SiPM digitization software

Edoardo Proserpio and Romualdo Santoro - University of Insubria and INFN

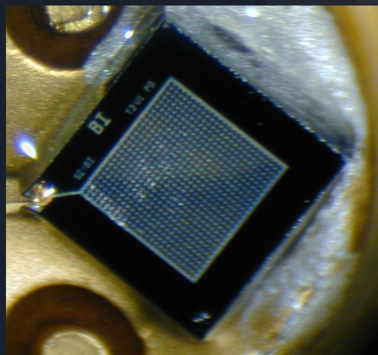
On behalf of IDEA Dual-Readout collaboration

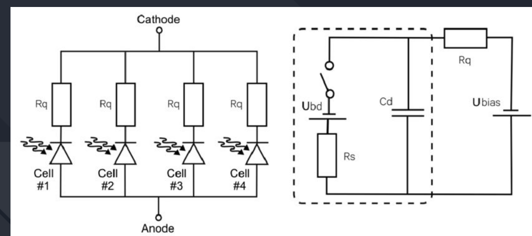
Idea behind

We want to develop a SiPM simulation that is:

- **Parameterized:** based on laboratory measured quantities or datasheet informations. No electronic circuit simulation!
- **General purpose** for all experiments
- With as **few dependencies** as possible
- Simple to use as **stand-alone** or in an **existing framework**

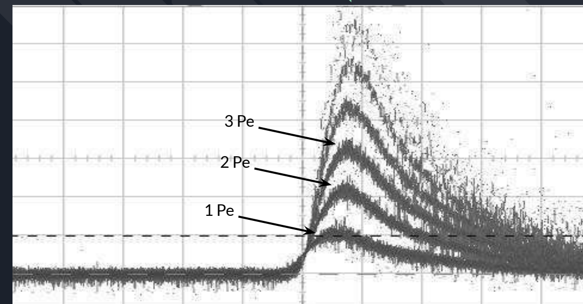
SiPM





Single-photon-sensitive device built as an array of **SPADs** implemented on a common substrate. Each SPAD is operated in **Geiger-Muller** mode and produces a discharge avalanche when detects a photon.

SPADs are connected in parallel and the output is the **sum of all SPAD** signals: it is expected to be **proportional** to the number of detected photons.



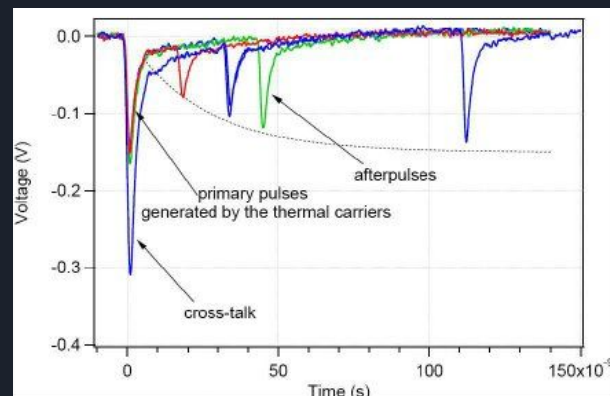
Effects considered

The simulation accounts for the following effects:

- **Dark Counts (DCR)**: thermally generated spurious avalanches
- **Optical Crosstalk (XT)**: photons generated in the avalanche can trigger adjacent cells.
- **Afterpulses (AP)**: delayed avalanches triggered by carriers trapped in silicon impurities

- **Non-linearity**: due to limited number of cells available
- **Cell recovery**: hitted cells recover as an RC circuit
- **Cell-to-cell gain variation**: small gain difference between cells

