1. Introduction

The CASYMIR package includes a standalone script and example YAML files that allow users to model two types of detectors: a direct-conversion detector used in a commercial mammography/DBT system, and a scintillator-based detector used in a commercial breast CT system. This script is intended for applications such as evaluating different spectra, system geometries, or exposure settings. These use cases are documented in the main README file and do not require modification of the underlying code.

For users who wish to simulate alternative detector architectures, implement new physical processes, or introduce custom branching pathways, the standalone script and predefined examples may not be sufficient. This manual provides the necessary information to:

- Use the built-in process blocks that represent individual gain and blurring stages;
- Construct parallel cascades using the parallel module;
- Assemble a complete model, illustrated through a direct-conversion DM/DBT detector example.

The theoretical background and validation results for CASYMIR are described in detail in the accompanying software article[1].

2. The *processes* module

The *processes.py* module defines the stages that make up the detector cascaded linear models. Each function corresponds to a physical process in the detection chain Signal, Detector, and Spectrum objects as arguments.

These blocks can be combined to form a serial cascaded model, or used as components within more complex parallel-path architectures. All built-in processes return an updated Signal object along with the corresponding gain and transfer function, unless otherwise noted.

Table 1 summarizes the process blocks included in CASYMIR. The underlying physical models, including the assumptions and equations used for each block, are described in Appendix A of the CASYMIR software article.

Table 1. CASYMIR built-in process blocks

Process **Arguments**	**Description**
initial_signal *Detector, Spectrum* *→ Signal*	Initializes a CASYMIR *Signal* object with spatial frequency information, and the magnitude equal to the fluence (photons/mm^2) reaching the detector. This function is to be called at the start of the model.
quantum_selection *Detector, Spectrum, Signal* *→ Signal, gain, transfer function*	Applies a stochastic gain to model photon absorption based on detector quantum efficiency.
absorption *Detector, Spectrum, Signal* *→ Signal, gain, transfer function.*	Models the conversion of x rays into secondary quanta (electron-hole pairs or optical photons for direct and indirect conversion detectors, respectively). Implemented as a Poisson gain stage.

local_k_absorption *Detector, Spectrum, Signal* *→ Signal, gain, transfer function.*	Models the local conversion of characteristic x rays into secondary quanta. Implemented as a Poisson gain stage.
remote_k_absorption *Detector, Spectrum, Signal* *→ Signal, gain, transfer function.*	Models the remote absorption and conversion of characteristic x rays. Implemented as a cascade consisting of a selection gain process followed by a stochastic blur process and a Poisson gain stage.
absorption_block *Detector, Spectrum, Signal* *→ Signal*	A parallel-cascade block implementing absorption, local_k_absorption, and remote_k_absorption. See Section 3 for details on the parallel path logic.
charge_trapping *Detector, Signal* *→ Signal, gain, transfer function.*	Models the spread due to charge trapping in direct conversion detectors. Implemented as a stochastic blur process.
optical_blur *Detector, Signal* *→ Signal, gain, transfer function.*	Models a stochastic blurring process caused by the spreading of optical photons in scintillator materials.
optical_coupling *Detector, Signal* *→ Signal, gain, transfer function.*	Models the coupling of optical photons to the photodiode of a detector. Implemented as a stochastic gain process with Bernoulli statistics.
q_integration *Detector, Signal* *→ Signal, gain, transfer function.*	Models pixel integration via a deterministic blur (pixel aperture) and applies gain based on the pixel's sensitive area.
noise_aliasing *Detector, Signal* *→ Signal*	Applies aliasing to the Wiener spectrum of the Signal object. The aliasing process is modeled by summing shifted copies of the Wiener spectrum at multiples of the sampling frequency.
model_output *Detector, Signal* *→ Signal with MTF and NNPS attributes*	Final block of the model. Adds electronic (white) noise to the Wiener spectrum and computes the model output (MTF and NNPS).

Note: Users wishing to define new physical processes must do so within the *processes.py* module. All process functions intended for use in parallel cascades must take Detector, Spectrum, and Signal objects as input and return a tuple containing:

- An updated Signal object,
- A float representing the mean gain factor,
- A NumPy array representing the corresponding transfer function.

3. The *parallel* module

This module defines the two core object classes used to create branching parallel paths within the models: Node and Path. The simple model shown in figure 1 will be used to illustrate the creation of parallel paths in CASYMIR.

