

---

# Documentation of the pw85 library

*Release 1.0rc1*

**Sébastien Brisard**

**Sep 22, 2019**



# CONTENTS

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Installation</b>  | <b>3</b>  |
| 1.1      | Building and installing the C library . . . . .            | 3         |
| 1.2      | Installation of the Python bindings . . . . .              | 4         |
| <b>2</b> | <b>Tutorial</b>  | <b>7</b>  |
| 2.1      | Python tutorial . . . . .                                  | 7         |
| 2.2      | C tutorial . . . . .                                       | 10        |
| <b>3</b> | <b>Theory</b>  | <b>13</b> |
| 3.1      | Mathematical representation of ellipsoids . . . . .        | 13        |
| 3.2      | The contact function of two ellipsoids . . . . .           | 14        |
| 3.3      | Geometric interpretation . . . . .                         | 14        |
| 3.4      | References . . . . .                                       | 15        |
| <b>4</b> | <b>Implementation of the function <math>f</math></b>       | <b>17</b> |
| 4.1      | Implementation #1: using Cholesky decompositions . . . . . | 17        |
| 4.2      | Implementation #2: using rational functions . . . . .      | 18        |
| 4.3      | Comparison of the two implementations . . . . .            | 19        |
| <b>5</b> | <b>Optimization of the function <math>f</math></b>         | <b>23</b> |
| <b>6</b> | <b>Testing the implementation of the contact function</b>  | <b>25</b> |
| <b>7</b> | <b>The C API</b>   | <b>27</b> |
| 7.1      | Representation of vectors and matrices . . . . .           | 27        |
| 7.2      | The new API . . . . .                                      | 28        |
| 7.3      | The “legacy” API . . . . .                                 | 30        |
| <b>8</b> | <b>The python API</b>                                      | <b>33</b> |
| 8.1      | The new API . . . . .                                      | 33        |
| 8.2      | The “legacy” API . . . . .                                 | 35        |
|          | <b>Bibliography</b>  | <b>37</b> |
|          | <b>Python Module Index</b>                                 | <b>39</b> |



### **Abstract**

This library implements the “contact function” defined by Perram and Wertheim (J. Comp. Phys. 58(3), 409–416, DOI:10.1016/0021-9991(85)90171-8) for two ellipsoids. Given two ellipsoids, this function returns the *square* of the common factor by which both ellipsoids must be scaled (their centers being fixed) in order to be tangentially in contact.

This library is released under a BSD 3-Clause License.



## INSTALLATION

**Contents**

- *Building and installing the C library*
- *Installation of the Python bindings*

This chapter describes how to install the C library as well as the Python bindings. The first step is to clone the Git repository:

```
git clone https://github.com/sbrisard/pw85.git
```

## 1.1 Building and installing the C library

PW85 requires a POSIX system. On Windows platforms, it is recommended that you install [MSYS2](#).

**Note:** Meson supports MSVC compilers. However, at the time of writing, MSVC-based installation has not been tested. Contributions are most welcome!

The C library depends on

1. The [GLib](#) (for testing purposes)
2. The [GNU Scientific Library \(GSL\)](#) (for the implementation of the Brent algorithm)
3. The [HDF5](#) (for testing purposes)

The installation procedure also requires Python 3.

For installation, we use the [Meson build system](#). We assume that GLib, GSL and Meson are installed on your system. Assuming that the project is built in the `src/build/` directory (no need to create it first), here is the whole installation procedure (you must first `cd` into the root directory of the PW85 project):

```
cd pw85/src
meson build
cd build
ninja install
```

A prefix can be specified in order for PW85 to be installed in a custom directory, like so:

```
cd pw85/src
meson build --prefix=C:/opt/pw85
cd build
ninja install
```

Congratulations, PW85 is now built and installed! You can then test the installation (stay in the `src/build` directory):

```
meson test
```

If you intend to use PW85 from within Python only, then go to [Installation of the Python bindings](#).

If you also intend to link the library to e.g. C executables, you must inform your system about the location of the library.

- On Windows platforms, you need to add the full path to `pw85.dll` to your PATH environment variable.
- On Linux or MacOS platforms, no further operation is required.

To further test your installation, build the example in the [C tutorial](#).

## 1.2 Installation of the Python bindings

The installation procedure is fairly standard and should be platform independent. It requires a fairly recent version of [NumPy](#). I successfully installed the Python bindings alongside v1.16.4 of NumPy. Please report if you are successful with older versions.

Installation in a virtual environment is not covered here, but is possible with little alterations to the procedure below.

Open a terminal and `cd` into the `wrappers/python-ctypes/` directory. Issue the following command:

```
$PYTHON_EXEC setup.py install
```

where `$PYTHON_EXEC` denotes your Python 3 executable (usually, `python` or `python3`). Then, you need to define the location of the dynamic libraries, for `ctypes` to be able to import it. This is done through the `pypw85.cfg` file, which you must create and place in the following directory

- Windows 10/8/7/Vista: `C:\Users\<User Name>\AppData\Roaming\pypw85`
- Windows XP/2000: `C:\Documents and Settings\<username>\Application Data\pypw85`
- Mac: `/Users/<username>/Library/Application Support/pypw85`
- Linux: `~/pypw85`

(in all cases, the `pypw85` subdirectory must be created). The contents of the `pw85.cfg` file should be:

```
[pw85]
libpw85 = full/path/to/the/pw85/dynamic/library
libpw85_legacy = full/path/to/the/pw85_legacy/dynamic/library
datadir = full/path/to/the/pw85/data/directory
```

where the `libpw85` and `libpw85_legacy` entries are the full path to the dynamic libraries (`*.dll`, `*.so` or `*.dylib`) *including their name*. All these configure options can be retrieved from the output of `ninja install`. For example, on a Windows machine, where the output was:

```
Installing libpw85.dll to C:/opt/pw85/bin
Installing libpw85.dll.a to C:/opt/pw85/lib
Installing libpw85.a to C:/opt/pw85/lib
```

(continues on next page)



(continued from previous page)

```
Installing libpw85_legacy.dll to C:/opt/pw85/bin
Installing libpw85_legacy.dll.a to C:/opt/pw85/lib
Installing libpw85_legacy.a to C:/opt/pw85/lib
Installing pw85_ref_data.h5 to C:/opt/pw85/share/pw85
Installing C:\path\to\pw85\src\pw85_legacy.h to C:/opt/pw85/include
Installing C:\path\to\pw85\src\build\pw85.h to C:/opt/pw85/include
```

the contents of pw85.ini is:

```
[pw85]
libpw85 = C:/opt/pw85/bin/libpw85.dll
libpw85_legacy = C:/opt/pw85/bin/libpw85_legacy.dll
datadir = C:/opt/pw85/share/pw85
```

Provided the [pytest](#) module is installed on your machine, you can run the tests as follows (from the wrappers/python-ctypes directory):

```
$PYTHON_EXEC -m pytest tests/test_pw85.py
```

You can also test the “legacy” API. This requires the [h5py](#) module. To run the tests, issue the command:

```
$PYTHON_EXEC -m pytest tests/test_pw85_legacy.py
```

(beware, these tests take some time!).



## TUTORIAL

**Contents**

- *Python tutorial*
  - *Checking the output*
- *C tutorial*

In this tutorial, we consider two ellipsoids, and check whether or not they overlap.

Ellipsoid  $E_1$  is an *oblate spheroid* centered at point  $x_1 = (-0.5, 0.4, -0.7)$ , with equatorial radius  $a_1 = 10$ , polar radius  $c_1 = 0.1$  and polar axis  $(0, 0, 1)$ .

Ellipsoid  $E_2$  is a *prolate spheroid* centered at point  $(0.2, -0.3, 0.4)$ , with equatorial radius  $a_1 = 0.5$ , polar radius  $c_1 = 5$  and polar axis  $(1, 0, 0)$ .

To carry out the overlap check, we must first create the representation of ellipsoids  $E_i$  as quadratic forms  $Q_i$  (see *Mathematical representation of ellipsoids*). Convenience functions are provided to compute the matrix representation of a *spheroid*.

**Note:** In principle, the contact function implemented in PW85 applies to *any* ellipsoids (with unequal axes). However, at the time of writing this tutorial (2019-01-01), convenience functions to compute the matrix representation of a general ellipsoid is not yet implemented. Users must compute the matrices themselves.

We first check for the overlap of  $E_1$  and  $E_2$  using the Python wrapper of pw85. We will then illustrate the C API.

