Extending IPOL to New Data Types and Machine-Learning Applications

Miguel Colom(✉)

Centre de mathématiques et de leurs applications, CNRS, ENS Paris-Saclay, Université Paris-Saclay, 94235 Cachan cedex, France
colom@cmla.ens-cachan.fr

Abstract. Image Processing On Line (IPOL) is a journal focused on mathematical descriptions of image processing (IP)/computer vision (CV) algorithms. Since the first article was published in 2010, it has started to become clear that the IP/CV discipline is mainly multi-disciplinary. For example, nowadays images are de-noised using convolutional neural networks (CNN), and fields such as neurophysiology need of the rudiments and techniques of IP/CV, general signal processing and artificial intelligence. IPOL wants to extend the capabilities of its demo system to cope with these needs. Specifically, in this article we review the state of the current demo system and its limitations. It enunciates a detailed project on how to build a more adapted system, and its minimal requirements: new data types, problematic heterogeneous data, the pre-processing and standardization, the possibility to chain different algorithms in a complex chain, and how to compare them.

1 Introduction

Image Processing on Line (IPOL) is a research journal founded in 2010 on Reproducible Research in the field of Signal Processing (mainly Image Processing, but also video, 3D pointclouds/meshes, and audio), giving a special emphasis on the role of mathematics in the design of the algorithms [1].

As pointed by Donoho et al. [2], there is a crisis of scientific credibility since in many published papers it is not possible for the readers to reproduce exactly the same results given by the authors. The causes are many, including incomplete descriptions in the manuscripts, not releasing the source code, or that the published algorithm does not correspond to what actually is implemented. Each IPOL article has an online demo associated which allows users to run the algorithms with their own data; the reviewers of the IPOL articles must carefully check that both the description and the implementation match.

Since it started in 2010, the IPOL demo system has been continuously improved and according to usage statistics collected along these years, it has about 250 unique visitors per day. However, several problems of design and potential improvement actions were identified and it was decided to build a second version of the system based on microservices [3].

B. Kerautret et al. (Eds.): RRPR 2018, LNCS 11455, pp. 3–24, 2019.
https://doi.org/10.1007/978-3-030-23987-9_1

The full redesign of the demo system proposed in the project will not only solve the problems enumerated before, but will allow also to expand IPOL to much more complex data types and thus to new applications. In particular, we are especially interested in multidimensional signals since they have applications in physiological analysis (data from accelerometers, oculometry, EGC, EMG, EEG), as well as mixtures of data types (for example, time-series signals along with text).

Another important extension of IPOL will be to allow Machine Learning applications. Specifically, to allow the algorithms to be able to explore the set of experiments performed by the users as training data. A typical task which can be learned is the choice of the best parameters to run the algorithm according to the characteristics of the input data. For example, in the case of images the values of the parameters needed to segment a hyperspectral image in general depend on the number of channels and other parameters on the image which could be learned automatically by the system. In the case of Machine Learning algorithms, the space of hyperparameters can vary from a few of them (up to ten for the main Machine Learning algorithms) to millions in the case of deep learning applications. The choice of the hyperparameters is an open question under intensive research[1] and the interest of a learning step to automatically choose the best parameters is clear.

Machine Learning algorithms require important changes in the architecture of the IPOL system to allow not only new data types, but also adding preprocessing steps, optimal data storage to allow fast queries, management of databases, and comparison of algorithms. The system will also require training processes which run continuously and explore the contents of the archive of experiments.

After the first four years of publications, the problems noted before were detected [4] and the architecture of the system was redesigned by a group of volunteers and part-time collaborators. We arrived at a working prototype of a new demo system which is scalable, easy to debug, and which implements the automatic generation of demos, thus alleviating the work load of the editor and thus allowing for fast demo publishing. Section 2 discusses the design of this preliminary system. Section 3 presents the plan to extend IPOL to a wide-purpose platform which allows to run algorithms with much more complex data types and multiple application fields (medical, research, industrial).

Section 4 discusses the business model of this new platform, and presents a plan to recover the investment and make the project auto-sustainable, with an estimation of the costs. Finally Sect. 5 concludes this article.

2 The Current (New) System

In 2014 we identified [4] several technical problems related to the architecture of the demo system, including the lack of modularity, tightly-coupled interfaces,

[1] See for example Cedric Malherbe's PhD dissertation: http://www.theses.fr/s144139.

difficulties to share the computational load along different machines, or complicated debugging of the system in case of malfunction. We found also editorial problems, such as the slow and rigid procedure that the editors needed to follow to create or modify demos.

After a careful analysis of the system and with the knowledge accumulated in those first years, we decided that the best option was to move to a more flexible architecture oriented to (micro)services (Sect. 2.1). We understood that the slow process to create and edit demos was a bottleneck in the first version of the demo system and thus we decided to create an abstract specification for the demos (the Demo Description Lines, or *DDL*) to allow automatic and fast demo generation (Sect. 2.2). And since we needed to have a tool which could be used by non-technical editors, we created a graphical web interface to create and edit demos, as well as the associated data, the Control Panel (Sect. 2.3).

After the improvements in the system, we have now:

- A function architecture of microservices, with 7 modules
- Fast demo creation. The new system accounts at this moment for 39 published demos, 6 pre-prints, and 72 workshops. The fact that there is a large number of workshops compared to the publications confirms that now it is easy and fast for our editors to create new demos
- Video demos
- Audio demos
- Interactive controls almost finished. They will be ready in a few weeks.
- The possibility to add custom Javascript code to the demos for editors with special requirements, without complexifying the demo system itself
- A system easier to debug when a bug is detected. Now it is really easy to track down bugs, since every module has its own logging system and automatic alerts. At this moment there are no known bugs in the system, and if they appeared they would be detect, the engineers warned, and fixed immediately.

Adding completely new features such as support for machine learning applications with a training step will require intense engineering efforts and more engineers in the team to build it in a reasonable time.