- Electronic noise

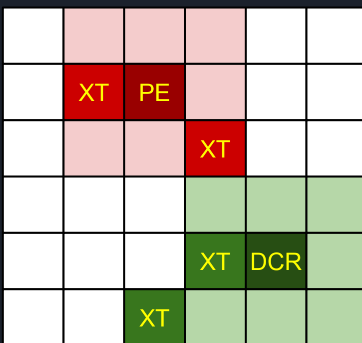


Stochastic noise

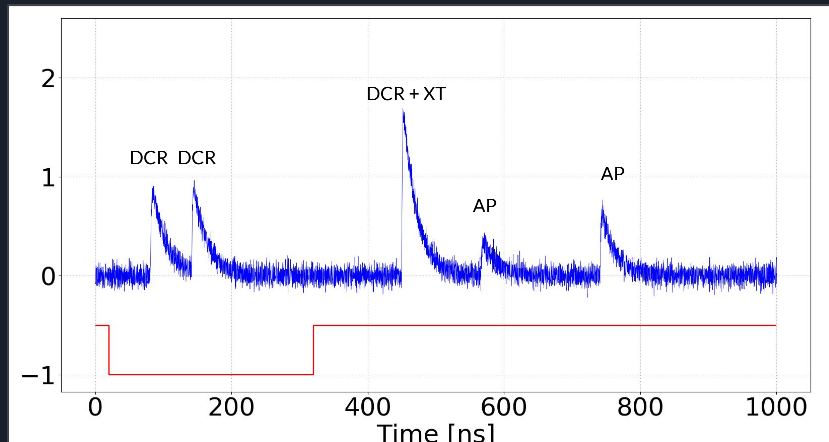
Dark counts, optical crosstalk and afterpulses are generated as **poisson processes**.

- Delay between DCR events is exponentially distributed
- XT is generated at the same time of the “main” event in one of the 8 adjacent cells
- AP is delayed from the “main” event with a double exponential distribution (fast/slow)

Any photoelectron can generate XT and/or AP (including DCR)



Representation of a SiPM matrix



Signal generation

The signal generated by an avalanche is modelled by:

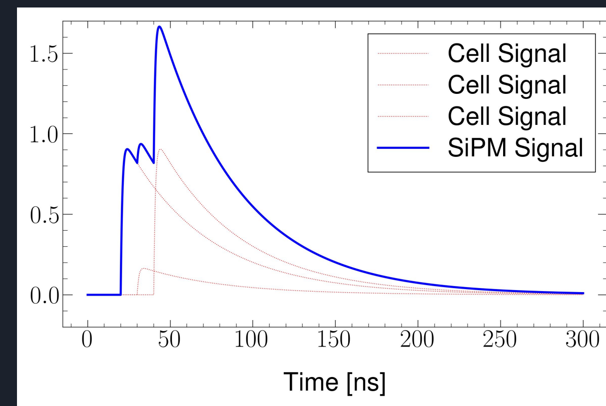
$$y(t) = a\left[e^{-(t-t_0)/\tau_f} - e^{-(t-t_0)/\tau_r}\right]\Theta(t-t_0)$$

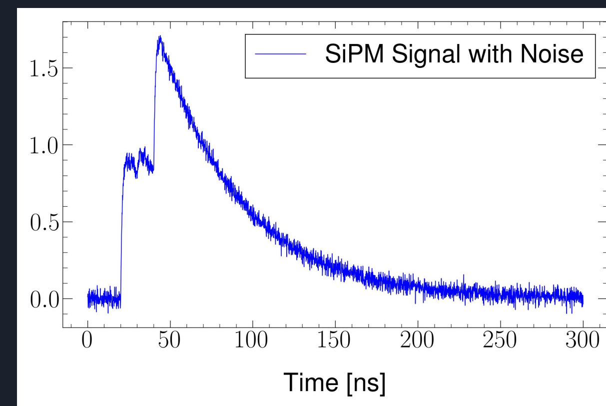
- t_0 Arriving time of the photoelectron
- a Amplitude of pulse
- τ_f Falling time of the signal
- τ_r Rising time of the signal
- Θ Heavyside function

DCR, XT and AP signals are modelled in the same way. The final signal is the **sum of all single cell pulses**.

