Computer project

World average values of neutral B_s meson parameters for the PDG

Dawid Gerstel

supervisor: Olivier Leroy

February 19, 2017

*This work has been carried out thanks to the support of the A*MIDEX grant (n ANR-11-IDEX-0001-02) funded by the French Government Investissements d'Avenir program.*

1 Introduction

The key world average values of the neutral B_s meson parameters are computed yearly by the Heavy Flavour Averaging Group (HFAG) [1] for the Particle Data Group (PDG). These parameters include the B_s lifetimes and decay width differences, as well as the CP-violating phase ϕ_s . Those averages have been computed so far using the Mathematica software. The objective of this project was to translate the Mathematica code to Python, as it is an open source language and, therefore, does not require any licence agreement unlike Mathematica. Also, the implementation had to be improved, for example, by avoiding code duplication whenever possible and the resulting program should be efficient and transparent. The world averages studied were $\Delta\Gamma_s$ with respect to (w.r.t.) ϕ_s , $\Delta\Gamma_s$ w.r.t. Γ_s and $1/\Gamma_H$ w.r.t. $1/\Gamma_L$. The results come from the experiments: ATLAS, CMS, CDF, D0 and LHCb.

2 Original application overview

The original (Mathematica) project is comprised of two macros:

- `DGsphis_Summer2016_zoom.nb` - which performs analysis of $\Delta\Gamma_s$ w.r.t. ϕ_s ,
- `DGsGs_worldSummer2016.nb` - which performs analysis of $\Delta\Gamma_s$ with respect to Γ_s as well as $1/\Gamma_H$ w.r.t. $1/\Gamma_L$.

Results of these macros were published in December 2016. [2]

In both cases the functionality looks similar and consists of the following:

- Inputting the values of parameters under study from various experiments. The parameters are mostly hard-coded within the code with the exception of LHCb data that are input from a text file. Usually, these values correspond to parameter's: mean value, systematic and statistical uncertainties, and correlation coefficient w.r.t. the other parameter under study (e.g. correlation of Γ_s and $\Delta\Gamma_s$).
- Preprocessing the data - some parameters are deducible from the input data, e.g., combined uncertainty.
- Implementing probability density functions of the parameters. These are, usually, two dimensional Gaussian distributions.
- Computing maximum likelihood functions based on probability density functions from all experiments considered.
- Minimising the parameters under study and outputting the results to respective text files.
- Drawing contour plots, both – for optimised values and for each experiment (or analysis type) separately – at a given confidence level and saving them.

The original code is quite unclear due to multiple duplications and obsolete comments.

The steps above will be addressed in the following sections describing the project in Python.

[1] http://www.slac.stanford.edu/xorg/hfag/osc/summer_2016/

[2] https://arxiv.org/pdf/1612.07233.pdf (sections 3.3.2 and 3.3.4)

3 Project evolution

The starting point was deciding to base the project on: Python2, `matplotlib` library for visualisation and `numpy` for extremely efficient numerical operations on arrays. Python3 could have been used instead, but in due course it turned out that the – described later – `iminuit` library is obsolete for many versions of Python3. [3] The choice of `matplotlib`, in turn, was obvious as it is a very powerful and well-maintained tool for graphics.

Because the two macros have similar functionality and are run during same annual updates – I chose to contain them in one program.

3.1 Input and data preprocessing

There are two ways of inputting data – from a separate file that contains them all or by hard-coding the values line by line. At first, the former way seemed much clearer as the user is not distracted by redundant code. Nonetheless, I decided to choose the latter solution for a number of reasons. First of all, the data do not come all at the same time – each experiment sends their parameters independently and it is often helpful if not necessary to comment on specific parameter, e.g., by pointing to an article from which it was taken. Writing the values line-by-line allows for easier comments than in the case of 2-dimensional array of numbers (one axis – experiment, the other – various parameters). Also, inputting parameters and updating them is a task of the same person (or team) as the subsequent computations and it is, therefore, more convenient to have all information at one place. Besides, experiments differ in providing data and some give results that need to be uniquely reshaped.

The original code stores every parameter as a separate real number variable, which results in unnecessary duplication of operations on the variables. I put the parameters inside a two-dimensional dictionary with experiments and parameters as the keys, as demonstrated in the excerpt below (that pertains to the ATLAS experiment). This solution, on the one hand is more compact and allows for iterative computations, and on the other hand – is clearer than array with no keys, but just indices.

