Boost.Python

Programación Técnica y Científica

Iván Garzón Segura

Práxedes Martínez Moreno

Vladislav Nikolov

UNIVERSIDAD
DE GRANADA

Contents

Introducción

Como bien sabemos, Python es un lenguaje interpretado. Esto tiene ciertas ventajas e inconvenientes, siendo uno de dichos inconvenientes la velocidad a la hora de realizar grandes cantidades de cómputo, algo en lo que justamente destacan los lenguajes compilados.

El objetivo de este trabajo es presentar una herramienta que nos permite combinar ambos tipos de lenguajes con el fin de aprovechar las ventajas de cada uno de ellos. Esta herramienta es *Boost.Python*, la cual emplearemos para introducir en Python módulos que fueron previamente escritos y compilados en C++.

Para comprender el funcionamiento de *Boost.Python* se expondrán una serie de ejemplos de clases y funciones en C++ y, para comparar las diferencias entre usar esta herramienta y no hacerlo, recurriremos a un ejemplo más complejo basado en la *supresión de no máximos* usada en Visión Computador para extraer regiones relevantes de una imagen.

Boost.Python

Boost.Python forma parte de la biblioteca *Boost*, la cuál expande el funcionamiento de C++ ofreciendo funcionalidad de todo tipo. *Boost.Python* nos permite crear módulos en C++, sin usar otras herramientas que no sean un editor y su respectivo compilador, que podremos importar desde Python. En estos módulos creados podremos exponer clases, funciones y objetos que podrán ser utilizados posteriormente en cualquier *script* de Python en los cuales este módulo sea importado. Uno de los objetivos de esta biblioteca es ser mínimamente intrusivo en el diseño del código en C++.

Antes de continuar, vamos a ver cómo se instala y cómo se compilan los módulos que utilian esta herramienta. Posteriormente, vamos a ver una serie de ejemplos, y finalmente, vamos a ver el ejemplo comparativo de la *supresión de no máximos*.

Instalación

Boost.Python se puede instalar de manera separada al resto de elementos de *Boost*, ya que puede ser compilada aparte. Sin embargo, para no complicar el proceso de instalación, la forma más sencilla de obtenerla es instalar toda la biblioteca, la cuál normalmente puede ser encontrada entre los repositorios del gestor de paquetes. Para instalar *Boost*, simplemente basta ejecutar la siguiente orden:

```
$ sudo apt install libboost-all-dev
```

Adicionalmente, se necesitan tener instaladas las cabeceras de Python, las cuáles se encuentran en el directorio `/usr/include/pythonX`, donde X es la versión de Python que queremos utilizar. En nuestro caso, utilizaremos la versión 3.7, ya que es de las últimas versiones estables. En caso de no tenerlas instaladas, las podríamos obtener del gestor de paquetes de la siguiente forma:

```
$ sudo apt install python3.7-dev
```

Compilación

Compilar un programa que utiliza *Boost.Python* es relativamente sencillo. Para ello, tenemos que utilizar g++ con una serie de *switches*. Las opciones genéricas que utilizaremos o que tenemos que utilizar son las siguientes:

- `-shared`: Indica que se tiene que compilar el archivo como una biblioteca dinámica. Esta es la forma en la que funciona *Boost.Python*: crea bibliotecas dinámicas (archivos con extensión `.so` en Linux) que funcionan como módulos de Python. Son estas bibliotecas las que son importadas y utilizadas posteriormente.
- `-fPIC`: Esta opción indica que se tiene que generar código independiente de la posición (**Position Independent Code**). De esta forma, el código no depende de que esté localizado en alguna posición de memoria específica. Para crear una biblioteca dinámica, se tiene que especificar esta opción.
- `-I /usr/include/python3.7 -I /usr/include/boost`: De esta forma indicamos que hay dependencias externas, y que éstas deben ser buscadas en los dos directorios especificados anteriormente. Se indica que se tienen que incluir las cabeceras de Python para la versión 3.7 (la que estamos utilizando) y que se va a incluir *Boost*.
- `lboost_python3`: Indicamos que vamos a utilizar la versión de *Boost.Python* para Python3.

Adicionalmente, *Boost.Python* permite utilizar una versión muy simplificada de *numpy*. Si queremos compilar algún fuente que utilice *numpy*, solo tendríamos que añadir la línea `-lboost_numpy3`, la cuál indica que estamos utilizando la biblioteca dicha parte de *Boost.Python* para Python3.

