



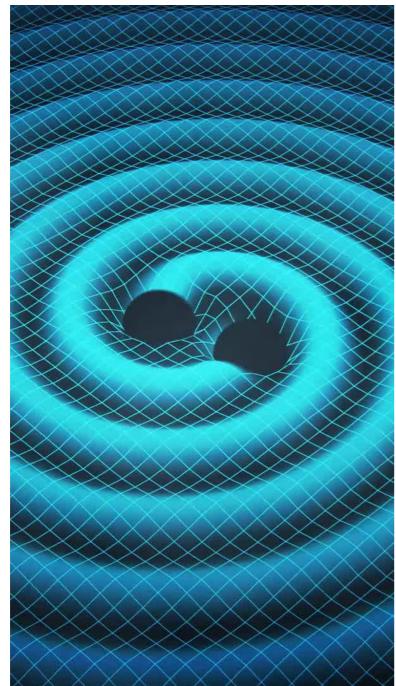
# *An End-to-End simulator design using graphs and nodes*

Marc Lilley (APC), JB Bayle (APC)



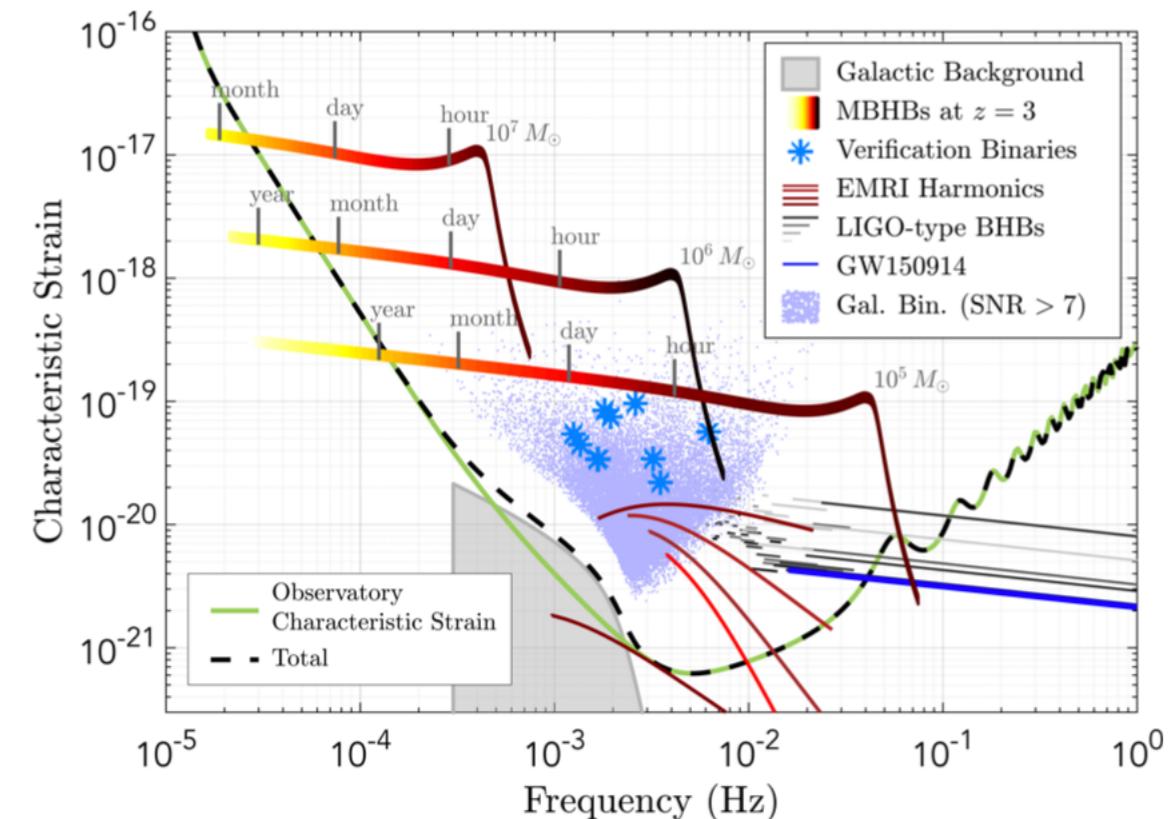
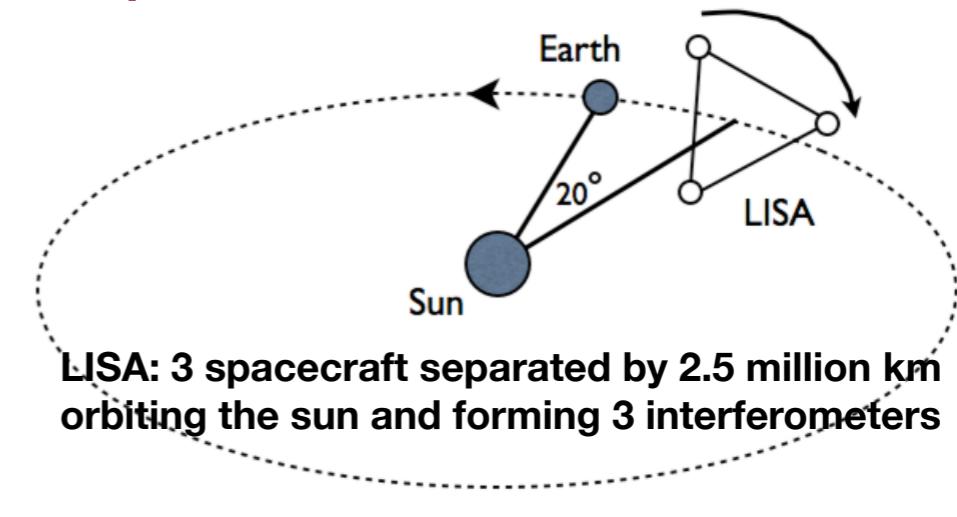
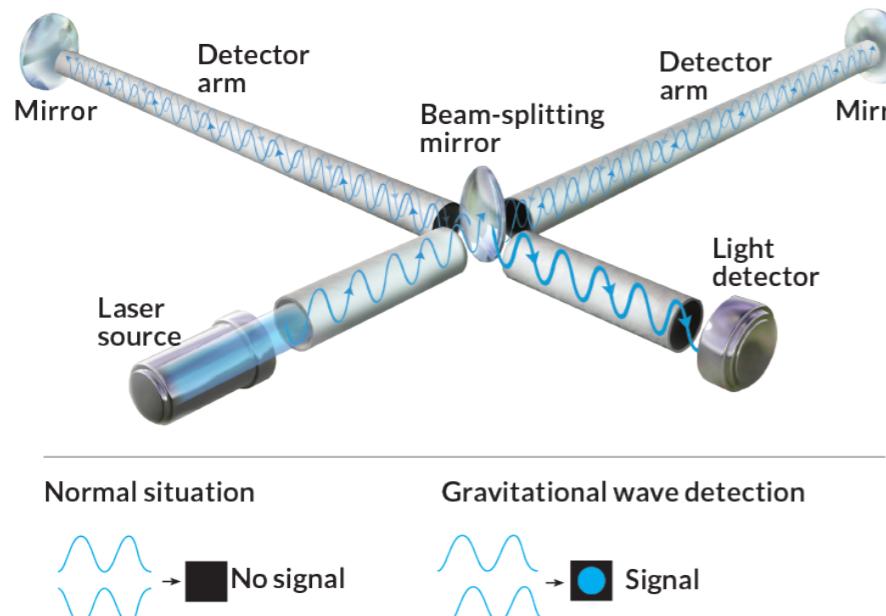
# What is LISA?

