The 9th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2024
3–7 June 2024, Lisboa, Portugal

ALGORITHMIC DIFFERENTIATION OF THE PYTHONOCC GEOMETRIC MODELING LIBRARY

MLADEN BANOVIĆ[1], THOMAS HAFEMANN[1], ARTHUR STÜCK[1]

[1] German Aerospace Center (DLR)
Institute of Software Methods for Product Virtualization
Zwickauer Str. 46, 01069 Dresden, Germany
e-mail: mladen.banovic@dlr.de, www.dlr.de

Key words: Algorithmic Differentiation (AD), Computer Aided Design (CAD), OpenCascade Technology (OCCT), Gradient-based Optimization, mixed-language AD

Summary. Shape optimization workflows in the aeronautical and automotive industry often rely on high-fidelity numerical simulations (e.g. Computational Fluid Dynamics) and involve CAD-based parametrizations. Since such workflows may impose large computational costs, the optimization itself can be driven by efficient gradient-based methods. This approach, however, requires gradient (sensitivity) information from each component used in the optimization workflow, where the missing link are typically the so-called geometric sensitivities from CAD systems or libraries. To retrieve the exact sensitivity information, one can apply algorithmic differentiation (AD) to the CAD library if its source code is available. For instance, this was successfully demonstrated in the past by differentiating the widely-used C++ geometric kernel OpenCASCADE Technology (OCCT) using the AD tool ADOL-C. This study continues on the previously mentioned work and introduces the following novel contribution: a mixed-language AD of a hybrid Python/C++ geometric modeling library, namely pythonOCC. As its name suggests, pythonOCC provides Python wrappers for OCCT. With the mixed-language AD approach, one can propagate geometric sensitivities from Python to C++ and vice-versa, thus allowing the utilization of pythonOCC in CAD-based shape optimization workflows.

1 INTRODUCTION

In the aerospace and automotive industry, shape optimization techniques are nowadays profoundly employed in the product development life cycle, especially in its preliminary phase to accelerate design studies. Here, one typically utilizes optimization algorithms that can systematically explore the design space of a given problem, often consisting of hundreds or even thousands of design variables.

Generally, the optimization algorithms are divided into two approaches: gradient-free or gradient-based. The gradient-based optimization methods are recognized for their computational efficiency, especially when dealing with problems defined by a large number of design variables. In the case of optimization problems related to aeronautical applications, the efficiency aspect is tremendously relevant because the objective function is usually evaluated by high-fidelity CFD simulations imposing large computational costs.

For the above-mentioned reason, the gradient-based optimization methods are considered in this work. To employ these methods, one requires the derivative calculation from each element or process executed in a given optimization chain, e.g.: (i) geometry generation driven by CAD tools, (ii) creation or deformation of a computational grid (mesh), and (iii) evaluation of the computational grid using Computer Aided Engineering (CAE) tools such as CFD or CSM (Computational Structural Mechanics). In CFD, the adjoint approach is considered as state-of-the-art [1, 2, 3, 4] for computing derivatives of an objective function w.r.t. grid node coordinates. Since the grid node coordinates depend on a CAD parametrization, the next key challenge is to systematically complement the overall sensitivity chain by the computation of the so-called geometric sensitivities in CAD, i.e. derivatives of surface mesh nodes with respect to design parameters of the CAD model to be optimized.

In commercial CAD systems (e.g. SIEMENS NX, SolidWorks, CATIA V5), this information is usually approximated using finite differences (FD) in a black-box manner, as the software sources are not available to many users. Here, a step size is introduced to each design parameter to compute perturbed geometries, which may result in topological changes (patch re-numbering and disappearance). A possible solution to this issue of topological changes is presented in [5] by introducing an additional step in which the original and perturbed geometries are approximated using surface tessellation of linear triangular elements (also referred to as a faceted representation). Then, distances between original and perturbed facets are computed by surface-to-surface projections. Nonetheless, this approach fits for applications where a closed-source CAD system is being integrated into a gradient-based shape optimization loop. However, approximation and cancellation errors of the FD approach are a source of gradient-inaccuracies. Moreover, the FD technique only allows for a forward sensitivity analysis, in which the numerical effort scales with the number of inputs, hampering an end-to-end adjoint sensitivity chain.