Ejemplo de funciones

Para comenzar vamos a ver un ejemplo detallado con el fin de comprender cómo se exponen funciones de C++ a Python.

En primer lugar, empecemos con el en el código en C++. Tenemos que importar la biblioteca de *Boost.Python*, además de aquellas otras que vayamos a necesitar:

```
#include <boost/python.hpp>
#include <string>
```

A continuación procederíamos a implementar las distintas funciones que luego emplearemos en el *script* de Python. Un ejemplo de función podría ser el siguiente:

```
template <typename T>
T genericSum(T a, T b)
{
    return a + b;
}
```

Como podemos observar, esta función simplemente calcula la suma de dos valores de un tipo genérico y la retorna. Para poder usarla en Python, hemos de crear un módulo de *Boost.Python* y, posteriormente, exponerla en el mismo. Para hacer esto último, tenemos que indicar:

```
BOOST_PYTHON_MODULE(basic)
{
    using namespace boost::python;

    // Exponer función genérica
    def("generic_sum", genericSum<int>);
    def("generic_sum", genericSum<float>);
    def("generic_sum", genericSum<std::string>);
}
```

En la primera línea de este fragmento de código se declara el módulo de *Boost.Python* que se importará desde Python posteriormente con el nombre que queremos asignarle, que en este caso es *basic*. A continuación, dentro de dicho módulo definimos el namespace que usa *Boost.Python* y exponemos la función definida previamente haciendo uso de `def(<nombre de la función en Python>, <función en C++>)`. Como es una función genérica, hace falta indicarle al compilador cuáles serán los tipos para los que estará expuesta. Si no fuese una función genérica, no se tendría que indicar.

Una vez hecho esto ya podemos compilar el código cpp. Para ello, usamos la siguiente orden de nuestro compilador:

```
$ g++ basic.cpp -I /usr/include/python3.7 -I /usr/include/boost
-lboost_python3 -shared -fPIC -o basic.so
```

Ahora veamos cómo usamos estas funciones en un script de Python. En primer lugar tenemos que importar el módulo que definíamos anteriormente como *basic*:

```
import basic
```

Ahora ya podríamos recurrir a la función previamente implementada. Para probar su funcionamiento, usaremos distintos tipos de datos (*int*, *float* y *string*):

```
print('\nGeneric functions')
print(basic.generic_sum(2, 2))
print(basic.generic_sum(19.1, 1.1))
print(basic.generic_sum("aaa", "bbb"))
```

Como vemos, llamamos a la función deseada recurriendo al módulo *basic* importado: `basic.<funcion>(<argumentos>))`. Ejecutemos el script para comprobar los resultados obtenidos:

```
Generic functions
4.0
20.200000762939453
aaabbb
```

Ejemplo de uso de objetos de Python en C++

También podemos trabajar con objetos típicos de Python en el código de C++. Por ejemplo, podemos definir una lista de Python de la siguiente manera:

```
bp::list lista;
```

Siendo bp el espacio de nombres de *Boost.Python*, el cuál es `boost::python`, que hemos debido de declarar previamente.

Sobre esta lista podemos actuar tal y como lo haríamos en Python. Por ejemplo, si quisiéramos añadir un elemento a la lista, podríamos hacer lo siguiente:

```
lista.append(<elemento>);
```

Un ejemplo de uso del tipo `list` es el siguiente, en el cuál recorremos una lista de entrada y generamos una nueva lista con todos los elementos de la lista original que sean números enteros:

```
bp::list getIntList(const bp::list& l)
{
    // Crear nueva lista
    bp::list intList;

    bp::ssize_t length = bp::len(l);

    for (bp::ssize_t i = 0; i < length; i++)
    {
        // Crear extractor
        bp::extract<int> intExtractor(l[i]);

        // Comprobar si el elemento puede ser extraído
        if (intExtractor.check())
        {
            // Añadir elemento a la lista
            intList.append(intExtractor());
        }
    }

    return intList;
}
```

La mayoría del código no es ningún misterio. Vemos que podemos utilizar algunas funciones básicas de Python, como por ejemplo `len` para obtener la longitud de la lista. `bp::ssize_t` es un tipo de *Boost.Python* que es igual al `size_t` de C++.