## 2.1 Python tutorial

The Python module relies on *NumPy* for passing arrays to the underlying C library. We therefore import both modules:

```
>>> import numpy as np
>>> import pypw85
```

and define the parameters of the simulation:

```
>>> x1 = np.array([-0.5, 0.4, -0.7])
>>> n1 = np.array([0., 0., 1.])
>>> a1, c1 = 10, 0.1
>>> x2 = np.array([0.2, -0.3, 0.4])
```

(continues on next page)

(continued from previous page)

```
>>> n2 = np.array([1., 0., 0.])
>>> a2, c2 = 0.5, 5.
>>> r12 = x2-x1
```

where  $r_{12}$  is the vector that joins the center of the first ellipsoid,  $x_1$ , to the center of the second ellipsoid,  $x_2$ .

We use the function `pypw85.spheroid()` to create the matrix representations  $q_1$  and  $q_2$  of the two ellipsoids:

```
>>> q1 = pypw85.spheroid(a1, c1, n1)
>>> q1
array([ 1.e+02, -0.e+00, -0.e+00,  1.e+02, -0.e+00,  1.e-02])
>>> q2 = pypw85.spheroid(a2, c2, n2)
>>> q2
array([25. ,  0. ,  0. ,  0.25,  0. ,  0.25])
```

We can now compute the value of the contact function — see the documentation of `pypw85.contact_function()`:

```
>>> mu2, lambda_ = pypw85.contact_function(r12, q1, q2)
>>> print('μ² = {}'.format(mu2))
>>> print('λ = {}'.format(lambda_))
μ² = 3.362706040638343
λ = 0.1668589553405904
```

We find that  $\mu^2 > 1$ , hence  $\mu > 1$ . In other words, both ellipsoids must be *swollen* in order to bring them in contact: the ellipsoids do not overlap!

### 2.1.1 Checking the output

The output of this simulation can readily be checked. First, we can check that  $q_1$  and  $q_2$  indeed represent the ellipsoids  $E_1$  and  $E_2$ . To do so, we first construct the symmetric matrices  $Q_1$  and  $Q_2$  from their upper triangular part

```
>>> Q1 = np.zeros((3, 3), dtype=np.float64)
>>> i, j = np.triu_indices_from(Q1)
>>> Q1[i, j] = q1
>>> Q1[j, i] = q1
>>> Q1
array([[ 1.e+02, -0.e+00, -0.e+00],
       [-0.e+00,  1.e+02, -0.e+00],
       [-0.e+00, -0.e+00,  1.e-02]])
```

```
>>> Q2 = np.zeros_like(Q1)
>>> Q2[i, j] = q2
>>> Q2[j, i] = q2
>>> Q2
array([[25. ,  0. ,  0. ],
       [ 0. ,  0.25,  0. ],
       [ 0. ,  0. ,  0.25]])
```

We can now check these matrices for some remarkable points, first for ellipsoid  $E_1$

```
>>> Q1_inv = np.linalg.inv(Q1)
>>> f1 = lambda x: Q1_inv.dot(x).dot(x)
>>> f1((a1, 0., 0.))
1.0
>>> f1((-a1, 0., 0.))
```

(continues on next page)

(continued from previous page)

```

1.0
>>> f1((0., a1, 0.))
1.0
>>> f1((0., -a1, 0.))
1.0
>>> f1((0., 0., c1))
0.99999999999994884
>>> f1((0., 0., -c1))
0.99999999999994884

```

then for ellipsoid  $E_2$

```

>>> Q2_inv = np.linalg.inv(Q2)
>>> f2 = lambda x: Q2_inv.dot(x).dot(x)
>>> f2((c2, 0., 0.))
1.0
>>> f2((-c2, 0., 0.))
1.0
>>> f2((0., a2, 0.))
1.0
>>> f2((0., -a2, 0.))
1.0
>>> f2((0., 0., a2))
1.0
>>> f2((0., 0., -a2))
1.0

```

Note that in the above tests, we have omitted the centers of ellipsoids  $E_1$  et  $E_2$  (both ellipsoids were translated to the origin).

We will now verify the correctness of the value found for the scaling factor  $\mu$ . To do so, we will find the coordinates of the contact point of the two scaled ellipsoids, and check that the normals to the two ellipsoids at that point coincide.

Although we use formulæ from the [Theory](#) section to find the coordinates of the contact point,  $x_0$ , it is not essential for the time being to fully understand this derivation. What really matters is to check that the resulting point  $x_0$  is indeed the contact point of the two scaled ellipsoids; how the coordinates of this point were found is irrelevant.

From the value of  $\lambda$  returned by the function `pw85.contact_function()`, we compute  $Q$  defined by Eq. (10) in section [Theory](#), as well as  $x = Q^{-1} \cdot r_{12}$

```

>>> Q = (1-lambda_)*Q1+lambda_*Q2
>>> x = np.linalg.solve(Q, r12)

```

From which we find  $x_0$ , using either Eq. (9a) or Eq. (9b) (and we can check that both give the same result)

```

>>> x0a = x1+(1-lambda_)*np.dot(Q1, x)
>>> x0a
array([ 0.16662271, -0.29964969, -0.51687799])
>>> x0b = x2-lambda_*np.dot(Q2, x)
>>> x0b
array([ 0.16662271, -0.29964969, -0.51687799])

```

We can now check that  $x_0$  belongs to the two scaled ellipsoids, that we first define, overriding the matrices of the unscaled ellipsoids, that are no longer needed. We observe that if ellipsoid  $E_i$  is scaled by  $\mu$ , then its matrix representation  $Q_i$  is scaled by  $\mu^2$ , and its inverse  $Q_i^{-1}$  is scaled by  $\mu^{-2}$ .

```
>>> x0 = x0a
>>> Q1 *= mu2
>>> Q2 *= mu2
>>> Q1_inv /= mu2
>>> Q2_inv /= mu2
```

```
>>> x = x0-x1
>>> Q1_inv.dot(x).dot(x)
1.00000000000058238
```

```
>>> x = x0-x2
>>> Q2_inv.dot(x).dot(x)
0.99999999999988334
```

Therefore  $x_0$  indeed belongs to both ellipsoids. We now compute the normal  $n_i$  to ellipsoid  $E_i$  at point  $x_0$ . Since ellipsoid  $E_i$  is defined by the level-set:  $(x-x_i)^T \cdot Q_i^{-1} \cdot (x-x_i) = 1$ , the normal to  $E_i$  is given by  $Q_i^{-1} \cdot (x-x_i)$  (which is then suitably normalized)

```
>>> n1 = Q1_inv.dot(x0-x1)
>>> n1 /= np.linalg.norm(n1)
>>> n1
array([ 3.64031943e-04, -3.82067448e-04,  9.99999861e-01])
```

```
>>> n2 = Q2_inv.dot(x0-x2)
>>> n2 /= np.linalg.norm(n2)
>>> n2
array([-3.64031943e-04,  3.82067448e-04, -9.99999861e-01])
```

We find that  $n_1 = -n_2$ . Therefore,  $E_1$  and  $E_2$  are in external contact. QED

Follow this link to download the above Python script.

## 2.2 C tutorial

The Python interface to PW85 has been kept close to the underlying C API. The following C program (download source file) defines the two ellipsoids, then computes  $\mu^2$  and  $\lambda$ :

```
#include <pw85.h>
#include <stdio.h>
#include <stdlib.h>

#define DIM 3
#define SYM 6

int main() {
    double x1[] = {-0.5, 0.4, -0.7};
    double n1[] = {0., 0., 1.};
    double a1 = 10.;
    double c1 = 0.1;

    double x2[] = {0.2, -0.3, 0.4};
    double n2[] = {1., 0., 0.};
    double a2 = 0.5;
    double c2 = 5.;
```

(continues on next page)

(continued from previous page)

```

double r12[DIM];
for (int i = 0; i < DIM; i++) r12[i] = x2[i] - x1[i];

double q1[SYM], q2[SYM];
pw85_spheroid(a1, c1, n1, q1);
pw85_spheroid(a2, c2, n2, q2);

double out[2];
pw85_contact_function(r12, q1, q2, out);
printf("mu^2 = %g\n", out[0]);
printf("lambda = %g\n", out[1]);
printf("\n");
}

```

A meson.build file is provided for the compilation of the tutorial using the [Meson build system](#). You can reuse it in one of your own projects (download):

```

project('PW85_C_Tutorial', 'c')

pw85_include = include_directories(get_option('pw85_include'))

cc = meson.get_compiler('c')
pw85_lib = cc.find_library('pw85', dirs: get_option('pw85_lib'))

exe = executable('tutorial',
                 'tutorial.c',
                 include_directories: pw85_include,
                 dependencies: pw85_lib)

```

The location of the PW85 dynamic library and header files is specified through two options that are defined in the meson\_options.txt file (download):

```

option('pw85_include', type: 'string', value: '/usr/include')
option('pw85_lib', type: 'string', value: '/usr/lib')

```

Compilation proceeds as follows:

```

meson setup -Dpw85_include=/c/opt/pw85/include/ -Dpw85_lib=/c/opt/pw85/bin/ build
cd build/
ninja

```

It produces an executable called (depending on the platform): tutorial.exe or tutorial. On execution, it prints the following lines to stdout:

```

mu^2 = 3.36271
lambda = 0.166859

```





#### Contents

- *Mathematical representation of ellipsoids*
- *The contact function of two ellipsoids*
- *Geometric interpretation*
- *References*

This chapter provides the theoretical background to the Perram–Wertheim algorithm [PW85]. We use matrices rather than tensors: a point/vector is therefore defined through the  $3 \times 1$  column-vector of its coordinates. Likewise, a second-rank tensor is represented by its  $3 \times 3$  matrix.

