alarms
- Client-side interfaces (Ethernet, OTN)
- Equipment lifecycle (planned, deployed, available, in-use)
- Vendor-specific device models
- Real-time telemetry integration
- Service protection schemes (SNCP, equipment protection groups)

4.5.3 What OpenROADM Models

- Complete equipment inventory (shelves to ports)
- Physical connectivity tracking (internal links, LGX jacks)
- Multi-layer service modeling (OTN, Ethernet, optical)
- Equipment lifecycle states (Table 14)
- Vendor-specific device models
- Operational mode catalog (standardized specifications)
- Protection groups and SNCP
- Turn-back links (IOWN architecture support)
- Client and line side interfaces
- Tail circuit management (pluggables, external connections)

4.5.4 What OpenROADM Does NOT Model

Not Directly Covered (delegated to planning tools or implicit):

- Detailed nonlinear impairment computation (GN/GGN)
- Automatic amplifier placement algorithms
- Physics-based path feasibility (relies on operational mode catalog)
- Raman solver (hybrid amplifiers supported, but no detailed modeling)
- Spectrum assignment algorithms (provides freq-maps, not algorithms)
- What-if network design scenarios

This is just an extensively way to say that OpenROADM is a top level model, the implementations are up to the lower level developers.

4.6 Integration Opportunities

4.6.1 GNPy → OpenROADM Translation

Feasible Mappings: 1. **Auto-design output → Network Model**: - GNPy computed gains → `initially-planned-gain` - GNPy tilt targets → `initially-planned-tilt` - GNPy amplifier placements → ILA section elements

2. **Equipment library → Operational Mode Catalog**:
   - GNPy transceiver modes → `specific-operational-modes`
   - GNPy amplifier specs → amplifier operational mode parameters
   - GNPy ROADM impairments → ROADM operational mode impairments
3. **Path computation → Service provisioning**:
   - GNPy feasible paths → OpenROADM service path-requests
   - GNPy spectrum assignment → OpenROADM frequency maps

Challenges: - GNPy has no equipment hierarchy → Manual mapping to shelves/slots needed - GNPy type_variety → OpenROADM circuit-pack-type requires mapping table - GNPy physical impairments → OpenROADM OSNR penalties (lossy conversion)

4.6.2 OpenROADM → GNPy Translation

Feasible Mappings: 1. **Network Model → GNPy Topology**: - OpenROADM nodes → GNPy elements (Edfa, Roadm, Transceiver) - Express/Add/Drop links → GNPy network connections - Span parameters → GNPy fiber parameters

2. **Operational Mode Catalog → Equipment Library**:
   - `specific-operational-modes` → GNPy transceiver modes
   - Amplifier operational modes → GNPy EDFA parameters
   - ROADM operational modes → GNPy ROADM impairments
3. **Measured telemetry → GNPy validation**:
   - Actual gains → Validate GNPy auto-design
   - Span losses → Validate GNPy fiber model
   - OSNR measurements → Validate GNPy impairment computation

Challenges: - OpenROADM detailed hierarchy → Collapse to single GNPy element - Measured values vs physics models (reconciliation needed) - OpenROADM penalties → GNPy penalty functions (format mismatch) - Multi-band modeling (OpenROADM implicit, GNPy explicit Multiband_amplifier)

4.6.3 Hybrid Workflow Example

```
[OpenROADM Controller] ← Discovered Network Topology
          ↓
[Extract Network Model] → Convert to GNPy JSON format
          ↓
[GNPy Path Computation] ← New service request
          ↓
[Feasibility + Impairments] → GSNR, OSNR, penalties
          ↓
[Translate to Service Model] → OpenROADM path-request
          ↓
```

```
[Provision via OpenROADM] → Service provisioning
          ↓
[Monitor via Telemetry] → Validate against GNPy predictions
```

4.7 Conclusions and Recommendations

4.7.1 7.1 Summary of Findings

1. **Complementary Models**: GNPy and OpenROADM serve different but complementary purposes in optical network management:
   - **GNPy**: Offline planning, path computation, impairment-aware design
   - **OpenROADM**: Online provisioning, device management, service lifecycle
2. **Abstraction Gap**: GNPy operates at a higher abstraction level (network elements) while OpenROADM provides detailed equipment hierarchy (device → shelf → circuit pack → port).
3. **Impairment Modeling**: GNPy provides detailed physics-based models (NLI, Raman), while OpenROADM relies on pre-computed operational mode catalogs with penalty tables.
4. **Equipment Representation**:
   - **GNPy**: Equipment library with type_variety → simplified parameters
   - **OpenROADM**: Multi-level hierarchy with vendor-specific device models

4.7.2 Mapping Completeness

Device Type	GNPy → OpenROADM	OpenROADM → GNPy	Fidelity
Amplifier (EDFA)	ok Good	w Lossy	Medium
ROADM	w Partial	w Partial	Medium-Low
Fiber/Span	ok Good	w Lossy	Medium
Transceiver	ok Good	ok Good	High
Fused/Passive	no Limited	no Implicit	Low

Legend: - ok Good: Direct mapping with minimal information loss - w Partial/Lossy: Mapping possible but with information loss or gaps - no Limited/Implicit: Weak or no direct mapping

4.7.3 Recommendations for Network Operators

For Planning Teams: 1. Use GNPy for: - Initial network design and dimensioning - Amplifier placement optimization - Path feasibility studies with physical impairments - What-if scenario analysis

2. Use OpenROADM for:
   - Service provisioning
   - Equipment inventory management
   - Operational state monitoring
   - Multi-vendor device interoperability

For SDN/Controller Development: 1. **Maintain both models**: - GNPy equipment library for planning - OpenROADM operational mode catalog for provisioning

2. **Bidirectional sync**:
   - GNPy auto-design → OpenROADM planned parameters
   - OpenROADM telemetry → GNPy model validation
3. **Translation layer**:
   - Build mapping tables: type_variety <-> circuit-pack-type
   - Convert impairment models: GNPy physics <-> OpenROADM penalties
   - Reconcile topology: GNPy network <-> OpenROADM device hierarchy

4.7.4 Future Enhancements

For GNPy: 1. Add optional equipment hierarchy (shelf/slot) for OpenROADM compatibility 2. Support operational-mode-id as alternative to format strings 3. Add SRLG modeling for diversity routing 4. Export to OpenROADM YANG format

For OpenROADM: 1. Enhance operational mode catalog with physics-based parameters (gamma, GVD) 2. Add optional simplified mode for planning tools (flat hierarchy) 3. Standardize impairment penalty formats for tool interoperability 4. Reference GNPy as example planning tool in whitepaper