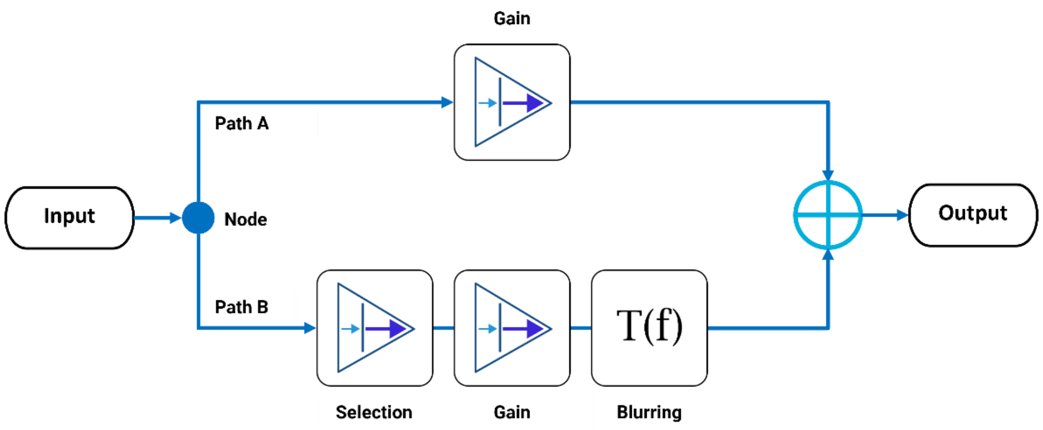

Figure 1. Schematic representation of a simple parallel cascade consisting of two paths separated by a quantum fork.

In this example, the input signal is split into two paths:

- Path A consists of a single gain process.
- Path B includes a sequence of processes: selection → gain → blurring.

A Node object represents the branching point and is defined by its type, either a Bernoulli split ("Bernoulli") or a quantum fork ("Fork"). The node type is specified at initialization:

```
Node = parallel.Node(node_type= “Fork” )
```

Each Path object is a list of process functions (e.g., gain, blurring) applied in sequence. **These functions must be defined in *processes.py* and follow the required input/output convention.** For the model in Figure 1, the paths are defined as follows:

```
Path_A = parallel.Path(processes = [gain])
Path_B = parallel.Path(processes = [selection, gain, blurring])
```

The *add()* method of the Node objects is used to attach branches and assign probabilities. Node and Path objects can be attached to other Nodes using the *add()* function:

```
Node.add(Path_A, probability=1)
Node.add(Path_B, probability=1)
```

Bernoulli Nodes support two mutually-exclusive paths with probabilities between 0 and 1. Quantum forks, by definition, require equal unity probabilities for all Paths.

Once the branching pathway architecture has been defined, the a*pply_parallel_process()* function is used to create the combined Signal:

```
combined_signal = parallel.apply_parallel_processes(Node, input_signal, detector, spectrum)
```

This function recursively traverses the defined structure, applies all processes, and combines the outputs by computing both individual and cross-spectral contributions. Equations governing this computation are presented in the software article (see Equations 1–3, 5, and 6).

4. Example: implementing characteristic x-ray production and reabsorption

The presence of a parallel block modeling characteristic x-ray production and reabsorption is a common feature of both direct and indirect detector models. The following code snippet illustrates how this process is implemented using CASYMIR's *parallel* module, following the architecture shown in Figure 2. The probability of k-fluorescence production is given by $\xi\omega$, where ξ and ω are the probability of a k-shell photoelectric interaction and the fluorescent yield, respectively.