Listing 1: Input parameters

```
# === ATLAS params ===
# taking this from altas run1 published paper (concerns phis vs DGs)
val["ATLAS"]['phis']               = -0.090
val["ATLAS"]['phis_estat']         = 0.078
val["ATLAS"]['phis_esyst']         = 0.041
val["ATLAS"]['Gs']                 = .675
val["ATLAS"]['Gs_estat']           = .003
val["ATLAS"]['Gs_esyst']           = .003
val["ATLAS"]['DGs']                = .085
val["ATLAS"]['DGs_estat']          = .011
val["ATLAS"]['DGs_esyst']          = .007
val["ATLAS"]['rho_Gs_DGs_stat']    = -.414
val["ATLAS"]['rho_Gs_DGs_syst']    = 0
val["ATLAS"]['rho_phis_DGs_stat']  = 0.097
val["ATLAS"]['rho_phis_DGs_syst']  = 0  # assume no correlation
```

Every parameter with mean values, uncertainties (systematic and statistical) and correlations (systematic and statistical) needed calculating total uncertainty and total correlation. The dictionary-based data made it possible to code these succinctly, as shown below. Moreover, the last two lines present exceptional consideration of experiments which did not provide systematic and statistical correlations separately, but only the total value.

[3] It works with Python3.4, but not with Python3.6 and I decided not to constrain the project by demanding very specific subversion of Python.

Listing 2: Data preprocessing

```
    # Compute and add to 'val' etots and rho_tots
# 2 utilities:
def etot(exp, paramName):
    return np.sqrt(val[exp][paramName + "_estat"] ** 2 + val[exp][paramName + "_esyst"] ** 2)

def rho(exp, param1, param2):
    return (val[exp][param1+"_estat"] * val[exp][param2+"_estat"] * val[exp]["rho_"+param1+"_
          "+param2+"_stat"] +\
            val[exp][param1+"_esyst"] * val[exp][param2+"_esyst"] * val[exp]["rho_"+param1+"_
                 "+param2+"_syst"])/\
            (val[exp][param1+"_etot"] * val[exp][param2+"_etot"])

for exp in experiments:
    for param in ["Gs", "DGs", "phis"]:
        val[exp][param+"_etot"] = etot(exp, param)
    val[exp]["rho_Gs_DGs_tot"] = rho(exp, "Gs", "DGs")
    if exp not in ["CDF", "D0"]:  # skip hard-coded ones
            val[exp]["rho_phis_DGs_tot"] = rho(exp, "phis", "DGs")
```

Besides the above, there are a few CP-related parameters for which simple real variables were sufficient:

Listing 3: Extra CP-related data

```
# (* tauDsDs constraint, only LHCb.  tauDsDs = tauL(1+phis^2 ys/2 )*)
tauDsDs1 = 1.37900
etauDsDs1 = 0.03106
# (* JpsiEta, LHCb ICHEP 2016, CP-even, tauL *)
tauJpsiEta1 = 1.479
etauJpsiEta1 = 0.03573513677
# (*tauJpsif0 constraint,only CDF. tauJpsif0=tauH(1-phis^2 ys/2)*)
# (* average of CDF, LHCb JpsiPiPi 1fb-1 and D0 2016 *)
tauf01 = 1.65765
etauf01 = 0.03188
# (* Flavour specific lifetime 2014, including tauDsD from LHCb
# previous tau_FS=1.463 pm 0.032. New average spring 2014, including LHCb
# tau_fs(DsD)=1.52 pm 0.15 pm 0.01 ps #=1.52 pm 0.15
# =>tau_FS_2014=1.465 pm 0.031 # \
# (1/sqrt(1/.15**2+1/.032**2))*(1.52/.15**2+1.463/.032**2)
# *)
tauFS1 = 1.516
etauFS1 = 0.014
phis1 = 0.
```

3.2 Probability density functions

The most important probability density function is the two-dimensional Gaussian with correlated variables, as depicted in the following code.