*LISA will measure gravitational waves emitted by astrophysical sources using laser interferometric techniques.*



**Massive object binaries emit spacetime distortions: gravitational waves (GWs).**

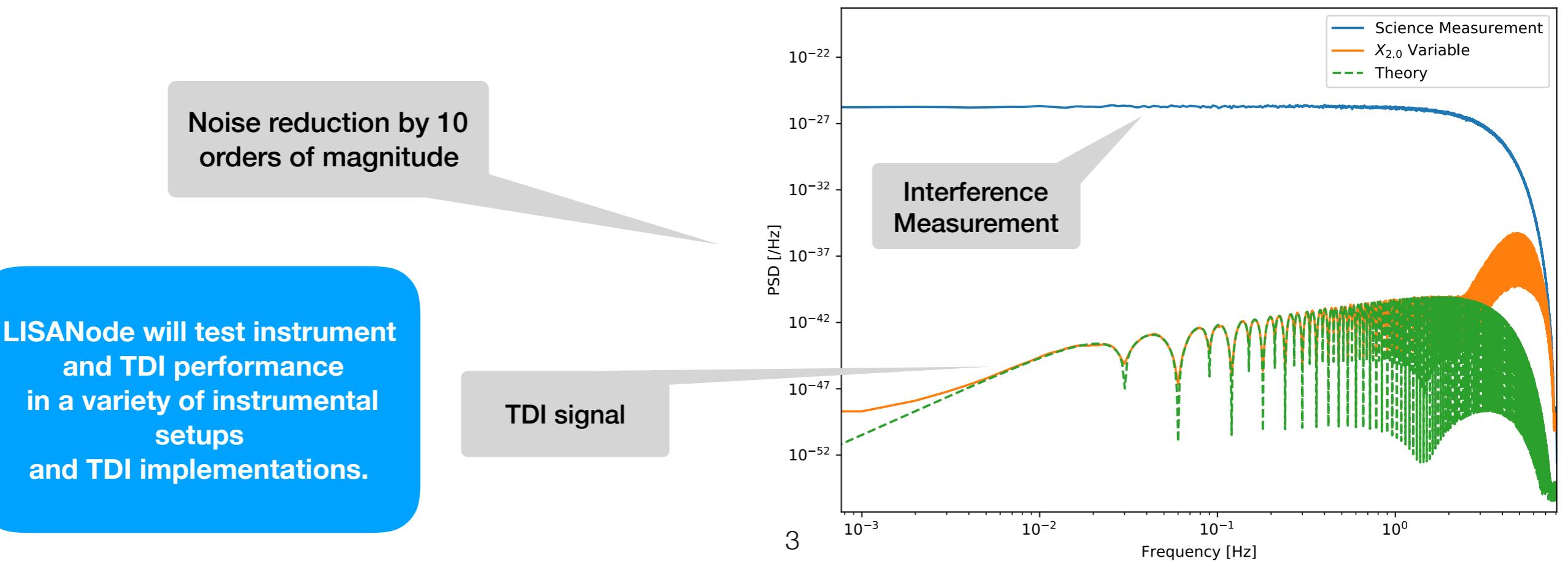
**They propagate at speed c & are of amplitude 1 picometer**



**Sensitivity curve and GW sources in frequency domain**

# Instrument requirements for GW detection

- GWs are tiny distortions of spacetime: 1 picometer.
- Instrumental design is challenging and critical for mission success (e.g. *LIGO*: *25 yrs of instrumental design upgrades before first detection*).
- In LISA, instrumental noise far outweighs the strength of GW signals.
- **Time delay interferometry (TDI)** is a complex mathematical strategy that reduces the noise levels by combining the interference measurements along arms 1, 2 and 3. The importance of testing its performance cannot be understated.