4.8 References

1. **GNPy Project**: https://github.com/Telecominfraproject/oopt-gnpy
2. **OpenROADM MSA Device Model Whitepaper v13.1** (April 2024)
3. **OpenROADM MSA Network Model Whitepaper v13.1** (April 2024)
4. **OpenROADM MSA Service Model Whitepaper v13.1**
5. GNPy source code: `gnpy.core.elements`, `gnpy.core.parameters`, `gnpy.core.network`

4.9 Appendix A: GNPy Element Class Hierarchy

```
_Node (base class)
|-- Transceiver
|-- Roadm
|-- Fused
|-- Fiber
|   --- RamanFiber
|-- Edfa
--- Multiband_amplifier
```

4.10 Appendix B: OpenROADM Device Model Structure

```
Device
|-- info (device-id, vendor, model, serial-number)
|-- shelves[]
|   |-- shelf-name
|   |-- shelf-type
|   |-- rack, shelf-position
|   |-- administrative-state
|   --- equipment-state
```

```
|-- circuit-packs[]
|   |-- circuit-pack-name
|   |-- circuit-pack-type
|   |-- shelf-name, slot-name, subSlot-name
|   |-- ports[]
|   |   |-- port-name
|   |   |-- port-type
|   |   |-- port-qual (xpdr-network, xpdr-client, roadm-internal, etc.)
|   |   |-- interfaces[] (optical interfaces, OTN, Ethernet)
|   |   --- partner-port (for internal connections)
|   --- parent-circuit-pack
|-- degree[]
|   |-- degree-number
|   |-- max-wavelengths
|   --- connection-ports[] (internal circuit pack ports)
|-- shared-risk-group[]
|   |-- srg-number
|   |-- max-add-drop-ports
|   --- connection-ports[]
|-- internal-links[]
|   |-- internal-link-name
|   |-- source (circuit-pack-name, port-name)
|   --- destination (circuit-pack-name, port-name)
--- ots-interface (optical transport section)
    |-- amplifier-type, amplifier-gain-range
    |-- span-loss, ingress-span-loss-aging-margin
    --- operational telemetry
```

4.11 Appendix C: Equipment Configuration Example Comparison

4.11.1 GNPy Equipment Library (eqpt_config.json excerpt)

```
{
  "Edfa": {
    "std_medium_gain": {
      "type_variety": "std_medium_gain",
      "type_def": "variable_gain",
      "gain_flatmax": 27,
      "gain_min": 15,
      "p_max": 21,
      "nf_model": {
        "enabled": true,
        "nf_coef": [-8.104e-4, 6.221e-2, -5.889e-1, 3.044]
      },
      "allowed_for_design": true
    }
  },
  "Fiber": {
    "SSMF": {
      "type_variety": "SSMF",
      "dispersion": 16.7,
      "gamma": 1.27e-3,
      "pmd_coef": 0.1
    }
  },
  "Transceiver": {
    "vendorA_trx": {
      "type_variety": "vendorA_trx",
```

```
      "frequency": {"min": 191.35e12, "max": 196.1e12},
      "mode": [
        {
          "format": "DP-16QAM",
          "baud_rate": 32e9,
          "OSNR": 15.0,
          "bit_rate": 200e9,
          "roll_off": 0.15,
          "tx_osnr": 40.0,
          "min_spacing": 50e9
        }
      ]
    }
  }
}
```

4.11.2 OpenROADM Operational Mode Catalog (simplified excerpt)

```
operational-mode-catalog:
  specific-operational-modes:
    - operational-mode-id: "vendor-A-200G-DP16QAM-C-band"
      min-central-frequency: 191.35  # THz
      max-central-frequency: 196.1   # THz
      baud-rate: 32  # Gbaud
      line-rate: 200  # Gbps
      modulation-format: "dp-16qam"
      min-TX-osnr: 40.0  # dB @ 0.1nm
      min-RX-osnr-tolerance: 15.0  # dB @ 0.1nm
      min-output-power: -10  # dBm
      max-output-power: 2    # dBm
      min-spacing: 50  # GHz
      penalties:
        - parameters-and-unit: "chromatic-dispersion-ps-per-nm"
          up-to-boundary: 10000
          penalty-value: 1.0
        - parameters-and-unit: "pmd-ps"
          up-to-boundary: 20
          penalty-value: 1.5
```

End of Guide

For questions, refer to:

- Official docs: https://gnpy.readthedocs.io/
- GitHub: https://github.com/TelecomInfraProject/oopt-gnpy
- Example files in `/example-data/` directory

5 GNPy Configuration Guide

5.1 Complete Reference for Equipment and Network Configuration

---

5.2 Table of Contents

5.3 1. Configuration File Overview

5.3.1 Core Configuration Files

GNPy uses several JSON files to configure optical networks:

File	Purpose	Required?
`eqpt_config.json`	Equipment library (EDFAs, fibers, ROADMs, transceivers)	Yes
`topology.json`	Network topology (nodes and links)	Yes
`services.json`	Service requests (path demands)	Optional
`sim_params.json`	Simulation parameters (NLI, Raman)	Optional
`initial_spectrum.json`	Spectrum definition for transmission-example	Optional
Advanced EDFA configs	Detailed amplifier models	Optional

5.3.2 File Relationships

```
eqpt_config.json
|-- References advanced_config files
|   |-- Juniper-BoosterHG.json
|   |-- std_medium_gain_advanced_config.json
|
topology.json
|-- Uses equipment types from eqpt_config.json
|   |-- type_variety: "std_medium_gain" -> defined in eqpt_config
|   |-- type_variety: "SSMF" -> defined in eqpt_config
|
sim_params.json
|-- Configures physical models (NLI, Raman)
```

5.4 2. Equipment Configuration

5.4.1 2.1 Structure of eqpt_config.json

The equipment configuration has 7 main sections:

```
{
  "Edfa": [...],           // Optical amplifiers
  "Fiber": [...],          // Fiber types
  "RamanFiber": [...],     // Raman-enabled fibers
  "Span": {...},           // Span design rules
  "Roadm": [...],          // ROADM configurations
  "SI": [...],             // Spectral Information defaults
  "Transceiver": [...]     // Transceiver modes
}
```

5.4.2 2.2 EDFA Configuration

```
{
  "type_variety": "std_medium_gain",
  "type_def": "variable_gain",
  "gain_flatmax": 26,      // Maximum flat gain (dB)
  "gain_min": 15,          // Minimum gain (dB)
  "p_max": 23,             // Maximum output power (dBm)
  "nf_min": 6,             // Minimum noise figure (dB)
  "nf_max": 10,            // Maximum noise figure (dB)
  "out_voa_auto": false,   // Automatic VOA adjustment
  "allowed_for_design": true  // Can auto-design use this?
}
```

