Ipanema v1.0 - Simulation Framework

Author: Gabriel Alejandro Fernández Fernández

A Coruña, August de 2025.

Abstract

High Energy physics is a branch of physics devoted to investigating the most fundamental constituents of matter as well as how they behave. Advancing in this field requires not only highly sophisticated experimentation methods but also specialized tools tailored to this purpose. Ipanema is defined as a statistical analysis framework used for this task. It is used by some of the research teams at *Conseil Européen pour la Recherche Nucléaire* (CERN).

Resumen

La física de partículas es una rama de la física que se dedica a investigar cuáles son los componentes más fundamentales de la materia y cómo se comportan. El hacer avances en este campo requiere de métodos de experimentación altamente sofisticados y de herramientas específicas preparadas para esta tarea. Ipanema forma parte de estas herramientas, siendo un sistema de análisis estadístico usado por equipos de investigación del *Conseil Européen pour la Recherche Nucléaire* (CERN).

Keywords:

- High Energy Physics
- Statistics
- Numerical Analysis
- Numerical Methods
- High Performance Computing
- Python

Palabras clave:

- Física de Partículas
- Estadística
- Análisis Numérico
- Métodos Numéricos
- Cómputo de Alto Rendimiento
- Python

Contents

Chapter 1

Description

This new version of Ipanema is designed to simplify the implementation process for any needed simulation. It separates the original workflow into a pipeline of simple plugins that users can modify. Ipanema handles the remaining steps automatically.

Any specific implementation will require at least three plugins:

1. **Input Plugin:** Prepares data for your model. All parameters required by the model must be stored in its dictionary.

2. **Model Plugin:** Model definition. FCN declaration, Minuit initialization, etc. Any process needed for a model fitting.

3. **Output Plugin(s):** Executes the model and processes the results for presentation. You may set multiple Output Plugins (e.g., one for printing results, another for plotting data, etc.).

The system contains two main modules inside `src`. `ipanema` (which is the core of the system) and `sdk` (which is a custom Software Development Kit designed for Ipanema).

1.1 Ipanema

This module contains 5 main packages:

1. **Config:** Inside this package resides `config.py`. This file is used to indicate to Ipanema which plugins should be executed. You can also specify any custom file paths used in your simulation.

2. **Core:** Contains the main pipeline Ipanema uses to dynamically load and execute plugins. It is unlikely that you will need to modify this package.

3. **Input:** Defines the interface which defines the required structure for Input Plugins. It also has a directory named `implementations/`. This directory contains a default and an example implementation of Input Plugins. You may use this directory to store your own Input Plugin implementations.

4. **Model:** Defines the interface which defines the required structure for Model Plugins. It also has a directory named `implementations/`. This directory contains a default and an example implementation of Model Plugins. You may use this directory to store your own Model Plugin implementations.

5. **Output:** Defines the interface which defines the required structure for Output Plugins. It also has a directory named `implementations/`. This directory contains a default implementation of an Output Plugin. You may use this directory to store your own Output Plugin implementations.

1.2 SDK

Its name stands for **Software Development Kit**. This module provides a set of support libraries users may use for their own implementations. Its present version has 2 main packages:

1. **CUDA Manager:** This package contains different implementations of a `CudaManager` designed for compiling and executing CUDA code in a simple and unified manner. It allows users to use High Performance Computing operations without having knowledge of any particular library. It also supports reduction operations over arrays, as well as element-wise operations. In this version, the implementations of this manager uses `PyCuda`.

2. **Math Utils:** This package is intended to contain different utilities involving mathematical operations users may need.

Chapter 2

The Plugins

This chapter explains how to properly implement plugins.

Note that for any given plugin a naming convention is used. The file which contains the plugin must have a `snake_case` name (e.g., `example_name_plugin.py`) while the class implementing the plugin must have the same name in `PascalCase` (e.g., `ExampleNamePlugin`).