Lo único que puede llamar la atención es `bp::extract<T>`. Esto lo que hace es intentar extraer a partir de un objeto de Python un elemento con el tipo especificado de C++, de forma que podamos realizar operaciones con él. En este caso, queremos extraer un `int` a partir del elemento *i*-ésimo de la lista. De la forma en la que se hace en el ejemplo se hace un primer intento de hacer una extracción, creando en el proceso un objeto llamato `intExtractor`. Para comprobar si podemos extraer o no el valor, tenemos que hacer una llamada al método `intExtractor.check()`, el cuál dice si podemos o no hacerlo. En caso de poder, añadimos el elemnto a la lista, obteniendo el valor directamente.

Un ejemplo un poco más sofisticado en el que recorremos una lista de elementos y mostramos el tipo de cada uno es el siguiente:

```
void checkListTypes(const bp::list& l)
{
    bp::ssize_t length = bp::len(l);

    for (bp::ssize_t i = 0; i < length; i++)
    {
        // Extraer el objeto en la posicion actual
        bp::object obj = bp::extract<bp::object>(l[i]);

        // Obtener el nombre de la clase
        std::string objClassName = bp::extract<std::string>(
            obj.attr("__class__").attr("__name__")
        );

        std::cout << "Clase del objeto: " << objClassName << std::endl;
    }
}
```

Aquí, a diferencia del ejemplo anterior, extraemos directamente el elemento a partir del elemento de la lista, sin comprobar si se puede hacer o no. En este caso, extraemos un objeto de Python (esto es un objeto de tipo `bp::object`). Para obtener información sobre la clase a la que pertence, tenemos que acceder al atributo `__class__` del objeto, el cuál guarda el nombre dentro del atributo `__name__`. Para ello, podemos extraer dicho accediendo de la forma especificada anteriormente y guardar dicho valor como un `string` para poder mostrarlo posteriormente.

Un último ejemplo con listas es una función que recibe una lista de listas (lista 2D), y la aplana (hace que sea una lista 1D). Todo lo que necesitamos saber ya lo hemos visto anteriormente, así que vamos a ver el ejemplo:

```
bp::list flatten2DList(const bp::list& l)
{
    bp::ssize_t length = bp::len(l);
    bp::list flattenedList;

    for (bp::ssize_t i = 0; i < length; i++)
    {
        bp::list innerList = bp::extract<bp::list>(l[i]);
        bp::ssize_t innerLength = bp::len(innerList);
```

```
        for (bp::ssize_t j = 0; j < innerLength; j++)
        {
            bp::object obj = bp::extract<bp::object>(innerList[j]);
            flattenedList.append(obj);
        }
    }

    return flattenedList;
}
```

También podemos utilizar diccionarios de Python. En el siguiente ejemplo, se crea una función que recibe como parámetro un diccionario e itera sobre las claves, mostrando los valores. Se espera que tanto las claves como los valores sean instancias de `string`, ya que si no se producirá un error:

```
void iterateDictionary(const bp::dict& dict)
{
    // Get keys
    bp::list keyList = dict.keys();

    bp::ssize_t numKeys = bp::len(keyList);

    for (bp::ssize_t i = 0; i < numKeys; i++)
    {
        std::string key = bp::extract<std::string>(keyList[i]);
        std::string val = bp::extract<std::string>(dict[key]);

        std::cout << key << ": " << val << std::endl;
    }
}
```

Primero se obtiene una lista con las claves. Se extrae el valor de cada una de las claves y se accede al valor asociado a dicha clave, el cuál también es extraído.

Exportamos las funciones de la siguiente forma:

```
BOOST_PYTHON_MODULE(objetos)
{
    using namespace bp;

    def("check_types", checkListTypes);
```

```
    def("get_int_list", getIntList);
    def("iterate_dict", iterateDictionary);
    def("flatten_2D_list", flatten2DList);
}
```

