


Implementation of FORMIDABLE: A generalized differential optical design library with NURBS capabilities

Jean-Baptiste Volatier^{1,*} , Stephane J. Beaussier², Guillaume Druart¹, Paul Jouglas³, and Fanny Keller⁴

¹ ONERA, Chemin de la Hunière, 91123 Palaiseau Cedex, France

² Valley Optics B.V., Molengraaffsingel 12, 2629 JD Delft, The Netherlands

³ Airbus Defence and Space, Rue des Cosmonautes 31, 31402 Toulouse, France

⁴ European Space Agency – ESTEC, Keplerlaan 1, PO box 299, 2200AG Noordwijk, The Netherlands

Received 15 August 2023 / Accepted 22 November 2023

Abstract. In this article we describe the implementation of Freeform Optics Raytracer with Manufacturable Imaging Design cApaBiLitiEs (FORMIDABLE): an optical design library capable of simulating optical systems by ray-tracing. Optical performance can be quantified and optimised using third-party optimisation algorithms. Compared to available commercial optical design and similarly to fast accurate NURBS optimization (FANO), our code can simulate and optimise Non-uniform rational B-Spline (NURBS). It also implements generalized differential capabilities that allows faster convergence compared to state-of-the-art. The implementation of FORMIDABLE and its innovative capabilities are described and illustrated with a representative case-study. The source code is available to eligible third-parties under the ECSL licence.

Keywords: Optics, Freeform, Optical design.

1 Introduction

An optical design problem consists of finding an appropriate combination of optical surfaces, reflective or refractive, and of refractive materials which will allow an optical instrument to perform a specified function.

The quality of the imaging function can be quantified with various metrics, wavefront error for diffraction limited systems, geometric spot radius for aberration limited ones, and modulation transfer function that can be computed for both. Distortion and transmission are also criterion of importance as are practical constraints such as volume, weight, cost etc.

An appropriate solution is a compromise between these aforementioned metrics and will be found by adjusting the degrees of freedom of the system. Adjusting a degree of freedom will for example modify the shape of an optical surface or its position.

To accomplish this task, designers leverages computed-aided design software which rely partly on non-linear numerical optimisation where the fitness of the system is encoded in a scalar function of the degrees of freedom. Optimisation algorithms are then used to find a minimum from a starting point supplied by the designer.

It is increasingly common to rely on surfaces that have no axial symmetry, usually referred to as freeform surfaces [1], to add degrees of freedom. Adding extra degrees of freedom allow finding solutions to an optical design problem that can outperform previous solutions on most metrics, though usually not on all since freeform systems are often more costly [2].

The choice of description of freeform surfaces has been the subject of a vigorous debate, Zernike polynomials being often the favored choice [3]. On the other hand, Chrisp [4] have pioneered Non-uniform rational B-splines in optical design of imaging systems and have shown advantages compared to other descriptions.

The advantages of NURBS comes from local control. Indeed, NURBS surfaces are controlled by a grid of control points together with their weights. Each control point influences the surface in only a limited region. However, to successfully optimise NURBS, a custom optical design software is required as commercial optical design codes empirically show poor performances with NURBS potentially because they require many degrees of freedom [4]. Fast accurate NURBS optimization (FANO) [5] is an example of such a software.

In the following article we describe our code Freeform Optics Raytracer with Manufacturable Imaging Design cApaBiLitiEs (FORMIDABLE) that also allows design of optical surfaces described as NURBS in optical systems.

* Corresponding author: jbv@pm.me

Compared to FANO, FORMIDABLE also implements the following functionalities: differential ray tracing [6] which greatly speeds up optimisation of problems with large number of degrees of freedom, and ray-aiming, which allows to optimise system with a physical pupil defined on one of the intermediate surfaces. In the case of ray-aiming, it is unclear to us if FANO implements it, one example [7] does have the physical stop defined on surface 2, which would require ray-aiming though most of the examples have the physical stop on the first surface.

Another objective of FORMIDABLE was to use its NURBS capabilities to simulate the impact of structural deformation on optical performance, as NURBS are a standard widely used in mechanical Computed-aided software (CAD) software hence the acronym Freeform Optics Raytracer with Manufacturable Imaging Design cApaBiLitiEs (FORMIDABLE).

In the following sections we will first introduce the concept of differential ray tracing and its implementation. We will then describe the challenges FORMIDABLE overcome in order to optimise a NURBS based system in the context of a case-study. Finally we will describe the architecture of FORMIDABLE.

2 Differential ray tracing

Differential ray tracing consist into simultaneously computing a ray trace and its derivatives.

We define a ray as a curve perpendicular to a wavefront of light. In this article we only consider mediums of homogeneous refractive index separated by optical surfaces, so rays can be described by a line.

Mathematically we represent a ray by a point \mathbf{p} and a normalized direction vector direction \mathbf{k} .

At each optical surface, the ray will either be refracted or reflected to a new ray. We call this sequence of rays the ray path.

In the context of optical ray tracing we are only interested in ray paths that originates from a point in the field of view and are not vignettted by the pupil of our instrument.

We call the phase space the combination of the field point, the pupil, and the wavelength define the ray tracing function in equation (1) as the function that associates a ray path ϕ to phase a point in the phase space and \mathbf{s} an optical system.

We note \mathbf{h} the part of ϕ that denotes the field point, \mathbf{r} the part that denotes the pupil point, and λ wavelength the wavelength.

$$\text{raytrace}(\phi, \mathbf{s}) = \mathbf{w}. \quad (1)$$

In equation (1), \mathbf{s} is the optical system in consideration, in concrete terms it is a vector of what might vary in an optical system (curvatures, element position etc.), or as referred earlier the degrees of freedom of the optical design problem.

The right hand side of equation (1), \mathbf{w} is the ray path of the system which can be represented as the sequence of rays $\{\{\mathbf{p}_1, \mathbf{k}_1\}, \dots, \{\mathbf{p}_N, \mathbf{k}_N\}\}$.

But we will often prefer a more compact representations eliminating redundant information, either the sequence of

intersection points $\{\mathbf{p}_1, \dots, \mathbf{p}_N\}$ (\mathbf{k}_i are omitted as redundant) or the sequence $\{\mathbf{u}_1, \dots, \mathbf{u}_N\}$ where \mathbf{u}_i is the argument of \mathbf{p}_i with respect to the surface function.