2.1 InputPlugin

User-defined plugins must implement the `InputPlugin` interface. The code shown below is a simplification of this plugin.

```
class InputPlugin():

    @staticmethod
    def get_params() -> dict:
        pass
```

Input Plugins implemented by users may have other methods but must provide their desired parameters in the dictionary returned by `get_params`.

2.2 ModelPlugin

User-defined plugins must implement the `ModelPlugin` interface. The code shown below is a simplification of this plugin.

```
class ModelPlugin():

    fit_manager: Minuit
    parameters: dict

    def __init__(self, params: dict) -> None:
```

```
            self._parameters = params

        def prepare_fit(self) -> None:
            pass
```

Model Plugins implemented by users may add logic inside `__init__` or have other methods, but must use `prepare_fit` to start the model preparation sequence.

2.3 OutputPlugin

User-defined plugins must implement the `OutputPlugin` interface. The code shown below is a simplification of this plugin.

```
    class OutputPlugin():

        def generate_results(self, model: ModelPlugin) -> None:
            pass
```

Model Plugins implemented by users may have other methods, but must use the function `generate_results` to start the sequence of actions leading to the execution of the model fit and results presentation.

Chapter 3

Example

Users who want to implement their own models might follow steps similar to the ones shown below:

1. Implement an Input Plugin. Declare your parameters and return them in a dictionary:

```
    # file: example_input.py
    class ExampleInput(InputPlugin):

        ...

        @staticmethod
        def get_params() -> dict:
            params: dict = {}

            param_1 = 1.0
            param_2 = "string"
            ...
            param_n = np.ndarray([1, 2, 3])

            params["param_1"] = param_1
            params["param_2"] = param_2
            ...
            params["param_n"] = param_n
            return params

        ...
```

2. Implement a Model Plugin. Initialize your model using the parameters previously declared:

```
# file: example_model.py
class ExampleModel(ModelPlugin):
```

```python

    ...

    def __init__(self, params):
        super().__init__(params)

    def prepare_fit(self) -> None:

        self.fit_manager = Minuit(
            self._generate_fcn(),
            a = 2.4,
            b = 1.3,
            c = 3
        )

        self.fit_manager.limits["a"] = (2., 3.)
        self.fit_manager.limits["b"] = (-1., 3.)
        self.fit_manager.limits["c"] = (-5, 15)

    def _generate_fcn(self):
        params = self.parameters
        param_1 = params["param_1"]
        param_n = params["param_n"]

        # Declaring FCN
        def fcn(a, b, c):
            result = a**2 + param_1
            result += b * param_n[1] / np.float(c)

            return result

        return fcn

    ...
```

3. Implement an Output Plugin. Execute your model and present the results:

```python
# file: example_output.py
class ExampleOutput(OutputPlugin):

    ...

    def generate_results(self, model: ModelPlugin) -> None:

        model.fit_manager.migrad()
        model.fit_manager.hesse()
```

```

        print(f"\nFit Manager Values:\n{model.fit_manager.values}\n")
        print(f"\nFit Manager Error:\n{model.fit_manager.errors}\n")

    ...
```

4. Modify `config.py` so that Ipanema uses your plugins. Note that Ipanema expects the plugin names without their `.py` file extension in its configuration file:

```
# file: config.py
CONFIG = {

    "custom_paths": ['if\\your\\plugins\\outside\\implementations\\directories'],

    "input": "example_input",

    "model": "example_model",

    "outputs": [
        "example_output"
    ],
}
```

5. Access the root directory of this project and run your simulation with the *Execution* command provided in chapter 8.

Chapter 4

Libraries

Ipanema's support libraries are in a module called `sdk` (Software Development Kit). These are its main components:

4.1 CUDA Manager

In this package the user will find implementations of the interface `CudaManager`. CUDA managers allow the user to register code fragments for its future execution:

```
def add_code_fragment(
    self,
    name: str,
    function: str | Path
) -> None:
    pass
```

Users may register code either by providing it as a string or by specifying the path to the source file of the file containing the source code. In both cases users must give a key name for the code fragment. You can safely add code fragments without worrying about duplicate`#include` statements.