Para probarlas, vamos a importar el módulo y vamos a hacer algunas llamadas a las funciones con algunos objetos:

```
import objetos

my_list = ['aa', 'bvbbb', 1, 3, 1.524, 1+4j, [1, 1], 3]
my_dict = {'cat': 'meow', 'dog': 'woof', 'cow': 'moo'}
my_2D_list = [[12, 1, 'a', '3'], [100, 1+4j, 3], ['asdf', 'ksdf']]

print('Check types')
objetos.check_types(my_list)

print('\nGet int list')
print(objetos.get_int_list(my_list))

print('\nFlatten 2D list')
print(objetos.flatten_2D_list(my_2D_list))

print('\nIterate over dictionary')
objetos.iterate_dict(my_dict)
```

```
Check types
Clase del objeto: str
Clase del objeto: str
Clase del objeto: int
Clase del objeto: int
Clase del objeto: float
Clase del objeto: complex
Clase del objeto: list
Clase del objeto: int

Get int list
[1, 3, 3]

Flatten 2D list
[12, 1, 'a', '3', 100, (1+4j), 3, 'asdf', 'ksdf']
```

```
Iterate over dictionary
cat: meow
dog: woof
cow: moo
```

Ejemplo de uso de una clase de C++ en Python

Para este ejemplo hemos implementado una clase sencilla de C++ y luego hemos sobrecargado uno de sus operadores, concretamente el de suma. La función de la clase es representar un punto 3D, por tanto, consta de tres coordenadas (x, y y z). Debido a esto último tenemos que implementar una serie de funciones que nos permitan acceder y modificar dichos atributos privados (*setters* y *getters*). También se han incluido otras funciones: una para calcular la distancia entre dos puntos y otra para obtener una representación en string del objeto.

```
template <typename T>
class Punto3D {
    private:
        T x, y, z;

    public:
        // Constructor
        Punto3D(T x, T y, T z): x(x), y(y), z(z) {}

        T getX() const
        {
            return this->x;
        }

        T getY() const
        {
            return this->y;
        }

        T getZ() const
        {
            return this->z;
        }
```

```
        void setX(T newX)
        {
            this->x = newX;
        }

        void setY(T newY)
        {
            this->y = newY;
        }

        void setZ(T newZ)
        {
            this->z = newZ;
        }

        double distancia(const Punto3D &p) const
        {
            return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2)
                        + pow(z - p.z, 2));
        }

        std::string toString() const
        {
            return "Punto3D -> x: "
                        + std::to_string(this->x)
                        + " y: " + std::to_string(this->y)
                        + " z: " + std::to_string(this->z);
        }
};
```

Como decíamos, además, hemos sobrecargado el operador de suma de la clase. El siguiente fragmento de código muestra como se ha llevado a cabo:

```
template <typename T>
Punto3D<T> operator+(const Punto3D<T>& p1, const Punto3D<T>& p2)
{
    T newX = p1.getX() + p2.getX(),
      newY = p1.getY() + p2.getY(),
      newZ = p1.getZ() + p2.getZ();

    Punto3D<T> newP = Punto3D<T>(newX, newY, newZ);
```

```
    return newP;
}
```

Una vez tenemos el código de C++, hemos de pasar a crear el módulo de *Boost.Python*. En este caso tenemos que exponer la clase y, por tanto, sus correspondientes funciones y atributos:

```
BOOST_PYTHON_MODULE(claseEjemplo)
{
    using namespace boost::python;

    class_<Punto3D<int>>("Punto3D", init<int, int, int>())
        .add_property("x", &Punto3D<int>::getX, &Punto3D<int>::setX)
        .add_property("y", &Punto3D<int>::getY, &Punto3D<int>::setY)
        .add_property("z", &Punto3D<int>::getZ, &Punto3D<int>::setZ)
        .def("distancia", &Punto3D<int>::distancia)
        .def("__str__", &Punto3D<int>::toString)
        .def(self + self);
}
```

Como vemos, lo que se ha hecho ha sido, en primer lugar, exponer la clase. Como estamos tratando con una clase genérica, tenemos que indicar el tipo de dato con el que queremos trabajar, también hemos de hacer esto a la hora de exponer el constructor: `class_<NombreClase<T>>("Nombre Clase Python", init<T, T, T>())`. Justo después se procede a incluir sus atributos y sus respectivas funciones de modificación y consulta, para esto recurrimos a la función `.add_property(<Nombre Attr Python>, <Getter>, <Setter>)`. De esta forma, podemos acceder y modificar los atributos en Python de forma normal, sin tener que romper para ello la encapsulación en C++. Por otro lado, tenemos que añadir también las dos funciones implementadas pero esta vez con `.def(<Nombre Python>, <Función>)`. De esta forma, se añaden como métodos de la clase en vez de como funciones del módulo. Hace falta destacar que el método `toString` se añade el método `__str__` de la clase, el cuál se encargará de ofrecer una representación en formato string. El operador sobrecargado hay que añadirlo también con `.def` pero indicando que intervienen dos objetos de la propia clase `((self + self))`.