Key Parameters Explained:

- **`type_def`**: Model type
- `"variable_gain"`: Simple NF model (linear interpolation)
- `"advanced_model"`: Detailed model with ripple and DGT
- `"openroadm"`: OpenROADM standard model
- `"fixed_gain"`: Fixed gain amplifier
- `"dual_stage"`: Two-stage amplifier (preamp + booster)
- `"multi_band"`: Multi-band amplifier (C+L band)
- **`allowed_for_design`**: Controls whether auto-design algorithm can select this amplifier type
- **Noise Figure Model** (`variable_gain`):

$$\text{NF}(G) = \text{nf\_min} + (\text{nf\_max} - \text{nf\_min}) \times \frac{\text{gain\_flatmax} - G}{\text{gain\_flatmax} - \text{gain\_min}}$$

Where G is the operating gain.

```
{
  "type_variety": "Juniper_BoosterHG",
  "type_def": "advanced_model",
  "gain_flatmax": 25,
  "gain_min": 10,
  "p_max": 21,
  "advanced_config_from_json": "Juniper-BoosterHG.json",
  "out_voa_auto": false,
  "allowed_for_design": false
}
```

This references an external file (`Juniper-BoosterHG.json`) containing:

- `nf_fit_coeff`: NF polynomial coefficients [a, a, a, a]
- `nf_ripple`: NF variation across channels (96 values)
- `gain_ripple`: Gain flatness profile (96 values)
- `dgt`: Dynamic Gain Tilt coefficients (96 values)
- `f_min`, `f_max`: Operating frequency range

```
{
  "type_variety": "openroadm_ila_low_noise",
  "type_def": "openroadm",
  "gain_flatmax": 27,
  "gain_min": 0,
  "p_max": 22,
  "nf_coef": [-8.104e-4, -6.221e-2, -5.889e-1, 37.62],
  "allowed_for_design": false
}
```

NF Model (OpenROADM):

$$\mathrm{NF}(G) = a_3 G^3 + a_2 G^2 + a_1 G + a_0$$

```
{
  "type_variety": "hybrid_4pumps_mediumgain",
  "type_def": "dual_stage",
  "raman": true,
  "gain_min": 25,
  "preamp_variety": "4pumps_raman",
  "booster_variety": "std_medium_gain",
  "allowed_for_design": true
}
```

Behavior: Signal passes through preamp, then booster. Total gain = preamp gain + booster gain.

```
{
  "type_variety": "std_low_gain_multiband",
  "type_def": "multi_band",
  "amplifiers": [
    "std_low_gain",      // C-band amplifier
    "std_low_gain_L"     // L-band amplifier
  ],
  "allowed_for_design": false
}
```

Each band amplifier must define `f_min` and `f_max`:

```
{
  "type_variety": "std_low_gain",
  "f_min": 191.25e12,   // C-band: 191.25 THz
  "f_max": 196.15e12,   // C-band: 196.15 THz
  "type_def": "variable_gain",
  ...
}
```

5.4.3 2.3 Fiber Configuration

```
{
  "type_variety": "SSMF",
  "dispersion": 1.67e-05,      // Dispersion (s/m)
  "effective_area": 83e-12,    // Effective area (m)
  "pmd_coef": 1.265e-15        // PMD coefficient (s/m)
}
```

Physical Meanings:

- **Dispersion**: 16.7 ps/(nm·km) typical for SSMF
- **Effective Area**: 83 μm^2 (determines nonlinear effects)
- **PMD Coefficient**: 1.265 ps/$\sqrt{\text{km}}$

Nonlinear Coefficient (calculated from effective area):

$$\gamma = \frac{2\pi n_2}{\lambda A_{\text{eff}}}$$

5.4.4 2.4 RamanFiber Configuration

For Raman amplification, fibers must be defined in the `RamanFiber` section:

```
"RamanFiber": [
  {
    "type_variety": "SSMF",
    "dispersion": 1.67e-05,
    "effective_area": 83e-12,
    "pmd_coef": 1.265e-15,
    "raman_coefficient": {
      "frequency_offset": [0, 13.2e12],
```

```
      "cr": [0, 0.7e-13]
    }
  }
]
```

Parameters:

- `frequency_offset`: Frequency offset from pump (Hz)
- `cr`: Raman gain coefficient (m/W)

5.4.5 2.5 Span Configuration

The `Span` section defines design rules for fiber spans:

```
{
  "power_mode": true,
  "delta_power_range_db": [-2, 3, 0.5],
  "max_length": 150,
  "length_units": "km",
  "max_loss": 28,
  "padding": 10,
  "EOL": 0,
  "con_in": 0.5,
  "con_out": 0.5
}
```

Key Parameters:

- **`power_mode`**: Enable power sweep optimization
- **`delta_power_range_db`**: [min, max, step] for power sweep (dB)
- **`max_length`**: Maximum span length
- **`max_loss`**: Maximum span loss (dB)
- **`padding`**: Loss margin for design (dB)
- **`EOL`**: End-of-life degradation margin (dB)
- **`con_in`**, `con_out`: Connector losses (dB)

Example: Power Mode Disabled

```
{
  "power_mode": false,
  "max_length": 120,
  "length_units": "km",
  "max_loss": 25,
  "padding": 8,
  "EOL": 1
}
```

In this mode, GNPy uses a fixed target power per channel.

5.4.6 2.6 ROADM Configuration

```
{
  "target_pch_out_db": -20,
  "add_drop_osnr": 38,
  "pmd": 0,
  "pdl": 0,
  "restrictions": {
    "preamp_variety_list": [],
    "booster_variety_list": []
  }
}
```

Parameters:

- **`target_pch_out_db`**: Target output power per channel (dBm)
- **`add_drop_osnr`**: OSNR penalty for add/drop (dB)
- **`pmd`**: PMD introduced by ROADM (ps)
- **`pdl`**: Polarization Dependent Loss (dB)
- **`restrictions`**: Allowed amplifier types before/after ROADM

OpenROADM Compliant ROADM:

```
{
  "target_pch_out_db": -20,
  "add_drop_osnr": 33,  // Ver 5.0 spec
  "pmd": 0.5,
  "pdl": 0.3
}
```

5.4.7 2.7 SI (Spectral Information) Configuration

Default spectrum configuration:

```
{
  "f_min": 191.3e12,
  "f_max": 195.1e12,
  "baud_rate": 32e9,
  "spacing": 50e9,
  "power_dbm": 0,
  "power_range_db": [0, 0, 1],
  "roll_off": 0.15,
  "tx_osnr": 40,
  "sys_margins": 2
}
```