On the contrary, computing exact geometric sensitivities can be achieved by applying algorithmic differentiation to the CAD sources. This approach has been demonstrated so far on several CAD tools tailored for CAD parametrizations in turbo-machinery [6, 7, 8], written in Fortran or C++. Moreover, a fully-developed C++ CAD kernel OpenCascade Technology (OCCT) has been differentiated by integrating the AD software tool ADOL-C into its source code [9].

The next advancement in this field is the application of AD to CAD libraries that provide Python front-end. In the community of industrial users, automated, Python-controlled workflows and frameworks are increasingly popular and established, e.g. in conjunction with the open-source library pythonOCC [11]. This imposes a new challenge for AD tools when, for instance, Python and C++ are combined in a single CAD environment. To this end, we consider pythonOCC in this study. It provides Python extensions for C++ OCCT, thus enabling the usage of OCCT to a broader audience and industrial workflows that require Python interfaces to CAD libraries. To differentiate it, one requires a mixed-language AD approach to ensure the propagation of derivatives from Python to C++ and vice-versa. Here, this has been enabled by ADOL-C.

This paper is organized as follows. Section 2 explains ADOL-C, its extension to support Python differentiation and why it has been used to differentiate pythonOCC. Section 3 describes how pythonOCC is differentiated. Section 4 shows verification of the differentiated sources by employing a simple blade parametrization. Finally, Sect. 5 offers conclusions.

2 ADOL-C AND ITS INTERFACE TO PYTHON

There are multiple tools and packages that enable the derivative computation in Python, several examples are mentioned as follows. The package SymPy provides capability for symbolic differentiation. Algorithmic differentiation is natively supported for Python and NumPy functions in the package Autograd, which is actively developed as a part of the JAX Python library [14]. Furthermore, one can consider extensions to the existing operator-overloading C++ AD tools. For example, PyADOLC [12] is a wrapper around ADOL-C developed using the Boost library. Moreover, ADOL-C itself provides SWIG-generated wrappers described in [13].

Here, we consider ADOL-C as the AD tool of choice for differentiating pythonOCC. The main reason is that the OCCT kernel is also differentiated with ADOL-C. Furthermore, one can use ADOL-C to differentiate Python codes. Nevertheless, additional developments regarding ADOL-C were required in this study, as described in Sect. 2.2.

2.1 Introduction to ADOL-C

ADOL-C (*Automatic Differentiation by Overloading in C++*) is an open-source AD tool based on operator-overloading concept [10]. It is mostly used to differentiate vector functions written in C++. As a result, one retrieves first or higher-order derivatives. To differentiate a code with ADOL-C, one replaces the declaration types of almost all real variables (e.g. declared as `float` or `double`) with the ADOL-C data-type named `adouble`.

There are two implementations of the `adouble` class (defined in different headers): *traceless* and *trace-based*, providing different ways of derivative computation. The difference is that the *traceless* option computes derivatives along the function (*primal*) evaluation by applying the forward mode of AD. On the other hand, the *trace-based* option first generates an internal representation of the function to be differentiated—called *trace*—which is later evaluated by employing ADOL-C drivers in order to calculate corresponding derivatives. The *trace-based* option supports both the forward and reverse mode of AD, where the reverse mode of AD has a significant potential for improved efficiency when dealing with functions that have a much smaller number of outputs (dependents) comparing to the number of inputs (independents).

Both options have been used to differentiate the OCCT kernel [9]. Since here the differentiation of a hybrid Python/C++ code is considered, we chose the *traceless* option as the first step of pythonOCC differentiation. There are a few practical reasons for such decision: (i) the forward mode of AD serves later as a reference for validating derivatives computed using the reverse mode of AD, (ii) the *traceless* option is much easier to debug than the *trace-based* option because the overloaded operators implement both *primal* and derivative statements, and (iii) it is straightforward to extract the *primal* and derivative information at any given point in the computation.