Listing 4: Gaussian P.D.F.

```
    def gaussian2D(x, y, x0, xsig, y0, ysig, rho):
    """Joint p.d.f. of Gs and DGs for JPsi_Phi-like channels
    params:
    =======
    x, y - specify point for the function to be evaluated at
    xsig, ysig - total uncertainties of these
    rho - total rho
    """
    return 1. / (2 * np.pi * xsig * ysig * np.sqrt(1 - rho ** 2)) *\
           np.exp( (-1. / (2 * (1 - rho **2))) * ((x - x0) ** 2 / xsig ** 2 +\
           (y - y0) ** 2 / ysig ** 2 - 2 * rho * (x - x0) * (y - y0) / (xsig * ysig)) )
```

The analyses are, however, enriched by CP-related constrains and require, therefore, a few extra p.d.f.'s. defined below. They include three additional constraints on the parameters, namely: `CP_even`, `CP_odd` and `flavour_specific`. Subsequent three functions pertain to analyses that take into account increasingly more constraints: `hadronic`, `hadronic_and_lifetimes` and `totpdf_Gs_DGs`. Finally, there are similar functions, but with respect to different variables – $1/\Gamma_L$ and $1/\Gamma_H$ (as opposed to Γ_s and $\Delta\Gamma_s$ used before).

Listing 5: Extra P.D.F.'s

```
def CP(x, y, tauCP, etauCP, phis, eta):
    """Generic joint pdf of Gs and DGs for CP-eigenstate channels (Bs2KK ??)"""
    return 1 / (np.sqrt(2 * np.pi) * etauCP) * np.exp( -1. / 2 * ((2. / (2 * x + eta * y) *\
               (1 + eta * phis ** 2 * y / (4 * x)) - tauCP) / etauCP) ** 2)

def CP_even(x, y):
    return CP(x, y, tauDsDs1, etauDsDs1, phis1, eta=1) * CP(x, y, tauJpsiEta1, etauJpsiEta1,
        phis1, eta=1)

def CP_odd(x, y):
    return CP(x, y, tauf01, etauf01, phis1, eta=-1)

def flavour_specific(x, y, tauFS=tauFS1, etauFS=etauFS1):
    """Joint pdf of Gs and DGs for flavour-specific lifetime tauFS"""
    return 1 / (np.sqrt(2 * np.pi) * etauFS) * np.exp(-1. / 2 * ((1. / x * (1 + (y / (2 * x))
         ** 2) / (1 - (y / (2 * x)) ** 2) - tauFS) / etauFS) ** 2)

def hadronic(x, y):
    """ pdf taking into account Bs->J/Psi hh only """
    return jointPDF(x, y, pm=("Gs", "DGs"), exps=experiments)

def hadronic_and_lifetimes(x, y):
    """ pdf taking into account Bs->J/Psi hh and CP even and odd """
    return hadronic(x, y) * CP_even(x, y) * CP_odd(x, y)

def totpdf_Gs_DGs(x, y):
    """Total pdf, including JpsiKK, JpsiPipi, CP and flavour specific lifetimes"""
    return hadronic_and_lifetimes(x, y) * flavour_specific(x, y, tauFS1, etauFS1)

# Change of variable to tau
def CP_tau(x, y, tauCP, etauCP, phis, eta):
    """Generic joint pdf of Gs and DGs for CP-eigenstate channels (Bs2KK ??)"""
    if (eta == 1):
        return 1. / (np.sqrt(2 * np.pi) * etauCP) * np.exp( -1./2 * ( (x*eta - tauCP)/etauCP
            )**2 )
    return 1. / (np.sqrt(2 * np.pi) * etauCP) * np.exp( -1./2 * ( (-y*eta - tauCP)/etauCP )
        **2 )

def CP_even_tau(x,y):
    return CP_tau(x, y, tauDsDs1, etauDsDs1, phis1, eta=1) * CP_tau(x, y, tauJpsiEta1,
        etauJpsiEta1, phis1, eta=1)

def CP_odd_tau(x,y):
    return CP_tau(x, y, tauf01, etauf01, phis1, -1)

def flavour_specific_tau(x, y, tauFS=tauFS1, etauFS=etauFS1):
    """Joint pdf of Gs and DGs for flavour-specific lifetime tauFS"""
    return 1 / (np.sqrt(2 * np.pi) * etauFS) * np.exp(-1. / 2 * ( ((x**2 + y**2)/(x+y) -
        tauFS )/etauFS)** 2)
```

3.3 Maximum likelihood

Most of the minimisation tasks dealt with the same 2-dimensional Gaussian pdf's with difference only in experiments taken into account and variables. As Minuit permits minimisation of 2-variable functions only and there were more parameters (i.e. list of experiments) – it would be natural to define separate functions for each case with parameters specifying all the necessary information. However this approach would require creating 5 separate functions with almost identical functionality. The alternative solution was to build a wrapper class `Maxlikelihood` – illustrated below – which allows for multiple constructor parameters, but behaves like a 2-dimensional function when invoked by the Minuit (method `__call__`) . Also note that the user can specify whether to return $-2 \cdot \ln$ max-likelihood or simply the max-likelihood.