Parameters:

- **`f_min`**, **`f_max`**: Frequency range (Hz)
- **`baud_rate`**: Symbol rate (baud)

- **spacing**: Channel spacing (Hz)
- **power_dbm**: Launch power per channel (dBm)
- **power_range_db**: [min, max, step] for power sweep
- **roll_off**: Pulse shaping roll-off factor (0-1)
- **tx_osnr**: Transmitter OSNR (dB)
- **sys_margins**: System margin (dB)

ITU Grid Example (100 GHz spacing):

```
{
  "f_min": 191.35e12,
  "f_max": 196.1e12,
  "baud_rate": 32e9,
  "spacing": 100e9,
  "power_dbm": 0
}
```

5.4.8 2.8 Transceiver Configuration

```
{
  "type_variety": "default",
  "frequency": {
    "min": 191.35e12,
    "max": 196.1e12
  },
  "mode": [
    {
      "format": "DP-QPSK 100G",
      "baud_rate": 32e9,
      "OSNR": 12,
      "bit_rate": 100e9,
      "roll_off": 0.15,
      "tx_osnr": 40,
      "min_spacing": 50e9,
      "cost": 1
    }
  ]
}
```

Mode Parameters:

- **format**: Modulation format description
- **baud_rate**: Symbol rate (baud)
- **OSNR**: Required OSNR at receiver (dB at 0.1 nm)
- **bit_rate**: Line rate (bits/s)
- **roll_off**: Nyquist filter roll-off
- **tx_osnr**: Transmitter OSNR (dB)

- **`min_spacing`**: Minimum channel spacing (Hz)
- **`cost`**: Relative cost (for mode selection)

```
{
  "type_variety": "Voyager",
  "frequency": {
    "min": 191.35e12,
    "max": 196.1e12
  },
  "mode": [
    {
      "format": "DP-16QAM 400G",
      "baud_rate": 69e9,
      "OSNR": 25,
      "bit_rate": 400e9,
      "roll_off": 0.15,
      "tx_osnr": 40,
      "min_spacing": 75e9,
      "cost": 1
    },
    {
      "format": "DP-QPSK 200G",
      "baud_rate": 69e9,
      "OSNR": 15,
      "bit_rate": 200e9,
      "roll_off": 0.15,
      "tx_osnr": 40,
      "min_spacing": 75e9,
      "cost": 1
    }
  ]
}
```

GNPy will automatically select the best mode based on path impairments.

```
{
  "format": "DP-16QAM 400G",
  "baud_rate": 69e9,
  "OSNR": 25,
  "bit_rate": 400e9,
  "chromatic_dispersion": 60000,  // ps/nm tolerance
  "pmd": 40,                       // ps tolerance
  "penalties": [
    {
      "chromatic_dispersion": 40000,
      "penalty_value": 1
    }
  ]
}
```

Penalty Model:

- Base required OSNR: 25 dB

- At 40,000 ps/nm CD: +1 dB penalty → 26 dB required
- Maximum CD: 60,000 ps/nm

5.5 3. Advanced EDFA Models

5.5.1 3.1 Advanced Config File Structure

An advanced EDFA configuration file contains detailed amplifier characteristics:

```
{
  "nf_fit_coeff": [0.0001, -0.015, 0.8, 5.2],
  "nf_ripple": [
    5.2, 5.3, 5.25, 5.28, ...  // 96 values
  ],
  "gain_ripple": [
    0.0, 0.1, -0.05, 0.08, ...  // 96 values
  ],
  "dgt": [
    0.0, 0.02, -0.01, 0.03, ...  // 96 values
  ],
  "f_min": 191.35e12,
  "f_max": 196.1e12
}
```

Components:

- **`nf_fit_coeff`**: Polynomial coefficients $[a_3, a_2, a_1, a_0]$ for NF vs. gain
- **`nf_ripple`**: Per-channel NF variation (96 channels)
- **`gain_ripple`**: Per-channel gain flatness (96 channels)
- **`dgt`**: Dynamic Gain Tilt coefficients (96 channels)
- **`f_min`**, **`f_max`**: Valid frequency range

5.5.2 3.2 NF Polynomial Model

The noise figure is calculated as:

$$\text{NF}_{\text{base}}(G) = a_3 G^3 + a_2 G^2 + a_1 G + a_0$$

Then modified by the ripple:

$$\text{NF}_{\text{channel}}(f_i, G) = \text{NF}_{\text{base}}(G) + \text{nf\_ripple}[i]$$

Example:

For gain $G = 20$ dB and coefficients $[0.0001, -0.015, 0.8, 5.2]$:

$$\text{NF}_{\text{base}} = 0.0001(20)^3 - 0.015(20)^2 + 0.8(20) + 5.2 = 15.2 \text{ dB}$$

At channel 48 (193.1 THz) with `nf_ripple[48] = 0.3`:

$$\text{NF}_{\text{channel}} = 15.2 + 0.3 = 15.5 \text{ dB}$$

5.5.3 3.3 Gain Ripple

The gain ripple array defines the gain flatness across the spectrum:

```
"gain_ripple": [
  0.0,   // Channel 1: no deviation
  0.2,   // Channel 2: +0.2 dB
  -0.1,  // Channel 3: -0.1 dB
  ...
]
```

Application:

If the target gain is $G_{\text{target}} = 20$ dB:

- Channel 1: $G = 20.0$ dB
- Channel 2: $G = 20.2$ dB
- Channel 3: $G = 19.9$ dB

5.5.4 3.4 Dynamic Gain Tilt (DGT)

DGT represents the change in gain tilt with input power:

$$\Delta G_{\text{tilt}}(f_i, \Delta P) = \text{dgt}[i] \times \Delta P$$

Where ΔP is the deviation from nominal input power.

Example:

For `dgt[48] = 0.05` dB/dB and input power 3 dB above nominal:

$$\Delta G_{\text{tilt}} = 0.05 \times 3 = 0.15 \text{ dB}$$

5.5.5 3.5 Building Advanced Config Files

GNPy provides tools to build advanced configuration files from measurement data.