Only the global, cartesian frame is considered here, and there is no ambiguity about the basis to which these column vectors and square matrices refer.

### 3.1 Mathematical representation of ellipsoids

Ellipsoids are defined from their center  $c$  and a positive-definite quadratic form  $Q$  as the set of points  $m$  such that:

$$(1) \quad (m-c)^T \cdot Q^{-1} \cdot (m-c) \leq 1.$$

$Q$  is a symmetric, positive-definite matrix:

$$(2) \quad Q = \sum_i a_i^2 v_i \cdot v_i^T,$$

where  $a_1, a_2, a_3$  are the lengths of the principal semi-axes and  $v_1, v_2, v_3$  their directions (unit vectors).

In the PW85 library,  $Q$  is represented as a `double[6]` array `q` which stores the upper triangular part of  $Q$  in row-major order:

$$(3) \quad Q = \begin{bmatrix} q[0] & q[1] & q[2] \\ & q[3] & q[4] \\ \text{sym.} & & q[5] \end{bmatrix}.$$

## 3.2 The contact function of two ellipsoids

Let  $E_1$  and  $E_2$  be two ellipsoids, defined by their centers  $c_1$  and  $c_2$  and quadratic forms  $Q_1$  and  $Q_2$ , respectively.

For  $0 \leq \lambda \leq 1$  and a point  $x$ , we introduce the function:

$$(4) \quad F(x, \lambda) = \lambda(x - c_1)^T \cdot Q_1^{-1} \cdot (x - c_1) + (1 - \lambda)(x - c_2)^T \cdot Q_2^{-1} \cdot (x - c_2).$$

For fixed  $\lambda$ ,  $F(x, \lambda)$  has a unique minimum [PW85]  $f(\lambda)$ , and we define:

$$(5) \quad f(\lambda) = \min\{ F(x, \lambda), x \in \mathbb{R}^3 \}, \quad 0 \leq \lambda \leq 1.$$

Now, the function  $f$  has a unique maximum over  $[0, 1]$ , and the “contact function”  $F(r_{12}, Q_1, Q_2)$  of ellipsoids  $E_1$  and  $E_2$  is defined as:

$$(6) \quad F(r_{12}, Q_1, Q_2) = \max\{ f(\lambda), 0 \leq \lambda \leq 1 \},$$

where  $r_{12} = c_2 - c_1$ . It can be shown that

- if  $F(r_{12}, Q_1, Q_2) < 1$  then  $E_1$  and  $E_2$  overlap,
- if  $F(r_{12}, Q_1, Q_2) = 1$  then  $E_1$  and  $E_2$  are externally tangent,
- if  $F(r_{12}, Q_1, Q_2) > 1$  then  $E_1$  and  $E_2$  do not overlap.

The contact function therefore provides a criterion to check overlap of two ellipsoids. The PW85 library computes this value.

## 3.3 Geometric interpretation

The scalar  $\lambda$  being fixed, we introduce the minimizer  $x_\theta(\lambda)$  of  $F(x, \lambda)$ . The stationarity of  $F$  w.r.t to  $x$  reads:

$$(7) \quad \nabla F(x_\theta(\lambda), \lambda) = 0,$$

which leads to:

$$(8) \quad \lambda Q_1^{-1} \cdot [x_\theta(\lambda) - c_1] + (1 - \lambda) Q_2^{-1} \cdot [x_\theta(\lambda) - c_2] = 0,$$

and can be rearranged:

$$(9a) \quad x_\theta(\lambda) - c_1 = (1 - \lambda) Q_1 \cdot Q^{-1} \cdot r_{12},$$

$$(9b) \quad x_\theta(\lambda) - c_2 = -\lambda Q_2 \cdot Q^{-1} \cdot r_{12},$$

with:

$$(10) \quad Q = (1 - \lambda) Q_1 + \lambda Q_2.$$

It results from the above that:

$$(11) \quad f(\lambda) = F(x_\theta(\lambda), \lambda) = \lambda(1 - \lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12}.$$

Maximization of  $f$  with respect to  $\lambda$  now delivers the stationarity condition:

$$(12) \quad 0 = f'(\lambda) = \nabla F(x_\theta(\lambda), \lambda) \cdot x_\theta'(\lambda) + \frac{\partial F}{\partial \lambda}(x_\theta(\lambda), \lambda).$$

Using Eqs. (4) and (7), it is found that  $f$  is minimum for  $\lambda = \lambda_\theta$  such that:

$$(13) \quad [x_\theta(\lambda_\theta) - c_1]^\top \cdot Q_1^{-1} \cdot [x_\theta(\lambda_\theta) - c_1] = [x_\theta(\lambda_\theta) - c_2]^\top \cdot Q_2^{-1} \cdot [x_\theta(\lambda_\theta) - c_2].$$

Let  $\mu^2$  be this common value. It trivially results from Eqs. (4) and (13) that  $\mu^2 = F(x_\theta(\lambda_\theta), \lambda_\theta)$ . In other words,  $\mu^2$  is the value of the contact function.

We are now in a position to give a geometric interpretation of  $\mu$ . It results from Eq. (13) and the definition of  $\mu$  that:

$$(14a) \quad [x_\theta(\lambda_\theta) - c_1]^\top \cdot (\mu^2 Q_1)^{-1} \cdot [x_\theta(\lambda_\theta) - c_1] = 1,$$

and:

$$(14b) \quad [x_\theta(\lambda_\theta) - c_2]^\top \cdot (\mu^2 Q_2)^{-1} \cdot [x_\theta(\lambda_\theta) - c_2] = 1.$$

The above equations mean that  $x_\theta(\lambda_\theta)$  belongs to both ellipsoids centered at  $c_j$  and defined by the symmetric, positive-definite quadratic form  $\mu^2 Q_j$  ( $j = 1, 2$ ). These two ellipsoids are nothing but the initial ellipsoids  $E_1$  and  $E_2$ , scaled by the *same* factor  $\mu$ .

Furthermore, Eq. (8) applies for  $\lambda = \lambda_\theta$ . Therefore, the normals to the scaled ellipsoids coincide at  $x_\theta(\lambda_\theta)$ : the two scaled ellipsoids are externally tangent.

To sum up,  $\mu$  is the common factor by which ellipsoids  $E_1$  and  $E_2$  must be scaled in order for them to be externally tangent at point  $x_\theta(\lambda_\theta)$ .

## 3.4 References



## IMPLEMENTATION OF THE FUNCTION $f$

In this chapter, we explain how the contact function is computed. From Eq. (12) in chapter *Theory*, the value of the contact function is found from the solution  $\lambda$  to equation  $f'(\lambda) = 0$ , where it is recalled that  $f$  is defined as follows:

$$(1) \quad f(\lambda) = \lambda(1-\lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12},$$

with:

$$(2) \quad Q = (1-\lambda)Q_1 + \lambda Q_2.$$

In the present chapter, we discuss two implementations for the evaluation of  $f$ . The *first implementation* uses the Cholesky decomposition of  $Q$ . The *second implementation* uses a representation of  $f$  as a quotient of two polynomials (rational fraction).