2.1 The IPOL Demo System Architecture

The architecture of the new IPOL demo system is an Service-Oriented Architecture (SOA) based on microservices[2]. This change was motivated by the problems found in the previous version of the demo system. First, it was designed as a monolithic program[3] which made it quite easy to deploy in the servers and to run it locally, but at the cost of many disadvantages. Given that it was a monolithic system, it was difficult to split it into different machines to share the computational load of the algorithms being executed. A simple solution would be to create specialized units to run the algorithms and to call them from the

[2] We use Python 3 but in fact the microservices may be written in any language.
[3] Of course, with a good separation of functionality among different classes.

monolithic code, but this clearly evokes the first step to move to a microservices architecture. Indeed, this first step of *breaking the monolith* [3] can be iterated until all the functions of the system have been delegated in different modules. In the case of IPOL, we created specialized modules and removed the code from the monolith until the very monolith became a module itself: Core. This Core module is in charge of all the system and delegates the operations to other modules. Figure 1 summarizes the IPOL modules and other components in the architecture of the system.

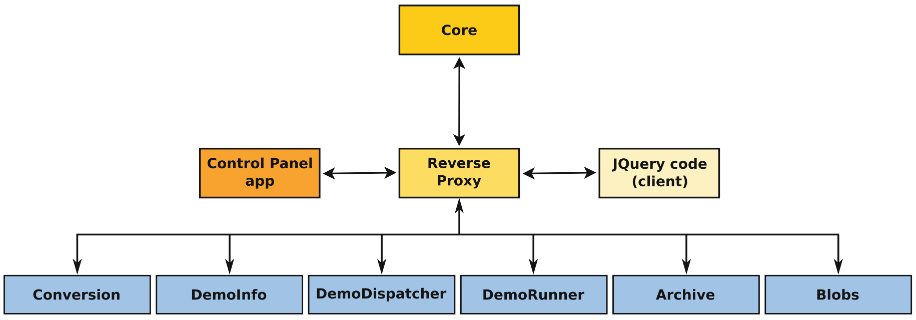


Fig. 1. Modular architecture of the current IPOL demo system.

Other problems we had in the previous version of the demo system got solved when we moved to the microservices architecture. Since there is a loose coupling between Core and the other modules (in the sense that they are independent programs which communicate via an HTTP API), different members of the development team can work at the same time without worrying about the implementation details or data structures used in other parts of the system. Also, tracking down malfunctions is easier: since Core centralizes all the operations, when a bug shows it can only be generated either at Core or at the involved module, but not at any other part of the system. In the old system a bug could be caused by complex conditions which depend on the global state of the program, making debugging a complex task. And as noted before, the fact that the architecture of the system is distributed and modular by design makes it very natural and simple to have mechanisms to share the computational load among several machines.

Hiding the internal implementation details behind the interfaces of the modules is an essential part of the system, and it is needed to provide loose coupling between its components. The internal architecture of the system is of course hidden from the users when they interact with the system, but it is also hidden *from the inside*. This means that any module (Core included) does not need to know the location of the modules. Instead, all of them use a published API.

Once the API is defined, the routing to the modules is implemented by a reverse proxy[4]. It receives the requests from the clients according to this pattern: `/api/<module>/<service>` and redirects them to the corresponding module. Figure 2 shows how the API messages received by the proxy are routed to the corresponding modules, thus hiding the internal architecture of the system.

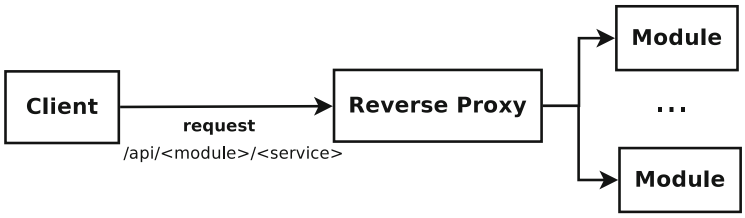


Fig. 2. The reverse proxy routes the API messages to the corresponding modules.

Modules: The IPOL demo system is made of several standalone units used by the Core module to delegate specialized and well isolated functions. This section describes briefly these microservices modules.

Blobs (Binary Large OBjects): each demo of IPOL offers the user a set of default blobs which can be tagged and linked to different demos. Thus, the users are not forced to supply their own files for the execution of the algorithms. The term *blob* in IPOL refers to any file that can be used as an input, regardless its type. In an image-processing demo they would be images, but they can be also videos, or even text files representing physiological signals, for instance. The system does not need to know their particular type since the execution flow is always the same (load inputs, eventually convert them, run the algorithm, store the results in the archive, and show results). Only when needed (for example, when the Conversion module needs to perform some kind of transformation in the data) the corresponding module will request its actual type. This module introduces the concept of *templates*, which are sets of blobs which can be associated to a particular demo. For example, this allows all the demos of a specific type (e.g., de-noising) to share the same images as default input data. Instead of editing each demo one by one, the editors can simply edit their template to make changes in all the demos, and then particular changes to each specific demo.

Archive: the Archive module stores all the experiments performed by the IPOL with their original data. The database stores the experiments and blobs, which are related with a junction table with a many-to-many relationship. It is worth noting that the system does not store duplicates of the same blob, but detects them from their SHA1 hash.

This module offers several services, such as adding (or deleting) an experiment or deleting all the set of experiments related to a particular demo. The

[4] We use Nginx as the reverse proxy.

archive also has services to show particular experiments or several pages with all the experiments stored since the first use of the archive.

Core: this module is the centralized controller of the whole IPOL system. It delegates most of the tasks to the other modules, such as the execution of the demos, archiving experiments, or retrieving metadata, among others.

When an execution is requested, it obtains first the textual description of the corresponding demo by using the DDL from DemoInfo and it copies the blobs chosen by the users as the algorithm's input. Then, it asks for the workload of the different DemoRunners and gives this information to Dispatcher in order to pick the best DemoRunner according to the Dispatcher's selection policy. Core asks the chosen DemoRunner to first ensure that the source codes are well compiled in the machine and then to run the algorithm with the parameters and inputs set by the user. Core waits until the execution has finished or a timeout happens. Finally, it delegates to Archive to store the results of the experiment. In case of any failures, Core terminates the execution and stores the errors in its log file. Eventually, it will send warning emails to the technical staff of IPOL (internal error) or to the IPOL editors of the article (compilation or execution failure).

Dispatcher: in order to distribute the computational load along different machines, this module is responsible of assigning a concrete DemoRunner according to a configurable policy. The policy takes into account the requirements of a demo and the workload of all DemoRunners and returns the one which fits best. The DemoRunners and their workloads are provided by Core. Figure 3 shows the communication between Core, Dispatcher, and the DemoRunner modules.