Listing 6: Max-likelihood

```
class Maxlikelihood(object):
    """ Class to be passed to Minuit constructor. Method `__call__(self, x, y)` is to be
        minimised.
        Using class has advantage over simple function, because it allows to set parameters
            (`par` and `exps`),
        which is not possible otherwise (Minuit accepts functions with no parameters but
            those to be optimised))

        If `log`==False, product of double gaussians for specified experiments `exps` is
            returned,
        unless a PDF `pdf` is specified - then `pdf` is returned.
        If `log`==True (default) -2*log of that product of pdf's is returned
    """
    def __init__(self, par, exps, pdf=None, log=True):
        self.par, self.exps, self.pdf, self.log = (par, exps, pdf, log)
        print '=' * 100
        print "Max lihelihood for parameters: ", self.par, "\nFor experiments: ", self.exps
        print '=' * 100

    def __call__(self, x, y):
        """ Function wrapper of function to be minimised """
        if self.log:
            if self.pdf == None:
                return jointPDFlog(x, y, pm=self.par, exps=self.exps)
            return -2 * np.log(self.pdf(x, y))
        else:
            if self.pdf == None:
                return jointPDF(x, y, pm=self.par, exps=self.exps)
            return self.pdf(x, y)
```

3.4 Minimisation

Pure Python has no native minimisation routine as Mathematica does and a third party library was, therefore, needed. At first, a module `scipy.optimise` was used, but it turned out it does not compute non-symmetrical uncertainties, which was essential in our subtle physics calculations. Then, we found an alternative – `iminuit` library [4], which is inspired on PyMinuit and ROOT Minuit. Apart from non-symmetrical uncertainties (computed by subroutine – MINOS), this package has the advantage that it relies on software that many particle physicists are familiar with.

The maximum likelihood function described above is the input of the minimising routine controlled by the class `Minimiser` listed below. Basically it provides a convenient interface to the Minuit object. Its core is the `minimise` method which initially minimises the function with MIGRAD routine and then, finds asymmetrical uncertainties with MINOS. Additionally, the class is equipped with getter methods enabling access to optimisation results and the output method `printall` that saves results in a desired format in a text file.

Listing 7: Minimiser class

```
class Minimiser(object):
def __init__(self, fun_obj, fitParams, fname, fmode='w', header="", parnames=('x','y')):
    self.minimise(fun_obj, fitParams)
    self.printall(fname, fmode, header, parnames)

def minimise(self, fun_obj, fitParams):
    self.m = Minuit(fun_obj, **fitParams)
    self.m.migrad()
    self.m.minos()

def x(self):
    return self.m.values['x']

def y(self):
    return self.m.values['y']

def xerr(self):
    return self.m.errors['x']

def yerr(self):
    return self.m.errors['y']

def eminus_x(self):
    return self.m.get_merrors()['x']['lower']

def eplus_x(self):
    return self.m.get_merrors()['x']['upper']

def eminus_y(self):
    return self.m.get_merrors()['y']['lower']

def eplus_y(self):
    return self.m.get_merrors()['y']['upper']

def rho(self):
    return self.m.matrix(correlation=True)[0][1]

def printall(self, fname, fmode, header, parnames):
        # Extract all fit params
        x, y = self.m.values['x'], self.m.values['y']
        errs = self.m.get_merrors()
        exminus, explus = errs['x']['lower'], errs['x']['upper']
        eyminus, eyplus = errs['y']['lower'], errs['y']['upper']
        rho = self.m.matrix(correlation=True)[0][1]

        with open(fname, fmode) as f:
            print >>f, '\n'
            print >>f, header
            print >>f, '=' * 30
            print >>f, parnames[0], "=", x, "^{+", explus, "}_{", exminus, "}"
            print >>f, parnames[1], "=", x, "^{+", eyplus, "}_{", eyminus, "}"
            print >>f, "rho(", parnames[0], ", ", parnames[1], ") = ", rho
```

In the end, the minimisation procedure can be simply invoked as in the example:

Listing 8: Minimising

```
# starting values for the Phis, DGs parameters
fitParams = dict(x=-0.3, y=0.085, error_x=0.0325, error_y=0.0065, limit_x=None, limit_y=None)
parnames = ('phis', 'DGs')
func = Maxlikelihood(par=parnames, exps=experiments)
m = Minimiser(func, fitParams, fname="Results_Phis_DGs.txt", header="Result from the global
    fit:", parnames=parnames)
```

[4]http://iminuit.readthedocs.io/en/latest/index.html

3.5 Contour plotting

For every so-called analysis channel defined by chosen constraints and experiments – a contour was plotted as in the attached listing. Firstly, log-likelihood of a corresponding maximum likelihood function is computed and then its minimum is subtracted, hence yielding $\Delta \log \mathcal{L}$. Statistical tables provide relationship of that function with given confidence levels. Contours are plotted with `matplotlib.pyplot.contourf` function that has a keyword `levels` that corresponds to z-axis levels at which contours should be drawn – being in our case the $\Delta \log \mathcal{L}$.

Listing 9: Contours

```python
# Draw contours
for i in range(len(channels)):
    lnL = jointPDFlog(X, Y, pm=("phis", "DGs"), exps=[exper for exper in channels[i]])
    dlnL = lnL - np.min(lnL)
    ## contour 2.30 <-> 1-sigma confidence interval on 2 delta log likelihood
    ## filled in contours and then edge contours (for pretty display)
    plt.contourf(X, Y, dlnL, levels=[0, 2.30], alpha=0.5, colors=colors[i])
    plt.contour(X, Y, dlnL, levels=[0, 2.30], alpha=1, colors=colors[i], linewidths=2)
    plt.text(coords[i][0], coords[i][1], labels[i], verticalalignment='bottom',
          horizontalalignment='right',
         transform=ax.transAxes, color=colors[i], fontsize=15)
```

Apart from the above, contour plotting for the first two pictures (see next section) involved rather not too difficult graphical operations, e.g.: positioning of labels, customising ticks and rendering LaTeXsymbols. The challenging part was the last plot. Although, it is merely a transformation of axes with respect to the second plot (from $\Delta\Gamma_s$ vs. Γ_s to $1/\Gamma_H$ vs. $1/\Gamma_L$) – only the two central elliptical contours could be easily transformed. Other contours transformed incompletely. After long time of debugging it turned out that the reason was that due to non-linear transformation of axes – some points *inside* the original contour (not the contour itself) should have become translated to the resulting contour. These points were skipped in the transformation procedure as only the contour was subject to it. There were two possible solutions to this problem: either to transform every point encircled by the original contours or redefine the functions in the new coordinates and draw their contours. The former solution proved too computationally costly and the latter was applied resulting in building correct plots.

4 Results comparison

In this section results obtained from the original project and the one developed in Python are compared. As the project has been successful the Mathematica and Python codes generated equivalent figures. The only differences have to do with aesthetics and the resulting ellipses and bands represent exactly the same constraints. In the Appendix are numerical results of optimisation of parameters as obtained by the two programs. These results are also compatible.

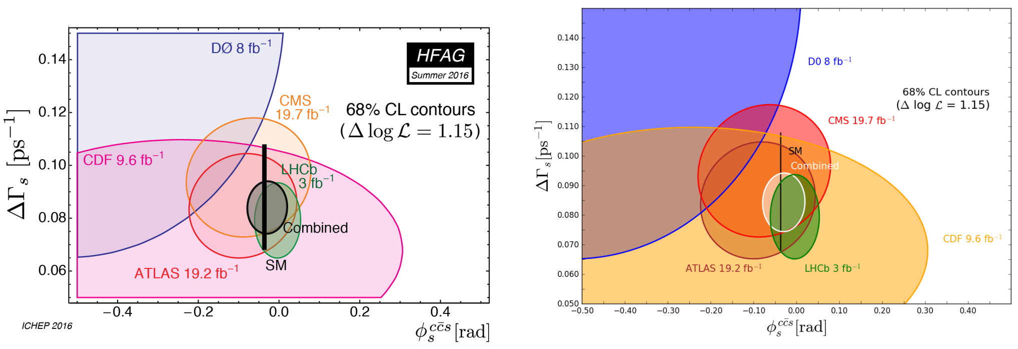

Figure 2: Comparison of the original plot (left) of $\Delta\Gamma_s$ vs. ϕ_s with the one made with the Python program (right).