Required Data Files:

- `DFG_96.txt`: Gain flatness (96 channels)
- `NFR_96.txt`: NF ripple (96 channels)
- `DGT_96.txt`: Dynamic gain tilt (96 channels)
- `NF_coeff.txt`: NF polynomial coefficients

File Format (DFG_96.txt):

```
0.0
0.1
-0.05
0.08
...
(96 lines total)
```

Build Command:

```
python gnpy/tools/build_oa_json.py \
  --pointer-file vendor_pointer.json \
  --output vendor_advanced_config.json
```

Pointer File (vendor_pointer.json):

```
{
  "nf_ripple": "NFR_96.txt",
  "gain_ripple": "DFG_96.txt",
  "dgt": "DGT_96.txt",
  "nf_fit_coeff": "NF_coeff.txt"
}
```

5.6 4. Topology Configuration

5.6.1 4.1 Topology File Structure

```
{
  "elements": [
    {
      "uid": "NodeA",
      "type": "Transceiver",
      ...
    },
    {
      "uid": "Fiber1",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {
        "length": 80.0,
```

```
        "length_units": "km"
      }
    },
    ...
  ],
  "connections": [
    {
      "from_node": "NodeA",
      "to_node": "Fiber1"
    },
    ...
  ]
}
```

5.6.2 4.2 Network Elements

```
{
  "uid": "NodeA",
  "type": "Transceiver",
  "metadata": {
    "location": {
      "city": "Paris",
      "region": "IDF",
      "latitude": 48.8566,
      "longitude": 2.3522
    }
  }
}
```

```
{
  "uid": "Fiber_A_to_B",
  "type": "Fiber",
  "type_variety": "SSMF",
  "params": {
    "length": 80.0,
    "length_units": "km",
    "loss_coef": 0.2,
    "con_in": 0.5,
    "con_out": 0.5
  }
}
```

Parameters:

- `length`: Fiber length
- `length_units`: "km" or "m"
- `loss_coef`: Attenuation (dB/km), overrides `type_variety`
- `con_in`, `con_out`: Connector losses (dB)

```
{
  "uid": "Amp1",
  "type": "Edfa",
  "type_variety": "std_medium_gain",
  "operational": {
    "gain_target": 20,
    "delta_p": 0,
    "tilt_target": 0,
    "out_voa": 0
  }
}
```

Operational Parameters:

- `gain_target`: Target gain (dB)
- `delta_p`: Power adjustment (dB)
- `tilt_target`: Gain tilt (dB)
- `out_voa`: Output VOA attenuation (dB)

```
{
  "uid": "ROADM_Paris",
  "type": "Roadm",
  "params": {
    "target_pch_out_db": -20,
    "add_drop_osnr": 38
  }
}
```

Fused A `Fused` element represents a passive optical component (splitter/coupler):

```
{
  "uid": "Splitter1",
  "type": "Fused",
  "params": {
    "loss": 3.5
  }
}
```

5.6.3 4.3 Connections

Connections define the network topology:

```
"connections": [
  {
    "from_node": "NodeA",
    "to_node": "Fiber1"
  },
  {
    "from_node": "Fiber1",
    "to_node": "Amp1"
  },
  {
    "from_node": "Amp1",
```

```
    "to_node": "ROADM_Paris"
  }
]
```

Rules:

- Each element can have multiple inputs/outputs
- Transceivers mark the start/end of paths
- Network must form a directed graph

5.6.4 4.4 Complete Example Topology

```
{
  "elements": [
    {
      "uid": "Paris",
      "type": "Transceiver"
    },
    {
      "uid": "Fiber_Paris_Lyon",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {
        "length": 80.0,
        "length_units": "km"
      }
    },
    {
      "uid": "Amp_preROADM_Lyon",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {
      "uid": "ROADM_Lyon",
      "type": "Roadm"
    },
    {
      "uid": "Amp_postROADM_Lyon",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {
      "uid": "Fiber_Lyon_Marseille",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {
        "length": 120.0,
        "length_units": "km"
      }
    },
    {
      "uid": "Amp_Marseille",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {
      "uid": "Marseille",
      "type": "Transceiver"
```

```
    }
  ],
  "connections": [
    {"from_node": "Paris", "to_node": "Fiber_Paris_Lyon"},
    {"from_node": "Fiber_Paris_Lyon", "to_node": "Amp_preROADM_Lyon"},
    {"from_node": "Amp_preROADM_Lyon", "to_node": "ROADM_Lyon"},
    {"from_node": "ROADM_Lyon", "to_node": "Amp_postROADM_Lyon"},
    {"from_node": "Amp_postROADM_Lyon", "to_node": "Fiber_Lyon_Marseille"},
    {"from_node": "Fiber_Lyon_Marseille", "to_node": "Amp_Marseille"},
    {"from_node": "Amp_Marseille", "to_node": "Marseille"}
  ]
}
```

5.7 5. Initial Spectrum Configuration

The `initial_spectrum.json` file defines the channel plan for transmission simulations.

5.7.1 5.1 Basic Spectrum Configuration

```
{
  "spectrum": [
    {
      "f_min": 191.3e12,
      "f_max": 195.1e12,
      "baud_rate": 32e9,
      "roll_off": 0.15,
      "delta_pdb": 0,
      "tx_osnr": 40,
      "slot_width": 50e9,
      "power": 0
    }
  ]
}
```

Parameters:

- `f_min`, `f_max`: Spectrum range (Hz)
- `baud_rate`: Symbol rate (baud)
- `roll_off`: Nyquist filter roll-off
- `delta_pdb`: Power pre-emphasis (dB)
- `tx_osnr`: Transmitter OSNR (dB)
- `slot_width`: Channel spacing (Hz)
- `power`: Launch power per channel (dBm)

5.7.2 5.2 Multi-Band Spectrum

```
{
  "spectrum": [
    {
      "f_min": 191.35e12,
      "f_max": 196.1e12,
      "baud_rate": 32e9,
```

```
      "slot_width": 50e9,
      "power": 0,
      "label": "C-band"
    },
    {
      "f_min": 186.0e12,
      "f_max": 190.9e12,
      "baud_rate": 32e9,
      "slot_width": 50e9,
      "power": 0,
      "label": "L-band"
    }
  ]
}
```

5.7.3 5.3 Power Pre-Emphasis

To compensate for gain tilt, apply pre-emphasis:

```
{
  "spectrum": [
    {
      "f_min": 191.35e12,
      "f_max": 193.1e12,
      "delta_pdb": 1.0,
      "label": "Low frequency channels"
    },
    {
      "f_min": 193.1e12,
      "f_max": 194.85e12,
      "delta_pdb": 0.0,
      "label": "Mid frequency channels"
    },
    {
      "f_min": 194.85e12,
      "f_max": 196.1e12,
      "delta_pdb": -1.0,
      "label": "High frequency channels"
    }
  ]
}
```

This creates a tilt: low frequencies at +1 dB, high frequencies at -1 dB.

5.8 6. Simulation Parameters

The `sim_params.json` file controls physical models used in simulation.