2.2 Python interfaces for ADOL-C

As described in the previous section, ADOL-C offers two differentiation options (*traceless* and *trace-based*) depending on which header file is being included. For the *trace-based* differentiation option, Python interfaces already exist [13] and they are generated using the software development tool SWIG (*Simplified Wrapper and Interface Generator*) [15]. Those developments have been considered as a fundamental work in order to implement the Python wrapper for the *traceless* option required in this study (header file `adolc/adtl.h`). Furthermore, pythonOCC also uses SWIG, which is an additional reason why this tool is chosen here (more details will follow in Sect. 3).

SWIG creates interfaces for scripting languages (in this case Python) based on a specific file that contains SWIG directives (macros), the so-called interface file (usually denoted with a `.i` or `.swg` suffix). The SWIG directives are preceded by the `%` delimiter in order to distinguish them from the C/C++ declarations. Furthermore, anything written between `%{` and `%}` block is copied as is to the resulting C++ wrapper file, i.e. it is not parsed or interpreted by SWIG.

A code snippet from the newly defined interface `adouble.i` is presented in Listing 1 and described as follows. Typically, the first thing to do is to define the module name using the `%module` directive, in this case `adtl`, as shown in Line 1. Lines 3–5 represent a block whose body (Line 4) will be copied as is into the wrapper file. Such blocks are used to include header files and other declarations in order to compile the generated interface. That is, just by using the `%include` macro as in Line 20 does not mean that this include statement will appear in the generated wrapper code. For this reason, the block represented by Lines 3–5 is required. Next, the `%ignore` directive in Line 8 is introduced, because the behavior of the assignment operator in Python is different than in C++. As the name suggests, this directive will cause SWIG to ignore a certain C/C++ identifier. Furthermore, SWIG is not able to fully handle operator overloading for operators that are not part of a class, i.e. the operators declared using the `friend` keyword. For them, SWIG issues a warning. Such declarations exist in `adolc/adtl.h` and they have been also ignored as shown, for example, in Lines 11–12 (the total number of the `%ignore` statements in the complete `adouble.i` file is equal to 53). Afterwards, there are `%import` and `%include` directives, listed in Lines 14–20. The `%include` directive (Line 20) instructs SWIG to process the header file of interest and create the corresponding wrapper code. The purpose of the `%import` directive is just to collect information from another SWIG interface file or a header file without generating any wrapper code. That is, the code block between Lines 14–19 represents what is required for the compilation purposes. Next, the code block between Lines 23–25 is an example of how the *tan* function is redefined because it was ignored in Line 11. Finally, there is an example how to extend the Python proxy `adouble` class (using the `%extend` directive) and overload the `rmul` operator (Lines 27–31) which is called when an `adouble` object is multiplied by a scalar (`double`) from the left. This operator is declared using the `friend` keyword in the C++ `adouble` class and was therefore previously ignored as shown in Line 12.

Listing 1: SWIG interface file `adouble.i` for the *traceless* `adouble` class of ADOL-C

```
 1  %module adtl
 2
 3  %{
 4  #include "adolc/adtl.h"
 5  %}
 6
 7  // ignore all assignments operators
 8  %ignore *::operator=;
 9
10  // ignore all declarations that use the "friend" keyword, e.g.:
11  %ignore adtl::tan;
12  %ignore operator*(const double v, const adtl::adouble& a);
13
14  %import <std_list.i>
15  %import <std_string.i>
```

```
%import <std_iostream.i>
%import "adolc/internal/common.h"
%import "adolc/internal/usrparms.h"
%import "adolc/internal/adolc_settings.h"
%include "adolc/adtl.h"

// redefine what was ignored previously:
adtl::adouble tan(const adtl::adouble& a){
    return adtl::tan(a);
}
// redefine what was ignored previously by extending the proxy class:
%extend adtl::adouble {
    adouble __rmul__(const double a){
        return (a * (*$self));
    }
}
```

Once the interface file is created, one can invoke SWIG to build the desired extension module, in this case `adtl`. There are multiple possibilities to invoke SWIG, here we employed CMake. Once the build process with SWIG is finished, the following module files appear as a result: `adtl.py` and `_adtl.so`. They have to be added to the user's Python system path such that the module files can be imported.