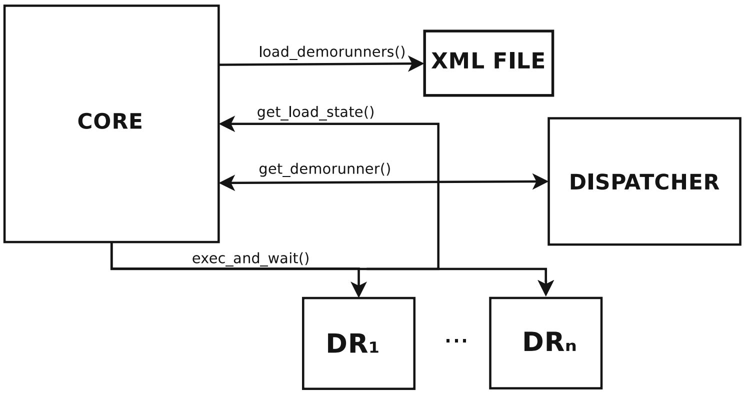


Fig. 3. Communication between Core, Dispatcher, and the DemoRunner modules.

Currently Dispatcher implements three policies:

- **random:** it assigns a random DemoRunner
- **sequential:** it iterates sequentially the list of DemoRunners;
- **lowest workload:** it chooses the DemoRunner with the lowest workload.

The choice of the actual policy is a configuration option in the module. It can be changed by the system operators at any moment, but the module can not change the policy depending on the execution needs of the system so far.

Any policy selects only those DemoRunners that satisfy the requirements (for example, having MATLAB installed, or a particular version of openCV).

DemoInfo: the DemoInfo module stores the metadata of the demos. For example, the title, abstract, ID, or its authors, among others. It also stores the abstract textual description of the demo (DDL). All this information can be required by Core when executing a demo or by the Control Panel when the demo is edited with its website interface.

It is possible that the demo requires non-reviewed support code to show results. In this case, the demo can use custom scripts to create result plots. Note that this only refers to scripts and data which is not peer-reviewed. In case they are important to reproduce the results or figures in the article, they need to be in the peer-reviewed source code package.

DemoRunner: this module controls the execution of the IPOL demos. The DemoRunner module is responsible of informing Core about the load of the machine where it is running, of ensuring that the demo execution is done with the last source codes provided by the authors (it downloads and compiles these codes to maintain them updated), and of executing the algorithm with the parameters set by the users. It takes care of stopping the demo execution if a timeout is reached, and to inform Core about the causes of a demo execution failure so Core can take the best action in response. In order to distribute and share the computational load of the whole system, several demoRunner instances can be deployed on different machines. Each of the instances can declare different capabilities, as for example being able to run MATLAB code, of the presence of a GPU.

2.2 Automatic Demo Generation

In the previous version of the IPOL demo system the demo editors had to write Python code to create a new demo. Specifically, to override some methods in a base demo class in order to configure its input parameters, to call the program implementing the algorithm, and also to design Mako HTML templates for the results page.

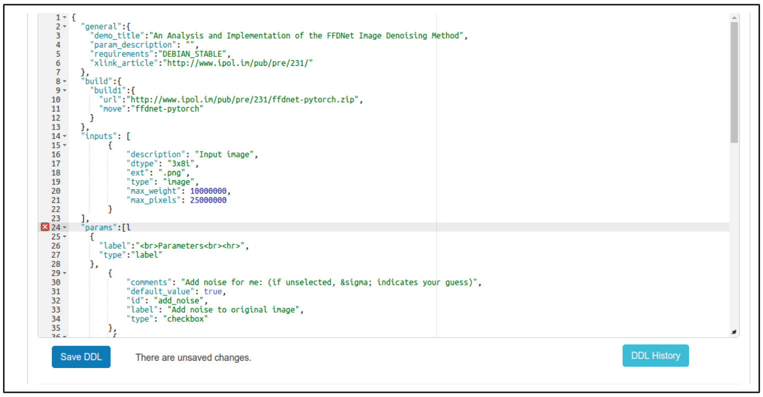
This approach does not really solve anything, and it simply passes the problem (the inability of the system to generate demos from a simple textual description) to the hands of the demo editors, that are forced to write code and HTML templates. Of course, these are the kind of tasks that can be automated and should not be made by human editors.

When Python code was written by the demo editors, it was prone to bugs which are specific to each demo. Moreover, fixing a bug in a particular demo does not guarantee that the same solution as-is could be applied to other demos with the same problem. Indeed, different demo editors might have written different codes in each demo to solve the same problem.

In fact, it is evident that this design can be simplified and the tasks automated. Indeed, to completely define a demo all one needs is:

1. The title of the demo;
2. the URL where the system should download the source code of the demo;
3. the compilation instructions;
4. the type interaction of the user with the demo (for example, drawing segments over the image, or annotating an input signal, or simply picking one image);
5. the input parameters along with their type and default values;
6. a description of what needs to be shown as results.

This information is not tied to any particular language or visualization technique, but it can be a simple abstract textual description. In IPOL we called this abstract textual description the Demo Description Lines (DDL). The IPOL editors only need to write such a description and the system takes care of making available the demo according to it. This not only avoids any coding problems (since there is nothing to code, but writing the corresponding DDL), but also allows IPOL to have non-expert demo editors, and makes it possible to edit and publish demos quickly. As an example, Fig. 4 shows the graphical tool used to edit the DDL of a demo within the Control Panel tool.