5.8.1 6.1 NLI Model Selection

```
{
  "nli_params": {
    "method": "gn_model_analytic",
    "dispersion_tolerance": 1,
    "phase_shift_tolerance": 0.1,
    "computed_channels": [0, 95]
  }
```

```
}
```

Available Methods:

- `"gn_model_analytic"`: Fast analytical GN model
- `"ggn_spectrally_separated"`: GGN model (more accurate)
- `"gn_model_numerical"`: Numerical integration

Tolerances:

- `dispersion_tolerance`: Integration step size control
- `phase_shift_tolerance`: Phase rotation threshold

5.8.2 6.2 Raman Amplification

```
{
  "raman_params": {
    "flag": true,
    "result_spatial_resolution": 10000,
    "solver_spatial_resolution": 50
  }
}
```

Parameters:

- `flag`: Enable Raman simulation
- `result_spatial_resolution`: Output resolution (points)
- `solver_spatial_resolution`: Solver step size (m)

5.8.3 6.3 Fiber Model Parameters

```
{
  "fiber_params": {
    "dispersion_slope": 0.058e-3,
    "raman_coefficient": {
      "frequency_offset": [0, 13.2e12],
      "cr": [0, 0.7e-13]
    }
  }
}
```

5.8.4 6.4 Complete sim_params.json Example

```
{
  "nli_params": {
    "method": "gn_model_analytic",
    "dispersion_tolerance": 1,
    "phase_shift_tolerance": 0.1
  },
  "raman_params": {
    "flag": false,
    "result_spatial_resolution": 10000,
    "solver_spatial_resolution": 50
  },
  "fiber_params": {
```

```
    "dispersion_slope": 0.058e-3
  }
}
```

5.9 7. Practical Examples

5.9.1 7.1 Simple Point-to-Point Link

Topology:

$$\texttt{Paris} \rightarrow \texttt{Fiber (80 km)} \rightarrow \texttt{EDFA} \rightarrow \texttt{Lyon}$$

Files Required:

1. `eqpt_config.json` (see Appendix)
2. `topology.json`:

```
{
  "elements": [
    {"uid": "Paris", "type": "Transceiver"},
    {
      "uid": "Fiber1",
      "type": "Fiber",
      "type_variety": "SSMF",
      "params": {"length": 80.0}
    },
    {
      "uid": "Amp1",
      "type": "Edfa",
      "type_variety": "std_medium_gain"
    },
    {"uid": "Lyon", "type": "Transceiver"}
  ],
  "connections": [
    {"from_node": "Paris", "to_node": "Fiber1"},
    {"from_node": "Fiber1", "to_node": "Amp1"},
    {"from_node": "Amp1", "to_node": "Lyon"}
  ]
}
```

Command:

```
gnpy-transmission-example \
  -e eqpt_config.json \
  topology.json
```

5.9.2 7.2 Multi-Span Link with ROADM

Topology:

$$\texttt{A} \rightarrow \texttt{Span1} \rightarrow \texttt{ROADM} \rightarrow \texttt{Span2} \rightarrow \texttt{B}$$

topology.json:

```
{
  "elements": [
    {"uid": "NodeA", "type": "Transceiver"},
    {"uid": "Fiber1", "type": "Fiber", "type_variety": "SSMF",
```

```
     "params": {"length": 80.0}},
    {"uid": "Amp1", "type": "Edfa", "type_variety": "std_medium_gain"},
    {"uid": "ROADM1", "type": "Roadm"},
    {"uid": "Amp2", "type": "Edfa", "type_variety": "std_medium_gain"},
    {"uid": "Fiber2", "type": "Fiber", "type_variety": "SSMF",
     "params": {"length": 100.0}},
    {"uid": "Amp3", "type": "Edfa", "type_variety": "std_medium_gain"},
    {"uid": "NodeB", "type": "Transceiver"}
  ],
  "connections": [
    {"from_node": "NodeA", "to_node": "Fiber1"},
    {"from_node": "Fiber1", "to_node": "Amp1"},
    {"from_node": "Amp1", "to_node": "ROADM1"},
    {"from_node": "ROADM1", "to_node": "Amp2"},
    {"from_node": "Amp2", "to_node": "Fiber2"},
    {"from_node": "Fiber2", "to_node": "Amp3"},
    {"from_node": "Amp3", "to_node": "NodeB"}
  ]
}
```

Command:

```
gnpy-transmission-example \
  -e eqpt_config.json \
  topology.json
```

5.9.3 7.3 Service Request Example

services.json:

```
{
  "path-request": [
    {
      "request-id": "request1",
      "source": "NodeA",
      "destination": "NodeB",
      "src-tp-id": "NodeA",
      "dst-tp-id": "NodeB",
      "bidirectional": false,
      "path-constraints": {
        "te-bandwidth": {
          "technology": "flexi-grid",
          "trx_type": "Voyager",
          "trx_mode": "DP-16QAM 400G",
          "spacing": 75000000000.0,
          "path_bandwidth": 400000000000.0
        }
      }
    }
  ]
}
```

Command:

```
gnpy-path-request \
  -e eqpt_config.json \
  topology.json services.json
```

5.9.4 7.4 Power Optimization

eqpt_config.json (Span section):

```
"Span": {
  "power_mode": true,
  "delta_power_range_db": [-2, 3, 0.5],
  "max_length": 150,
  "max_loss": 28
}
```

This sweeps launch power from -2 dB to +3 dB in 0.5 dB steps.

Command:

```
gnpy-transmission-example \
  -e eqpt_config.json \
  topology.json
```

GNPy will output optimal power levels for each span.

5.9.5 7.5 Multi-Band C+L Simulation

eqpt_config.json:

```
"Edfa": [
  {
    "type_variety": "std_low_gain_multiband",
    "type_def": "multi_band",
    "amplifiers": ["std_low_gain", "std_low_gain_L"]
  },
  {
    "type_variety": "std_low_gain",
    "f_min": 191.25e12,
    "f_max": 196.15e12,
    ...
  },
  {
    "type_variety": "std_low_gain_L",
    "f_min": 186.0e12,
    "f_max": 190.9e12,
    ...
  }
]
```

initial_spectrum.json:

```
{
  "spectrum": [
    {"f_min": 191.35e12, "f_max": 196.1e12, "label": "C-band"},
    {"f_min": 186.0e12, "f_max": 190.9e12, "label": "L-band"}
  ]
}
```

Command:

```
gnpy-transmission-example \
  -e eqpt_config.json \
  -s initial_spectrum.json \
  topology.json
```

5.10 8. Common Patterns and Best Practices

5.10.1 8.1 Equipment Library Organization

Recommended Structure:

```
equipment/
|-- eqpt_config.json          // Main equipment library
|-- advanced_configs/
|   |-- vendor1_booster.json
|   |-- vendor2_preamp.json
|   |-- ...
|-- measurement_data/
|   |-- vendor1_DFG_96.txt
|   |-- vendor1_NFR_96.txt
|   |-- ...
```

