Python fit with ceres

[C++ ceres 2](#_Toc16003099)

[D+ ceres 2](#_Toc16003100)

[Classes summary 3](#_Toc16003101)

[Python ceres 4](#_Toc16003102)

[wrap python function to C++ 4](#_Toc16003103)

[call\_obj module: 4](#_Toc16003104)

[C++ residual class: 5](#_Toc16003105)

[python residual class: 6](#_Toc16003106)

[summary 7](#_Toc16003107)

[Wrap C++ function to python 8](#_Toc16003108)

[ceres\_static.pxd 8](#_Toc16003109)

[cyres\_static.pyx 9](#_Toc16003110)

This document explains how the wrapping of ceres with python works.

I would start with explanation with how the C++ ceres works and then how python ceres works.

Remainer – fit runs generate multiple times in order to calculate params that give the most “close” curve to the a given curve.

# C++ ceres

The must important function of ceres is:

void Solve(const Solver::Options& options, Problem\* problem, Solver::Summary\* summary).

Options: The params that determine how the fit will run (example: optimization methods, stop conditions)

Problem: The problem that fit should solve, will contain - cost function, loss function, mute params, and mute parameters bound).

Summary: Ceres will fill this params with values, so that in the end of the run the user will be able to get all the needed data about the fit.

## D+ ceres

The function that calls ceres fit in D+ is PerformModelFitting in modelfitting.cpp .

The function creates CeresOptimizer.

CeresOptimizer: This class is d+ class. It initializes instance of problem and option.

The option is filled with the json from the frontend. The problem receives pointer to a function which run the calculations (generate). **The generate the cost function runs is the most important part.**

StructXRayResiduals/XRayRatioResiduals/XRayLogResiduals:

Those structs are the cost function ceres solver is going to run. Each one of them has:

* constructor which receives x, y (the wanted curve), numParams (number of mutable params), numResiduals( number of points in x), model (the model we want to run the generate on), bestParam (pointer to array with the params that receive the best cost value), bestEval (pointer to double with the best cost value).

***notice*** that bestParam and bestEval are pointers so when we change their values other variables that point them change (ceresOptimizer is also point there).

* GetCeresCostFunction static function which returns ceres:costFunction. This cost function receives the struct as param. **Please notice that because the structs implement () operator this is simulate sending the struct as function!**
* bool operator()(double const\* const\* p, double\* residual) const

this is the most key element. When you implement cost function to CERES it should had a function that receives p, and residual.

The operator runs CalculateVectorForCeres (generate). Then calculate the residuals cost. The calculation is the difference between the 3 structs (the generate is the same).

CalculateVectorForCeres –This function update the model with the new fit mutable params value and then run generate (CalculateIntensityVector).

## Classes summary

-----ceresOptimizer

----problem

---- costFunction (XRay residual structs)

----summary

# Python ceres

There are 2 main motivations to wrap ceres for python. The first one is that we want simple way to add residual cost calculation for fit. The second one is that we want a simple way to add calculation to the exists cost function (like add resolution function).

These 2 motivations mean that **we want to be able to create cost function classes in python and run it with the C++ code!**

In order to be able sending python class/ function to C++ code we need to do 3 things:

1. wrap python code so it will be callable from C++
2. wrap C++ code so it will be callable from python
3. convert C++ params to python params

## wrap python function to C++

### call\_obj module:

The base was taken from <https://stackoverflow.com/questions/39044063/pass-a-closure-from-cython-to-c>

We are aimed to pass callable python object to C++ code that receives std::function object (we should remember that ceres::problem receive costFunction – and this is the callable object we want to pass from python code to C++).

call\_obj.pyx:

Cython code (which will be compiled to C++). It has one function call\_obj.

public bool call\_obj(obj, const double\* x, double \* p, double\* residual, int numResiduals, int numParams)

|  |  |  |
| --- | --- | --- |
| Param name | Type | Description |
| obj | Python function | The callable object we want to run (the python function) |
| x | Const double \* | The x values (we want really use that because our residual class already contains it) |
| P | Double \* | The mutable params we will change in the model |
| Residual | Double \* | Pointer to array that will contain the residuals calculations.  **NOTICE:** this is empty we will fill that at the end of the function |
| numResidual | Int | The number of x point/ residual point we have. **Notice** that because the array arrives as double\* we need to know this value otherwise we won’t know how many items exist in the array |
| numParam | Int | The number of mutable params. **Notice** that because the array arrives as double\* we need to know this value otherwise we won’t know how many items exist in the array |

The function creates python lists with the arrays values and send them to obj item (python function). The obj return array of double and we set the new residuals values to the double \* residual array.

py\_obj\_wrapper.h

C++ code of PYObjWrapper:

We need to create C++ class that implement the operator () and inside this operator call Cython function (call\_obj) which receives python object and params and call the python object with the C++ params.

**bool operator**()(**const double**\* x, **double const**\* **const**\* p, **double**\* residual, **int** numResiduals, **int** numParams)

This class should mimic std::function (the cost function). Therefore, the signature of this function will be the signature of the std::function.