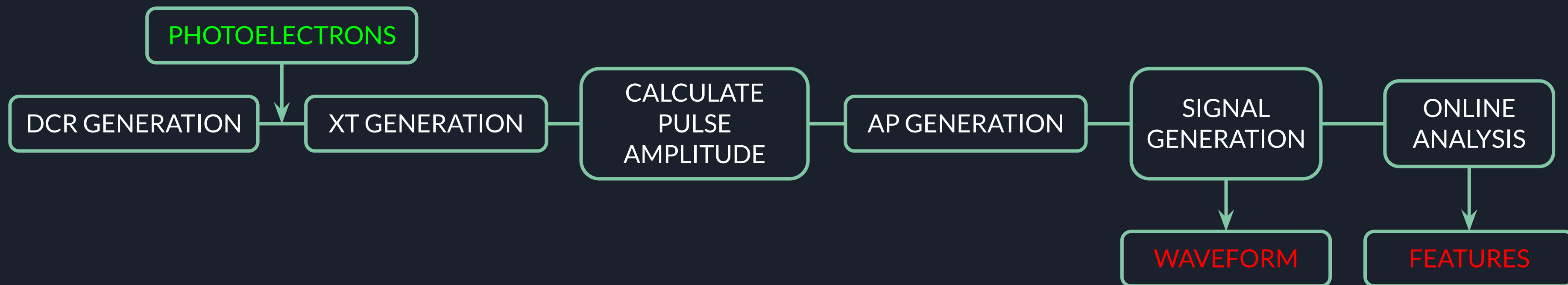
Electronic noise is added at the end with a given SNR.

C++ version has also a three exponential model with an additional slow component. $y(t) = a\left((1-\alpha)e^{-t/\tau_{ff}} + \alpha e^{-t/\tau_{fs}} - e^{-t/\tau_r}\right)$





Simulation chain



The **input** used to generate events is the **arrival time of photons** on the SiPM surface.

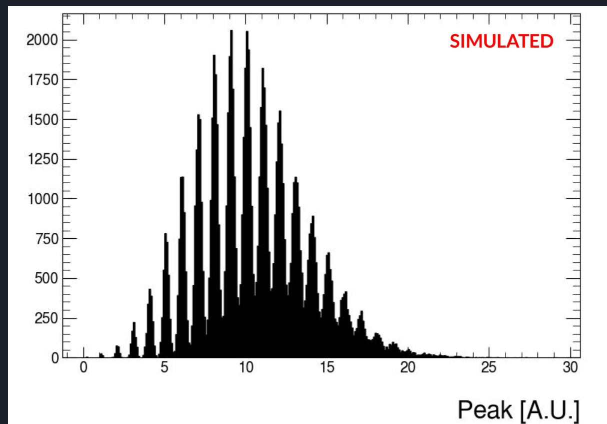
Photons are **uniformly distributed** on the sensor.

Photoelectrons statistic can be parameterized before the SiPM simulation. In this case the input is the arriving time of **photoelectrons** (like in Geant4 DR Calorimeter simulation)

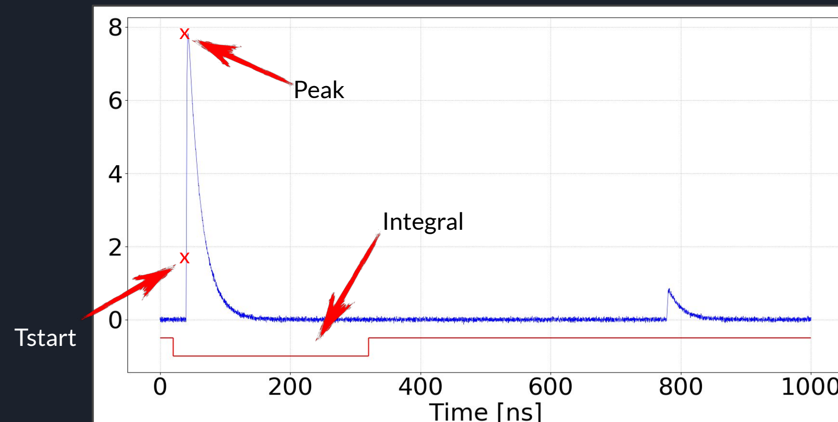
Output

The output can be either the signal or a list of signal features:

- **Peak**: max value of the signal
- **Integral**: sum of all samples
- **Tstart**: time of the first sample above threshold
- **Tot**: time over threshold
- **Top**: time position of the peak



Example of multiphoton peak spectrum



Features are calculated considering an user defined trigger and integration gate. Time units are calculated starting from the trigger

Validation

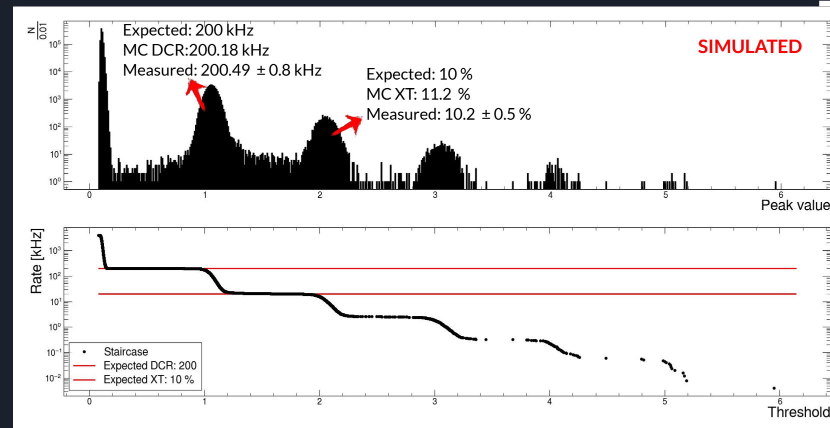
The simulation has been validated considering the **MC-truth** values saved during the generation and by performing a **“laboratory-like” measurement** on the produced data.

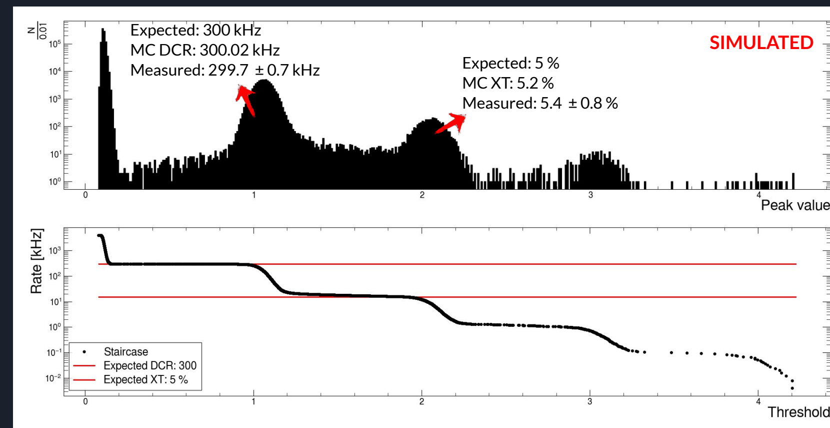
Stochastic noise

The **true number** of dark counts and optical crosstalk events generated is saved and their respective rates are calculated.

Staircase measurement is performed by counting the rate of events above a moving threshold. Direct measurement of DCR and XT.

Simulated data shows good agreement with the parameters set by the user.