# What is LISANode?

*Prototype simulator for the LISA space experiment (launch 2034).*

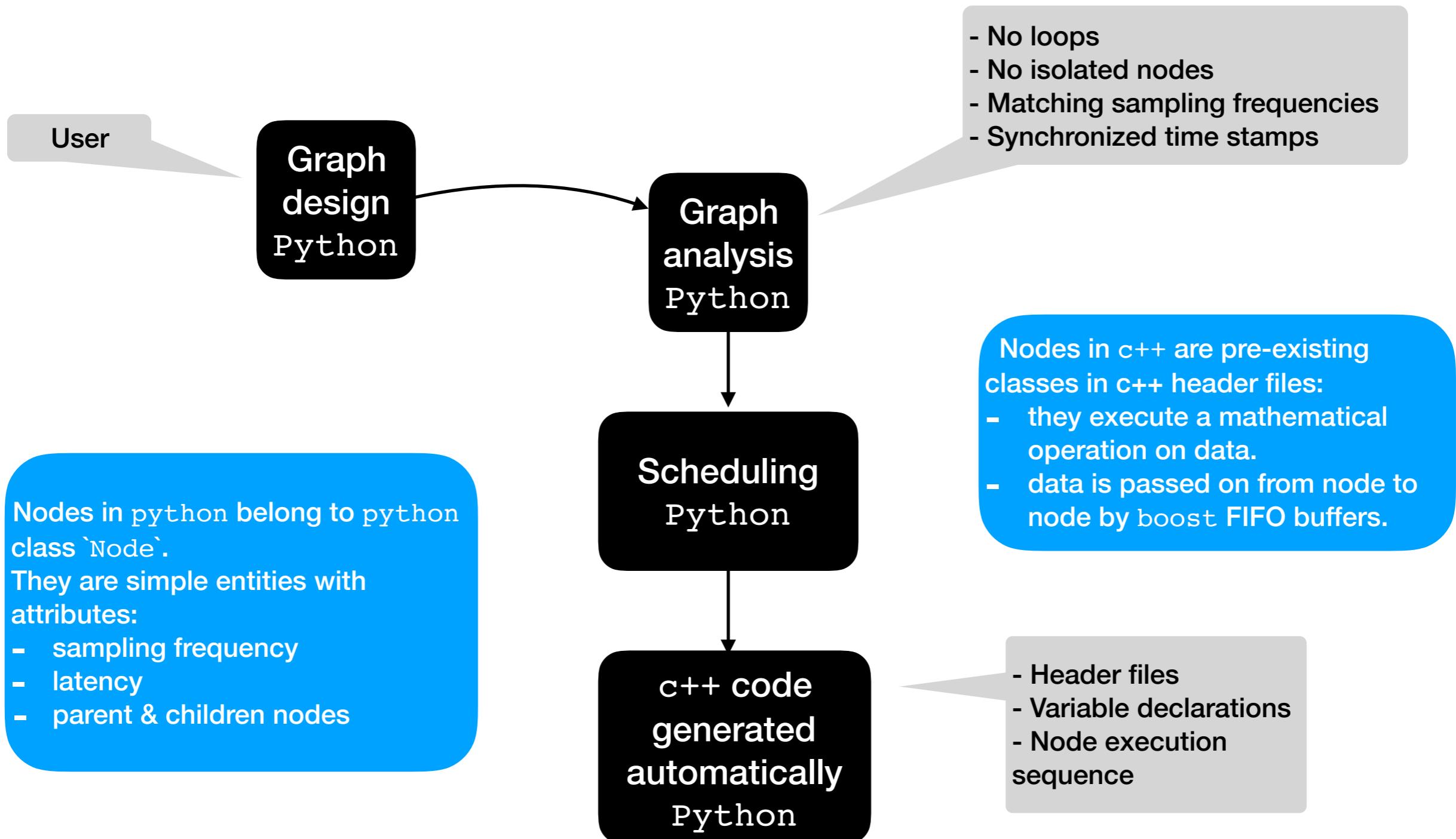
- Scientific goal:
  - Simulate the exchange of laser beams among three spacecraft in orbit.
  - Simulate interference measurements among each spacecraft.
  - Simulate multiple sources of noise.
  - Simulate pre-processing steps (TDI).
- Flexibility:
  - multiple/evolving configurations.
  - Various degrees of complexity.
- Performance requirements:
  - Parallelization.
  - Portability.

# Concept

**To respond to the above requirements:**

- Design the simulator as a graph connecting nodes.
- Each simple node performs a single operation.
- Simple nodes can be embedded into complex nodes.
- Data flows from top to bottom, being passed from node to node.
- Any choice of graph can be easily composed by respecting a minimal set of rules.