The following section describes how the Python wrapper of the *traceless* `adouble` class can be employed to differentiate pythonOCC.

3 MIXED-LANGUAGE AD OF PYTHONOCC

The open-source pythonOCC geometry modeling library provides Python wrappers to almost all OCCT C++ classes. On top of that, it provides utility functions, e.g. for various topological operations, and 3-D visualization in Python GUI (*Graphical User Interface*) libraries such as PyQt5/6 or PySide2. More information and additional features can be found in [11]. The version *7.6.2* is considered in this study.

pythonOCC is differentiated in the forward mode of AD using the SWIG-generated Python extension for the *traceless* `adouble` class (described in the previous section). Moreover, it is compiled and linked against the AD-enabled OCCT C++ kernel *v7.6.2*, also differentiated with ADOL-C.

ADOL-C is integrated into OCCT by changing the OCCT's alias `Standard_Real` (defined in `Standard_TypeDef.hxx`) from `double` to the `adouble` data-type. Such a change triggered a substantial amount of compile- and run-time issues that had to be tackled to finalize the differentiation process. More details about these obstacles and the corresponding solutions are described in [9].

The differentiation of pythonOCC is described as follows. Since pythonOCC uses the CMake build system, the first step is to modify the `CMakeLists.txt` file in this fashion: (i) add paths to the ADOL-C installation directories containing the header `adtl.h` and the library file, (ii) add path the interface file `adouble.i` described in the previous section, and (iii) add `adolc` to each existing `swig_link_libraries` command to ensure that all pythonOCC modules are linked against ADOL-C.

Next, the source files located under the path `src/SWIG_files` are modified as follows. First, an include statement for `adolc/adtl.h` is written to `headers/Standard_module.hxx`. This is one of the top-level header files and therefore a good candidate to make `adtl.h` present all over the pythonOCC sources. Afterwards, the interface file `wrapper/Standard.i` is modified as shown in a code snippet of Listing 2. As one can notice, the `adouble.i` is imported (Line 2). Followed by that is the change of the alias `Standard_Real` (Line 7). It is important to note that this `typedef` is not propagated to the generated wrapper code. One possibility to make it accessible also on the Python level is to use the `%pythoncode` directive—an example is shown between Lines 10–13. Nonetheless, this step is optional, i.e. one could also work directly with `adouble` on the Python side.

Listing 2: Modifications done in the SWIG interface file `Standard.i`

```
 1  // add import directive for the traceless adouble interface
 2  %import "adouble.i"
 3
 4  // --- original code ---
 5  typedef double Standard_Real;
 6  // --- AD code ---
 7  typedef adtl::adouble Standard_Real;
 8
 9  // optional AD code - an example of alias definition
10  %pythoncode {
11  import adtl as ad
12  Standard_Real = ad.adouble
13  }
```

Thereafter, the directory `src/SWIG_files/wrapper` contains about 300 additional SWIG interface files, corresponding to each source package of OCCT. These interfaces were checked and accordingly modified such that each class and its methods match the signature of the corresponding classes from the differentiated OCCT kernel. For most cases, no adjustments were required, as most of the interfaces make correct use of the alias `Standard_Real`. Nevertheless, there were certain mismatches reported by SWIG during the compilation, mainly because the differentiated OCCT has slightly different method signatures comparing to the original version. That is, the `adouble` class is not used everywhere in the source code, as reported in [9].

For every interface file there is a corresponding *stub* file (extension: `.pyi`), installed together with the pythonOCC modules. These files contain useful type hints, i.e. signatures of classes and methods that can be queried by a type checker. In the original version, all *real* variables were declared using the `float` type—which is now changed to `Standard_Real` in the AD version wherever required to match the signatures from the SWIG interfaces.

In light of this information, it is important to note that the user experience in the differentiated pythonOCC is now changed, since the *real* variables (in most cases) have to be initialized as `adouble` objects. As an example, a code snippet is presented in Listing 3. Here, Line 5 shows how a 3-D point object can be initialized in the original version, while Line 8 demonstrates the corresponding initialization in the AD version.