### 4.1 Implementation #1: using Cholesky decompositions

Since  $Q$  is a symmetric, positive definite matrix, we can compute its *Cholesky decomposition*, which reads as follows:

$$(3) \quad Q = L \cdot L^T,$$

where  $L$  is a lower-triangular matrix. Using this decomposition, it is straightforward to compute  $s = Q^{-1} \cdot r$  (where we write  $r$  as a shorthand for  $r_{12}$ ), so that:

$$(4) \quad f(\lambda) = \lambda(1-\lambda) r^T \cdot s.$$

In order to solve  $f'(\lambda) = 0$  numerically, we use a *Newton–Raphson* procedure, which requires the first and second derivatives of  $f$ . It is readily found that:

$$(5) \quad s' = -Q^{-1} \cdot Q' \cdot Q^{-1} \cdot r = -Q^{-1} \cdot u \quad \text{and} \quad r^T \cdot s' = -r^T \cdot Q^{-1} \cdot u = -s^T \cdot u,$$

with  $u = Q_{12} \cdot s$  and  $Q_{12} = Q_2 - Q_1$ . Therefore:

$$(6) \quad f'(\lambda) = (1-2\lambda) r^T \cdot s - \lambda(1-\lambda) s^T \cdot u.$$

Similarly, introducing  $v = Q^{-1} \cdot u$ :

$$(7) \quad s^T \cdot u' = s^T \cdot Q_{12} \cdot s' = -s^T \cdot Q_{12} \cdot Q^{-1} \cdot u = -u^T \cdot v,$$

and:

$$(8) \quad u^T \cdot s' = -u^T \cdot Q^{-1} \cdot u = -u^T \cdot v.$$

Therefore:

$$(9) \quad f''(\lambda) = -2r^T \cdot s - 2(1-2\lambda)s^T \cdot u + 2\lambda(1-\lambda)u^T \cdot v.$$

## 4.2 Implementation #2: using rational functions

We observe that  $f(\lambda)$  is a rational function [see Eq. (1)], and we write:

$$(10) \quad f(\lambda) = \frac{\lambda(1-\lambda)a(\lambda)}{b(\lambda)},$$

with:

$$(11a) \quad a(\lambda) = r_{12}^T \cdot \text{adj}[(1-\lambda)Q_1 + \lambda Q_2] \cdot r_{12} = a_0 + a_1\lambda + a_2\lambda^2,$$

$$(11b) \quad b(\lambda) = \det[(1-\lambda)Q_1 + \lambda Q_2] = b_0 + b_1\lambda + b_2\lambda^2 + b_3\lambda^3,$$

where  $\text{adj}(Q)$  denotes the adjugate matrix of  $Q$  (transpose of its cofactor matrix), see e.g. [Wikipedia](#).

The coefficients  $a_i$  and  $b_i$  are found from the evaluation of  $a(\lambda)$  and  $b(\lambda)$  for specific values of  $\lambda$ :

$$(12a) \quad a_0 = a(0),$$

$$(12b) \quad a_1 = \frac{a(1) - a(-1)}{2},$$

$$(12c) \quad a_2 = \frac{a(1) + a(-1)}{2} - a(0),$$

$$(12d) \quad b_0 = b(0),$$

$$(12e) \quad b_1 = \frac{8b(\frac{1}{2})}{3} - 2b(0) - \frac{b(1)}{2} - \frac{b(-1)}{6}$$

$$(12f) \quad b_2 = \frac{b(1) + b(-1)}{2} - b(0),$$

$$(12g) \quad b_3 = -\frac{8b(\frac{1}{2})}{3} + 2b(0) + b(1) - \frac{b(-1)}{3}.$$

This requires the implementation of the determinant and the adjugate matrix of a  $3 \times 3$ , symmetric matrix, see `pw85__det_sym()` and `pw85__xT_adjA_x()`.

Evaluating the derivative of  $f$  with respect to  $\lambda$  is fairly easy. The following [SymPy](#) script will do the job:

```
import sympy

from sympy import Equality, numer, pprint, Symbol

if __name__ == '__main__':
    sympy.init_printing(use_latex=False, use_unicode=True)
```

(continues on next page)

(continued from previous page)

```

λ = Symbol('λ')
a = sum(sympy.Symbol('a{}'.format(i))*λ**i for i in range(3))
b = sum(sympy.Symbol('b{}'.format(i))*λ**i for i in range(4))
f = λ*(1-λ)*a/b
f_prime = f.diff(λ).ratsimp()
c = numer(f_prime)
c_dict = c.collect(λ, evaluate=False)
for i in range(sympy.degree(c, gen=λ)+1):
    pprint(Equality(Symbol('c{}'.format(i)), c_dict[λ**i]))

```

It is readily found that:

$$(13) \quad f'(\lambda) = \frac{c(\lambda)}{b(\lambda)^2},$$

where  $c(\lambda)$  is a sixth-order polynomial in  $\lambda$ :

$$(14) \quad c(\lambda) = c_0 + c_1\lambda + c_2\lambda^2 + c_3\lambda^3 + c_4\lambda^4 + c_5\lambda^5 + c_6\lambda^6,$$

with:

$$\begin{aligned}
 (15a) \quad c_0 &= a_0b_0, \\
 (15b) \quad c_1 &= 2(a_1-a_0)b_0, \\
 (15c) \quad c_2 &= -a_0(b_1+b_2) + 3b_0(a_2-a_1) + a_1b_1, \\
 (15d) \quad c_3 &= 2[b_1(a_2-a_1) - a_0b_3] - 4a_2b_0, \\
 (15e) \quad c_4 &= (a_0-a_1)b_3 + (a_2-a_1)b_2 - 3a_2b_1, \\
 (15f) \quad c_5 &= -2a_2b_2, \\
 (15g) \quad c_6 &= -a_2b_3,
 \end{aligned}$$

Solving  $f'(\lambda) = 0$  for  $\lambda$  is therefore equivalent to finding the unique root of  $c$  in the interval  $0 \leq \lambda \leq 1$ . For the sake of robustness, the `bisection method` has been implemented (more efficient methods will be implemented in future versions).

Once  $\lambda$  is found,  $\mu$  is computed from  $\mu^2 = f(\lambda)$  using Eq. (10).

## 4.3 Comparison of the two implementations

High precision reference data was generated using the `mpmath` library. The reference dataset is fully described and freely downloadable from the [Zenodo platform](https://zenodo.org/doi/10.5281/zenodo.3323683) (DOI:10.5281/zenodo.3323683). Accuracy of both implementations is then evaluated through the following script (download source file):

```

#include <glib.h>
#include < hdf5.h>
#include < hdf5_hl.h>
#include < pw85.h>
#include < pw85_legacy.h>

void read_dataset_double(hid_t const hid, char const *dset_name, size_t *size,
                        double **buffer) {
    int ndims;
    H5LTget_dataset_ndims(hid, dset_name, &ndims);
    hsize_t *dim = g_new(hsize_t, ndims);
    H5LTget_dataset_info(hid, dset_name, dim, NULL, NULL);
}

```

(continues on next page)

(continued from previous page)

```

*size = 1;
for (size_t i = 0; i < ndims; i++) {
    *size *= dim[i];
}
*buffer = g_new(double, *size);
H5LTread_dataset_double(hid, dset_name, *buffer);
g_free(dim);
}

void update_histogram(double act, double exp, size_t num_bins, size_t *hist) {
    double const err = fabs((act - exp) / exp);
    int prec;
    if (err == 0.0) {
        prec = num_bins - 1;
    } else {
        prec = (int)(floor(-log10(err)));
        if (prec <= 0) {
            prec = 0;
        }
        if (prec >= num_bins) {
            prec = num_bins - 1;
        }
    }
    ++hist[prec];
}

int main() {
    hid_t const hid = H5Fopen(PW85_REF_DATA_PATH, H5F_ACC_RDONLY, H5P_DEFAULT);

    size_t num_directions;
    double *directions;
    read_dataset_double(hid, "/directions", &num_directions, &directions);
    num_directions /= PW85_DIM;

    size_t num_lambdas;
    double *lambdas;
    read_dataset_double(hid, "/lambdas", &num_lambdas, &lambdas);

    size_t num_radai;
    double *radai;
    read_dataset_double(hid, "/radai", &num_radai, &radai);

    size_t num_spheroids;
    double *spheroids;
    read_dataset_double(hid, "/spheroids", &num_spheroids, &spheroids);
    num_spheroids /= PW85_SYM;

    size_t num_expecteds;
    double *expecteds;
    read_dataset_double(hid, "/F", &num_expecteds, &expecteds);

    double *exp = expecteds;
    double params[2 * PW85_SYM + PW85_DIM];

    size_t num_bins = 16;
    size_t hist1[num_bins];
    size_t hist2[num_bins];