# Simulation design (python) and production (c++)

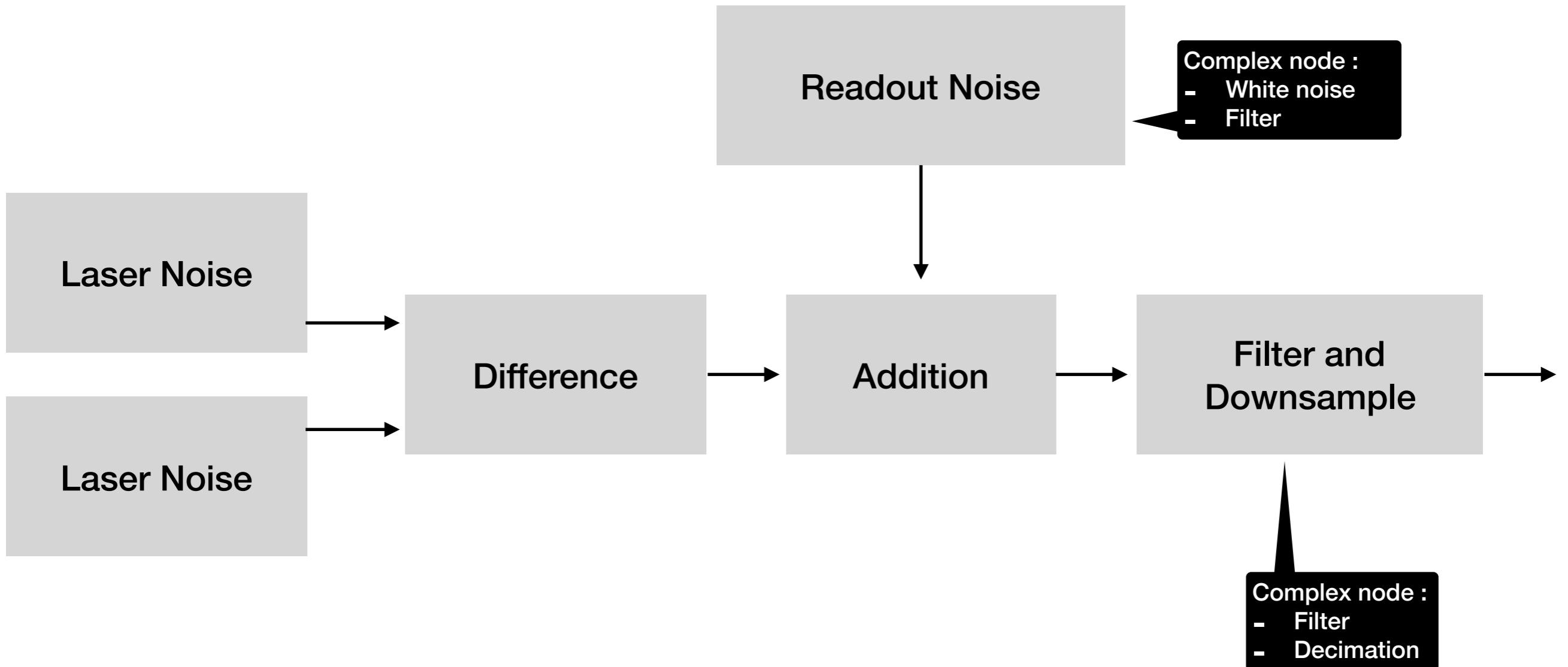


# Why this design?

- The machinery of graph design and analysis is easy to code in Python
- Users can easily design their graphs in Python
- Fast execution and portability of c++
- Object oriented languages are adapted to simulation designs in the form of graphs connecting nodes.

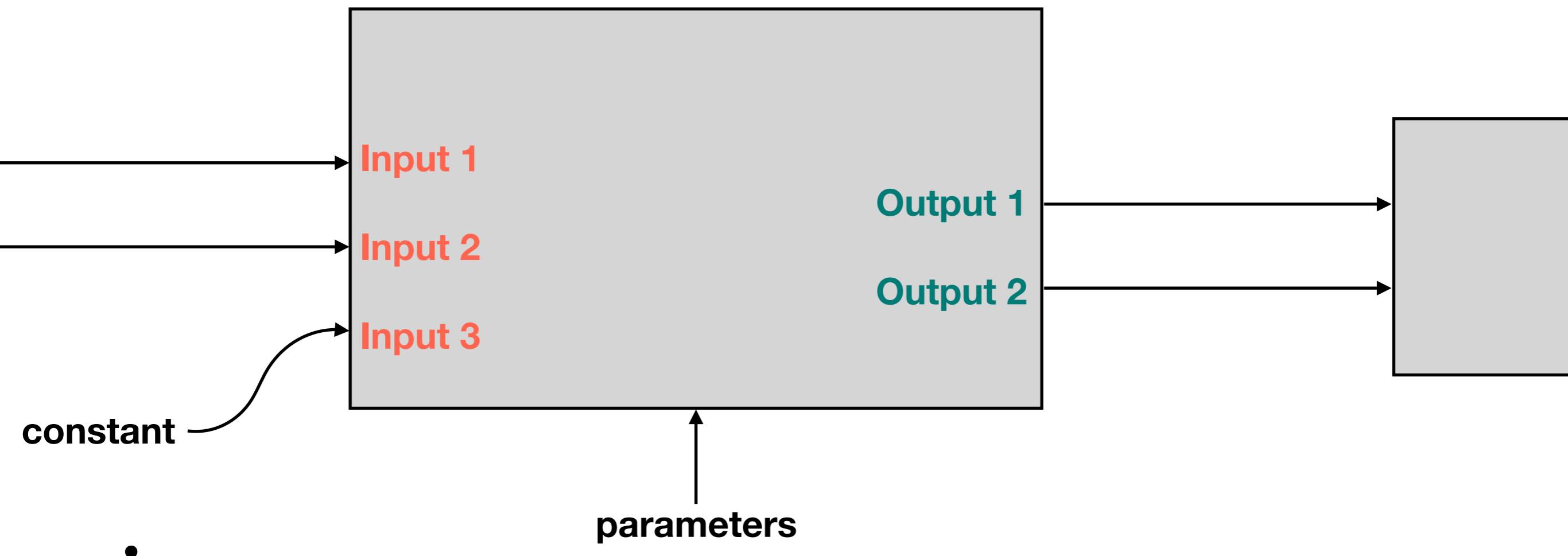
# A graph connecting nodes

This graph describes an interference between two lasers, the addition of instrumental noise, band-pass filtering and downsampling.