Listing 3: Example of initializing a 3-D point in the original vs. differentiated pythonOCC

```
1  from OCC.Core.Standard import *
2  from OCC.Core.gp import gp_Pnt
3
4  # --- original code ---
5  aPnt = gp_Pnt(1., 2., 3.)
```

```

# --- AD code ---
aPnt = gp_Pnt(Standard_Real(1.), Standard_Real(2.), Standard_Real(3.))
```

During the execution of the tests provided with the pythonOCC sources, a lot of run-time errors had to be resolved, mainly due to the type change from `float` to `Standard_Real`, which is now `adouble`. Furthermore, `adouble` is not used everywhere in pythonOCC. For instance, the `ShapeTesselator` class of pythonOCC is used by WebGL (*Web Graphics Library*) which requires the `float` data-type. Thus, the SWIG interface of the `ShapeTesselator` class remained unchanged (i.e. `floats` are used). However, the underlying C++ code was adapted because it uses the classes of the AD-enabled OCCT. In this process, the *primal* values of the `adouble` objects (e.g. point coordinates) were extracted by calling the `getValue` method wherever required.

The following section describes the validation of the differentiated pythonOCC with respect to: (i) its original (*primal*) functionality, and (ii) calculation of geometric sensitivities.

4 VERIFICATION OF DIFFERENTIATED PYTHONOCC

After a successful compilation of the AD-enabled pythonOCC using SWIG, it is important to check its original functionality. For this purpose, one employs the test suite provided together with the sources. In this process, many changes were made to the utility classes of pythonOCC and to the test suite itself, because everything has to be adapted such that `adoubles` are correctly used. The latest results are shown in Table 1. The remaining four failed tests require further investigation, however it is important to note that they are not related to the geometric modeling functionality of pythonOCC.

Table 1: Test results for the pythonOCC *primal* functionality

Number of succeeded tests	Number of failed tests	Success rate
126	4	96.9%

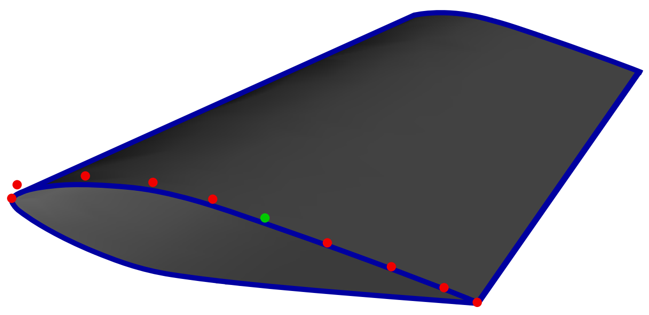

Figure 1: Blade geometry used for sensitivity verification

In the following step, the correctness of the computed derivatives is verified using a simple blade geometry, as illustrated in Fig. 1. Its 2-D base profile is the NACA0012 airfoil approximated using two B-splines that are joined at the leading and the trailing edge. This base profile is cloned, scaled and transformed to another position. Finally, the *lofting* algorithm named `BRepOffsetAPI_ThruSections` is employed, which takes two profiles (cross-sections) as input and generates the final blade stored as a topological data structure. Afterwards, one obtains the underlying geometric information—surfaces in this case—to calculate surface point coordinates and their corresponding derivatives.

In terms of working with the *traceless* `adouble` class of ADOL-C to calculate the derivatives using the *scalar* forward mode of AD, the following steps are important to note: (i) one activates (in terms of AD) a certain design parameter (i.e. independent variable) by calling the method `setADValue` with the derivative seed equal to 1., and (ii) one retrieves the derivative information on the surface point coordinates (i.e. dependent variables) by calling the method `getADValue`.

As a representative example, the surface point derivatives are computed w.r.t. the weight parameter of the control point marked with green color (in Fig. 1). The correctness of the computed derivatives is verified in a few steps described as follows.