```

(continues on next page)



(continued from previous page)

```

for (size_t i = 0; i < num_bins; i++) {
    hist1[i] = 0;
    hist2[i] = 0;
}
for (size_t i1 = 0; i1 < num_spheroids; i1++) {
    memcpy(params + PW85_DIM, spheroids + PW85_SYM * i1,
           PW85_SYM * sizeof(double));
    for (size_t i2 = 0; i2 < num_spheroids; i2++) {
        memcpy(params + PW85_DIM + PW85_SYM, spheroids + PW85_SYM * i2,
               PW85_SYM * sizeof(double));
        for (size_t i = 0; i < num_directions; i++) {
            memcpy(params, directions + PW85_DIM * i, PW85_DIM * sizeof(double));
            for (size_t j = 0; j < num_lambdas; j++, exp++) {
                double const act1 = -pw85_f_neg(lambdas[j], params);
                update_histogram(act1, *exp, num_bins, hist1);
                double out[2];
                pw85_legacy_f2(lambdas[j], params, params + PW85_DIM,
                              params + PW85_DIM + PW85_SYM, out);
                double const act2 = out[0];
                update_histogram(act2, *exp, num_bins, hist2);
            }
        }
    }
}

FILE *f = fopen(HISTOGRAM_PATH, "w");
for (size_t i = 0; i < num_bins; i++) {
    fprintf(f, "%d,%g,%g\n", (int)i,
           100. * ((double)hist1[i]) / ((double)num_expecteds),
           100. * ((double)hist2[i]) / ((double)num_expecteds));
}
fclose(f);

g_free(spheroids);
g_free(radii);
g_free(lambdas);
g_free(directions);
H5Fclose(hid);

return 0;
}

```

**Note:** To compute this program, you might need to pass the options `-Dpw85_include=...`, `-Dpw85_lib=...` and `-Dpw85_data=...` to meson (see [C tutorial](#)).

We get the histograms shown in [Fig. 4.1](#). These histograms show that *Implementation #1* is more accurate than *Implementation #2*. The former will therefore be selected as default.

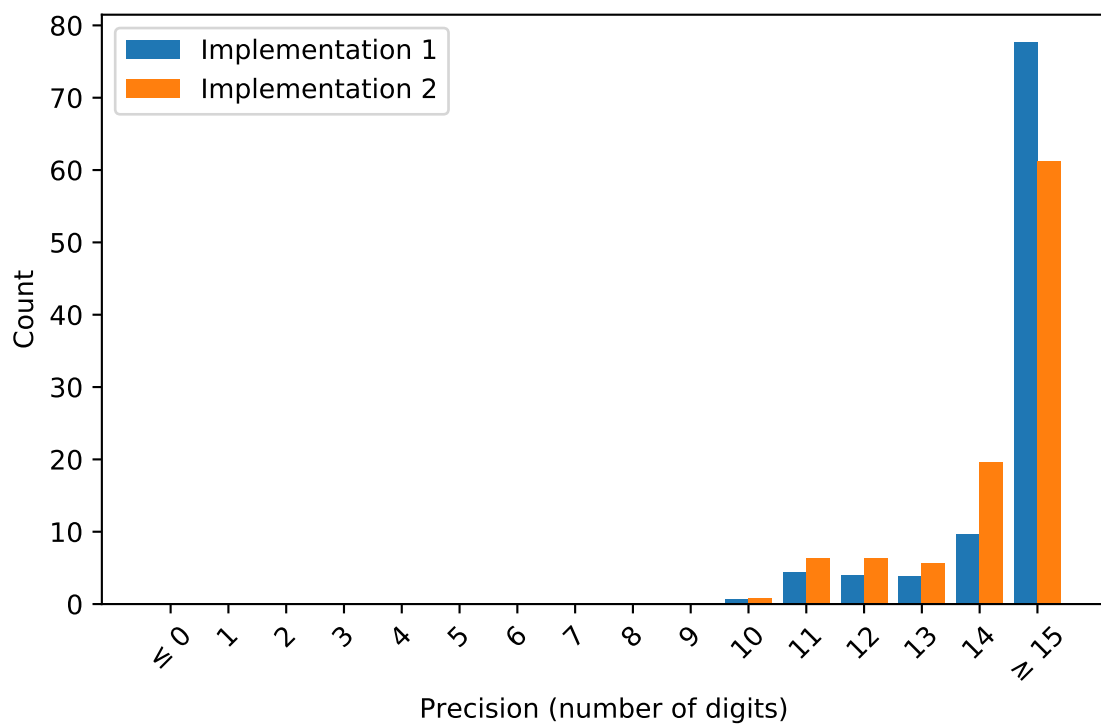


Fig. 4.1: Accuracy of the two implementations.

## OPTIMIZATION OF THE FUNCTION $f$

It was shown in chapter *Theory* [see Eq. (6)] that the contact function was defined as the maximum for  $0 \leq \lambda \leq 1$  of the function  $f$  discussed in chapter *Implementation of the function  $f$* .

Given that the first and second derivatives of  $f$  can be computed explicitly (see section *Implementation #1: using Cholesky decompositions* in chapter *Implementation of the function  $f$* ) it would be tempting to use the Newton–Raphson method to solve  $f'(\lambda)$  iteratively. However, our experiments show that this method performs very poorly in the present case, because the variations of  $f$  can be quite sharp in the neighborhood of  $\lambda = 0$  or  $\lambda = 1$ . To carry out the optimization of  $f$ , we therefore proceed in two steps.

In the first step, we use a robust optimization algorithm. We selected here *Brent’s method*, as implemented in the *GNU Scientific Library (GSL)*. However, this method delivers a relatively low accuracy of the maximizer and the maximum.

Therefore, in the second step, we use a few Newton–Raphson iterations to refine the previously obtained estimates of the minimizer and minimum of  $f$ . In the remainder of this chapter, we describe how these Newton–Raphson iterations are performed.

Our starting point is Eqs. (9) and (13) in chapter *Theory*, from which it results that for a given value of  $\lambda$  we can define two values of  $\mu^2$ : one is provided by Eq. (9a), the other one is given by Eq. (9b) (both in chapter *Theory*):

$$\begin{aligned} (1a) \quad \mu_1^2 &= [x_0(\lambda_0) - c_1]^T \cdot Q_1^{-1} \cdot [x_0(\lambda_0) - c_1] = (1-\lambda)^2 s^T \cdot Q_1 \cdot s, \\ (1b) \quad \mu_2^2 &= [x_0(\lambda_0) - c_2]^T \cdot Q_2^{-1} \cdot [x_0(\lambda_0) - c_2] = \lambda^2 s^T \cdot Q_2 \cdot s, \end{aligned}$$

where we have introduced  $s = Q^{-1} \cdot r_{12}$ . We further define the matrix  $Q_{12} = Q_2 - Q_1$ , so that:

$$(2) \quad Q_1 = Q - \lambda Q_{12} \quad \text{and} \quad Q_2 = Q + (1-\lambda) Q_{12}.$$

Combining Eqs. (1) and (2) and recalling that  $Q \cdot s = r$  then delivers the following expressions:

$$\begin{aligned} (3a) \quad \mu_1^2 &= (1-\lambda)^2 r^T \cdot s - \lambda(1-\lambda)^2 s^T \cdot u, \\ (3b) \quad \mu_2^2 &= \lambda^2 r^T \cdot s + \lambda^2(1-\lambda) s^T \cdot u, \end{aligned}$$

where we have introduced  $u = Q_{12} \cdot s$ .

The above expressions seem to behave slightly better from a numerical point of view. Our problem is now to find  $\lambda$  such that  $\mu_1^2 = \mu_2^2$ . We therefore define the following residual:

$$(4) \quad g(\lambda) = \mu_2^2 - \mu_1^2 = (2\lambda-1) r^T \cdot s + \lambda(1-\lambda) s^T \cdot u,$$

and we need to find  $\lambda$  such that  $g(\lambda) = 0$ . In order to implement Newton–Raphson iterations, we need the expression of the derivative of the residual. Using results that are presented in section *Implementation #1: using Cholesky decompositions*, we readily find that:

$$(5) \quad g'(\lambda) = 2r^T \cdot s + 2(1-2\lambda) s^T \cdot u - 2\lambda(1-\lambda) u^T \cdot v.$$

Eqs. (4) and (5) are then used for the final, refinement step of determination of  $\lambda$ .