Veamos cómo el uso de esta clase en Python es realmente sencillo:

```
# Importamos módulo
import claseEjemplo
```

```
# Declaración
p1 = claseEjemplo.Punto3D(1,1,1)
p2 = claseEjemplo.Punto3D(1,2,4)
p3 = p1

# Consulta
print(f"Punto 1 {p1}")
print(f"Punto 2 {p2}")
print(f"Punto 3 {p3}")

# Modificación
p3.y = 4

# Consulta
print(f"\nPunto 1 {p1}")
print(f"Punto 3 {p3}")

# Cálculo de distancia
dist = p1.distancia(p2)
print(f"\nDistancia entre los dos puntos: {dist}")

# Suma de dos objetos
print(f"\nSuma de los dos puntos: {p1 + p2}")
```

```
Punto 1 Punto3D -> x: 1 y: 1 z: 1
Punto 2 Punto3D -> x: 1 y: 2 z: 4
Punto 3 Punto3D -> x: 1 y: 1 z: 1

Punto 1 Punto3D -> x: 1 y: 4 z: 1
Punto 3 Punto3D -> x: 1 y: 4 z: 1

Distancia entre los dos puntos: 3.605551275463989

Suma de los dos puntos: Punto3D -> x: 2 y: 6 z: 5
```

Comparativa: supresión de no máximos de una imagen

La supresión de no máximos de una imagen es un procedimiento costoso y de alto cómputo, es por esto por lo que seleccionamos este ejemplo para mostrar la diferencia entre usar un código en Python con *numpy* y un código de Python que use un módulo implementado en C++, el cuál utiliza *Boost.Python*.

Este proceso consiste en recorrer todos y cada uno de los píxeles de la imagen y comparar dicho píxel con su vecindario de 3×3: si el valor de dicho píxel es el máximo de su vecindario, entonces lo mantenemos, si no, lo eliminamos (lo ponemos a 0).

Para hacer la comparación, vamos a implementar la función primero en Python y luego haremos una implementación en C++. En ambos casos, las imágenes de entrada están representadas como arrays de *numpy*. Las dos funciones tienen que devolver un array de *numpy* con las mismas dimensiones que la entrada. Para poder usar arrays de *numpy* en C++, podemos servirnos de *Boost.Python*, el cuál recordemos que tiene una parte que nos permite representar dichos elementos.

Una vez dicho esto, vamos a ver las dos implementaciones, y después vamos a ver los tiempos obtenidos en cada caso.

Implementación en Python

La implementación en Python es bastante directa, y se puede ver en el siguiente código:

```
def non_max_supression(img):
    # Crear imagen inicial para la supresion de maximos
    # (inicializada a 0)
    supressed_img = np.zeros_like(img)

    # Para cada pixel, aplicar una caja 3x3 para determinar
    # el maximo local
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            # Obtener la region 3x3 (se consideran los bordes
            # para que la caja tenga el tamaño adecuado,
            # sin salirse)
            region = img[max(i-1, 0):i+2, max(j-1, 0):j+2]

            # Obtener el valor actual y el maximo de la region
            current_val = img[i, j]
            max_val = np.max(region)

            # Si el valor actual es igual al maximo, copiarlo
            # en la imagen de supresion
            # de no maximos
            if max_val == current_val:
                supressed_img[i, j] = current_val

    return supressed_img
```

Implementación en C++ con Boost.Python

La implementación en C++ es un poco más intrincada, ya que no disponemos de las funciones de *numpy* ni podemos usar el *slicing*, tal y como hacíamos anteriormente.