Cython can’t wrap double const\* const\* so we convert it to double\* , we can do it because the values won’t be change inside the python function (it const const).

### C++ residual class:

I created a c++ residual class.

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| Calc\_vector\_ | std::function<bool(const double\* , double const\* const\* ,double\* , int, int )> | This is the main function. Will run generate |
| x\_ | Const double \* |  |
| y\_ | Const double \* | The y curve to fit for |
| numResidual | Int | Number of points to calculate (length of x and y) |
| numParams\_ | int | Number of mutable params |
| pBestParam\_ | VectorXd | Array of the best mutable param values |
| pBestEval\_ | Double\* | The best cost value |
| bestEval\_ | Mutable double | The nest cost value |

|  |  |  |
| --- | --- | --- |
| Function name | params | description |
| Residual | X, y, numParam, calc\_vecor, bestParam, bestEval | Constractor of Residual |
| GetCeresCostFunction | X, y, numParam, calc\_vecor, bestParam, bestEval, eps, step\_size  Return -ceres::CostFunction | Static function which creates DynamicNumericDiffCostFunction with the residual class and option and return it as cosFunction. DynamicNumericDiffCostFunction is a costFunction class that receives classes that implement bool operator()(double const\* const\* p, double\* residual) const. |
| Operator () | P, residual  Return- bool | This function call calcVector\_ with p, residual and the class values: x\_, numResiduals\_ and numParams |

### python residual class:

I added new python module Residuals.py. It contains the following classes: XRayResiduals, XRayLogResiduals, XRayRatioResiduals.

The base class is XRayResiduals:

|  |  |  |
| --- | --- | --- |
| Property name | Type | description |
| calc\_runner | calculationRunner | The generate is running with it |
| calc\_input | CalculationInput | The generate will be running on calc\_input params. |
| np\_y | numpy array | Contains the y curve we want to do fit to. |
| best\_params | numpy array | Contains the mutable params that receive the best cost value.  **NOTICE:** when we update the values. We don’t change the array we change the values **inside** the array – because we want to mimic pointer behavior. |
| best\_eval | numpy array | array with one value. Contain the best cost value.  **NOTICE:** when we update the value. We don’t change the array we change the value **inside** the array – because we want to mimic pointer behavior. |
| save\_amp | Bool | Params to generate that determine if we want to save the amp grid (we will want that just at the end of all the fit calculations) |

|  |  |  |
| --- | --- | --- |
| Function name | params | description |
| run\_generate | Params, num\_residual | This function receives new mute param values and set them in the calc\_input instance. Then it runs the generate function and send the calc\_result to the function calc\_residual in order to get the cost value. It returns the calc residual values. |
| calc\_residual | Residual, num\_residual | This is the function that calculate the cost.  **NOTICE:** This function is the one who changes between the class. If user wants to add new cost function, he should inherit from XRayResiduals and override this function |

XRayLogResiduals and XRayRatioResiduals inherit from XRayResiduals and override the function calc\_residual.

### summary

we will send run\_generate from XRayResiduals as an std::function to Residual.

you will also notice that there are a lot of unneeded properties in Residual. They are there because we want to expose ceres not just for d+ and we will need them for that.

The flow will be:

create XRayResiduals , send XRayResiduals.run\_generate to PyObjWrapper. Send PyObjWrapper to Residual. GetCeresCostFunction. Residual. GetCeresCostFunction will return ceres::CostFunction and this we will send to ceres::problem.

## Wrap C++ function to python

We will use Cython to wrap the C++ code.

### ceres\_static.pxd

ceres\_static.pxd- declaration file (like h file). We announce about each c++ class/ property we will want to expose to python.

Two interesting wrapping:

1. *CostFunction \*GetCeresCostFunction(const double \*x, const double \*y,  
   int numParams, int numResiduals****, PyObjWrapper calcVector****, double stepSize,  
   double eps, vector[double] pBestParams, double \*pBestEval).*

*ceres::CostFunction \*GetCeresCostFunction(const double \*x, const double \*y,int numParams, int numResiduals,* ***std::function<bool(const double\*, double const\* const\*, double\*, int, int)> calcVector****, double stepSize ,double ep, std::vector<double> pBestParams ,double \*pBestEval)*

* You can see that we declare PyObjWrapper as param to GetCeresCostFunction. However, we should have declare **std::function –** we can do it because we tricked the Cython compiler- the signature of PyObjWrapper looks like **std::function** because we implement operator () that looks the same

*cdef extern from r"ceres/problem.h" namespace "ceres::Problem":  
  
 ctypedef struct ProblemOptions "ceres::Problem::Options":*

* We declare struct *ProblemOptions* although the class name is option. We did that in order to make a difference between the option class from the different namespaces.

**NOTICE:** we include only the .h file because we will compile Cython with ceres\_static.lib. if you don’t want to use lib you should include the cpp file too, **this is highly not recommended for ceres**, you can find example in CythonWrapping (wrapping grid class with Cython).

### cyres\_static.pyx

cyres\_static.pyx- somewhat similar to cpp file. We will create Cython class (cdef class) that contains properties from the C++ code.