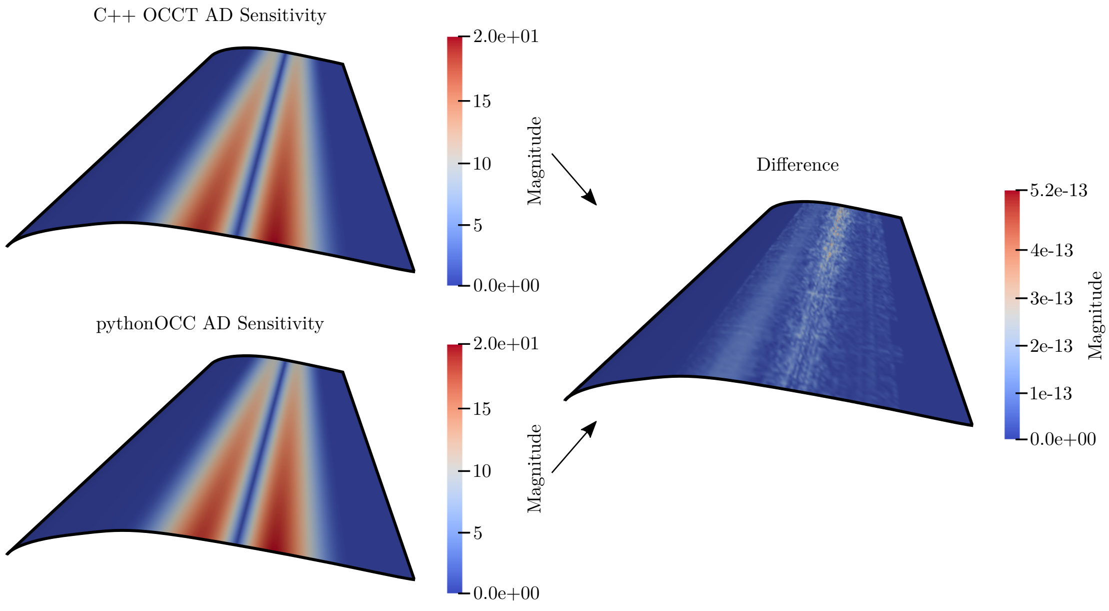

Figure 2: C++ OCCT AD vs. pythonOCC AD sensitivities

First, the blade parametrization written in Python (pythonOCC) is also written in C++ (OCCT). The sensitivities are compared as demonstrated in Fig. 2 and they show mutual consent. This concludes that the derivatives are correctly propagated from Python to C++ (and vice-versa) using the SWIG-generated ADOL-C interface. An additional remark to this comparison is that the workflow regarding ADOL-C is the same between the AD-enabled pythonOCC and C++ OCCT, but one can benefit from the flexibility that Python as a high-level programming language has to offer, for example, in terms of rapid prototyping.

Second, the derivatives computed with AD are compared against FD. A qualitative comparison is presented in Fig. 3. As one can notice, the overall magnitude plots between AD and FD match to a high extent.