Non-linearity

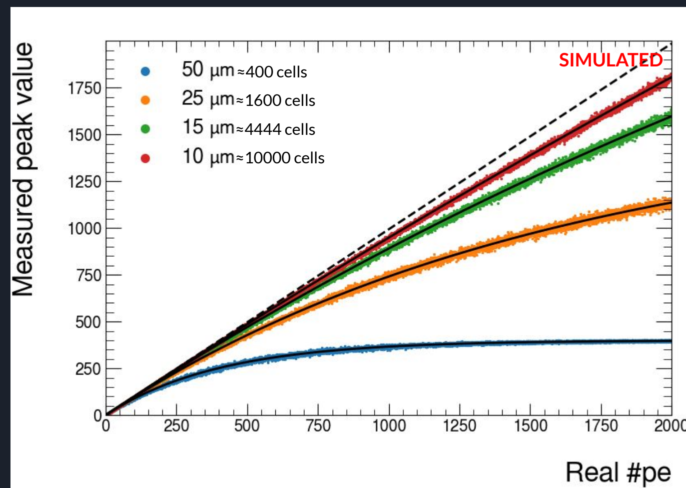
A sipm cannot measure more photoelectrons than the number of its cells. This causes **intrinsic non-linearity:**

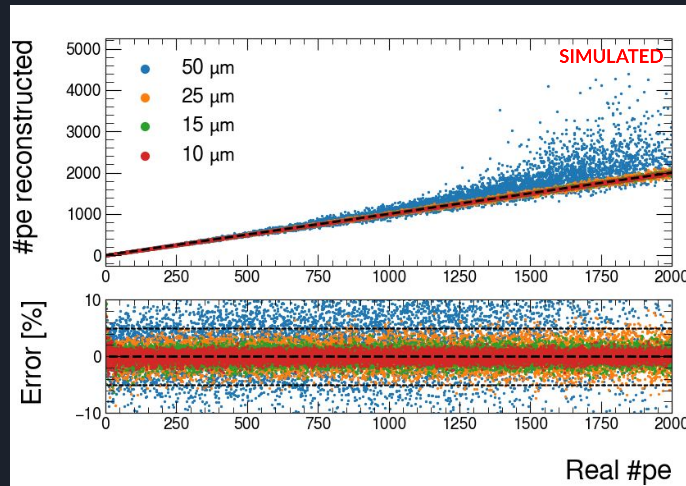
Intrinsic non-linearity of a sipm can be described by:

$$Signal \propto N_{cell}\left(1 - e^{-\frac{N_{pe}}{N_{cell}}}\right)$$

Corrections may be applied by inverting the formula

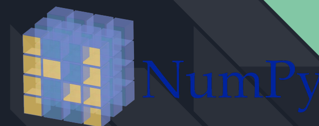
The simulation correctly reproduces the expected non-linearity effect considering different numbers of cells.





Performance

To speed up the python simulation **F2PY Fortran API** (included in Numpy) has been heavily used to compile the demanding parts of code.



C++ uses (where possible and useful) **AVX2 intrinsics** for vectorization.