Indeed, we limit ourselves to the design of optical surfaces that can be represented parametrically, so for each optical surfaces it exists a function associating a point in three-dimensional space \mathbf{p} to a point in two-dimensional parameter space \mathbf{u} (Eq. (2)),

$$\mathbf{u} \in \mathbb{R}^2 \mapsto \mathbf{p} \in \mathbb{R}^3. \quad (2)$$

To compute the derivative of equation (1) in the general case we need a mathematical framework that allows to differentiate functions that do not necessarily have an analytical expression. The framework was introduced in literature [6] and is described here in updated terms.

2.1 Implicit differentiation

Given \mathbf{y} and \mathbf{x} two vector variables such that there is a function mapping \mathbf{x} to \mathbf{y} but without access to the analytic expression of such function.

What we do have is the analytic expression of \mathbf{f} that satisfies $\mathbf{f}(\mathbf{x}, \mathbf{y}) = \mathbf{0}$.

Knowing \mathbf{x} it is possible to compute \mathbf{y} by solving \mathbf{f} with an algorithm such as Newton's descent, but how to compute the jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$?

To do so, we first introduce two differential operators: the partial differential operator $\partial \cdot / \partial \cdot$ and the total differential operator $d \cdot / d \cdot$.

The total differential operator applied to \mathbf{f} takes into account the fact that \mathbf{y} depends on \mathbf{x} and can be expressed as a linear combination of partial differential operators that neglect cross terms by applying the chain rules as in equation (3).

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \mathbf{0}. \quad (3)$$

Since we have the analytic expression of \mathbf{f} , if it is differentiable we obtain the jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by solving the linear system. Of course equation (4) implies that \mathbf{f} and \mathbf{y} have the same dimensionality.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}^{-1}}{\partial \mathbf{y}} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}. \quad (4)$$

We can apply the implicit differentiation technique to the intersection calculation of a ray and a surface.

For surfaces that can be described as quadrics (such as spheres, conics etc.) the intersection calculation accepts a closed form solution therefore differentiation is trivial.

For intersection of a ray with a NURBS we use an iterative algorithm [8] which searches for the zero of a two-dimensional error function.

We can therefore differentiate \mathbf{u} using the implicit function theorem applied to this two-dimensional intersection error function.

To further compute the derivatives of \mathbf{w} from the derivatives of \mathbf{u} we use the method that we introduce in Section 2.3.

In the following section, we describe another method implemented in FORMIDABLE, implicit differentiation using Fermat path principle that allows to differentiate directly a ray path.

Whether differentiating using the fermat error function or using the intersection error function will come down to computation time, typically intersection method are more efficient when more surfaces are considered.

2.2 Fermat path principle

We recall that the Fermat path principle states that rays travel along a stationary path throughout the system. Let L be the optical path throughout an optical system and $n_{i,i+1}$ the refractive index between surface i and $i + 1$ and $\|\mathbf{p}_i\mathbf{p}_{i+1}\|$ is the Euclidean distance between \mathbf{p}_i and \mathbf{p}_{i+1} .

$$L = \sum_{i=0}^N n_{i,i+1} \|\mathbf{p}_i\mathbf{p}_{i+1}\|. \quad (5)$$

As a consequence of Fermat's path principle we have:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{0}. \quad (6)$$

Let $\mathbf{p}_i = (x_i, y_i, z_i)$, we define in equation (7), the vector \mathbf{F} composed of the numerators of the $\partial L/\partial \mathbf{w}$ terms.

$$\begin{aligned} \mathbf{F}_i = & (\|\mathbf{p}_{i-1}\mathbf{p}_i\|(x_i - x_{i+1}) + \|\mathbf{p}_i\mathbf{p}_{i+1}\|(x_i - x_{i-1})) \frac{\partial x_i}{\partial \mathbf{w}_i} \\ & + (\|\mathbf{p}_{i-1}\mathbf{p}_i\|(y_i - y_{i+1}) + \|\mathbf{p}_i\mathbf{p}_{i+1}\|(y_i - y_{i-1})) \frac{\partial y_i}{\partial \mathbf{w}_i} \\ & + (\|\mathbf{p}_{i-1}\mathbf{p}_i\|(z_i - z_{i+1}) + \|\mathbf{p}_i\mathbf{p}_{i+1}\|(z_i - z_{i-1})) \frac{\partial z_i}{\partial \mathbf{w}_i} \\ \mathbf{F} = & (\mathbf{F}_1, \dots, \mathbf{F}_i, \dots, \mathbf{F}_{N-1}). \end{aligned} \quad (7)$$

We call \mathbf{F} the offence against Fermat path principle. \mathbf{F} must be equal to $\mathbf{0}$ given equation (6) for the ray path to be physical.

We note that there the dimensionality of \mathbf{F} is equal to the dimensionality of \mathbf{w} .

Therefore, we can apply the implicit differentiation technique to \mathbf{F} to differentiate \mathbf{w} with respect to ϕ or \mathbf{s} (Eq. (8)).

$$\frac{\partial \mathbf{w}}{\partial \phi} = \frac{\partial \mathbf{F}^{-1}}{\partial \mathbf{w}} \frac{\partial \mathbf{F}}{\partial \phi}, \quad (8)$$

$$\frac{\partial \mathbf{w}}{\partial \mathbf{s}} = \frac{\partial \mathbf{F}^{-1}}{\partial \mathbf{w}} \frac{\partial \mathbf{F}}{\partial \mathbf{s}}. \quad (9)$$

The term $\frac{\partial \mathbf{F}}{\partial \mathbf{w}}$ represents the sensitivity of the offence against fermat path principle to small errors in the ray path. This term can be computed analytically if the curvatures of the optical surfaces can be computed analytically.

The term $\frac{\partial \mathbf{F}}{\partial \phi}$ represents the sensitivity of the offence against fermat path principle to small changes in the field or the pupil coordinate of the ray.

The term $\frac{\partial \mathbf{F}}{\partial \mathbf{s}}$ represents the sensitivity of the offence against fermat path principle to small changes in the geometry of the system. To be computed it will require that the optical surfaces are differentiable with respect to the variable parameters of the optical system. For example, if the optical system has a spherical optical surface with a variable curvature, it is possible to differentiate a point on the surface with respect to the curvature and therefore it is possible to compute analytically $\frac{\partial \mathbf{F}}{\partial \mathbf{s}}$.

In most cases the three terms above will be computable analytically and therefore the ray tracing will be differentiable.

2.3 Differentiating merit functions

In the previous section we have described how to differentiate a ray tracing.

However, merit functions are typically calculated from the results of multiple operations applied on multiple ray traces.