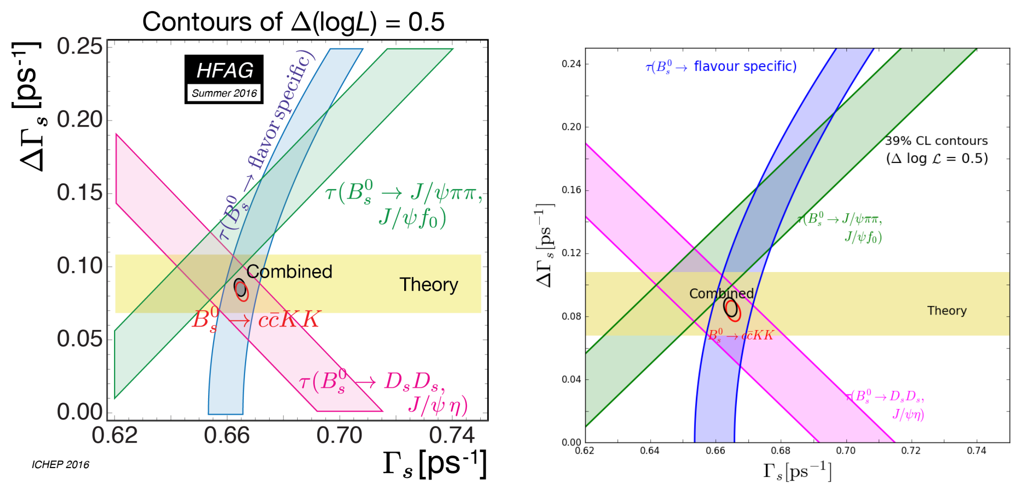

Figure 3: Comparison of the original plot (left) of $\Delta\Gamma_s$ vs. Γ_s with the one made with the Python program (right).

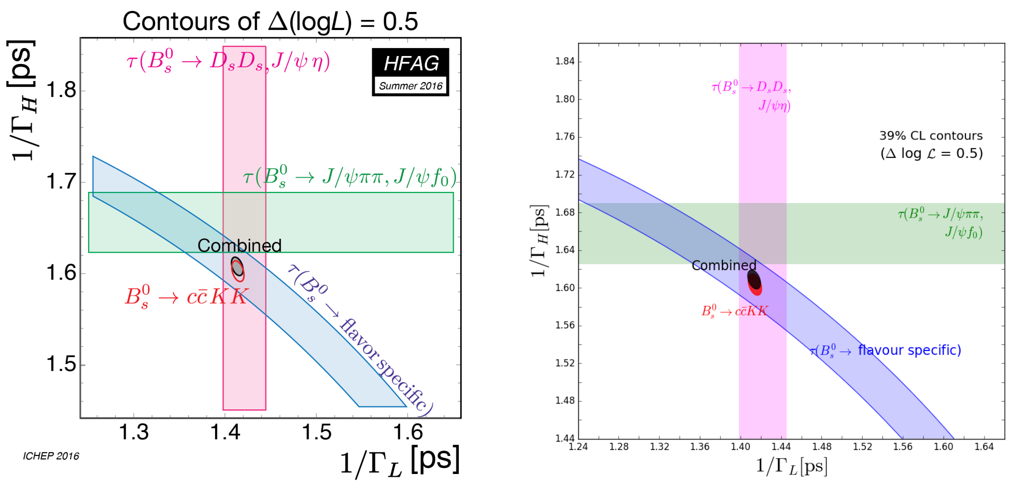

Figure 4: Comparison of the original plot (left) of $1/\Gamma_H$ vs. $1/\Gamma_L$ with the one made with the Python program (right).

5 Conclusion and outlook

The project has been successfully developed: the results obtained – plots and optimised parameters – proved to be consistent with those from the original macros. The program will replace the original one due to economical reasons (no need to pay for the licence agreement) as well as its easier maintenance.

There are, however, a few questions to answer in the long run. It needs to be decided whether to use `iminuit` package, as it seems not to have been upgraded since one year. Consequently, an adequate Python (sub)version should be chosen.

Although, the plots obtained are correct – small aesthetic adjustments might be applied in order to make them indeed of publication quality. Once the project is shared with other members of HFAG – their feedback might be expected and appreciated.