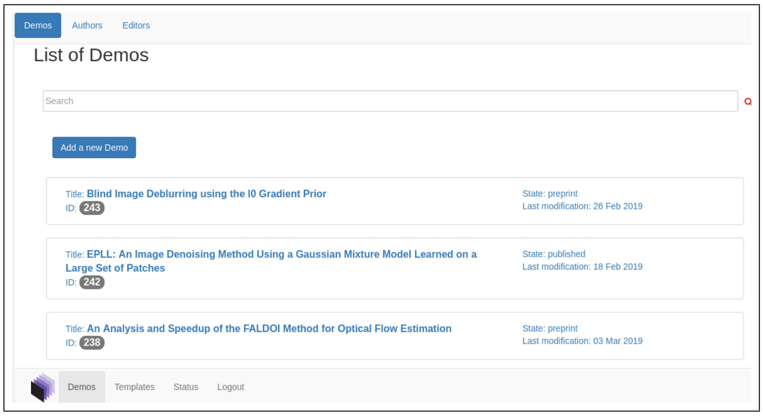
```
{
  "general":{
    "demo_title":"An Analysis and Implementation of the FFDNet Image Denoising Method",
    "param_description": "",
    "requirements":"DEBIAN_STABLE",
    "xlink_article":"http://www.ipol.im/pub/pre/231/"
  },
  "build":{
    "build1":{
      "url":"http://www.ipol.im/pub/pre/231/ffdnet-pytorch.zip",
      "move":"ffdnet-pytorch"
    }
  },
  "inputs": [
        {
            "description": "Input image",
            "dtype": "3x8i",
            "ext": ".png",
            "type": "image",
            "max_weight": 10000000,
            "max_pixels": 25000000
        }
  ],
  "params":[l
    {
      "label":"<br>Parameters<br><hr>",
      "type":"label"
    },
        {
            "comments": "Add noise for me: (if unselected, &sigma; indicates your guess)",
            "default_value": true,
            "id": "add_noise",
            "label": "Add noise to original image",
            "type": "checkbox"
        },
```

There are unsaved changes.

Fig. 4. Graphical tool to edit the DDL of a demo in the Control Panel.

2.3 Editorial Management: The Control Panel

The Control Panel is a web application which offers a unified interface to configure and manage the platform, intended for non-technical editors. It allows to create/modify/delete the list of IPOL editors, the demos along their textual descriptions, upload new testing dataset, and to customize the demos with extra code for advanced users who need it. It allows also editors to remove specific experiments upon request (for example, in the case of copyright images) or if



List of Demos

Title: Blind Image Deblurring using the l0 Gradient Prior
ID: 243
State: preprint
Last modification: 26 Feb 2019

Title: EPLL: An Image Denoising Method Using a Gaussian Mixture Model Learned on a Large Set of Patches
ID: 242
State: published
Last modification: 18 Feb 2019

Title: An Analysis and Speedup of the FALDOI Method for Optical Flow Estimation
ID: 238
State: preprint
Last modification: 03 Mar 2019

Fig. 5. List of demos as shown in the Control Panel.

inappropriate content is found. Figure 5 shows a screenshot of the Control Panel application as shown in the browser.

Even if it is an internal tool for the journal management, it has been designed with these main objectives:

- To allow fast and easy demo creation and edition, to avoid that the generation of demos becomes a bottleneck. The DDL description was created in the new system to this purpose (see Sect. 2.2).
- To incorporate as many editors as possible to the journal. To achieve this objective the Control Panel needs to be a tool intended for editors without no special technical background. All the technical system administration tools are outside the Control Panel.
- To hide the complexity of the distributed system to the editor. Indeed, IPOL is a complex system running in several machines, but from the point of view of the editors and the demo users they simply interact with a web page front-end.

3 Plan to Extend to a Wide-Purpose Platform

IPOL has been evolving since it was funded in 2010, from a very simple demo system which required the editors to write actual Python code and design HTML templates, to a system of a distributed system started in 2015 with load balancing along several machines and automatic demo generation from an abstract description syntax.

The next step is first to move to new data types (audio, video, 3D) and finally to allow different applications, including Machine Learning. Adapting the system to a wide-purpose platform will require some changes in the architecture of the system (new modules) and better data organization (Archive, mainly) but still the current modular architecture of the system will be able to support it. In any case, moving to new application fields and at the same time keeping a generalist platform will require strong engineering efforts and a stable team with permanent positions. This is not the case at this moment, and we need to work on this point to be able to extend IPOL with advanced features reliably.

The following sections present and discuss the steps needed to build the next evolution of the IPOL system.

3.1 Extended Architecture: Support for Applications

There is a fundamental difference between the current IPOL demo system and the complete platform offering Software as a Service (SaaS) applications and learning capabilities: their execution time. The complete platform differs from a demo system in the fact that it allows applications whose lifetime[5] is largely over the short period of execution of a demo. In the case of a demo the input data is loaded, the algorithm executed, the results shown, and the demo has totally finished. In the case of algorithms which can learn from the archive, they need to be running indefinitely. Once they are run, they can receive events indicating the there is new data in the archive, or to explore it regularly.

We shall call these new processes with a large lifetime *Applications*. The system will have two different kinds of processes:

– Demos
– Applications

Nevertheless, the architecture of the new system needs not be redesigned in order to have these new functionalities, but new modules need to be added and they will share information within the system.

The first module which needs to be added is the equivalent to Core which delegates the demo operations along different specialized modules, but for the Applications. We will call this new module *AppCore* and rename the existing one to *DemoCore*.

The DemoCore will delegate some specialized functions in already existing modules, for example in Conversion or Blobs since they still be needed in the context of the applications. But new modules will be exclusive to AppCore, such as:

– *Databases*: storage and management of training and testing datasets
– *UserAccess*: lists of users and authorization management

The same way DemoCore offers an API of microservices that the web application uses to render demos in the client's browser, AppCore will offer microservices to be used by external applications. We can think for example of a MATLAB application or a Jupyter notebook which uses the AppCore's services to log in and then visualize classification results obtained using the corresponding learning databases. Figure 6 shows the architecture of the complete platform.

[5] Here we refer to the "wall clock time" of the process, not their CPU usage time. The application is expected to have a large lifetime when most of the time the process in idle, whereas in the case of a demo the process is CPU intensive.

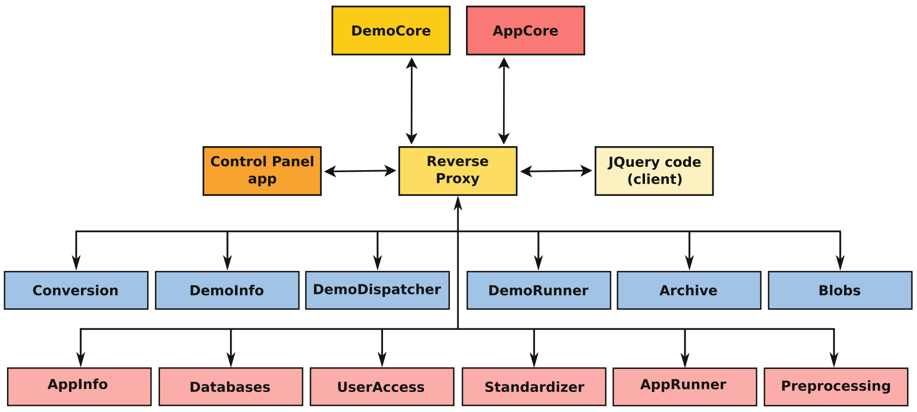


Fig. 6. The complete modular architecture of the future IPOL platform.