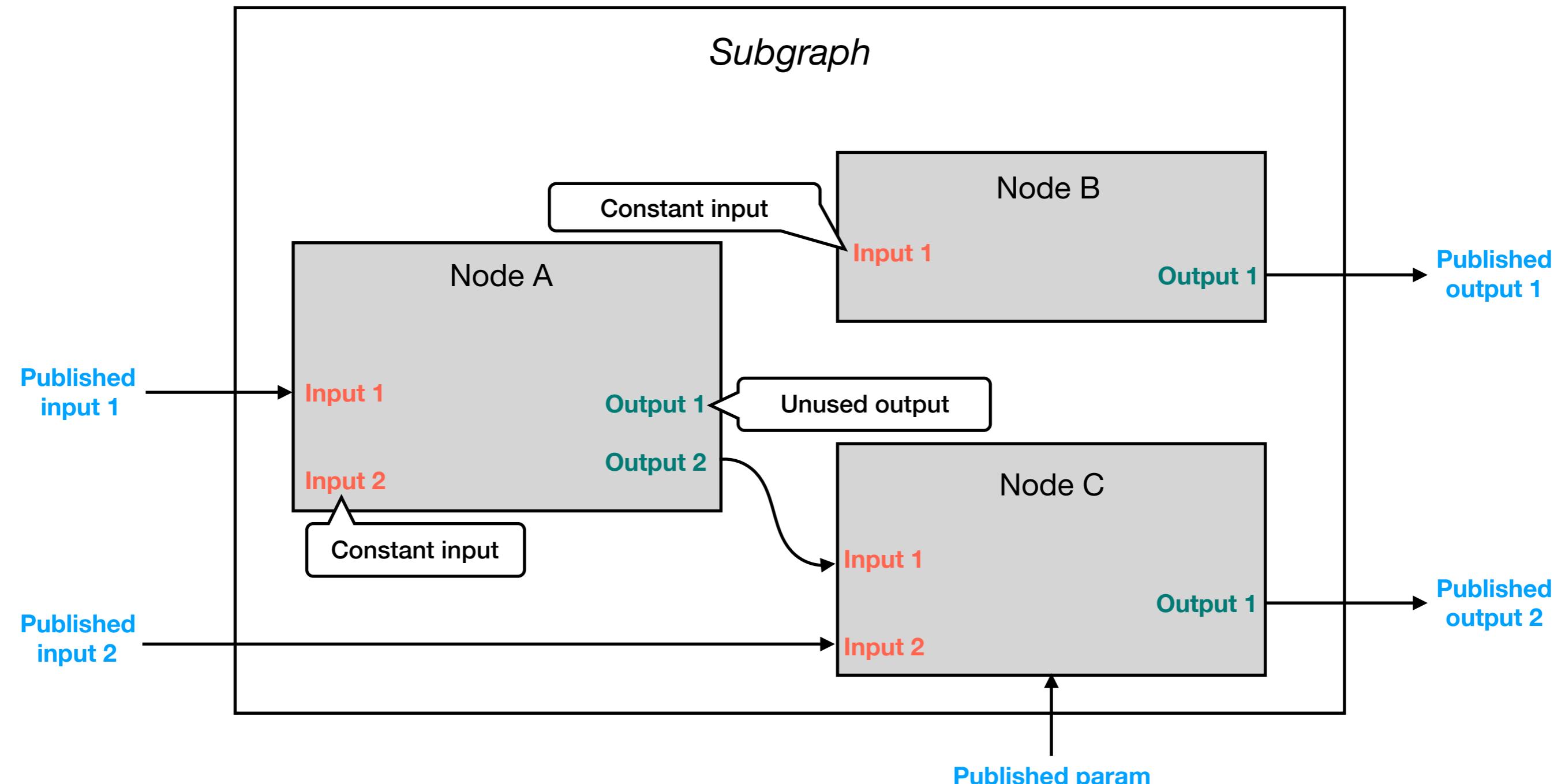


# A simple node

A simple node has inputs that can be connected to parent nodes or set constant, and also has parameters such as sampling frequency for example.