We will assume that merit functions are composed of operations that are all differentiable, but that are too complex for hand-derivation of derivatives to be practical.

In this case, automatic differentiation (AD) is the tool of choice [9]. It is one of the main algorithms behind the progresses of deep learning the recent years. In the following section we will illustrate the concepts behind AD.

Let us first assume that our merit function can be described as an expression which can be put in the form of a directed acyclic graph (DAG). In Figure 1 such a DAG is represented for the calculation of the spherical sag. The sub-expression that is reused corresponds to the expression of $r^2 = x^2 + y^2$.

The DAG can be converted into a list of elementary operations that a computer will perform in sequence to compute the expression. In the first column of Table 1 each intermediary calculation is denoted by an intermediary variable a_i with a_{12} corresponding to the result of our computation.

All elementary operations of Table 1 can be differentiated. AD assumes that the computer has a database of differentiation rules for each elementary steps.

There are two modes to aggregate the results of elementary rules into a program, the forward mode and the reverse mode [9]. FORMIDABLE uses the forward mode that we introduce next.

In parallel of the computation of Table 1 we will also perform the calculation of its derivative. We introduce the notation:

$$\frac{\partial a}{\partial x} = \dot{a}.$$

The second column of Table 1 introduces the calculations of the derivative.

The main advantage compared to the tree approach introduced earlier is that each operation can be done in sequence, forwarding the results of the differentiation to the final result.

The disadvantage is that it requires to set the value of \dot{x} , \dot{y} and \dot{z} . These values are called the seeds and will be set for

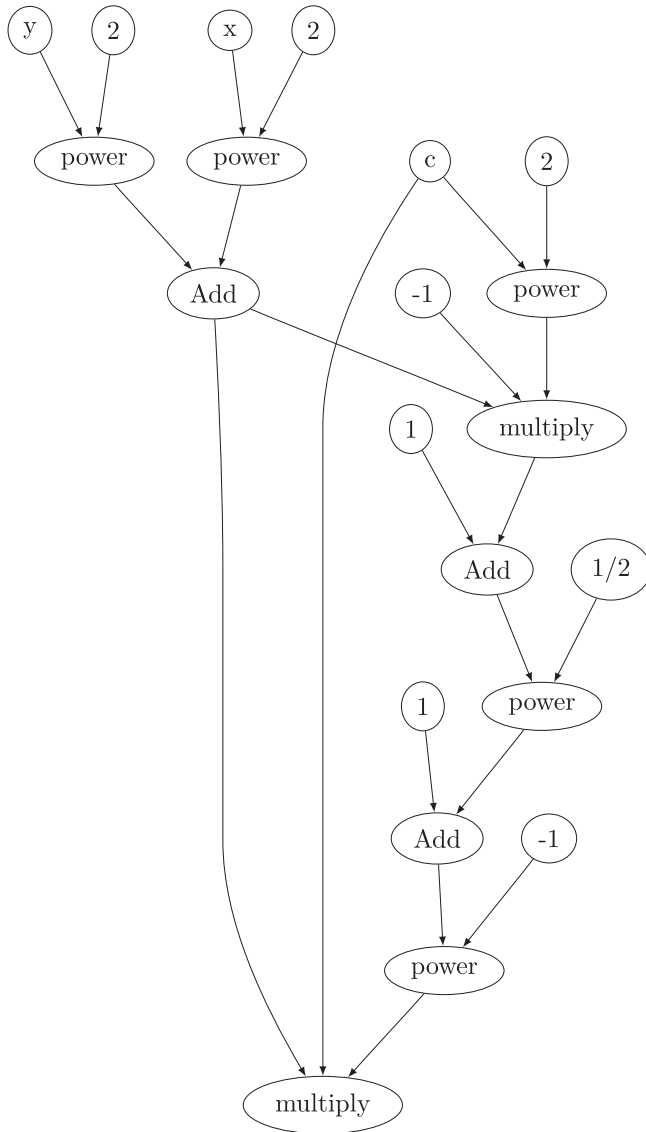


Figure 1. Directed acyclic graph for the calculation of the sphere sag.

example to $\dot{x} = 1$, $\dot{y} = 0$ and $\dot{c} = 0$ to compute the derivatives with respect to x .

So if we want to compute all three derivatives (because we have three inputs) we would have to repeat the computation 3 times with different seed values.

However if our calculation had multiple outputs, (more than one root in the DAG illustrated in Fig. 1) we would only have to perform the calculation once to have the derivatives of all the outputs with respect to one input.

This is one key property of forward mode differentiation, it is efficient for many outputs with few inputs, so it is well suited for ray-tracing as we will have typically more rays than degrees of freedom.

There are many different strategies to implement AD we will focus on the concepts behind the library `ForwardDiff.jl` [10] which FORMIDABLE is built upon.

This library is implemented in the Julia [11] programming language as is the core of FORMIDABLE.

Table 1. Operations to compute simulatenously sag and derivative in forward mode.

i	a_i	\dot{a}_i
0	y	\dot{y}
1	c	\dot{c}
2	x	\dot{x}
3	a_2^2	$2\dot{a}_2 a_2$
4	a_0^2	$2\dot{a}_0 a_0$
5	a_1^2	$2\dot{a}_1 a_1$
6	$a_3 + a_4$	$\dot{a}_3 + \dot{a}_4$
7	$-a_5 a_6$	$-\dot{a}_5 a_6 - \dot{a}_6 a_5$
8	$a_7 + 1$	\dot{a}_7
9	$\sqrt{a_8}$	$\frac{\dot{a}_8}{2\sqrt{a_8}}$
10	$a_9 + 1$	\dot{a}_9
11	$\frac{1}{a_{10}}$	$-\frac{\dot{a}_{10}}{a_{10}^2}$
12	$a_{11} a_1 a_6$	$\dot{a}_{11} a_1 a_6 + \dot{a}_1 a_{11} a_6 + \dot{a}_6 a_1 a_{11}$

In Figure 1, leaf nodes represent inputs and constant values, while the other nodes are operations.

For example multiply denotes the application of the multiplication operator to the inputs.

The idea behind `ForwardDiff.jl` implementation is to change the implementation of the operations in a way that allows the computation of the derivative.

For that `ForwardDiff.jl` uses Dual numbers. They bundle in the same variable a value (called primal value) and its derivatives.

Dual numbers can then be used to compute derivatives in the forward mode For example in Table 1 dual numbers would represent an entire row of the table (a_i, \dot{a}_i).

Then operators like the multiplication operator are re-defined to operate on dual numbers.