Note that the platform is not tied to a particular visualization tool (say, a website) or to any specific framework (MATLAB, R, or the Android system, for example) but instead it offers generic webservices that can be used with them. This way the platform remains generic but at the same time it can be customized with the frameworks preferred by its users.

3.2 Extension to New Data Types

The new architecture of the IPOL demo system (Sect. 2) already allows to work with arbitrary data types. By design, we chose to refer to the data generically as *blob*s, instead of being attached to a particular data type such as images and others. The system manages the data in terms of blobs, and only a particular module ("Conversion") (see Fig. 1) needs to know about the actual details of the type to make conversions. This means that when adding a new data type only Conversion needs to be modified whereas the rest of the system modules remain the same.

The execution flow given an input blob is the following:

1. The blob arrives to Core module. The origin can be either one of the input blobs offered by the demo or a new blob uploaded by the user.
2. Core examines the DDL to check if the demo authorizes data conversion and also it that conversion is actually needed. For example: reduce an image to the maximum allowed size and convert from JPEG to PNG.
3. In case the modification is authorized and needed, Core delegates the conversion of the blob to Conversion.
4. The Conversion module converts the blob to meet the requirements written at the DDL. Only in the context of Conversion the actual data type is taken into account. Outside it is only a generic *blob*.
5. Core goes on with the execution of the algorithm. With the original blob or with the converted.

The kind of conversions needed depend on the data type. For images, audio, video, and 3D they are basically two: resampling (or decimation) and format conversion. Other data types might need different and specific transformations.

While the conversion of the data itself is supported by the current system, depending on the application it might not be enough. Indeed, two more steps are needed:

– Data standardization
– Pre-processing

Given that the aim of the platform is to support multiple applications and data types, the input data is likely to be in different file formats and in different ranges. We can think for example in the data of two different accelerometer sensors: one of them could give values of the three spatial axes in m/s units, while the other might give only two of the three axes in cm/s. Before attempting the process the data, the system needs to put the input data in a common format.

Even if the data has been standardized, it might require pre-processing. For example, an algorithm might not work with missing ("NA" - Not Available) values, while for others this could be acceptable. The system might decide to pre-process the data in order to remove NA samples by interpolation, or simply to remove them before passing the data to the algorithm for execution.

The solution is to add two new modules to the system:

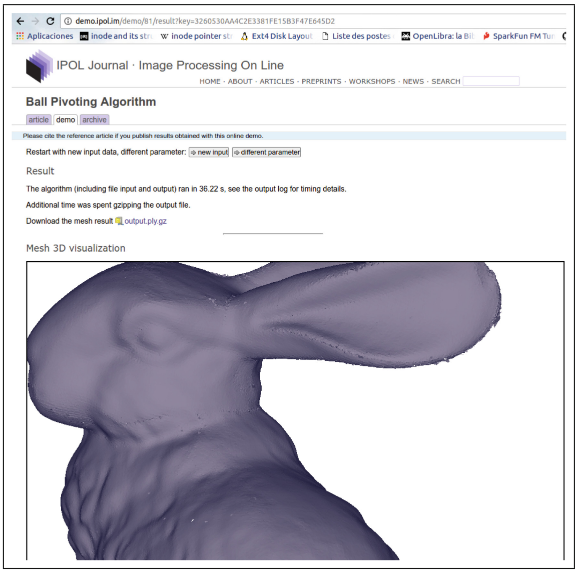
– *Standardizer*: to structure the data in a format understood by the system and the algorithms;
– *Preprocessing*: to perform pre-processing steps before execution according to the needs of the algorithm.

Core would systematically pass the data to these two modules to check if these steps are needed and to actually perform them in that case.

About the web interface, it needs to be expanded to support the visualization of the new data types. At the moment of writing this text, the system has full support for video processing and visualization and the interactive controls (masks for inpainting demos, points for Poisson editing, segments for lens correction, etc.) are expected to be finished in a few weeks. Figure 7 shows the IPOL demo of the published article *Ball Pivoting Algorithm* for which the system shows the result with a 3D mesh renderer.

3.3 Comparison of Algorithms

A desired feature in the future platform is the comparison of algorithms. At this moment every demo is an isolated unit which takes the input data, applies an algorithm with the configured parameters, and finally shows the results. However, being able to measure the performance of the algorithms is of great interest it this would allow to score them and to know which is the best parameter choice according to the algorithm and the input data.


demo.ipol.im/demo/81/result?key=3260530AA4C2E3381FE15B3F47E645D2
Aplicaciones
inode and its stru
inode pointer str
Ext4 Disk Layout
Liste des postes
OpenLibra: la Bib
SparkFun FM Tun
IPOL Journal · Image Processing On Line
HOME · ABOUT · ARTICLES · PREPRINTS · WORKSHOPS · NEWS · SEARCH
article
demo
archive
new input
different parameter

Ball Pivoting Algorithm

Please cite the reference article if you publish results obtained with this online demo.

Restart with new input data, different parameter:

Result

The algorithm (including file input and output) ran in 36.22 s, see the output log for timing details.

Additional time was spent gzipping the output file.

Download the mesh result output.ply.gz

Mesh 3D visualization

Fig. 7. IPOL demo of the published article *Ball Pivoting Algorithm* showing the results with a 3D mesh renderer.

In order to compare the algorithms a metric of quality must be available. This metric can depend solely in the results (for example, exploiting structural similarity in the case of images [5]) or a metric with respect to a given ground-truth.

The inputs and outputs of the algorithm might vary largely and therefore we need to define common data formats in the platform. Clearly, to compare two algorithms their outputs need to be encoded with compatible formats to allow automatic comparisons.

The role of Preprocessing Standardizer presented in Sect. 3.2 becomes clear here. Indeed, the comparison of algorithms feature relies on this two modules since the platform needs to:

– Define formats to homogenize the outputs of diverse algorithms. The Standardizer module will be the responsible for this task, delegated from the DemoCore or AppDemo controller modules.
– In the case of missing data or if pre-processing is needed, then Preprocessing will be invoked too.