# A complex node



# What are simple nodes in c++?

## Basic building block

Node.hpp

```
#include <boost/circular_buffer.hpp>

class Node {

public:

    double latency;
    double time_origin;
    double sampling_frequency;

};
```

The instance variables  
for 'Node' are the  
same as in the Python  
class Node

Pre-existing c++ classes  
and Python are used to  
generate the c++  
simulation

## Nodes inherit 'node' properties

Product.hpp

```
using namespace boost;

template <typename T>
class Product: public Node {

public:

    // MARK: Inputs

    /** First input */
    circular_buffer<T> a;

    /** Second input */
    circular_buffer<T> b;

    // MARK: Outputs

    /** Product \f[a * b\f] */
    T result;

    void prepare() {
        a = circular_buffer<T>(1, T());
        b = circular_buffer<T>(1, T());
    }

    void fire(double time) {
        result = a[0] * b[0];
    }
}
```

Generic data type :  
One class template  
for all types!

Instance variables  
for  
'Product'  
in FIFO buffer.

Buffer size varies as a  
function of the  
associated node (e.g.  
interpolating node)

'prepare' sets  
up the node

'fire' executes  
the node

FIFO buffer  
for data flow

# Graph design in Python

## Example

```
from lisanode.compiler.graph import Graph

class Decimate(Graph):
    """ Test the decimation node """

    def __init__(self):
        super().__init__("Decimate")

        fs = 20.0
        ds = 2

        self.add("Sinus", "sine")
        self.nodes["sine"].fs = fs

        self.add("Decimation<double>", "decimate")
        self.nodes["decimate"].downsampling = ds

        self.connect("sine.result", "decimate.input")

        self.publish_output("sine.result", "sine.txt")
        self.publish_output("decimate.result", "decimated.txt")
```

connect  
adds a node  
to its parent's  
list of  
children...

publish\_outut  
inserts an output  
node

`Graph` is a Python class containing methods:

- add and connect nodes
- publish inputs and outputs
- graph analysis
- c++ code generation
- etc.

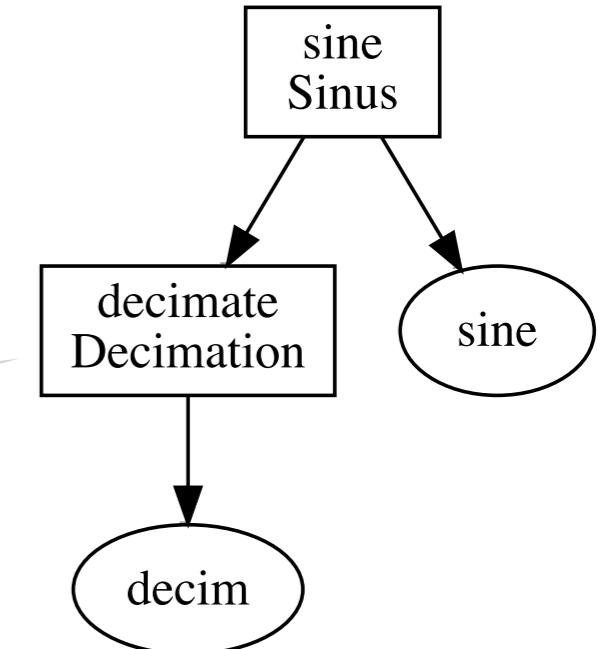
`Node` is a Python class defining the attributes of a node:

- parameters
- inputs
- parent and child nodes

Add a simple node to the graph

Nodes have parameters

Automated Graph visualisation using dot



# Graph analysis in Python & automatic c++ code generation

## • Graph analysis

- Check that all nodes are connected (no isolated nodes)
- Detect cycles
- Deduce sampling rate and latency for each node
- Compute execution schedule

## • C++ code generation

- Write down a set of rules that translate the graph and schedule into c++ code.

```
#include <type_traits>
#include "cxxopts.hpp"
#include "Decimation.hpp"
#include "Output.hpp"
#include "Sinus.hpp"

int main(int argc, char** argv) {
    // MARK: Node definitions
    Sinus _n;
    Decimation<double> _decimate;
    Output<std::remove_reference<decaytype(_n.result)>::type> _output_for_sine;
    Output<std::remove_reference<decaytype(_decimate.result)>::type> _output_for_decim;

    // MARK: Argument parsing
    cxxopts::Options parser = cxxopts::Options("Decimate");
    parser.add_options()
        ("h,help", "show this help message and exit")
        ("d,duration", "simulation duration in seconds", cxxopts::value<double>())
        ("time-origin", "origin of times in seconds", cxxopts::value<double>())
        ("precision", "set precision on output_for_sine, set precision on output_for_decim",
         cxxopts::value<decaytype(_output_for_sine.precision)>())
        ("print-time", "set print_time on output_for_sine, set print_time on output_for_decim",
         cxxopts::value<decaytype(_output_for_sine.print_time)>());
    ;

    auto args = parser.parse(argc, argv);
    if (args.count("help"))
    {
        std::cout << parser.help() << std::endl;
        exit(0);
    }
}
```