Best Practices:

- Use consistent naming: `vendor_model_type`
- Keep advanced configs separate from main file
- Document measurement conditions in config files
- Use version control for equipment updates

5.10.2 8.2 Amplifier Selection Guidelines

Use Case	Gain Range	Type	Example
Short spans (<50 km)	8-15 dB	Low gain	`std_low_gain`
Medium spans (50-80 km)	15-26 dB	Medium gain	`std_medium_gain`
Long spans (>80 km)	26-35 dB	High gain	`std_high_gain`
ROADM sites	Booster+Preamp	Dual stage	`hybrid_*`
High capacity	Multi-band	C+L	`*_multiband`

5.10.3 8.3 Span Design Rules

Maximum Span Loss:

$$L_{\text{max}} = G_{\text{flatmax}} - \text{margin}$$

Typical margins:

- Standard design: 3-5 dB
- High reliability: 8-10 dB
- Submarine: 1-2 dB

Launch Power Limits:

- Standard SSMF: 0 dBm/channel
- Long spans: +2 to +4 dBm/channel
- Consider nonlinear effects above 0 dBm

5.10.4 8.4 ROADM Cascade Limits

OSNR Budget:

Each ROADM adds:

- Insertion loss: 15-20 dB
- OSNR penalty: 0.3-0.5 dB (express)
- OSNR penalty: 1-2 dB (add/drop)

Maximum Cascaded ROADMs:

For 100G DP-QPSK (12 dB OSNR required):

- Standard ROADMs: 15-20 nodes
- High-performance ROADMs: 25-30 nodes

For 400G DP-16QAM (25 dB OSNR required):

- Standard ROADMs: 8-12 nodes
- High-performance ROADMs: 15-20 nodes

5.10.5 8.5 Troubleshooting Common Issues

Issue: Insufficient OSNR Symptoms:

```
OSNR = 10.5 dB < Required 12 dB
```

Solutions:

1. Increase launch power (if not at limit)
2. Use lower-NF amplifiers
3. Reduce span count (add regeneration)
4. Switch to higher OSNR tolerance mode

Issue: Excessive Nonlinear Penalties Symptoms:

```
NLI noise > -20 dBm/channel
```

Solutions:

1. Reduce launch power
2. Increase channel spacing
3. Use lower-order modulation
4. Check fiber effective area

Issue: Amplifier Saturation Symptoms:

```
Output power exceeds p_max
```

Solutions:

1. Reduce number of channels

2. Reduce per-channel power
3. Use higher `p_max` amplifier
4. Add output VOA

5.10.6 8.6 Performance Validation

Key Metrics to Check:

1. **OSNR Margin:**
$$\text{Margin} = \text{OSNR}_{\text{actual}} - \text{OSNR}_{\text{required}}$$
Target: > 2 dB
2. **Chromatic Dispersion:**
$$\text{CD}_{\text{total}} = \sum D_i \times L_i$$
Check against transceiver limits
3. **PDL Budget:**
$$\text{PDL}_{\text{total}} = \sqrt{\sum \text{PDL}_i^2}$$
Typical limit: < 2 dB
4. **Amplifier Gain:**
$$\text{gain\_min} < G < \text{gain\_flatmax}$$

5.10.7 8.7 Configuration Workflow

1. **Define Equipment Library**
   - Collect vendor datasheets
   - Create `eqpt_config.json`
   - Add advanced configs if available
2. **Build Topology**
   - Map physical network
   - Create `topology.json`
   - Verify connections
3. **Configure Simulation**
   - Set NLI model in `sim_params.json`
   - Define spectrum in `initial_spectrum.json`
   - Enable Raman if needed
4. **Run Initial Simulation**
```
gnpy-transmission-example -e eqpt_config.json topology.json
```
5. **Optimize**
   - Enable power mode

- Adjust amplifier types
- Fine-tune launch powers

6. **Validate Results**
   - Check OSNR margins
   - Verify amplifier operation
   - Compare with measurements

5.11 9. Advanced Topics

5.11.1 9.1 Custom EDFA Models

To create a custom EDFA model from measurements:

Required Measurements:

- Gain flatness across 96 channels (DFG_96.txt)
- NF vs. gain curve (fit to polynomial)
- NF ripple across channels (NFR_96.txt)
- Dynamic gain tilt (DGT_96.txt)

1. **Create pointer file (VendorX.json):**

```
{
  "nf_ripple": "vendorX_NFR_96.txt",
  "gain_ripple": "vendorX_DFG_96.txt",
  "dgt": "vendorX_DGT_96.txt",
  "nf_fit_coeff": "vendorX_NF_coeff.txt"
}
```

2. **Run build script:**

```
python build_oa_json.py VendorX.json
```

3. **Add to equipment config:**

```
{
  "type_variety": "VendorX_BoosterHG",
  "type_def": "advanced_model",
  "gain_flatmax": 25,
  "gain_min": 15,
  "p_max": 21,
  "advanced_config_from_json": "default_edfa_config.json",
  "allowed_for_design": false
}
```

5.11.2 9.2 Hybrid Raman+EDFA Amplification

Configuration:

```
{
  "type_variety": "hybrid_4pumps_mediumgain",
  "type_def": "dual_stage",
```

```
  "raman": true,
  "gain_min": 25,
  "preamp_variety": "4pumps_raman",
  "booster_variety": "std_medium_gain",
  "allowed_for_design": true
}
```

Raman preamp:

```
{
  "type_variety": "4pumps_raman",
  "type_def": "fixed_gain",
  "gain_flatmax": 12,
  "gain_min": 12,
  "p_max": 21,
  "nf0": -1,  // Negative NF indicates Raman gain
  "allowed_for_design": false
}
```

Fiber must support Raman:

```
"RamanFiber": [
  {
    "type_variety": "SSMF",
    "dispersion": 1.67e-05,
    "effective_area": 83e-12,
    "pmd_coef": 1.265e-15
  }
]
```

5.11.3 9.3 OpenROADM Compliance

GNPy supports OpenROADM MSA ver. 4.0 and 5.0 models.