Sometimes it will be needed to express the same data in different forms. For example, some metrics would need a temporal series while others could work only with their spectra. The role of Standardizer is to put the data in a flexible format when it is entered into the platform for the first time, not to perform data format conversions during execution (which is a different task and performed by Conversion, see Fig. 6).

The platform needs to provide large datasets to test the algorithms with and also several metrics to perform the comparisons. In the most favorable scenario, these datasets are previously annotated by experts and serve as a ground-truth to evaluate the algorithms. The module responsible for storing the testing and learning databases is the Databases module (Sect. 3.5).

3.4 Chaining of Algorithms

The *chaining* of algorithms consists in using the output of one algorithm as the input for another. For example, a de-noising algorithm could be applied to an input image before attempting object recognition. Another example may be an algorithm to interpolate missing samples in the data acquired from an accelerometer could be run in a first step before trying to determine the trajectory followed by the sensor. In the case of satellite images, we could think of a chain which first de-noises pairs of images with one algorithm, then rectifies them with a second algorithm, and finally performs a 3D reconstruction with yet another different algorithm.

This problem is utterly related to the comparison of algorithms sketched in Sect. 3.3, since the key to be able to connect one algorithm with a different one is that both have a common format for their input and outputs. The platform needs to define general formats for each possible input and output for the algorithm.

Some general data types that should be defined within the platform according to the current editorial needs:

- Temporal series of k dimensions, allowing missing values.
- Sets of points in a 3D point cloud. In general, sets of hyperpoints with an arbitrary number of points.
- Sets of edges and vertices to define 3D meshes. In general, edges and vertices defined in a hyperspace.
- Images and Video. In general, k-dimensional hypercubes of data.

Other generic formats might be defined according to the evolution of the platform and the needs of the users.

We propose to create input/output wrappers in the algorithms to adapt the data to the formats of the platform, to avoid forcing the implemented algorithm to follow specific formats. This way the platform would allow most of the algorithms available without the need of adapting them.

In the future system a final result might not come only from the execution of a single algorithm, but built from the concatenation of many.

3.5 Machine Learning Applications

Machine Learning algorithms have an extra step which makes them different to most of the Signal Processing algorithms: a first training step. Indeed, they normally proceed in three different phases:

1. A training (or *learning*) step with some examples and one or more metrics to evaluate the performance of the algorithm. For example, a neural network

classifier intended for face recognition needs to be trained with many samples previously labeled as faces and non-faces. Typically the learning step is slow, requires many samples, and can be improved by adding more samples to the learning database.
2. A validation step, to ensure that the training is not biased.
3. Prediction. Once the algorithm has already learned according to the training samples (say, a neural network with well-adjusted weights or a SVM[6] classifier with the right hyperplane) it can proceed to predict the result given a sample that the algorithm has not seen before.

On the other hand, the assessment phase needs:

1. A metric (i.e. a formula), which typically is different from the metric used during training.
2. A test set, different from the training set.

The two-step process described above might not be directly applicable when the training of the neural network needs a complex customized training and therefore can not be reduced to uploading learning data and applying standard stochastic optimization. Also, depending on the application the learning step could consist on several phases.

For those complex cases we expect the learning to be performed offline and to work with two types of data:

- The learning set that must be kept for records and
- the generated neural network.

The researcher should upload a script (as *demoExtras*, see Sect. 3.6) governing the learning, in addition to data and to the back-propagation codes. In order to make it reproducible it would be needed that all the details on the training process are documented, including any random seeds, the hyperparameters, the initialization of the weights when learning (for example, Xavier's initialization), number of mini-batches, and in general any other learning parameters which are relevant to reproduce the same results.

The typical demos in the current IPOL system are not well adapted to this new schema since their cycle is simple to read the input data, apply some algorithm with the given parameters, and finally show the results. It is possible however to give some pre-learned databases, but the new platform will be able to update its own training datasets according to the input data from authorized users.

Let us present a typical case of use of the platform. In order to make it more understandable, we will explain it in a concrete application such as healthcare.

The platform will be used by persons with different roles, such as clinicians, technicians, and researchers from different fields (for example, physiologists, signal processing researchers, statisticians, expert systems researchers, etc). Their diverse interests converge at the platform. Doctors are mostly interested in

[6] Support Vector Machine.

obtaining relevant information after they have introduced data of a particular patient (for example, the possibility of discovering a neurological problem from the analysis of saccadic signatures in ocular movements [6]), as well as tracking the evolution of a patient over time.

One of the objectives of the platform is to obtain substantial results by applying Machine Learning algorithms. In this example our aim would be helping clinicians extract relevant medical information from large and multidimensional datasets.

Doctors will decide if the data of their patients (after mandatory anonymization [7]) can be incorporated to the databases of the platform, or not. For example, if the clinician has labeled data it can be used as a ground-truth for many algorithms of the platform and thus over colleagues benefit from it. Or they can decide not to share but still use it as a tool with their own data.

Not all users will be authorized to upload training data (or at least, it needs to be peer-reviewed first), since the performance of the platform relies on the quality of its learning base and thus it needs to be carefully controlled. Thus, depending on the role of the users there will be several combinations of rights and authorization levels: possibility of uploading training data, possibility of executing the algorithms on uploaded test data, possibility to label an already existent dataset, possibility of creating groups of users with common right for certain datasets, etc. Instead of fixing the set of possibilities it is better to leave them open as configurable options and to set up rights for users or groups of users depending on their roles. As advanced in Sect. 3.1, the platform will use a new *Databases* module to store and manage the permissions for the training and test datasets.

In order to take advantage of both the practitioners' extended knowledge and the presence of numerous healthcare datasets, the platform could also include a crowdsourcing aspect (see e.g. [8]). Using trained classification or scoring algorithms, unlabelled data with the largest amount of uncertainty could be presented to the platform medical user for additional input – thus both improving the quality of the dataset, the performance of the algorithm, and presenting practitioner with interesting usecases.