Vamos a ver cómo se ha realizado dicha implementación y luego comentaremos los aspectos más relevantes:

```
#include <boost/python.hpp>
#include <boost/python/numpy.hpp>
#include <vector>

namespace bp = boost::python;
namespace np = boost::python::numpy;

np::ndarray nonMaxSupression(const bp::object& img)
{
    // Crear entorno que usa numpy
    Py_Initialize();
    np::initialize();

    // Obtener tamaño y tipo de imagen de salida
    bp::tuple shape = bp::extract<bp::tuple>(img.attr("shape"));
    np::dtype dtype = np::dtype::get_builtin<double>();

    // Crear imagen de salida con 0's
    np::ndarray supr = np::zeros(shape, dtype);

    int rows = bp::extract<int>(shape[0]),
        columns = bp::extract<int>(shape[1]);

    std::vector<double> region;

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            double currentValue = bp::extract<double>(img[i][j]);
            region.clear();

            // Procesar region 3x3
            for (int xReg = std::max(i-1, 0);
                 xReg <= std::min(i+1, rows-1);
```

```
                 xReg++)
            {
                for (int yReg = std::max(j-1, 0);
                     yReg <= std::min(j+1, columns-1);
                     yReg++)
                {
                    region.push_back(
                        bp::extract<double>(img[xReg][yReg])
                    );
                }
            }

            // Obtener maximo de la region
            auto max = std::max_element(
                region.begin(), region.end()
            );

            // Copiar valor actual si es el maximo
            if (*max == bp::extract<double>(img[i][j]))
            {
                supr[i][j] = img[i][j];
            }
        }
    }

    return supr;
}
```

Es relevante destacar que se ha pasado la imagen como un objeto de Python, ya que posteriormente nos interesa acceder a uno de sus atributos.

Lo primero que se hace es inicializar el intérprete de Python y el módulo de *numpy*. El primer paso es necesario, ya que si no, se produce un error.

A continuación extraemos el tamaño de la imagen original como una tupla de Python accediendo al atributo `shape`. También obtenemos el `dtype` asociado al tipo `double` de C++, el cuál es `float64`. Con esta información, creamos un *array* de *numpy* relleno de ceros, en el cuál escribiremos la salida.

A continuación, obtenemos el número de filas y columnas e iteramos sobre cada píxel de la imagen. Para cada píxel, nos escogemos una región 3×3, accediendo a la matriz en función de los índices i, j, y controlando que no nos salgamos de la matriz (lo cuál podría producir un fallo de segmentación). Guardamos cada elemento en un vector y después obtenemos el máximo. Si el máximo coincide con

el elemento actual, se copia en la matriz de salida.

Podemos exponer la función anterior de la siguiente forma:

```
BOOST_PYTHON_MODULE(imageModule)
{
    def("non_max_supression", nonMaxSupression);
}
```

Comparativa

Para comparar el rendimiento de cada función, vamos a ejecutar las dos funciones pasándoles 100 imágenes. Vamos a medir el tiempo que tarda cada función para cada imagen y el timpo total para las 100 imágenes.

Ya que en uno de los casos estamos utilizando C++ y no tenemos la mayoría de funciones de *numpy* y tampoco tenemos acceso al *slicing*, vamos a compilar el código con optimización. Para ello, vamos a compilarlo con optimización `-Os`, reduciendo por tanto el tamaño del archivo de salida y aplicando una optimización no agresiva.

El tiempo obtenido implementando el código con Python es el siguiente:

```
    Total time spent in non-max supression: 87.28784799575806
```

El tiempo empleado durante la ejecución del código que usa la biblioteca *Boost.Python* es el siguiente:

```
    Total time spent in non-max supression: 53.51091933250427
```

Como vemos, hay una diferencia de tiempos notable. Es más, en el código de *Boost.Python*, por cada imagen se emplea alrededor de 0.55 segundos:

```
[...]
Time in non-max supression: 0.6244661808013916
Time in non-max supression: 0.6220667362213135
Time in non-max supression: 0.5682306289672852
Time in non-max supression: 0.5632765293121338
Time in non-max supression: 0.5228853225708008
Time in non-max supression: 0.5326223373413086
[...]
```

En cambio, usando *Python* se tarda alrededor de 0.9 segundos:

```
[...]
Time in non-max supression: 0.9579119682312012
Time in non-max supression: 0.9638268947601318
Time in non-max supression: 0.970787763595581
Time in non-max supression: 0.8839678764343262
Time in non-max supression: 0.7947449684143066
Time in non-max supression: 0.812824010848999
[...]
```

Conclusiones

En resumen, *Boost.Python* es un módulo potente y que ofrece muchas posibilidades para conectar C++ con Python. Como principal ventaja tenemos que es bastante completo, y permite hacer un *wrapper* sobre código de C++ de forma relativamente sencilla. Además, la documentación es bastante completa, y en internet se pueden acceder a foros donde se resuelven las dudas.

No obstante, como punto negativo encontramos que tiene una curva de aprendizaje un poco elevada, ya que cuesta acostumbrarse a él al principio.