For example, the multiplication operator would be defined as in equation (10).

$$((a, \dot{a}), (b, \dot{b})) \mapsto (ab, \dot{a}b + a\dot{b}). \quad (10)$$

In terms of implementation, the library `ForwardDiff.jl` uses Julia multiple dispatch to call the correct function.

This is also made possible, because Julia is a functional language meaning that multiplication is implemented with conventional functions that can be extended by the user.

We conclude this section by noting that we have shown how AD can be used to differentiate complex programs as long as the elementary functions that compose this program have known differentiation rules.

Since we have also defined the differentiation rule for ray-tracing we now have all the components required to implement a generalized differential ray-tracer which we will describe in the next section.

3 Case study

The case study we chose to test FORMIDABLE and uncover potential limitations is inspired from the literature [7]. Primary properties are in Table 2.

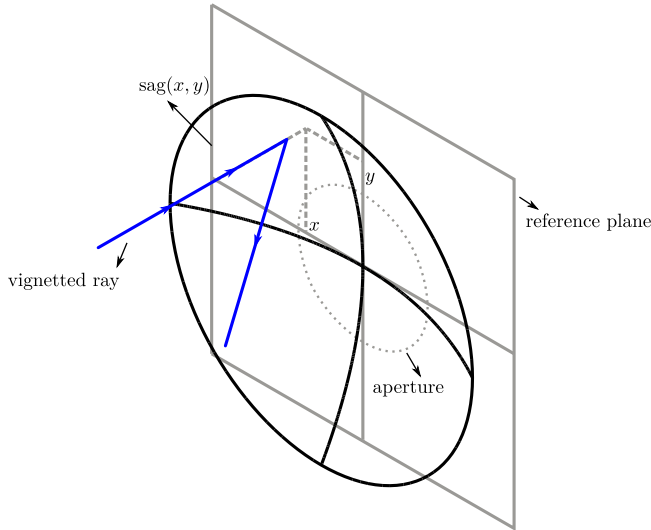


Figure 2. A ray vignetted by an aperture on a surface defined by a sag function.

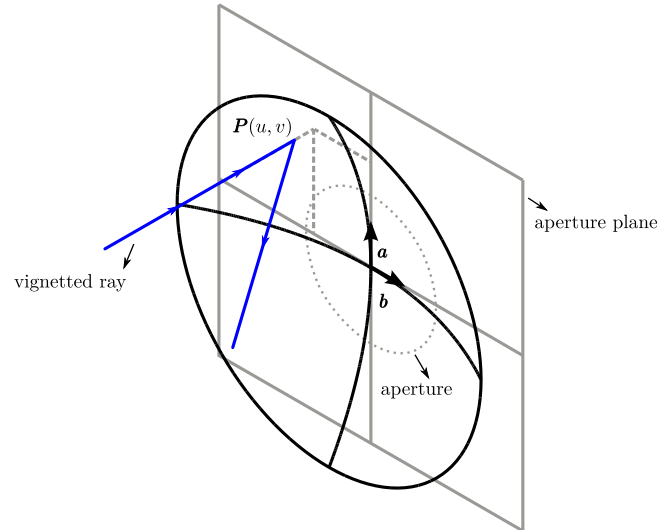


Figure 3. A ray vignetted by an aperture on a generic surface. The aperture plane is defined independently using \mathbf{a} and \mathbf{b} vectors (offset point omitted).

Table 2. Case study primary properties.

Focal length	357 mm
Field of view	$\pm 5 \times \pm 4.5^\circ$
Entrance pupil diameter	180 mm

In the remaining of the section we will first focus on three difficulties that were encountered: definition of aperture on NURBS surfaces, ray-aiming, NURBS and the choice of the optimisation algorithm, before presenting the results.

3.1 Projected aperture

Optical design software usually only consider surfaces that can be defined as sag surfaces.

A sag surface defines a point in a local coordinate system as in equation (11).

$$\mathbf{P} = (x, y, \text{sag}(x, y)). \quad (11)$$

NURBS surfaces are of a more general form described in equation (12).

$$\mathbf{P} = (x(u, v), y(u, v), z(u, v)). \quad (12)$$

In the case of a sag surfaces, we can define an aperture A as a function of x and y which are projection of \mathbf{P} on a reference plane as in Figure 2.

For example for a centered circular aperture A would be defined as in equation (13).

$$A = R^2 - x^2 - y^2. \quad (13)$$

In equation (13) R defines the radius of the aperture and A is negative if the ray is blocked by the aperture.

In the case of a NURBS u and v do not correspond to a projection on a plane. Often they do not even correspond to physical quantities as they are normalized between 0 and 1.

ProjectedAperture allows bypassing this problem by defining an aperture plane independent of the surface definition (see Fig. 3) and adds the following fields:

- aperture: a “classical” aperture,
- \mathbf{a} : tangent vector of the aperture plane,
- \mathbf{b} : tangent vector of the aperture plane (2nd direction),
- p : offset point, moves the coordinate system of the aperture plane.

A projected aperture will then project the intersection point onto the aperture plane plane, and apply the aperture on this plane. This allows to completely decouple the surface definition from the aperture definition.

3.2 Ray-aiming

The first preliminary tests of optimisation showed that it is critical to eliminate as much as possible discontinuities in the merit function. Some discontinuities are unavoidable, when the optimizer tries to evaluate the merit function for an unphysical system (impossibly large curvature for example). Some discontinuities are due to a Formidable calculation artifacts and are a false positive to be avoided.

False positives were found to come from ray-aiming and from the definition of the physical pupil shape.

Ray-aiming is particularly difficult in off-axis systems because it is difficult to locate the image of the physical pupil in the object space.

A simple ray-aiming implementation can be done naively by an optimisation whose guess was that the ray would originate from the center of the first surface (Fig. 4).

This could fail in two ways (in red in Fig. 4):

- Either because the guess provided was too far from the minima, so it would not converge.
- Or because they were multiple ray-aiming solutions.