## TESTING THE IMPLEMENTATION OF THE CONTACT FUNCTION

This chapter describes how our implementation of the contact function is tested. The source of the unit tests can be found in the file `src/test_pw85.c`. Note that the tests described here are repeated over a large set of tests case, including very flat and very slender spheroids, for various relative orientations and center-to-center distances.

In the present chapter, we assume that the two ellipsoids (their matrices  $Q_1$  and  $Q_2$  are given), as well as their center-to-center radius vector  $r_{12}$ . Then, a call to `pw85_contact_function()` delivers an estimate of  $\lambda$  and  $\mu^2$ .

We first assert that  $\mu_1^2$  and  $\mu_2^2$  as defined by Eq. (3) in chapter *Optimization of the function  $f$*  are close to the value returned by `pw85_contact_function()`. For all the cases considered here, this is true up to a relative error of  $10^{-10}$ .

We also check that  $f'(\lambda) = 0$ , up to an absolute error of  $\Delta\lambda f''(\lambda)$  where  $\Delta\lambda$  is the absolute tolerance on  $\lambda$  for the stopping criterion of the Brent iterations, as defined by the macro `PW85_LAMBDA_ATOL`.

---

**Note:** The PW85 library offers a “new” API and a “legacy” API. The latter is kept for reference. The new API is generally more accurate and robust, and should be preferred by most users. Both APIs are thoroughly tested; however, we adopted two different testing strategies.

The legacy API is solely (but fully) tested through its Python wrapper using `pytest`.

The new API is fully tested through pure C tests (using `GLib`). Then the Python wrapper is also tested (using `pytest`). However, the python tests do not need to be as thorough, since only the validity of the wrapper itself must be checked, not the validity of the underlying C library.

---



## THE C API

### Contents

- *Representation of vectors and matrices*
- *The new API*
- *The “legacy” API*

---

**Note:** we use the following naming convention

- “public” functions are prefixed with `pw85_` or `pw85_legacy_` (single underscore),
- “private” functions are prefixed with `pw85__` or `pw85_legacy__` (double underscore).

Note that “public” and “private” is a matter of convention here, since all functions are exposed (mostly, for testing purposes). However, double underscored functions should not be considered as part of the public API and should not be used, since they are susceptible of incompatible changes (or even removal) in future versions.

---

## 7.1 Representation of vectors and matrices

An ellipsoid is defined from its center  $c$  (a  $3 \times 1$ , column-vector) and quadratic form  $Q$  (a  $3 \times 3$ , symmetric, positive definite matrix) as the set of points  $m$  such that:

$$(m - c)^T \cdot Q^{-1} \cdot (m - c) \leq 1.$$

In this module, objects referred to as “vectors” are `double[3]` arrays of coordinates. In other words, the representation of the vector  $x$  is the `double[3]` array  $x$  such that:

$$x = \begin{bmatrix} x[0] \\ x[1] \\ x[2] \end{bmatrix}.$$

Objects referred to as “symmetric matrices” (or “quadratic forms”) are of type `double[6]`. Such arrays list in row-major order the coefficients of the triangular upper part. In other words, the representation of a the symmetric matrix  $A$  is the `double[6]` array  $a$  such that:

$$A = \begin{bmatrix} a[0] & a[1] & a[2] \\ & a[3] & a[4] \\ \text{sym.} & & a[5] \end{bmatrix}.$$

## 7.2 The new API

The functions and macros gathered below form the new API that should be invoked by most users. To use these functions and macros in your code, you must include the following header:

```
#include <pw85.h>
```

and use the following link directive:

```
-lpw85
```

### **PW85\_VERSION**

The current version of the library.

### **PW85\_DIM**

The dimension of the physical space (3).

### **PW85\_SYM**

The dimension of the space of symmetric matrices (6).

### **PW85\_LAMBDA\_ATOL**

The absolute tolerance for the stopping criterion of Brent's method (in function `pw85_contact_function()`).

### **PW85\_MAX\_ITER**

The maximum number of iterations of Brent's method (in function `pw85_contact_function()`).

### **PW85\_NR\_ITER**

The total number of iterations of the Newton–Raphson refinement phase (in function `pw85_contact_function()`).

void **pw85\_\_cholesky\_decomp**(double const *a*[PW85\_SYM], double *l*[PW85\_SYM])

Compute the Cholesky decomposition of a symmetric, positive matrix.

Let *A* be a symmetric, positive matrix, defined by the `double`[6] array *a*. This function computes the lower-triangular matrix *L*, defined by the `double`[6] array *l*, such that  $L^T \cdot L = A$ .

The array *l* must be pre-allocated; it is modified by this function. Note that storage of the coefficients of *L* is as follows:

$$L = \begin{bmatrix} l[0] & 0 & 0 \\ l[1] & l[3] & 0 \\ l[2] & l[4] & l[5] \end{bmatrix}.$$

void **pw85\_\_cholesky\_solve**(double const *l*[PW85\_SYM], double const *b*[PW85\_DIM], double *x*[PW85\_DIM])

Compute the solution to a previously Cholesky decoposed linear system.

Let *L* be a lower-triangular matrix, defined by the `double`[6] array *l* (see `pw85__cholesky_decomp()` for ordering of the coefficients). This function solves (by substitution) the linear system  $L^T \cdot L \cdot x = b$ , where the vectors *x* and *b* are specied through their `double`[3] array of coordinates; *x* is modified by this function.

void **pw85\_\_residual**(double *lambda*, double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double *out*[3])

Compute the residual  $g(\lambda) = \mu_2^2 - \mu_1^2$ .

See [Optimization of the function \*f\*](#) for the definition of *g*. The value of  $\lambda$  is specified through the parameter *lambda*. See `pw85_contact_function()` for the definition of the parameters *r12*, *q1* and *q2*.

The preallocated `double`[3] array *out* is updated with the values of  $f(\lambda)$ ,  $g(\lambda)$  and  $g'(\lambda)$ :



$$\text{out}[0] = f(\lambda), \quad \text{out}[1] = g(\lambda) \quad \text{and} \quad \text{out}[2] = g'(\lambda).$$

This function is used in function `pw85_contact_function()` for the final Newton–Raphson refinement step.

void **pw85\_spheroid**(double *a*, double *c*, double *n*[PW85\_DIM], double *q*[PW85\_SYM])

Compute the quadratic form associated to a spheroid.

The spheroid is defined by its equatorial radius *a*, its polar radius *c* and the direction of its axis of revolution, *n*.

*q* is the representation of a symmetric matrix as a double[6] array. It is modified in-place.

double **pw85\_f\_neg**(double *lambda*, double cons\* *params*)

Return the value of the opposite of the function *f* defined as (see [Theory](#)):

$$f(\lambda) = \lambda(1-\lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12},$$

with:

$$Q = (1-\lambda)Q_1 + \lambda Q_2,$$

where ellipsoids 1 and 2 are defined as the sets of points *m* (column-vector) such that:

$$(m - c_i) \cdot Q_i^{-1} \cdot (m - c_i) \leq 1$$

In the above inequality, *c<sub>i</sub>* is the center; *r<sub>12</sub>* = *c<sub>2</sub>* - *c<sub>1</sub>* is the center-to-center radius-vector, represented by the first 3 coefficients of the array *params*. The symmetric, positive-definite matrices *Q<sub>1</sub>* and *Q<sub>2</sub>* are specified through the next 12 coefficients. In other words, if *r12*, *Q1* and *Q2* were defined as usual by their double[3], double[6] and double[6] arrays *r12*, *q1* and *q2*, then *params* would be formed as follows:

```
double params[] = {r12[0], r12[1], r12[2],
                  q1[0], q1[1], q1[2], q1[3], q1[4], q1[5],
                  q2[0], q2[1], q2[2], q2[3], q2[4], q2[5]};
```

The value of *λ* is specified through the parameter *lambda*.

This function returns the value of  $-f(\lambda)$  (the “minus” sign comes from the fact that we seek the maximum of *f*, or the minimum of  $-f$ ).

This implementation uses [Cholesky decompositions](#). Its somewhat awkward signature is defined in accordance with `gsl_min.h` from the GNU Scientific Library.

int **pw85\_contact\_function**(double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double *out*[2])

Compute the value of the contact function of two ellipsoids.

The center-to-center radius-vector is specified by the double[3] array *r12*. The symmetric, positive-definite matrices *Q<sub>1</sub>* and *Q<sub>2</sub>* that define the two ellipsoids are specified through the double[6] arrays *q1* and *q2*.