Since there is a method for registering code user will also need a code deleting one:

```
def pop_code_fragment(self, name: str) -> str:
    pass
```

Users will retrieve the code fragment as a string based on the given key name.

The function `run_program(...)` allow users to execute kernels from their registered CUDA code:

```
def run_program(self,
    func_name: str,
    outputs_idx: list[int],
```

```
        outputs_details: dict[
            int, tuple[tuple[int, ...], Any]
        ],
        block: tuple[int, int, int],
        grid: tuple[int, int],
        *args
    ) -> list:
        pass
```

Users must provide the following parameters:

- `func_name`: name of the global kernel function to be executed.
- `outputs_idx`: list with the positions of the kernel outputs.
- `outputs_details`: dictionary where users indicate the `shape` and `dtype` of each output argument.
- `block`: CUDA block dimension.
- `grid`: CUDA grid dimension.
- `*args`: arguments for the CUDA kernel.

`single_operations(...)` method allows users to perform element-wise operations to arrays. Users must provide the name of the desired operation and its required arguments.

```
    def single_operation(self, func_name: str, *args) -> Any:
        pass
```

`reduction_operations(...)` function allows users to perform reduction operations over arrays. Users must provide the name of the desired reduction operation and the array to be reduced.

```
    def reduction_operation(self, op_name: str, array: Any) -> Any:
        pass
```

In the present version of the system, there are two possible implementations:

- `AutoCudaManager`: implementation of a CUDA manager based on PyCUDA. PyCUDA's automatic context management is used in this implementation.
- `InteractiveCudaManager`: implementation of a CUDA manager based on PyCUDA. This implementation allows users to select any of the available devices for context initialization.

4.2 Math Utils

In this package users will find various mathematical utilities prepared to facilitate model implementations.

In the present version there is one utility named `rotate`. It is used as an example of how to use a `CudaManager`. This strategy allows users to transform float32 vectors or matrices using a transformation matrix `'T'`.

Inside this package, users will find an interface defining the algorithm named `AbstractRotationAlgorit`

```
class AbstractRotationAlgorithm(ABC):

    @abstractmethod
    def transform_f32(
        self,
        in_matrix: np.ndarray,
        t_matrix: np.ndarray,
        n: int
    ) -> np.array[np.double]:
        pass
```

Users will also find the following implementation using a `CudaManager` (the CUDA kernel of the algorithm is in the file `'src/sdk/math_utils/rotate/_support_files/_impl_rotate.cu'`).

```
class RotationAlgorithm(AbstractRotationAlgorithm):

    cuda_manager: CudaManager

    def __init__(self):
        super().__init__()
        self.cuda_manager = AutoCudaManager()
        self.cuda_manager.add_code_fragment(
            "rotate",
            Path(
    r"src\sdk\math_utils\rotate\_support_files\_impl_rotate.cu"
            )
        )

    def transform_f32(
        self,
        in_matrix: np.ndarray,
        t_matrix: np.ndarray,
        n: int
    ) -> np.array[np.double]:
        transform_f32_out: list = self.cuda_manager.run_program(
            "transform_f32",
```

```
            [1],
            {1: [(len(in_matrix),), np.double]},
            (1, 1, 1),
            (int(n), 1, 1),
            in_matrix,
            np.empty_like(in_matrix),
            t_matrix,
            n
        )
        return transform_f32_out[0]
```

Chapter 5

Complete Example

In this example, a signal peak is fitted on top of an exponential background, using an unbinned maximum likelihood fit. The signal probability density function (PDF) is implemented in a file called psIpatia.cu as a device function, `__device__ double log apIpatia`. A device function is only accessible by the GPU.