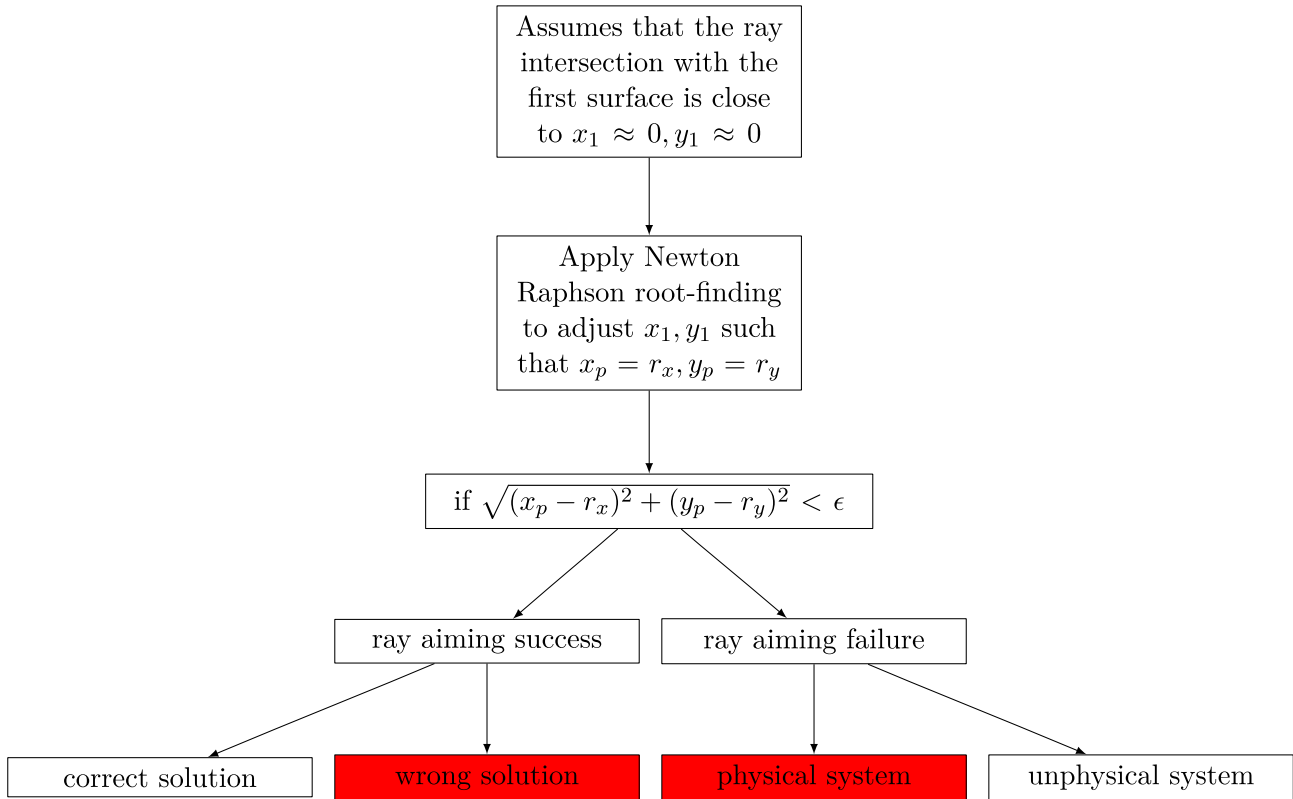


Figure 4. First naive algorithm for ray aiming.

The first case requires a better guess than the center of the first surface, the second case is harder to solve, and can typically happen where there is a reflection at a very shallow angle that manages to intersect the pupil at the correct location.

We solved the ray-aiming problem by an algorithm in three parts.

The first part deals with finding the On-axis Chief Ray (OAR) that is the chief ray of the center field.

Since this search is done relatively rarely we invest a significant amount of computational resources to ensure that the correct OAR is found.

This is done in the following steps:

- Tracing in reverse from the center of the pupil, we scan the full sphere of incidence angle to search for directions that trace back to the entrance. We obtain reverse OAR candidates that intersect the pupil at the correct location but do not originate from the correct field points.
- Tracing in the direct direction and using as a guess the intersection of the OAR candidates with the first surface we adjust this intersection to find direct OAR candidates that intersect the pupil at the correct location and come from the correct field points. We obtain less direct OAR candidates than reverse OAR candidate because the adjustment is not always successful. Furthermore, most of the direct OAR candidates converge to the same ray-aiming solution.
- We eliminate non-physical rays that is rays whose direction is sometime opposite the physical

propagation of light. To allow tracing to virtual surfaces, (exit pupil for example) we disregard this rule when the direction violation occurs before or after a dummy surface.

- If there is still multiple candidates we select the one that has paraxial derivatives that come closer to an imaging function.

This process is depicted in [Figure 5](#). If the system is axial symmetric, this procedure can be disabled.

Now that we have an OAR we can compute the physical pupil shape. For this we trace a sample of rays along the edge of the target pupil shape in the object space to the physical pupil and we fit the physical pupil to this sample.

This is done with typically 100 rays, much more than what optical design software will typically do. This large number avoids a situation where a couple of NURBS control points could be used by the optimizer to artificially reduce the pupil size.

Once we have the physical pupil shape determined we can compute a raymap.

We progressively fill a hyper cube sampling field, pupil and wavelength of rays. Each ray is traced using the closest ray already traced as a guess back to the OAR, ensuring continuity of the ray map.

If the sampling is sufficiently dense, the expensive procedure for the OAR does not need to be repeated for each ray.

Once the raymap is filled, each ray trace will first look in the raymap for the closest ray that has been traced and use