In the case of executing an algorithm with a test dataset (say, to perform prediction), the execution will be similar to a typical IPOL demo. It can be sketched as follows:

1. The users upload or choose one of their datasets;
2. they chain several algorithms and configure their parameters;
3. the algorithm is run. The user can wait until it finishes, or it can be stored and recovered later;
4. the results are shown;
5. the results are archived for further analysis.

This is a complex interaction, perhaps for the most advanced users. However, most users (say, medical doctors) would just upload their data and wait for their automatic results. However, the system needs of course to be designed such a way that it copes with both the complex and common cases.

Technically, the request arrives to Core, which delegates the selection of the most available DemoRunner for the execution, DemoRunner will perform the execution, and then Core will retrieve the results. Finally, Core will send a command to Archive to store all relevant information on the experiment. See Fig. 3.

While the current system could support easily Machine Learning learning algorithm to perform *testing* (say, classification or prediction), it needs to be extended to reach the new architecture depicted in Fig. 6). Indeed, as explained in Sect. 3.1, the execution cycle of a simple demo does not allow to have a long learning phase.

This operative does not adapt well to the needed learning phase and thus a new entity (the *applications*). The applications are processes which are started by AppCore and instead of finishing after a fixed period has elapsed, they simply enter in a *sleep* mode. They can be woken up by AppCore for several reasons, as for example when new data arrives to Archive, or they can wake up by themselves at regular intervals to perform learning on the data available in the platform.

To better understand the concept of the applications, let us think of an algorithm to detect faces in pictures using an SVM. The users will upload their own pictures to the demo to check if the system is able to detect or not the faces that might be present in the image. The users will provide feedback about if the system managed to detect a face in the right position (positive), or if it detected a face in a place where it was not any (false positive), or if it missed an actual face (false negative). To do it, the demo could use an SVM classifier whose hyperplane is shaped according to a learning step. Thus, the published algorithm will implement two tasks: testing (used by the demo), and training (performed by the application). We can imagine the algorithm's package as a ZIP file with two folders with independent codes, one for testing and the other for training.

Once the algorithm has been installed in the system, AppCore will start the corresponding application which uses the training code. It will wake up regularly the application with an event indicating that it has new feedback, and then the application will run the training code to update the SVM hyperplane. This will improve the performance of the demo. Once the training with the new data has finished the application will enter in a sleep mode until it is woken up by AppCore again.

Note that when we write *the AppRunner* or *the DemoRunner* modules, actually we refer to an instance of many of them. Indeed, while the other modules are unique in the system, the runners are many distributed processes. Each time Core needs to execute a process (say, a demo or a learning phase), it uses Dispatcher module to pick a particular runner and send the execution. For example, in the current system we have three distributed demoRunners available. One is physically installed at CMLA laboratory, and the other two are in the installations of an external provider. This is easily configured with simple configuration files.

In the case of the applications they will be also distributed along several servers. The AppCore will ask Dispatcher for an AppRunner, then AppCore will wake the runner up and ask to update its learning database when needed. Note that AppCore has the possibility of running several instances of the same learning process along different runners if necessary. For example, it can split the input data into several batches and make different runners process each of the parts at the same time.

Apart of adding these new AppCore and AppRunner modules to the architecture of the system, the existing Archive module will be improved to allow queries and receive structured information quickly. For example, we can think in a demo that runs a classification algorithm in images (let's say, it tells which is the animal in the image). If the demo configures Archive to store extra information about the inputs (say, level of JPEG compression, the mean intensity of the image, ...), it would be possible to extract meaningful information afterwards. For example, it how relevant is the compression level of the image when performing classification, or how images with very low contrast or saturated impact the performance. This could be done with a query language very similar to SQL.

Improving Archive will allow not only Machine Learning applications, but also for more complex Signal Processing algorithms in the demo system.

Finally, already existing Control Panel (Sect. 3.6) will be extended to allow the system operators to manage[7] the installed applications.

3.6 System Management with a Unified Tool

A tool to allow non-technical administrators to manage the platform is needed. Such a tool is known as the *Control Panel* in the current system (see Sect. 2.3).

The Control Panel allows the IPOL editors to create new demos and modify any existing demos. This tool is a web application where the editors have access after typing their username and password and one of the design requirements is that it should be used by editors without any technical background. It allows however to add extra code to the demos (known as *demoExtras*) in order that advanced editors can customize them but nevertheless most of the demos are defined from a simple textual description (see Sect. 2.2).

In order to allow the complete system with the extensions presented, the Control Panel also needs to be completed. Specifically, it needs new sections to:

- Install, configure, and remove Applications, in a similar way to what does already for the demos.
- Create, edit, and remove users.
- Add, remove, and configure the access right of the learning and testing databases for each user or group of users according to their roles in the platform.

[7] Add, remove, start, stop, and configure user database rights.

The existence of a unified tool such as the Control Panel[8] and the use of an API (see Sect. 2.1) to interact with the system hides not only the implementation details but also the fact that the system is distributed along different machines. Indeed, the editors manage the system with a simple tool which hides all its complexity and lets them configure it quickly and easily.

4 Business Model: The IPOL Case

This section discusses briefly about the business model that RR platforms could adopt, focusing on the experience that the authors have as editors of the IPOL journal.

The scientific interest of IPOL a platform for reproducible research is clear after look at the number of visitors (about 250 unique visits per day) and the number of citations of the published papers (see Fig. 8, total citations: 3497, h-index: 29, i10-index: 61). It has also proved its utility as a tool to prove the competence of the research group and to obtain research contracts (DxO, CNES, ONR, and others).

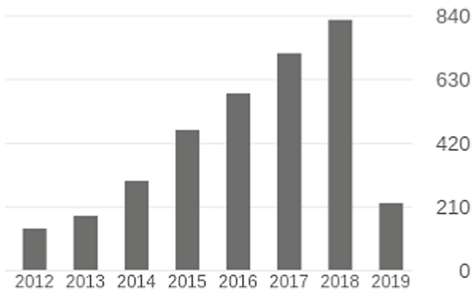


Fig. 8. IPOL citations from 2012 to 2019. Source: Google Scholar. Total citations: 3497, h-index: 29, i10-index: 61.

A very pertinent question is how to obtain a business return which ensures the sustainability of the project for the long term, out of the support of the public agencies (CNRS, ERC, and others).

A possible move would be to move to the Software as a Service (SaaS) model and to receive incomes from:

– The use of already learned databases. This model is already used by large big-data companies such as Google, for example with their TensorFlow product[9].