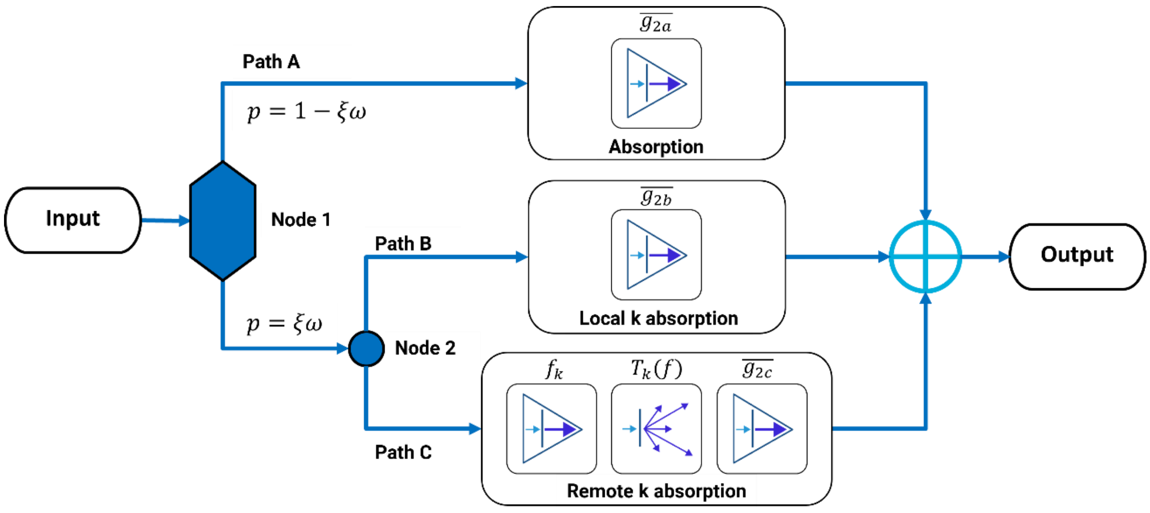

Figure 2. Schematic representation of a parallel cascade block that models the conversion of x-rays into secondary quanta considering K-fluorescence.

```
# Create Paths
PathA = parallel.Path(processes=[processes.absorption])
PathB = parallel.Path(processes=[processes.local_k_absorption])
PathC = parallel.Path(processes=[processes.remote_k_absorption])

# Create Nodes
# First split: Absorption vs. K-fluorescence production
Node1 = parallel.Node(node_type="Bernoulli")
# Second split: Local vs. Remote Fluorescence Absorption
Node2 = parallel.Node(node_type="Fork")

# Attach Paths to Nodes.
Node1.add(PathA, probability=1 - detector.material["xi"] * detector.material["omega"])
Node2.add(PathB, probability=1)
Node2.add(PathC, probability=1)

# Attach second Node to parent Node. The probability of the fork split is equal to xi*omega
Node1.add(Node2, probability=detector.material["xi"] * detector.material["omega"])

# Apply parallel process goes through all Nodes and Paths
output_signal = parallel.apply_parallel_process(Node1, input_signal, detector, spectrum)
```

This code defines the same logic used in the absorption_block() function found in the *processes* module. The output is a Signal object representing the combined effect of all three processes, including cross-spectral noise terms between Paths B and C.

5. Model example: direct-conversion detector of a DM/DBT system

This section provides an example implementation of a direct-conversion amorphous selenium (a-Se) detector used in a commercial DM/DBT system. The structure follows the parallel-cascaded architecture introduced by Zhao et al[2]. This configuration corresponds to the example DM/DBT system included in the CASYMIR repository and is the same model used for experimental validation in the CASYMIR software article.

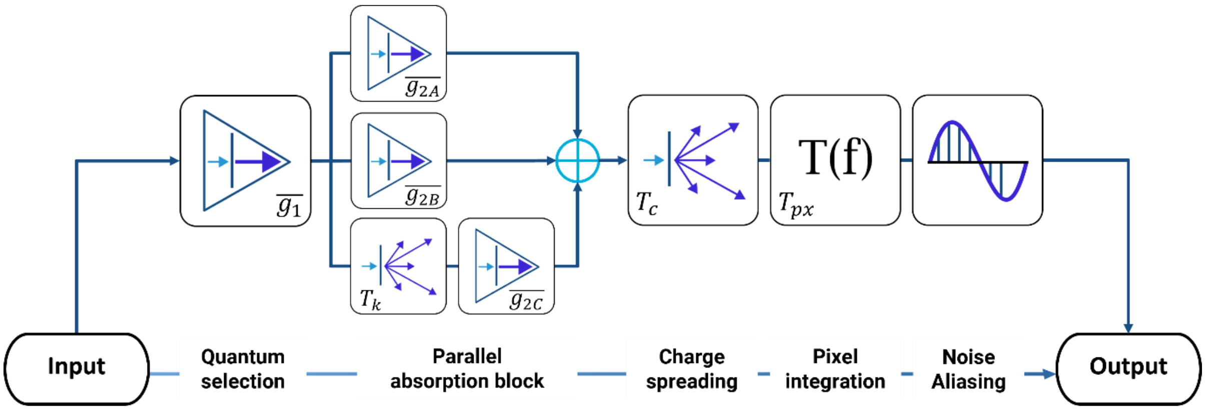

Figure 3. Block diagram showing the parallel-cascaded model of a direct-conversion a-Se detector used, for example, in a DM/DBT system.

In this example, we use a YAML file and a *System* object to set and load the system parameters. For more details on the structure and contents of the System YAML files, see the included "example_dbt.yaml" file and Appendix B of the CASYMIR software article.

```
from casymir import casymir, processes

system = casymir.System("example_dbt.yaml")
```

After creating a System object, the Tube, Detector and Spectrum objects are initialized. For this example, the potential of the x-ray tube is set to 28 kV, while the time-current product is set to 4.2 mAs.

```
det = casymir.Detector(sys.detector)

tube = casymir.Tube(sys.source)

spec = casymir.Spectrum(spec_name, kV=28, mAs=4.2, det, tube)
```

Next, a Signal object is initialized using the initial_signal process The processes shown in Figure 2 are then implemented using the functions from the *processes* module (see Table 1). The absorption_block process was defined as shown in Figure 2.

```
sig = processes.initial_signal(det, spec)

sig = processes.quantum_selection(det, spec, sig)

sig = processes.absorption_block(det, spec, sig)

sig = processes.charge_trapping(det, spec, sig)

sig = processes.q_integration(det,spec, sig)
sig = processes.noise_aliasing(det, sig)
sig = processes.model_output(det, sig)
```

The outputs of the model (frequency vector, MTF, and NNPS) can then be accessed from the Signal attributes and cast as a NumPy array for further processing:

```
output = np.array([sig.freq, sig.mtf, sig.nnps])
```

This concludes the example model. Additional configuration details and supporting code can be found in the CASYMIR repository.

References

1. Pacheco G, Pautasso JJ, Michielsen K, Sechopoulos I. Software Article: A generalized cascaded linear system model implementation for x-ray detectors. Med Phys. (Manuscript under revision) 2025.

2. Zhao W, Ji WG, Debrie A, Rowlands JA. Imaging performance of amorphous selenium based flat-panel detectors for digital mammography: Characterization of a small area prototype detector. Med Phys. 2003;30(2):254-263. doi:10.1118/1.1538233