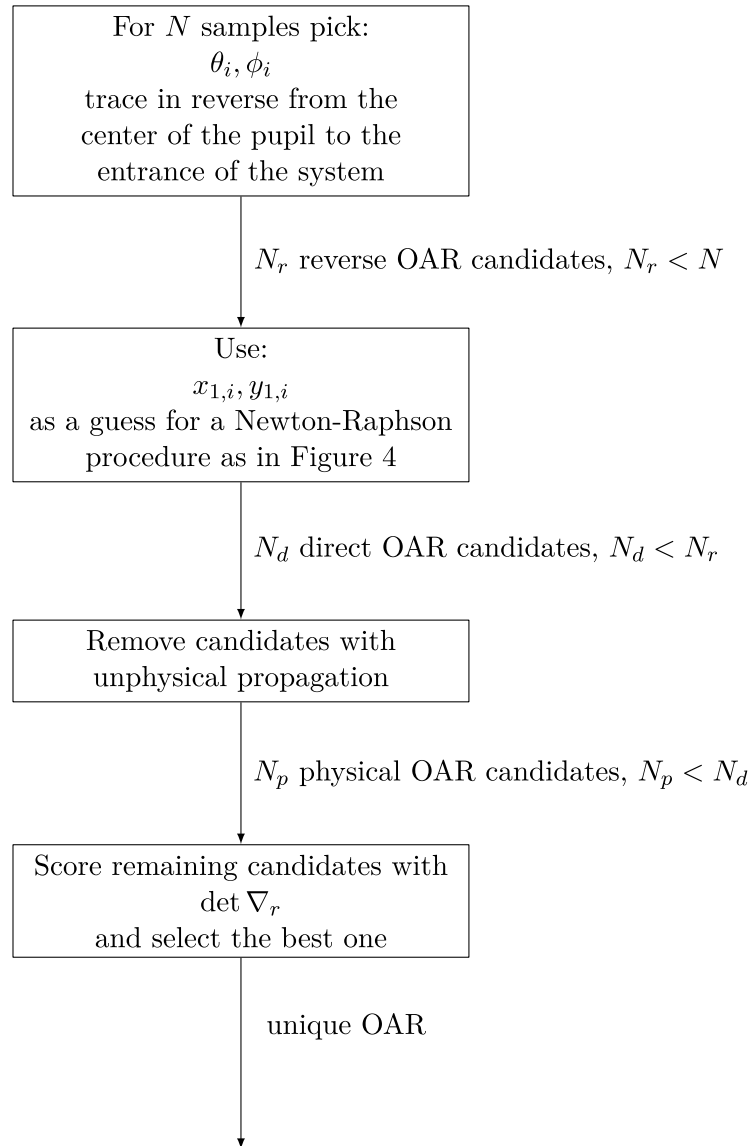


Figure 5. OAR determination.

it as a guess. This ensures very robust ray-aiming and is also a speed improvement.

3.3 Optimisation

Testing showed that the Levenberg–Marquardt algorithm (also known as damped-least squares) has remarkably good performance on optical problems.

Since we are using forward-mode differentiation, we get the jacobian matrix for free, which would not be the case in reverse-mode differentiation.

Typical optical merit function try to minimize the quadratic sum of components that we will call residuals.

Levenberg–Marquadt tries to bring as close to zero as possible the residuals. This should be completely equivalent but as explained in literature [12] under the assumption that this residuals will indeed come closer to zero it allows Levenberg–Marquardt to behave like 2nd order derivative

information was provided. Of the open-source implementations available Levenberg–Marquardt implementation available in SciPy [13] was found to be the most efficient implementation in terms of speed of convergence and quality of the solution found.

Therefore, the direction of descent chosen is usually more efficient than just following the gradient of the merit function.

With this observation, convergence was faster but the final performance still disappointing. A more careful investigation showed that the jacobian matrix conditioning was poor and the effective rank as calculated numerically was deficient.

This was solved by first removing degrees of freedom that were redundant. For example, the effect of decenters of entire NURBS on the merit function was found numerically to be strongly correlated with the control points displacement.

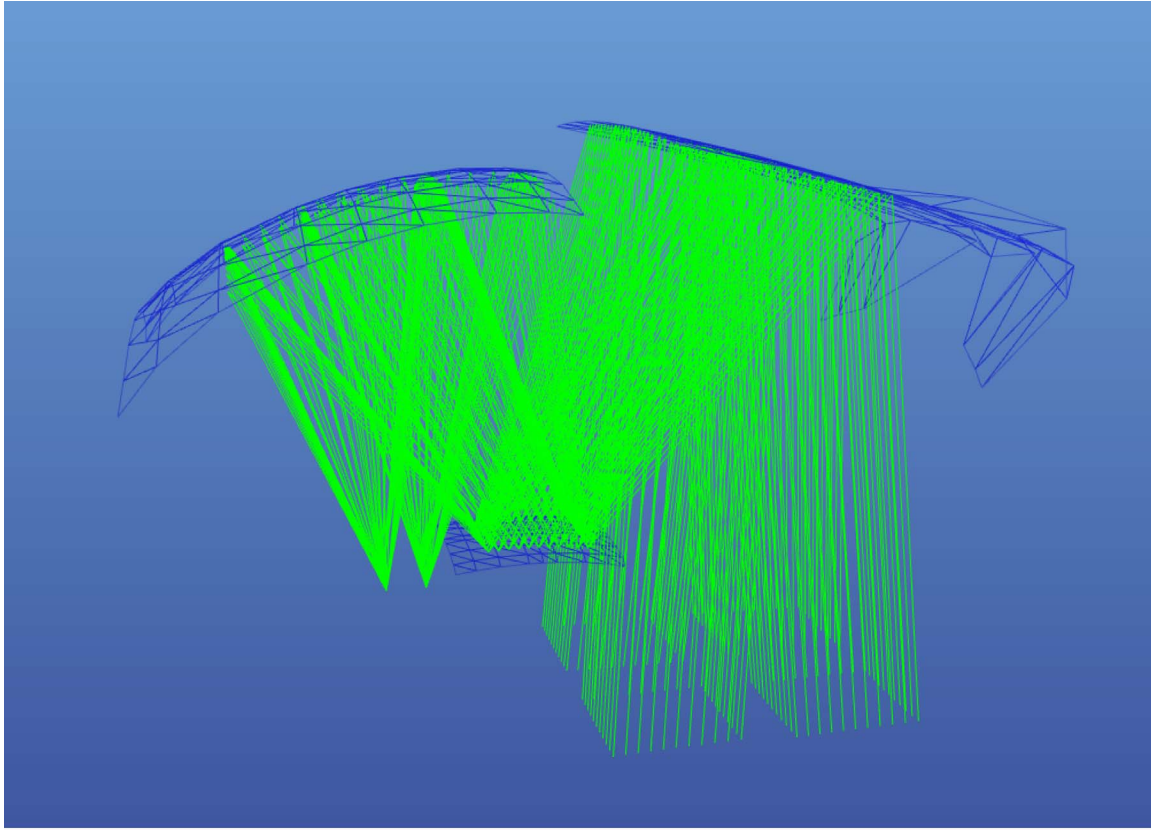


Figure 6. System with optimised NURBS.

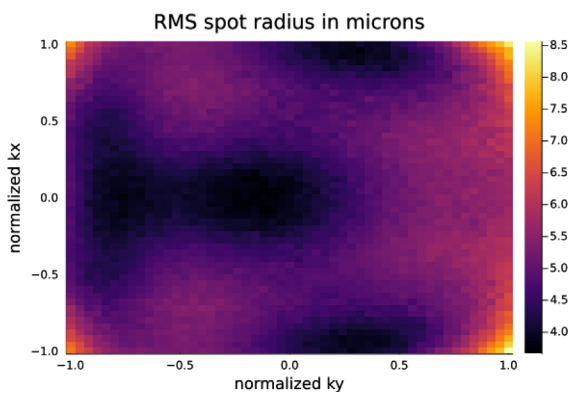


Figure 7. RMS spot radius field map.