Firstly, an `InputPlugin` that processes the parameters is required:

```
class SignalPeakInput(InputPlugin):

    @staticmethod
    def get_params() -> dict:

        sd = "float64"
        dtype = getattr(np, sd)

        params: dict = {}

        with open(
            Path(
    r"src\ipanema\input\implementations\support_files\data_SnB.ext"
            ),
            "rb"
        ) as file:
            data = pickle.load(file, encoding="latin1")
        mydat = dtype(data[0])
        n_dat = len(mydat)
        massbins = dtype(data[1])
        d_m = dtype(massbins[1] - massbins[0])
        m_max = max(massbins)
        m_min = min(massbins)

        params["mydat"] = mydat
```

```
        params["n_dat"] = n_dat
        params["d_m"] = d_m
        params["m_max"] = m_max
        params["m_min"] = m_min
        params["massbins"] = massbins

        return params
```

Secondly, a `ModelPlugin` for the model definition is implemented:

```
class SignalPeakModel(ModelPlugin):

    _cuda_manager: CudaManager

    def __init__(self, params):
        super().__init__(params)
        self.cuda_manager = InteractiveCudaManager(None, False)

    def prepare_fit(self) -> None:
        n_dat = self.parameters["n_dat"]
        self.cuda_manager.add_code_fragment(
            "ipatia",
            Path(

    r"src\ipanema\model\implementations\_support_files\ipatia.cu"
            )
        )

        # Minuit Fit Manager Initialization
        self.fit_manager = Minuit(
            self._generate_fcn(),
            mu = 5365.,
            sigma = 7.,
            l = -3.,
            beta = 0.,
            a = 3.,
            n = 1,
            a2 = 6,
            Ns = 0.3*n_dat,
            Nb = 0.7*n_dat,
            n2 = 1,
            k = -0.05
        )

        self.fit_manager.limits["mu"] = (5360., 5370.)
        self.fit_manager.limits["sigma"] = (5., 9.)
        self.fit_manager.limits["l"] = (-5., -1.)
```

```python
        self.fit_manager.limits["beta"] = (-1e-3, 1e-3)
        self.fit_manager.limits["k"] = (-0.05, 0)
        self.fit_manager.limits["Ns"] = (0.1*n_dat, 1.1*n_dat)
        self.fit_manager.limits["Nb"] = (0.1*n_dat, 1.1*n_dat)

        self.fit_manager.fixed["a"] = True
        self.fit_manager.fixed["a2"] = True
        self.fit_manager.fixed["n"] = True
        self.fit_manager.fixed["n2"] = True

    def _generate_fcn(self):
        # Obtaining parameters
        params = self.parameters
        d_m = params["d_m"]
        m_max = params["m_max"]
        m_min = params["m_min"]
        mydat = params["mydat"]
        massbins = params["massbins"]
        n_dat = params["n_dat"]

        # Declaring FCN
        def fcn(mu, sigma, l, beta, a, n, a2, n2, k, Ns, Nb):
            # Calling ipatia for mass_bins
            grid_x = math.ceil(len(mydat) / 512)
            grid = (grid_x, 1)
            block = (512, 1, 1)
            ipatia_bins_out: list = self.cuda_manager.run_program(
                "Ipatia",
                [1],
                {1: [(len(massbins),), np.double]},
                block,
                grid,
                massbins,
                None,
                mu,
                sigma,
                l,
                beta,
                a,
                n,
                a2,
                n2,
                len(mydat)

            )
            integral_ipa = np.sum(ipatia_bins_out[0])*d_m
```

```
        if k!= 0 :
            integral_exp = (np.exp(k*m_max)-np.exp(k*m_min))*1./k
        else :
            integral_exp = (m_max - m_min)

        invint_b = 1./integral_exp
        invint_s = 1./integral_ipa
        Nexp = Ns+Nb
        fs = np.float64(Ns*1./Nexp)
        fb = np.float64(1.-fs)

        # Calling ipatia for my_dat
        ipatia_data_out: list = self.cuda_manager.run_program(
            "Ipatia",
            [1],
            {1: [(len(massbins)), np.double]},
            block,
            grid,
            mydat,
            None,
            mu,
            sigma,
            l,
            beta,
            a,
            n,
            a2,
            n2,
            len(mydat)
        )
        # Exponential background
        bkg_gpu = self.cuda_manager.single_operation("exp",k*mydat)
        term1 = bkg_gpu * invint_b * fb
        term2 = ipatia_data_out[0] * invint_s * fs
        sum_terms = term1 + term2
        # Calculate total likelihood
        LL_gpu = self.cuda_manager.single_operation(
            "log",
            sum_terms
        ) - Nexp
        extendLL =  n_dat*math.log(Nexp) -(Nexp)
        LL = np.float64(
            self.cuda_manager.reduction_operation("sum",LL_gpu)
```

```python
            ) + extendLL

            chi2 = -2*LL
            return chi2

        return fcn

    @property
    def cuda_manager(self) -> dict:
        """Getter for cuda_manager property."""
        return self._cuda_manager

    @cuda_manager.setter
    def cuda_manager(self, manager: CudaManager):
        """Setter for cuda_manager property."""
        self._cuda_manager = manager
```