[8] Nowadays, even configuration tools like this are regarded as part of an experimental system, and techniques like A/B testing are used to improve the performance (usability) of the tool.

[9] https://cloud.google.com/products/machine-learning/.

- The use of computational services. This modality is implemented by similar platforms such as Code Ocean[10] and others.
- Consulting services. The start-up created to build the final platform will collaborate closely with the scientific staff at CMLA (professors, associate researchers, postdocs, and PhD candidates) and eventually incorporate them into the company. This way the company will not only offer the services of the platform, but also scientific consulting and advice.
- Expanding to areas other than Image Processing and Computer Vision. Indeed, in IPOL we are exploring the possibility to accept articles on more general signal processing, including physiological signals.

Other ways of receiving incomes such as crowdfunding or donations might help too. For example, the Jupyter project is funded mainly by donations[11] and GV[12] (Google Ventures) has invested in projects of companies such as Uber, Light, Slack, Periscope, and many others.

Some companies have already expressed their interest on our future platform. Other companies[13] use or have used IPOL for research purposes, with reference source codes and detailed algorithmic descriptions.

4.1 Resources Needed

IPOL began as part of the PhD project of a single person [1], and since then it has been continuously improved. First by the generous contributions of several collaborators, and finally with a group of interns and software engineers working part time or hired for short periods to build the new version of the demo system.

At this point, the project needs highly qualified engineers. It is no longer a small project but a complex system with requires a professional and dedicated team with stable positions. One of the problems at this moment is that it takes months to train a new engineer in the details of the project and most of the time they leave when they start to be really productive. The reason why the engineers leave the project even if they would like to go on is precisely the lack of stability and perspectives, since the public organizations are not able to create stable positions. When the engineers leave the project the group loses all the accumulated knowledge and again we need to train new members.

Only one person in the team has a permanent position, and even though the dedication to the project is part time.

It is important to create stable positions for the project and also to obtain funding to go on with the engineering tasks. A minimum requirement is to obtain at least one permanent position for the technical director and to have funding for a team of one director and three software engineers during two years. The current system is distributed along three servers rented from an external provided, with a cost of 400 euros/month. We estimate that the final platform will need about

[10] https://codeocean.com/pricing.
[11] http://jupyter.org/donate.html.
[12] https://www.gv.com/portfolio/.
[13] We could cite DxO, Thales, or Technicolor, among others.

8 servers with a cost of $400 \times 8 = 3200$ euros/month. The number of servers is flexible and can be adapted to the computational needs of the platforms at any time.

5 Conclusions

The first version of the IPOL demo system has been working since the first article was published in 2010, with a total of 152 articles, 3364 citations and h- and i10-indexes of 29 and 61 respectively. While it is clear that the system is functional, some problems were detected: the system was difficult to debug to track down malfunctions, it suffered from tightly coupled interfaces, it was complicated to distribute the computational load among different machines, and the editors needed to write Python code to create and edit demos. These problems compromised the durability of the system at the same time they started to create a bottleneck that prevented to create and edit demos quickly. These problems are already solved with the new version of the demo system, a distributed architecture of microservices, presented in Sect. 2.

Of course, even if the system is already functional it still needs strong engineering efforts before it is considered a finished product. The plan for the very short term is to integrate new data types such as video, audio, and 3D. We are quite confident on this task since the system is able to manage generic types (even if we refer to *images*, for the system they are simply *blobs*) and it will come down to a visualization problem in the website interface.

With more than one hundred published demos, more than seven years of activity, about 250 unique visitors/day, thousands of experiments performed by the users with their original data, IPOL has been proved to be a useful tool not only for research but also as a value-added tool for the industry.

The existence of similar initiatives such as Jupyter, Code Ocean, RunMyCode, and openCV indicate a great interest of the scientific community and the industry for this kind of platforms, for which IPOL is a clear precursor. These other platforms do not need to be understood as a threat for IPOL, but instead as an indication of the market trend. We refer the reader to [9] for a comparison and discussion on platforms that might be used to implement reproducible research. Nevertheless, IPOL needs to go a step beyond and move from a demo system to a wide-scope platform to build complete applications over it, following the Software as a Service model. Machine learning algorithms, the use of learning databases, and the ability of the platform to improve these databases is a unique business that is worth exploiting.

As explained in Sect. 4.1, while the technical competence is already met and a working system has already been built, it is fundamental for the project to have a stable team. We can afford some temporal contributors (say, interns and engineers hired for specific tasks), but the fact that nobody in the team has a permanent position (not even the technical director) puts the continuity the whole project in obvious danger.

Without any doubt, this is an ambitious project but nevertheless realistic, and the fact that it has been running for almost ten years already is the best proof of feasibility.

Acknowledgments. I would like to thank Jean-Michel Morel, Nicolas Vayatis, Bertrand Kerautret, and Daniel Lopresti for their valuable comments that helped improve this work.

References

1. Limare, N.: Reproducible Research, Software Quality, Online Interfaces and Publishing for Image Processing. Ph.D. thesis, École Normale Supérieure de Cachan (2012)
2. Donoho, D.L., Maleki, A., Rahman, I.U., Shahram, M., Stodden, V.: Reproducible research in computational harmonic analysis. Comput. Sci. Eng. **11**, 8–18 (2009)
3. Neuman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Sebastopol (2015)
4. Colom, M., Kerautret, B., Limare, N., Monasse, P., Morel, J.M.: IPOL: a new journal for fully reproducible research; analysis of four years development. In: 2015 7th International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. IEEE (2015)
5. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**, 600–612 (2004)
6. Purves, D., Augustine, G., Fitzpatrick, D.: Neural Control of Saccadic Eye Movements. Sinauer Associates, Sunderland (2001)
7. Neubauer, T., Heurix, J.: A methodology for the pseudonymization of medical data. Int. J. Med. Inform. **80**, 190–204 (2011)
8. Foncubierta Rodríguez, A., Müller, H.: Ground truth generation in medical imaging: a crowdsourcing-based iterative approach. In: Proceedings of the ACM Multimedia 2012 Workshop on Crowdsourcing for Multimedia, pp. 9–14. ACM (2012)
9. Colom, M., Kerautret, B.: An Overview of Platforms for Reproducible Research and New Ways of Publications. Springer, New York (2018)