This function returns the value of  $\mu^2$ , defined as (see [Theory](#)):

$$\mu^2 = \max\{ \lambda(1-\lambda) r_{12}^T \cdot [(1-\lambda)Q_1 + \lambda Q_2]^{-1} \cdot r_{12}, \quad 0 \leq \lambda \leq 1 \},$$

and the maximizer *λ*. Both values are stored in the preallocated double[2] array *out*:

$$\text{out}[0] = \mu^2 \quad \text{and} \quad \text{out}[1] = \lambda.$$

*μ* is the common factor by which the two ellipsoids must be scaled (their centers being fixed) in order to be tangentially in contact.

This function returns 0

---

**Todo:** This function should return an error code.

---

## 7.3 The “legacy” API

The functions described below belong to the legacy API. These are functions that have been superseded by equivalent (more accurate or more efficient) implementations in the core library. To use these functions in your code, you must include the following header:

```
#include <pw85_legacy.h>
```

and use the following link directive:

```
-lpw85_legacy
```

double **pw85\_legacy\_\_det\_sym**(double *a*[PW85\_SYM])

Return the determinant of A.

The symmetric matrix A is specified through the double[6] array *a*.

double **pw85\_legacy\_\_xT\_adjA\_x**(double *x*[PW85\_DIM], double *a*[PW85\_SYM])

Return the product  $x^T \cdot \text{adj}(A) \cdot x$ .

The column vector *x* is specified as a double[3] array. The symmetric matrix A is specified through the double[6] array *a*.

$\text{adj}(A)$  denotes the adjugate matrix of A (transpose of its cofactor matrix), see e.g. [Wikipedia](#).

void **pw85\_legacy\_\_detQ\_as\_poly**(double *q1*[PW85\_SYM], double *q2*[PW85\_SYM], double *q3*[PW85\_SYM], double *q4*[PW85\_SYM], double *b*[PW85\_DIM+1])

Compute the coefficients of the polynomial  $\lambda \mapsto \det[(1-\lambda)Q_1 + \lambda Q_2]$ .

The symmetric, positive definite, 3×3 matrices  $Q_1$  and  $Q_2$  are specified as arrays *q1* and *q2*. The arrays *q3* and *q4* must hold the difference  $2Q_1 - Q_2$  and average  $(Q_1 + Q_2)/2$ , respectively:

```
q3[i] = 2*q1[i] - q2[i] and q4[i] = 0.5*(q1[i] + q2[i]),
```

for  $i = 0, \dots, \text{PW85\_SYM}-1$ . The returned polynomial has degree *PW85\_DIM*:

```
det[(1-λ)Q1+λQ2] = b0 + b1λ + b2λ2 + b3λ3.
```

The coefficients  $b_i$  are stored in *b* in *increasing* order:  $b[i] = b_i$ .

double **pw85\_\_rT\_adjQ\_r\_as\_poly**(double *r*[PW85\_DIM], double *q1*[PW85\_SYM], double *q2*[PW85\_SYM], double *q3*[PW85\_SYM], double *a*[PW85\_DIM])

Compute the coefficients of the polynomial  $\lambda \mapsto r^T \cdot \text{adj}[(1-\lambda)Q_1 + \lambda Q_2] \cdot r$ .

The symmetric, positive definite, 3×3 matrices  $Q_1$  and  $Q_2$  are specified as arrays *q1* and *q2*. The array *q3* must hold the difference  $2Q_1 - Q_2$ :

```
q3[i] = 2*q1[i] - q2[i],
```

for  $i = 0, \dots, \text{PW85\_SYM}-1$ . The returned polynomial has degree  $\text{PW85\_DIM} - 1$ :

```
rT·adj[(1-λ)Q1+λQ2]·r = a0 + a1λ + a2λ2.
```

The coefficients  $a_i$  are stored in  $a$  in *increasing* order:  $a[i] = a_i$ .

double **pw85\_legacy\_f1**(double *lambda*, double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double\* *out*)

Return the value of the function  $f$  defined as (see [Theory](#)):

$$f(\lambda) = \lambda(1-\lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12},$$

with:

$$Q = (1-\lambda)Q_1 + \lambda Q_2,$$

where ellipsoids 1 and 2 are defined as the sets of points  $m$  (column-vector) such that:

$$(m - c_i) \cdot Q_i^{-1} \cdot (m - c_i) \leq 1$$

In the above inequality,  $c_i$  is the center;  $r_{12} = c_2 - c_1$  is the center-to-center radius-vector, represented by the double[3] array *r12*. The symmetric, positive-definite matrices  $Q_1$  and  $Q_2$  are specified through the double[6] arrays *q1* and *q2*.

The value of  $\lambda$  is specified through the parameter *lambda*.

This function returns the value of  $f(\lambda)$ . If *out* is not NULL, then it must be a pre-allocated double[3] array which is updated with the values of the first and second derivatives:

$$\text{out}[0] = f(\lambda), \quad \text{out}[1] = f'(\lambda) \quad \text{and} \quad \text{out}[2] = f''(\lambda).$$

This implementation uses [Cholesky decompositions](#).

double **pw85\_legacy\_f2**(double *lambda*, double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double\* *out*)

Alternative implementation of [pw85\\_legacy\\_f1\(\)](#).

See [pw85\\_legacy\\_f1\(\)](#) for the meaning of the parameters *lambda*, *r12*, *q1* and *q2*.

This function returns the value of  $f(\lambda)$ . If *out* is not NULL, then it must be a pre-allocated double[1] array which is updated with the value of  $f(\lambda)$ .

This implementation uses [rational fractions](#).

---

**Todo:** This function should also compute the first and second derivatives.

---

int **pw85\_legacy\_contact\_function1**(double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double *out*[2])

Compute the value of the contact function of two ellipsoids.

See [pw85\\_contact\\_function\(\)](#) for the invocation of this function.

Implementation of this function relies on Newton–Raphson iterations on  $f$ ; it is not robust.

This function returns 0

---

**Todo:** This function should return an error code.

---

int **pw85\_legacy\_contact\_function2**(double const *r12*[PW85\_DIM], double const *q1*[PW85\_SYM], double const *q2*[PW85\_SYM], double *out*[2])

Compute the value of the contact function of two ellipsoids.

See [pw85\\_contact\\_function\(\)](#) for the invocation of this function.

This implementation uses the representation of  $f$  as *rational fractions*. Finding the maximum of  $f$  is then equivalent to finding the root of the numerator of the rational fraction of  $f'$ . For the sake of robustness, bisection is used to compute this root.

This function returns 0

---

**Todo:** This function should return an error code.

---

## THE PYTHON API

### Contents

- *The new API*
  - *Representation of vectors and matrices*
- *The “legacy” API*

## 8.1 The new API

Overlap test of two ellipsoids.

This module provides a wrapper around the PW85 C library that implements the “contact function” defined by Perram and Wertheim (J. Comp. Phys. 58(3), 409–416, DOI:10.1016/0021-9991(85)90171-8) for two ellipsoids. Given two ellipsoids, this function returns the *square* of the common factor by which both ellipsoids must be scaled (their centers being fixed) in order to be tangentially in contact.

This module is released under a BSD 3-Clause License.

### 8.1.1 Representation of vectors and matrices

An ellipsoid is defined from its center  $c$  (a  $3 \times 1$ , column-vector) and quadratic form  $Q$  (a  $3 \times 3$ , symmetric, positive definite matrix) as the set of points  $m$  such that:

$$(m - c)^T \cdot Q^{-1} \cdot (m - c) \leq 1.$$

In this module, objects referred to as “vectors” are length-3 arrays of `double` coordinates. In other words, the representation of the vector  $x$  is the `double[3]` array  $x$  such that:

$$x = \begin{bmatrix} x[0] \\ x[1] \\ x[2] \end{bmatrix}.$$

Objects referred to as “symmetric matrices” (or “quadratic forms”) are length-6 arrays of `double`. Such arrays list in row-major order the coefficients of the triangular upper part. In other words, the representation of a the symmetric matrix  $A$  is the array  $a$  such that:

$$A = \begin{bmatrix} a[0] & a[1] & a[2] \\ & a[3] & a[4] \\ \text{sym.} & & a[5] \end{bmatrix}.$$

The present wrapper around the PW85 C library relies on the NumPy library. “double[n] array” should be understood here as “NumPy array with shape == (n,) and dtype == numpy.float64.”