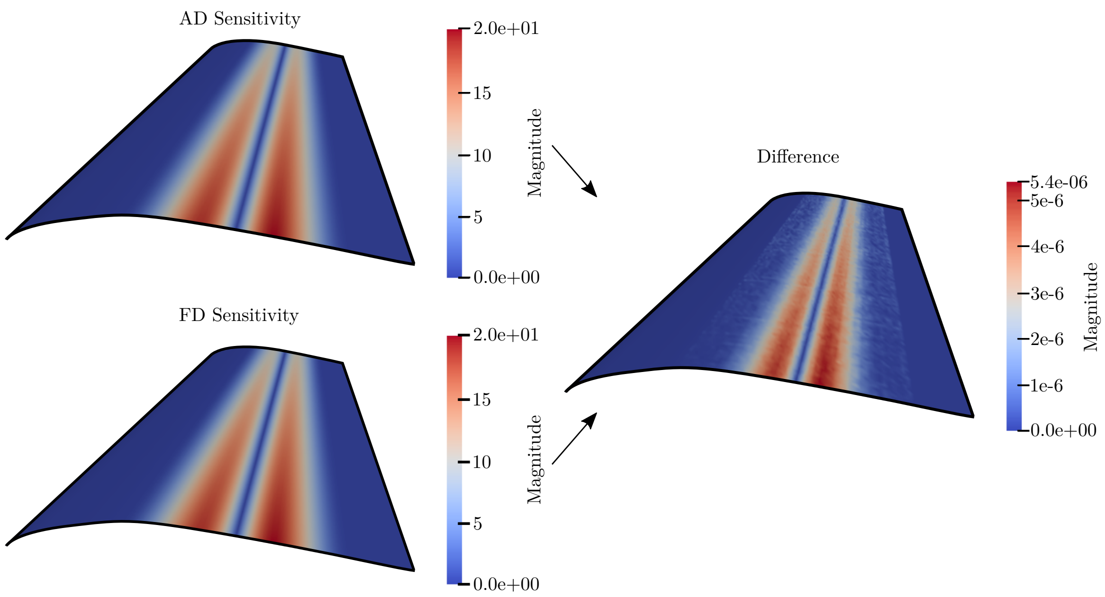

Figure 3: pythonOCC sensitivities: AD vs. FD

Finally, the AD-enabled surface sensitivities are additionally verified with a Taylor test:

$$f(x+h) - f(x) - h\frac{\partial f}{\partial x}(x) = \mathcal{O}(h^2). \tag{1}$$

The test was performed on eight different surface point coordinates, considering the following range of step sizes $h \in [10^0, 10^{-13}]$. The error plots (the left-hand side of Eq. 1) are presented in Fig. 4. One can observe that the errors follow the theoretical convergence rate of h^2 until they reach machine precision around $h = 10^{-6}$. The verification demonstrates that AD derivatives are correct for this particular use-case.

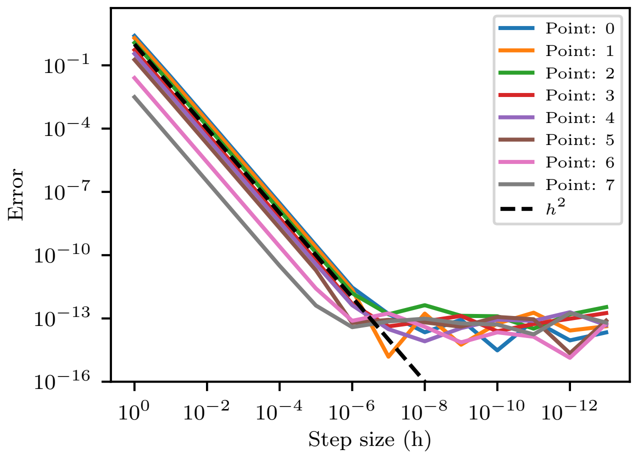

Figure 4: Taylor test evaluation using the differentiated pythonOCC

5 CONCLUSION AND OUTLOOK

This paper presents the algorithmic differentiation of the hybrid Python/C++ geometric modeling library pythonOCC. For this purpose, a SWIG interface for the *traceless* `adouble` class was introduced to generate its Python extension. Then, this interface was employed to differentiate pythonOCC.

In this process, the pythonOCC interfaces and the corresponding *stub* files were modified to match class- and method-signatures of the differentiated OCCT in order to resolve the compile- and run-time issues. The obtained derivatives were verified against FD, showing mutual agreement. Furthermore, the results of the Taylor test aligned with the theoretical expectations. This verification proved the correctness of the computed derivatives for a simple blade parametrization and serves as a proof of concept. The next step is to include more complex parametrizations to verify the derivative computation of different functionalities of the differentiated pythonOCC.

The derivatives were computed using the forward mode of AD. Currently, the reverse mode of AD is not supported in pythonOCC. To allow it, further work will be about integrating the *trace-based* `adouble` class into pythonOCC.