Besides, in spite of the common belief that Python is a rather slow tool for computation – thanks to the `numpy` library – execution speed was even better (roughly 2 times higher) than that of Mathematica software. The total word count was 643 lines in Python and 1129 in Mathematica.

A Appendix: numerical results from Mathematica and Python

Mathematica results

```
Fit using ONLY Bs->Jpsihh :
 ===========================================================================
DGs = 0.0833382^{+0.00652198} _{-0.00652233}
Gs = 0.665388^{+0.00221678} _{-0.00221377}
rhoGs,DGs = -0.292348
taus = 1/Gs = 1.50288^{+0.00500694} _{-0.00500014}
tauL = 1/GL = 1.41431+/-0.00672943
tauH = 1/GH = 1.60329+/-0.0114283
DGs/Gs = 0.125248+/-0.00993195

Fit using Jpsihh and effective lifetimes CP-even and odd  :
 ===========================================================================
DGs = 0.0859665^{+0.00609698} _{-0.00610083}
Gs = 0.664433^{+0.00213922} _{-0.00213589}
rhoGs,DGs = -0.273149
taus = 1/Gs = 1.50504^{+0.00484567} _{-0.00483812}
tauL = 1/GL = 1.41359+/-0.00641539
tauH = 1/GH = 1.60914+/-0.0108101
DGs/Gs = 0.129383+/-0.00930143

DEFAULT fit using Jpsihh, effective lifetimes and tauFS :
 ===========================================================================
DGs = 0.0858241^{+0.00595442} _{-0.00596891}
Gs = 0.664519^{+0.00199595} _{-0.00199365}
rhoGs,DGs = -0.216788
taus = 1/Gs = 1.50485^{+0.00451996} _{-0.00451476}
tauL = 1/GL = 1.41356+/-0.00640862
tauH = 1/GH = 1.60873+/-0.0101702
DGs/Gs = 0.129152+/-0.00906334
Result from the global fit :
 ===========================================================================
DGs = 0.0843489^{+0.00653478} _{-0.00653952}
phis = -0.029572^{+0.032527} _{-0.0325274}
rhophis,DGs = 0.0282987
```

Python results

```
Fit using ONLY Bs->Jpsihh:
==============================
Gs = 0.6653943296 ^{+ 0.00221694630623 }_{ -0.00221694630622 }
DGs = 0.6653943296 ^{+ 0.00653202704158 }_{ -0.00653202704159 }
rho Gs ,  DGs  =  -0.291378059144
taus = 1/Gs = 1.50286823244 ^{+ 0.00500722357921 }_{ -0.0050072235792 }
tauL = 1/GL =  1.41419830992  +/-  0.00674142856442
tauH = 1/GH =  1.6034011301  +/-  0.0114401038745
DGs/Gs =  0.125399559445  +/-  0.00994654785436


Fit using Jpsihh and effective lifetimes CP-even and odd:
==============================
Gs = 0.664426342568 ^{+ 0.00215547012516 }_{ -0.00212551694998 }
DGs = 0.664426342568 ^{+ 0.00606192913942 }_{ -0.00615246619601 }
rho Gs ,  DGs  =  -0.272017924778
taus = 1/Gs = 1.50505772564 ^{+ 0.00488256824935 }_{ -0.00481471835415 }
tauL = 1/GL =  1.41347151941  +/-  0.00642730414403
tauH = 1/GH =  1.60933494686  +/-  0.0108223276663
DGs/Gs =  0.129590451545  +/-  0.00931347105391


DEFAULT fit using Jpsihh, effective lifetimes and tauFS:
==============================
Gs = 0.664527664707 ^{+ 0.00199735403447 }_{ -0.00199442246608 }
DGs = 0.664527664707 ^{+ 0.0059629629965 }_{ -0.00597710049055 }
rho Gs ,  DGs  =  -0.215554602131
taus = 1/Gs = 1.50482824585 ^{+ 0.00452302428878 }_{ -0.00451638572855 }
tauL = 1/GL =  1.41345540527  +/-  0.00641980552768
tauH = 1/GH =  1.6088311539  +/-  0.0101779751281
DGs/Gs =  0.129290022508  +/-  0.00907559113868


Result from the global fit:
==============================
phis = -0.0295740494561 ^{+ 0.0325295360458 }_{ -0.0325254983935 }
DGs = -0.0295740494561 ^{+ 0.00654569400363 }_{ -0.00653384476125 }
rho phis ,  DGs  =  0.0283125138892
```