Note that true NumPy array *must* be passed (array-likes will *not* work).

`pypw85._cholesky_decomp(a, l=None)`

Compute the Cholesky decomposition  $A = L \cdot L^T$  of a 3×3 matrix.

A is a symmetric matrix, L is a lower matrix, both represented by double[6] arrays.

This function returns l, suitably updated with the coefficients of the Cholesky decomposition. If l is None, then a new array is allocated.

`pypw85._cholesky_solve(l, b, x=None)`

Compute the solution of the 3×3 linear system  $L \cdot L^T \cdot x = b$ .

L is a lower matrix, represented by the double[6] array l; x and b are vectors (double[3] arrays).

This function returns x, suitably updated with the solution to the system. If x is None, then a new array is allocated.

`pypw85.contact_function(r12, q1, q2, out=None)`

Return the value of the contact function of two ellipsoids.

See [f\(\)](#) for the meaning of the parameters r12, q1 and q2.

This function returns the pair  $(\mu^2, \lambda)$ , defined as (see [Theory](#)):

$$\mu^2 = \max\{ \lambda(1-\lambda) r_{12}^T \cdot [(1-\lambda)Q_1 + \lambda Q_2]^{-1} \cdot r_{12}, \ 0 \leq \lambda \leq 1 \}$$

(the returned value of  $\lambda$  is the actual maximizer).

$\mu$  is the common factor by which the two ellipsoids must be scaled (their centers being fixed) in order to be tangentially in contact.

If out is not None, it must be a pre-allocated double[2] array. It is updated with the values of  $\mu^2$ , and the maximizer  $\lambda$ :

$$\text{out}[0] = \mu^2 \quad \text{and} \quad \text{out}[1] = \lambda.$$

`pypw85.f(lambda_, r12, q1, q2)`

Return the value of the function f defined as:

$$f(\lambda) = \lambda(1-\lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12},$$

with:

$$Q = (1-\lambda)Q_1 + \lambda Q_2,$$

where ellipsoids 1 and 2 are defined as the sets of points m (column-vector) such that:

$$(m - c_i) \cdot Q_i^{-1} \cdot (m - c_i) \leq 1$$

In the above inequality,  $c_i$  is the center;  $r_{12} = c_2 - c_1$  is the center-to-center radius-vector, represented by the double[3] array r12. The symmetric, positive-definite matrices  $Q_1$  and  $Q_2$  are specified through the double[6] arrays q1 and q2.

The value of  $\lambda$  is specified through the parameter `lambda_`.

This function returns the value of  $f(\lambda)$ .

`pypw85.spheroid(a, c, n, q=None)`

Return the quadratic form associated to a spheroid.

The spheroid is defined by its equatorial radius  $a$ , its polar radius  $c$  and the direction of its axis of revolution,  $n$  (vector, a.k.a. `double[3]` array).

If  $q$  is not `None`, then it must be a pre-allocated `double[6]` array. It is modified in place.

## 8.2 The “legacy” API

Python wrapper to the legacy API of PW85.

This module offers some implementations of the function  $f$  that were eventually discarded (for accuracy reasons). These functions are kept for reference.

The present wrapper around the legacy PW85 C library relies on the NumPy library. “`double[n]` array” should be understood here as “NumPy array with `shape == (n,)` and `dtype == numpy.float64`.”

Note that true NumPy array *must* be passed (array-likes will *not* work).

`pypw85.legacy._detQ_as_poly(q1, q2, q3=None, q4=None, b=None)`

Compute the coefficients of the polynomial  $\lambda \mapsto \det[(1-\lambda)Q_1 + \lambda Q_2]$ .

The symmetric, positive definite,  $3 \times 3$  matrices  $Q_1$  and  $Q_2$  are specified as arrays  $q1$  and  $q2$ . If  $q3$  is not `None`, it must hold the difference  $2Q_1 - Q_2$ ; if  $q4$  is not `None`, it must hold the average  $(Q_1 + Q_2)/2$ :

```
q3[i] = 2*q1[i] - q2[i] and q4[i] = 0.5*(q1[i] + q2[i]),
```

for  $i = 0, \dots, 5$ . The returned polynomial has degree 3:

$$\det[(1-\lambda)Q_1 + \lambda Q_2] = b_0 + b_1\lambda + b_2\lambda^2 + b_3\lambda^3.$$

If  $b$  is not `None`, it must be a pre-allocated `double[4]` array. It is modified in place with the coefficients  $b_i$ , stored in *increasing* order:  $b[i] = b_i$ .

If  $b$  is `None`, a new `double[4]` array is created and returned.

`pypw85.legacy._det_sym(a)`

Return the determinant of  $A$ .

The symmetric matrix  $A$  is specified through the `double[6]` array  $a$ .

`pypw85.legacy._rT_adjQ_r_as_poly(r, q1, q2, q3=None, a=None)`

Compute the coefficients of the polynomial  $\lambda \mapsto r^T \cdot \text{adj}[(1-\lambda)Q_1 + \lambda Q_2] \cdot r$ .

The symmetric, positive definite,  $3 \times 3$  matrices  $Q_1$  and  $Q_2$  are specified as arrays  $q1$  and  $q2$ . If  $q3$  is not `None`, it must hold the difference  $2Q_1 - Q_2$ :

```
q3[i] = 2*q1[i] - q2[i],
```

for  $i = 0, \dots, 5$ . The returned polynomial has degree 2:

$$r^T \cdot \text{adj}[(1-\lambda)Q_1 + \lambda Q_2] \cdot r = a_0 + a_1\lambda + a_2\lambda^2.$$

If  $a$  is not `None`, it must be a pre-allocated `double[3]` array. It is modified in place with the coefficients  $a_i$ , stored in *increasing* order:  $a[i] = a_i$ . The function returns  $a$ .

If  $a$  is `None`, a new `double[3]` array is created and returned.

`pypw85.legacy._xT_adjA_x(x, a)`

Return the product  $x^T \cdot \text{adj}(A) \cdot x$ .

The column vector  $x$  is specified as a `double[3]` array. The symmetric matrix  $A$  is specified through the `double[6]` array  $a$ .

$\text{adj}(A)$  denotes the adjugate matrix of  $A$  (transpose of its cofactor matrix), see e.g. [Wikipedia](#).

`pypw85.legacy.f1(lambda_, r12, q1, q2, out=None)`

Return the value of the function  $f$  defined as (see [Theory](#)):

$$f(\lambda) = \lambda(1-\lambda) r_{12}^T \cdot Q^{-1} \cdot r_{12},$$

with:

$$Q = (1-\lambda)Q_1 + \lambda Q_2,$$

where ellipsoids 1 and 2 are defined as the sets of points  $m$  (column-vector) such that:

$$(m - c_i) \cdot Q_i^{-1} \cdot (m - c_i) \leq 1$$

In the above inequality,  $c_i$  is the center;  $r_{12} = c_2 - c_1$  is the center-to-center radius-vector, represented by the `double[3]` array `r12`. The symmetric, positive-definite matrices  $Q_1$  and  $Q_2$  are specified through the `double[6]` arrays `q1` and `q2`.

The value of  $\lambda$  is specified through the parameter `lambda_`.

This function returns the value of  $f(\lambda)$ . If `out` is not `None`, then it must be a pre-allocated `double[3]` array which is updated with the values of the first and second derivatives:

$$\text{out}[0] = f(\lambda), \quad \text{out}[1] = f'(\lambda) \quad \text{and} \quad \text{out}[2] = f''(\lambda).$$

This implementation uses [Cholesky decompositions](#).

`pypw85.legacy.f2(lambda_, r12, q1, q2, out=None)`

Alternative implementation of [f1\(\)](#).

See [f1\(\)](#) for the meaning of the parameters `lambda_`, `r12`, `q1` and `q2`.

This function returns the value of  $f(\lambda)$ . If `out` is not `None`, then it must be a pre-allocated `double[1]` array which is updated with the value of  $f(\lambda)$ .

This implementation uses [rational fractions](#).



## BIBLIOGRAPHY

- [PW85] Perram, J. W., & Wertheim, M. S. (1985). Statistical mechanics of hard ellipsoids. I. Overlap algorithm and the contact function. *Journal of Computational Physics*, 58(3), 409–416. [https://doi.org/10.1016/0021-9991\(85\)90171-8](https://doi.org/10.1016/0021-9991(85)90171-8)



## PYTHON MODULE INDEX

### p

`pypw85`, [33](#)  
`pypw85.legacy`, [35](#)