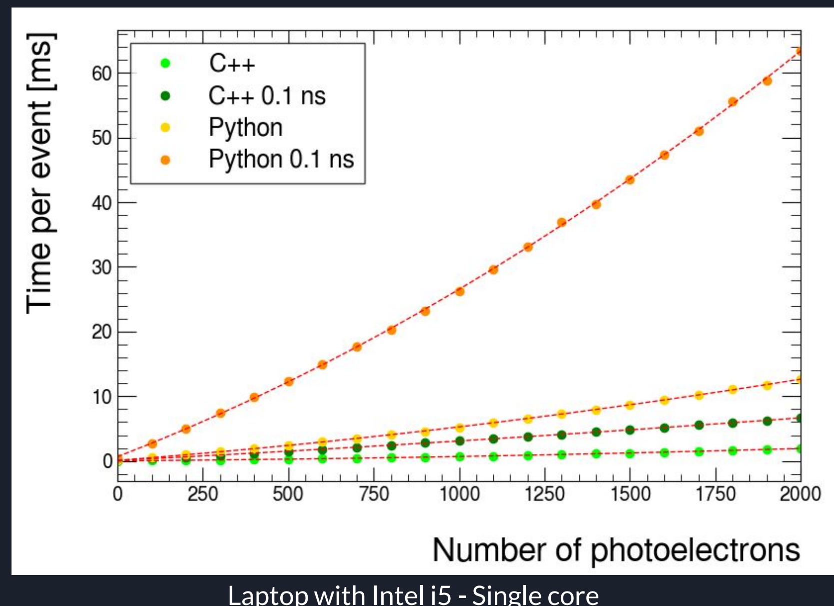
Speed

The two main parameters responsible for the computational time are:

- Number of **photons** in the event
- Number of points in the signal (**sampling**)

C++ version takes **<1ms (5ms)** to digitize a signal considering 1ns (0.1ns) of sampling and <1000 photoelectrons.

Python version is slower but still usable. In IDEA Dual Readout calorimeter it takes ≈ **1-2 s** to digitize a **full single-particle event**.



Laptop with Intel i5 - Single core

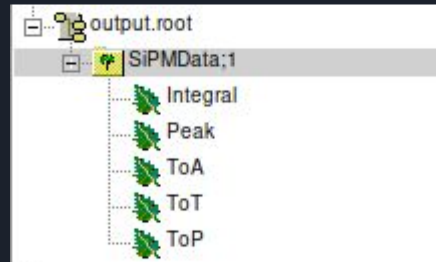
Output files

Python version offers functions to **save features** and/or **waveforms.**

Features are saved in a ROOT file (uproot). Resulting file size is very small and manageable.

Waveforms are saved in a compressed HDF5 file. Saving waveforms is a **slow process** and considering 0.1ns sampling it takes ≈ 11 Mb per 1000 signals (10^5 signals = 1Gb).

If all signals cannot be stored in memory waveforms have to be saved in batches. It is recommended to write your own code in this case.



30Mb per 10^6 signals

Example file: "wavedump.py" in dual-readout repo

User interface

Python version works as a **script** and relies on a user defined **configuration file**. Settings cannot be modified during runtime. It is up to the user to write a “wrapper” function to feed the main function of the simulation with data and to store results.

C++ version is **object oriented**, allowing an easier setup and the possibility to run multiple instances at the same time with different parameters.

Practical examples in backup slides

Work in progress

The simulation is being developed in Como and currently used in the Dual-Readout calorimeter software framework.

Python version is tested on: Python 3.6 - 3.8 - 3.9 (Linux and MacOS)
C++ version compiles with: GCC 10.2 and Clang 11.0 (Linux and MacOS)

Python version

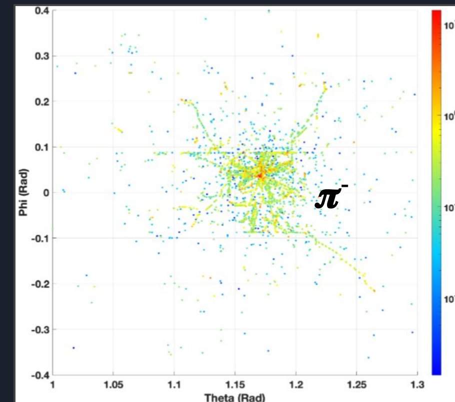
Python simulation is continuously being optimized.

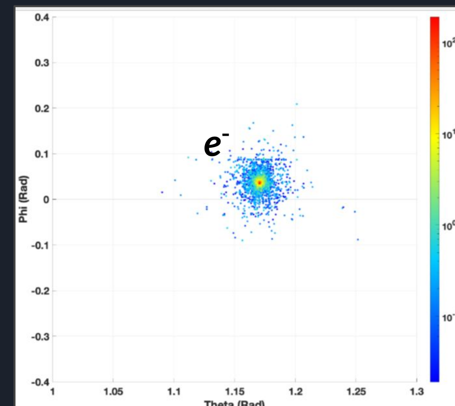
Being written as a script this version is **difficult to add new features** to.