Next, the differentiated pythonOCC is going to be integrated in the FlowSimulator environment, which is a high-performance computing framework for multidisciplinary simulations being developed by DLR and partners in academia and industry. The benefits of this integration are: (i) additional flexibility for the creation of CAD models and model constraints, and (ii) the direct access to the associated geometric sensitivities with the convenience of the Python layer. PythonOCC geometries can then be linked to the numerical meshes by the in-house FlowSimulator plugin FSOCCT, which already serves as an entry point for certain OCCT functionalities to the FlowSimulator framework with its Data Manager (FSDM). Moreover, other plugins of the FlowSimulator ecosystem could easily access metadata attached to the geometry, for example, required for boundary conditions, mesh deformation, etc. Finally, the whole framework will be used to perform large-scale gradient-based optimizations of an aircraft backed by an end-to-end adjoint sensitivity chain that is HPC-ready and features exact gradient information.

REFERENCES

[1] Giles, M.B.; Duta, M.C.; Müller, J.D.; Pierce, N.A.: Algorithm developments for discrete adjoint methods. AIAA journal, 41(2), 198–205, 2003. `http://doi.org/10.2514/2.1961`.

[2] Jameson, A.: Aerodynamic design via control theory. Journal of Scientific Computing, 3, 233–260, 1988. `http://doi.org/10.1007/BF01061285`.

[3] Pironneau, O.: On optimum design in fluid mechanics. Journal of Fluid Mechanics, 64(1), 97–110, 1974. `http://doi.org/10.1017/S0022112074002023`.

[4] Yu, G.; Müller, J.D.; Jones, D.; Christakopoulos, F.: CAD-based shape optimisation using adjoint sensitivities. Computers & Fluids, 46(1), 512–516, 2011. ISSN 0045-7930. `http://doi.org/10.1016/j.compfluid.2011.01.043`. 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[5] Agarwal, D.; Robinson, T.T.; Armstrong, C.G.; Marques, S.; Vasilopoulos, I.; Meyer, M.: Parametric design velocity computation for CAD-based design optimization using ad-

joint methods. Engineering with Computers, 34, 225–239, 2018. http://doi.org/10.1007/s00366-017-0534-x.

[6] Sanchez Torreguitart, I.; Verstraete, T.; Mueller, L.: Optimization of the LS89 axial turbine profile using a CAD and adjoint based approach. Int J Turbomach Propuls Power 3(3):20, 2018.

[7] Voß, C.; Siggel, M.; Backhaus, J.; Pahs, A.: A Differentiated Geometry Blade Parameterization Methodology for Gas Turbines. Available at SSRN (2023).

[8] Banović, M.; Vasilopoulos, I.; Walther, A.; Meyer, M.: Algorithmic differentiation of an industrial airfoil design tool coupled with the adjoint CFD method. Optim Eng 21, 1221–1242 (2020). https://doi.org/10.1007/s11081-019-09474-x.

[9] Banović, M.; Mykhaskiv, O.; Auriemma, S.; Walther, A.; Legrand, H.; Müller, J.D.: Algorithmic differentiation of the Open CASCADE Technology CAD kernel and its coupling with an adjoint CFD solver. Optimization Methods and Software, 33(4-6), 813–828, 2018. http://doi.org/10.1080/10556788.2018.1431235.

[10] Walther, A.; Griewank, A.: Getting started with ADOL-C. Dagstuhl seminar proceedings 09061, pp 181–202, 2009.

[11] Paviot, T.: pythonocc (7.7.0). Zenodo. 2022. https://doi.org/10.5281/zenodo.3605364.

[12] Walter, S.F.: PyADOLC, https://github.com/b45ch1/pyadolc.

[13] Kulshreshtha, K.; Narayanan, S.H.K.; Bessac, J.; MacIntyre, K: Efficient computation of derivatives for solving optimization problems in R and Python using SWIG-generated interfaces to ADOL-C. Optimization Methods and Software, 33(4-6), 1173–1191 (2018).

[14] Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M.J.; Leary, C.; Maclaurin, D.; Necula, G.; Paszke, A.; VanderPlas, J.; Wanderman-Milne, S.; Zhang, Q.: JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax.

[15] Beazley, D. M.: Automated scientific software scripting with SWIG. Future Generation Computer Systems, 19(5), 599–609 (2003).