Ver 4.0 differences:

```
{
  "type_variety": "OpenROADM MSA ver. 4.0",
  "mode": [{
    "format": "100 Gbit/s, 31.57 Gbaud, DP-QPSK",
    "tx_osnr": 35,  // Ver 4.0: 35 dB
    ...
  }]
}
```

Ver 5.0 improvements:

```
{
  "type_variety": "OpenROADM MSA ver. 5.0",
  "mode": [{
    "format": "100 Gbit/s, 31.57 Gbaud, DP-QPSK",
    "tx_osnr": 36,  // Ver 5.0: 36 dB (improved)
    "chromatic_dispersion": 48e3,  // Ver 5.0: higher tolerance
    ...
  }]
}
```

ROADM differences:

```
// Ver 4.0:
{"add_drop_osnr": 30, ...}

// Ver 5.0:
{"add_drop_osnr": 33, ...}  // Improved by 3 dB
```

5.12 10. Quick Reference

5.12.1 File Extensions

Extension	Type	Tool
`.json`	JSON	Any text editor, VS Code
`.xls, .xlsx`	Excel	Excel, LibreOffice
`.txt`	Text data	For import into JSON

5.12.2 Command Examples

```
# Transmission example (full spectrum)
gnpy-transmission-example \
  -e eqpt_config.json \
  -s initial_spectrum.json \
  topology.json

# Path request (service-specific)
gnpy-path-request \
  -e eqpt_config.json \
  topology.json services.json

# With simulation parameters
gnpy-transmission-example \
  -e eqpt_config.json \
  -sim sim_params.json \
  topology.json

# With extra equipment
gnpy-path-request \
  -e eqpt_config.json \
  --extra-eqpt extra_eqpt_config.json \
  topology.json services.json
```

5.12.3 Frequency Bands

Band	Frequency Range	Common Use
S-band	1460-1530 nm (206-196 THz)	Rarely used
C-band	1530-1565 nm (196-191 THz)	Standard WDM
L-band	1565-1625 nm (191-184 THz)	High capacity
U-band	1625-1675 nm (184-179 THz)	Research

ITU Grid (C-band):

- Center: 193.1 THz

- Standard spacing: 50, 100 GHz

5.12.4 Unit Conversions

```
Wavelength (nm) <-> Frequency (THz):
f[THz] = 299792.458 / lambda[nm]

Power:
P[dBm] = 10 x log(P[mW])
P[dBm] = 30 + 10 x log(P[W])

OSNR (0.1nm) <-> OSNR (BW):
OSNR_BW = OSNR_0.1nm - 10xlog(BW[GHz]/12.5)

Dispersion:
D[ps/(nm*km)] x L[km] = Total_CD[ps/nm]
```

5.12.5 Typical Parameter Ranges

Parameter	Typical Range	Units
Launch power	-3 to +3	dBm/channel
EDFA gain	8 to 35	dB
EDFA NF	4 to 11	dB
Fiber loss	0.18 to 0.22	dB/km
Span length	40 to 120	km
ROADM loss	15 to 20	dB
Required OSNR (100G)	10 to 13	dB
Required OSNR (400G)	20 to 25	dB

5.13 Appendix: Complete Example

File: complete_example_eqpt_config.json

```
{
  "Edfa": [
    {
      "type_variety": "std_medium_gain",
      "type_def": "variable_gain",
      "gain_flatmax": 26,
      "gain_min": 15,
      "p_max": 23,
      "nf_min": 6,
      "nf_max": 10,
      "out_voa_auto": false,
      "allowed_for_design": true
    }
  ],
  "Fiber": [
    {
      "type_variety": "SSMF",
      "dispersion": 1.67e-05,
      "effective_area": 83e-12,
```

```
      "pmd_coef": 1.265e-15
    }
  ],
  "Span": [
    {
      "power_mode": true,
      "delta_power_range_db": [-2, 3, 0.5],
      "max_length": 150,
      "length_units": "km",
      "max_loss": 28,
      "padding": 10,
      "EOL": 0
    }
  ],
  "Roadm": [
    {
      "target_pch_out_db": -20,
      "add_drop_osnr": 38,
      "pmd": 0,
      "pdl": 0
    }
  ],
  "SI": [
    {
      "f_min": 191.3e12,
      "f_max": 195.1e12,
      "baud_rate": 32e9,
      "spacing": 50e9,
      "power_dbm": 0,
      "roll_off": 0.15,
      "tx_osnr": 40
    }
  ],
  "Transceiver": [
    {
      "type_variety": "default",
      "frequency": {
        "min": 191.35e12,
        "max": 196.1e12
      },
      "mode": [
        {
          "format": "DP-QPSK 100G",
          "baud_rate": 32e9,
          "OSNR": 12,
          "bit_rate": 100e9,
          "roll_off": 0.15,
          "tx_osnr": 40,
          "min_spacing": 50e9,
          "cost": 1
        }
      ]
    }
  ]
}
```

field	type	description
`type_variety`	(string)	a unique name to ID the fiber in the JSON or Excel template topology input file
`dispersion`	(number)	In $s \times m^{-1} \times m^{-1}$.
`dispersion_slope`	(number)	In $s \times m^{-1} \times m^{-1} \times m^{-1}$
`dispersion_per_frequency`	(dict)	Dictionary of dispersion values evaluated at various frequencies, as follows: `{"value": [], "frequency": []}`. `value` in $s \times m^{-1} \times m^{-1}$ and `frequency` in Hz.
`effective_area`	(number)	Effective area of the fiber (not just the MFD circle). This is the A_{eff}, see e.g., the Corning whitepaper on MFD/EA. Specified in m^2.
`gamma`	(number)	Coefficient $\gamma = 2\pi \times n^2/(\lambda * A_{eff})$. If not provided, this will be derived from the `effective_area` A_{eff}. In $w^{-1} \times m^{-1}$. This quantity is evaluated at the reference frequency and it is scaled along frequency accordingly to the effective area scaling.
`pmd_coef`	(number)	Polarization mode dispersion (PMD) coefficient. In $s \times \sqrt{m}^{-1}$.
`lumped_losses`	(array)	Places along the fiber length with extra losses. Specified as a loss in dB at each relevant position (in km): `{"position": 10, "loss": 1.5}`
`raman_coefficient`	(dict)	The fundamental parameter that describes the regulation of the power transfer between channels during fiber propagation is the Raman gain coefficient (see [**?**] for further details); f_{ref} represents the pump reference frequency used for the Raman gain coefficient profile measurement ("reference_frequency"), Δf is the frequency shift between the pump and the specific Stokes wave, the Raman gain coefficient in terms of optical power g_0, expressed in $1/(m\ \ W)$. Default values measured for a SSMF are considered when not specified.

Table 93: Fiber parameters

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 central channel frequency.
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.

Table 111: Transceiver parameters

field	type	description
chromatic_dispersion or pdl or pmd	(number/string)	In ps/nm/. Value of chromatic dispersion. In dB. Value of polarization dependant loss. In ps. Value of polarization mode dispersion.
penalty_value	(number)	in dB. Penalty on the transceiver min OSNR corresponding to the impairment level

Table 113: Penalty parameters