Lastly, the remaining problem was with control points of the NURBS that were not contributing in the first order of the merit function. This is typically control points outside of the circular footprint because typical NURBS will have a rectangular bounding curve.

We first tried to solve this problem by adding a penalty either on the curvature of the NURBS (penalized to stay as constant as possible) or on the departure from the starting point. This was found to converge to poor results. What was successful was to optimise a rectangular pupil shape to ensure that every grid point of the NURBS contributes.

This requires each NURBS to be tailored to be just slightly larger than its effective aperture before optimisation, except the pupil which needs a larger oversize due to the way the physical stop is computed.

The cause of this problem is that we define NURBS with a regular grid of points. To directly optimize circular pupils and NURBS, it will be necessary to investigate optimisation of irregular grids in FORMIDABLE.

A system obtained with this method is depicted in [Figure 6](#), with its RMS spot radius field map depicted in [Figure 7](#). For comparison a system was design with the same properties in CodeV using polynomials with inferior performance as depicted in [Figure 8](#).

4 Software architecture

4.1 The choice of Julia

Julia [11] is a high-performance language that aims at a familiar syntax for users of Matlab [14] and Python [15].

It solves the two language problems encountered by Python users when a language convenient to use (Python) must be often supplemented by another language for the performance demanding parts.

The two language problems has the following drawbacks:

- Developer of the software need to be proficient in two languages.

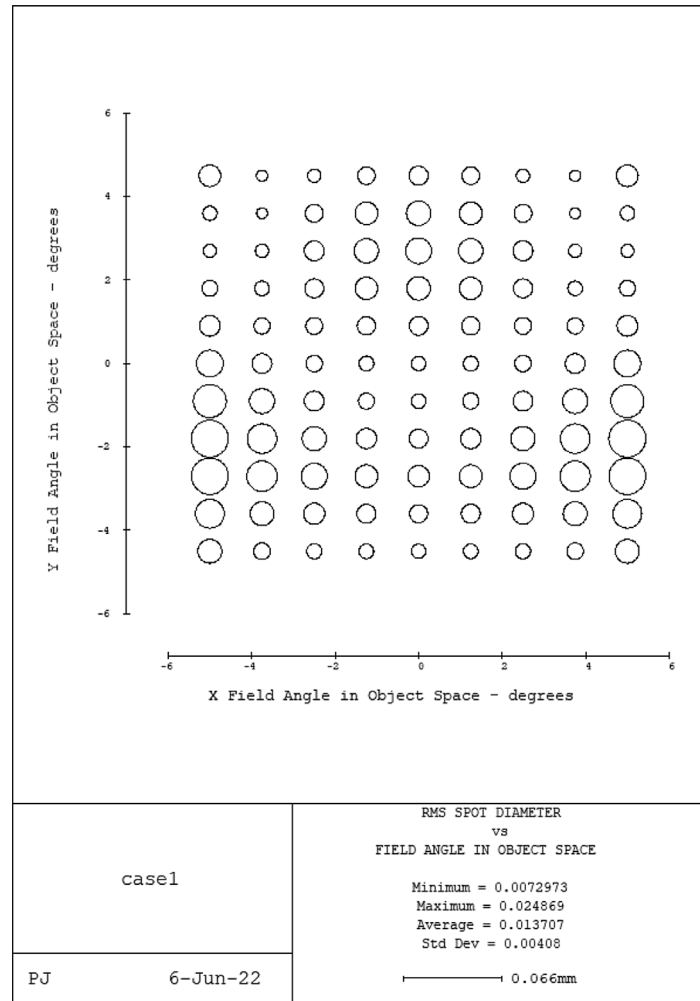


Figure 8. RMS spot radius field map.

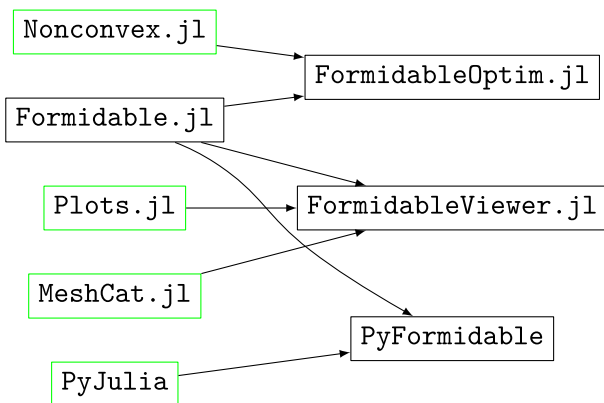


Figure 9. Package hierarchy and main dependencies.

- Users of such software also need to be learned in two languages if they want to have a deep understanding of the internal workings.
- External tools such as debuggers, profilers are difficult to use.

Solving the two language problem is achieved by just-in-time compilation. In a nutshell, when a Julia function is called, Julia analyses the arguments type (e.g. float, integer or more complex types) and compiles an optimised version of this function (called a method) for this particular argument type combination (called a signature).

This strategy is also employed by Numba [16], a Python library aimed at high performance.

It has the advantage that while Numba is limited to a subset of Python functionalities in Julia no such limitations exist.

Finally Julia has great interoperability with Python [17], allowing Python users to use FORMIDABLE easily.

FORMIDABLE is divided in packages to simplify the management of the third-party dependencies. It allows the end-user to install only necessary packages without many useless dependencies.

Therefore, functions and data structures are grouped based on bringing-in similar dependencies as illustrated in Figure 9.

Formidable.jl is the core packages including everything relating to the raytrace of an optical system and the definition of the optical system.

It also includes dependencies required to perform the differential ray-tracing.

All other packages depend on it.

`FormidableViewer.jl` includes the plotting functions and depends on third party plotting libraries such as `Plots.jl` [18] and `MeshCat.jl` [19] that we did not want to add as dependencies to `Formidable.jl` core.

`FormidableOptim.jl` implements conversion utilities to allow optimisation of optical merit functions with third-party optimisation libraries.

`FormidableOptim.jl` only implements a bridge to `Nonconvex.jl` [20] and to `SciPy` [21].

`PyFormidable` is a python package that implements the bridge to python.

4.2 Numerical performance

FORMIDABLE has been developed not with the goal of having the ultimate performance but with the goal of being performant enough, comparable with performance indicated in literature [7].

On a CPU clocked at 2.2 FORMIDABLE is capable of computing a ray, NURBS intersection in 0.6 μ s.