simple node  
C++ class

The scheduler  
determines which  
nodes are executed  
in each sequence

```
// MARK: Parameter assignment
_n.sampling_frequency = 20.0;
_n.time_origin = 0.0;
_n.latency = 0.0;

_decimate.sampling_frequency = 10.0;
_decimate.time_origin = 0.0;
_decimate.latency = 0.05;

_output_for_sine.sampling_frequency = 20.0;
_output_for_sine.time_origin = 0.0;
_output_for_sine.latency = 0.05;
_output_for_sine.path = "./sine.txt";

_output_for_decim.sampling_frequency = 10.0;
_output_for_decim.time_origin = 0.0;
_output_for_decim.latency = 0.15;
_output_for_decim.path = "./decim.txt";

if (args.count("precision")) {
    _output_for_sine.precision = args["precision"].as<decaytype(_output_for_sine.precision)>();
    _output_for_decim.precision = args["precision"].as<decaytype(_output_for_decim.precision)>();
}

if (args.count("print-time")) {
    _output_for_sine.print_time = args["print-time"].as<decaytype(_output_for_sine.print_time)>();
    _output_for_decim.print_time = args["print-time"].as<decaytype(_output_for_decim.print_time)>();
}

if (args.count("time-origin")) {
    double time_origin = args["time-origin"].as<double>();
    _n.time_origin = time_origin + 0.0;
    _decimate.time_origin = time_origin + 0.0;
    _output_for_sine.time_origin = time_origin + 0.0;
    _output_for_decim.time_origin = time_origin + 0.0;
}

// MARK: Node preparation
_n.prepare();
_decimate.prepare();
_output_for_sine.prepare();
_output_for_decim.prepare();

// MARK: Input assignment
// MARK: Simulation loop

double duration = 10.0;
if (args.count("duration")) { duration = args["duration"].as<double>(); }
int iloop = ceil(duration * 10.0);
for (int iloop = 0; iloop < iloop; iloop += 1) {

    // Sequence 1
    _n.fire(2 * iloop + 0);
    _decimate.fire(2 * iloop + 0);
    _output_for_sine.fire(2 * iloop + 0);
    _output_for_decim.fire(2 * iloop + 0);

    _decimate.input.push_front(_n.result);
    _output_for_sine.input.push_front(_n.result);
    _output_for_decim.input.push_front(_decimate.result);

    // Sequence 2
    _n.fire(2 * iloop + 1);
    _output_for_sine.fire(2 * iloop + 1);
    _decimate.input.push_front(_n.result);
}
```

Attribute inherited  
from node.hpp

The method  
'prepare' is used

The method  
'fire' is used

The FIFO in 'boost'  
is used for data flow  
from node to node

# Application to the LISA mission

- Number of inter-connected simple nodes: 680.
- `instrument.py`: 835 lines composing the graph describing the instrument.
- `preprocessing.py`: 300 lines composing the graph describing the data pre-processing pipeline.
- c++ code: 9,490 lines implementing the combination of the instrument and the preprocessing graphs.

# Summary & perspectives

- LISANode has a graph structure linking simple nodes.
- The graph is composed and analyzed in Python .
- The executable simulation is c++ code automatically generated by Python .
- This simulation framework is highly *flexible* and *modular*, and allows *users to compose their own graphs* for testing.
- Unit nodes, complex nodes, and a full simulation design are available to users.
- The scheduling scheme allows massive parallelization of node exec.
- Available on gitlab and installable using pip.
- Increase usage (advertising, documentation, user interface)
- Develop the existing simulation graphs / bring in contributions from LISA consortium labs
- Maintenance: evolving content, but similar structure.
- Parallelize
- Docker and Singularity deployment

La collaboration LISA est très dynamique, n'hésitez pas à venir nous parler pour nous rejoindre et travailler sur l'infrastructure informatique du projet dont la France a la responsabilité, ainsi que sur les divers aspects de simulation de la mission.