The stable version of the simulation **is being used** in the IDEA Dual-Readout Calorimeter software chain to obtain informations at the SiPM level allowing timing studies.

Available at:
https://github.com/EdoPro98/pySiPM

Some documentation available at:
https://edopro98.github.io/pySiPM/docs/html/index.html





C++11 version of the simulation is under development and testing.

The code is being written following the FCCSW C++ rules and style guides.

Advantages of C++ version:

- Object Oriented
- Additional features
- Compatible with other C++ software as a shared library
- No dependencies
- Faster
- Easy to add new features

Still **missing documentation** and detailed **validation**

Available at:
https://github.com/EdoPro98/SimSiPM

Backup

Python version configuration

```
# Configuration file example
SIGLEN = 500
SAMPLING = 0.1
SIZE = 1
CELLSIZE = 25
DCR = 200e3
XT = 0.05
AP = 0.03

RISETIME = 1
FALLTIME = 50

INTSTART = 20
INTGATE = 300
```

Run with:

```
$ python script.py -f config.txt
```

Where “script.py” is a script that calls the simulation and “config.txt” a user defined configuration file

If the config file is missing default values will be used.

Python version wrapper

```python
# Script.py example (single event)
from sipm import SiPM

time = [10,20,30]
info = ("Event 1", "Sensor 0", "Endcap")

result = SiPM(time,info)

peak,integra,toa,tot,top = result[0]
info = result[1]
signal = result[2]
debug = result[3]
```

```python
# Script.py example (multiple events)
from sipm import SiPM, initializeRandomPool
from multiprocessing import Pool

times = [[1,2,3],[10,20,30,40]]
infos = [("Event 1","Sensor 0"),("Event 2","Sensor 0")]
# times is a list of lists with phe times
# infos is a list of tuples with event details

pool = Pool(4, initializer=initializeRandomPool)
# Run in batch
results = pool.starmap(SiPM,zip(times,infos))
pool.close();pool.join()

for res in results.get():
    peak = res[0][0]
    ...
```

Since the core of the simulation is a function it is possible to use **multiprocessing** module or even **pySPARK** to **scale up** the simulation on multiple cores/workers.

C++ version configuration

```
#include "SiPM.h"

SiPMProperties aProp;                // Create properties object
aProp.setSignalLength(500);          // It contains all the properties
aProp.setSampling(0.1);              // needed to describe a SiPM sensor
aProp.setSize(1);
aProp.setPitch(25);

SiPMSensor sipm0(aProp);             // Create two SiPM objects
SiPMSensor sipm1(aProp);             // with the same properties

sipm0.setProperty("Dcr",200e3);      // Tune sensor's parameters individually
sipm1.setProperty("Dcr",300e3);

sipm0.properties.setXtOff();         // Turn off XT on this one

sipm0.setPde(0.25);                  // Set PDE on this one
sipm1.properties().setPdeOff();      // PDE = 1 on this one
```

C++ version execution

```
...
sipm0.resetState();    // Reset internal state of sensor from
sipm1.resetState();    // previously stored events (in case of loop)

sipm0.addPhotons(t);   // t = std::vector<double> containing arriving time
sipm1.addPhotons(t);   // of photons/photoelectrons

sipm0.runEvent();     // Run the event
sipm1.runEvent();     // Run the event

SiPMAnalogSignal signal0 = sipm0.signal();        // Gather signal object

double peak0 = signal0.peak(20,300,1.5);          // Calculate peak
double peak1 = sipm1.signal().peak(20,300,1.5);  // Calculate peak directly

SiPMDebugInfo = sipm0.debugInfo();              // Gather MC-Truth info
```

C++ version can be easily **parallelized with OpenMP** but it requires instantiating a SiPMSensor object per each worker.