Furthermore, in the merit function calculation, batch ray tracing is parallelized on all available CPU threads.

Automatic differentiation brings advantages in performance as well, the merit function of the case study presented above is composed of 228 degrees of freedom and is evaluated in 116 ms while its gradient takes 4.581 s to evaluate.

So the finite differentiation logically takes $2 \times 228 \times 116$ ms = 52.9 s on computation only, automatic differentiation is about 11.5 \times faster.

This is due to the fact that some expensive calculations (especially the ray-aiming and computation of the raymap) are done only once, and the derivatives calculated without having to redo the ray-aiming twice for each degree of freedom.

4.3 Functionalities

At the time of writing this article FORMIDABLE implements the following functionalities:

- Surfaces
 - Conic/aspherical surfaces
 - XY polynomials
 - ZERNIKE polynomials
 - NURBS
 - Linear diffraction gratings
- Analysis
 - 2D and 3D layout (discontinuities plotted on a 3D layout)
 - Transverse ray diagrams
 - Spot diagrams
 - Wavefront error maps
 - Distortion maps
 - NURBS curvature maps

Due to its nature (julia programming and available source) it is comparatively very easy to implement new functionalities.

FORMIDABLE [22] is available under the ESA Software Community Licence [23] following registration.

Funding

This activity was funded by the European Space Agency under contract 4000136450/21/NL/AR.

References

- 1 Rolland J.P., Davies M.A., Suleski T.J., Evans C., Bauer A., Lambropoulos J.C., Falaggis K. (2021) Freeform optics for imaging, *Optica* **8**, 2, 161–176.
- 2 Schiesser E.M., Bauer A., Rolland J.P. (2019) Effect of freeform surfaces on the volume and performance of unobscured three mirror imagers in comparison with off-axis rotationally symmetric polynomials, *Opt. Express* **27**, 15, 21750–21765. <https://doi.org/10.1364/OE.27.021750>.
- 3 Forbes G.W. (2012) Characterizing the shape of freeform optics, *Opt. Express* **20**, 3, 2483–2499. <https://doi.org/10.1364/OE.20.002483>.
- 4 Chrisp M.P. (2014) New freeform NURBS imaging design code, in: *International Optical Design Conference*, Optical Society of America, paper ITh3A-7.
- 5 *FANO: fast accurate Nurbs optimization*. Available at <https://sc22.mghpcc.org/project/fast-accurate-nurbs-optimization-fano/>.
- 6 Volatier J.B., Mendiña-Fernández Á., Erhard M. (2017) Generalization of differential ray tracing by automatic differentiation of computational graphs, *J. Opt. Soc. Am.* **34**, 7, 1146–1151. <https://doi.org/10.1364/JOSAA.34.001146>.
- 7 Chrisp M.P., Primeau B., Echter M.A. (2016) Imaging freeform optical systems designed with NURBS surfaces, *Opt. Eng.* **55**, 7, 071208. <https://doi.org/10.1117/1.OE.55.7.071208>.
- 8 Abert O.P. (2005) Interactive ray tracing of NURBS surfaces by using SIMD instructions and the GPU in parallel. *Diploma Thesis*, Nanyang Technological University. Available at https://userpages.uni-koblenz.de/~cg/Diplomarbeit/DA_Oliver_Abert.pdf.
- 9 Baydin A.G., Pearlmutter B.A., Radul A.A., Siskind J.M. (2017) Automatic differentiation in machine learning: a survey, *J. Mach. Learn. Res.* **18**, 5595–5637.
- 10 Revels J., Lubin M., Papamarkou T. (2016) *Forward-mode automatic differentiation in Julia*. Available at <http://arxiv.org/abs/1607.07892> (visited on 07.08.2019).
- 11 Bezanson J., Karpinski S., Shah V.B., Edelman A. (2012) *Julia: A fast dynamic language for technical computing*. Available at <https://arxiv.org/abs/1209.5145>.
- 12 Schittkowski K. (1988) Solving constrained nonlinear least squares problems by a general purpose SQP-method, in: *Trends in Mathematical Optimization: 4th French-German Conference on Optimization*, Springer, pp. 295–309.
- 13 Moré J.J. (1978) The Levenberg–Marquardt algorithm: implementation and theory, in: Watson G.A. (ed), *Numerical Analysis*, Springer, Berlin Heidelberg, pp. 105–116. ISBN: 978-3-540-35972-2.
- 14 MATLAB (2010) *Version 7.10.0 (R2010a)*, The MathWorks Inc., Natick, Massachusetts.

- 15 Van Rossum G., Drake F.L. (2009) *Python 3 reference manual*, CreateSpace, Scotts Valley, CA. ISBN: 1441412697.
- 16 Lam S.K., Pitrou A., Seibert S. (2015) Numba: A LLVM-based python JIT compiler, in: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*, Association for Computing Machinery, New York, NY, pp. 1–6. <https://doi.org/10.1145/2833157.2833162>.
- 17 Arakaki T., Bolewski J., Deits R., Fischer K., Johnson S.G., Bussomier M., Norton I., Haraldsson P., Rocklin M., Shah V.B., Soto D. (2020) *JuliaPy/pyjulia: PyJulia v0.5.6*. Version v0.5.6. <https://doi.org/10.5281/zenodo.4294940>.
- 18 Christ S., Schwabeneder D., Rackauckas C., Borregaard M. K., Breloff T. (2023) Plots.jl – a user extendable plotting API for the julia programming language, *J. Open Res. Soft.* **11**, 1, 5. <https://doi.org/10.5334/jors.431>.
- 19 *MeshCat.jl*. Available at <https://github.com/rdeits/MeshCat.jl> (visited on 11.20.2023).
- 20 *Nonconvex.jl*. Available at <https://github.com/JuliaNonconvex/Nonconvex.jl> (visited on 11.20.2023).
- 21 Virtanen P., Gommers R., Oliphant T.E., Haberland M., Reddy T., Cournapeau D., Burovski E., Peterson P., Weckesser W., Bright J., Van der Walt S.J. (2020) SciPy 1.0: fundamental algorithms for scientific computing in Python, *Nat. Methods* **17**, 3, 261–272. <https://doi.org/10.1038/s41592-019-0686-2>.
- 22 *Formidable*. Available at <https://gitlab.space-codev.org/formidable/formidable> (visited on 11.20.2023).
- 23 *ESCL*. Available at <https://essr.esa.int/license/european-space-agency-communitylicense-v2-4-strong-copyleft-type-1> (visited on 11.20.2023).