Finally, an `OutputPlugin` for functions minimization, error calculation and results presentation is declared:

```python
class CommandLineOutput(OutputPlugin):

    def generate_results(self, model: ModelPlugin) -> None:

        model.fit_manager.migrad()
        model.fit_manager.hesse()

        print(f"\nFit Manager Values:\n{model.fit_manager.values}\n")
        print(f"\nFit Manager Error:\n{model.fit_manager.errors}\n")
```

5.1 Execution Times

This example was tested on the following three devices:

1. **MSI GP66 Leopard 10UG**, featuring an **Intel(R) Core(TM) i7-10870H**, **NVIDIA GeForce RTX 3070 Laptop GPU** and **16 GB** RAM

2. **HP Victus 15**, featuring an **Intel(R) Core(TM) i5-12450H**, **NVIDIA GeForce RTX 3050 Laptop GPU** and **16 GB** RAM

3. **OMEN Laptop 15**, featuring an **AMD Ryzen 7 4800H**, **NVIDIA GeForce GTX 1650 Ti Mobile** and **32 GB** RAM

The average model fitting times (in seconds) for each device are listed below:

Equipo	Windows	Ubuntu
1	12478	612
2	10398	457
3	-	672

Table 5.1: Average model fitting times (in seconds) for each device under Windows and Ubuntu.

Chapter 6

Errors

Ipanema defines its own set of exceptions. When something goes wrong, it will raise one of the following errors:

- `IpanemaImportError`: Raised when there are issues importing a module or resolving a class. Users should check files names and plugin paths.
- `IpanemaInitializationError`: Raised during the execution of an `InputPlugin`.
- `IpanemaFittingError`: Raised during the execution of a `ModelPlugin`.
- `IpanemaOutputError`: Raised during the exection of an `OutputPlugin`.

Chapter 7

Getting Started

Note that for this process it is assumed users have previously installed and updated their CUDA drivers. Users using Windows will also require Visual Studio Build Tools.

7.1 Clone the repository

```
git clone https://a-specific-url/Ipanema.git
```

7.2 Install Hatch

If your do not have hatch installed in your computer:

```
pip install hatch
```

7.3 Set up the environment

Use hatch to create and activate a development environment:

```
hatch shell
```

7.4 Modify the configuration

Adjust `config.py` to your necessities.

7.5 Run your simulation

Use the *Execution* command provided in chapter 8.

Chapter 8

Basic Commands

Execution:

```
hatch run python src/main.py
```

Testing:

```
hatch run pytest
```

Bibliography