`pykitPIV`: PyTorch-compatible synthetic image generation for training optical flow estimation algorithms for velocimetry

Kamila Zdybał*[a], Claudio Mucignat[a] and Ivan Lunati[a]

[a]*Laboratory for Computational Engineering, Swiss Federal Laboratories for Materials Science and Technology, Empa, Dübendorf, Switzerland*

ARTICLE INFO

ABSTRACT

We describe `pykitPIV`, a PyTorch-compatible Python library for generating synthetic images that mimic those obtained from particle image velocimetry (PIV) experiments. The library can be readily ported to training machine learning algorithms, such as convolutional neural networks (CNNs), or reinforcement learning (RL). Our image generation exploits the kinematic relationship between two PIV snapshots and advects particles from one time frame at t_1, to the next at $t_1 + \Delta t$, with a second-order accurate numerical scheme. This results in paired image intensities, I_1 and I_2, that are separated by Δt in time. The goal of this library is to give the user a lot of flexibility in selecting various parameters that would normally be available in an experimental setting such as particle seeding density, thickness of the laser plane, camera exposure, particle loss due to out-of-plane movement, or Δt between images. The richness of image generation can help answer outstanding questions in training CNNs for optical flow estimation or RL algorithms.

1. Motivation and significance

The last decade has seen many advances in training convolutional neural networks (CNNs) for optical flow estimation, *i.e.*, predicting motion information from recorded image frames separated by Δt in time. To date, numerous interesting network architectures have been developed, along with general advancements in training case-specific CNNs. This includes various implementations of FlowNets [? ? ?], spatial pyramid network (SPyNet) [?], pyramid, warping and cost-volume network (PWC-Net) [?], and recurrent all-pairs field transforms (RAFT) [?]. In addition, the idea of the iterative residual refinement (IRR) [?] allowed for significant reduction in the number of trainable parameters thanks to weight-sharing at several levels of successively upscaling image resolution.

Particle image velocimetry (PIV) can especially profit from those architectures, since its main goal is to predict flow targets, such as displacement fields, velocity components, velocity magnitude, or vorticity, from paired snapshots of illuminated tracer particles injected into the flow. Recently, RAFT-PIV [?] and lightweight image-matching architecture (LIMA) [?] were proposed as versions of CNNs that are specifically optimized for predicting flow targets from velocimetry experiments. Thanks to this targeted architecture and optimization of training parameters, both RAFT-PIV and LIMA achieve high accuracy and per-pixel spatial resolution. LIMA is also significantly leaner in terms of the number of trainable parameters than its predecessors. Such networks can be trained on GPUs within the matter of hours and then ported to laboratory hardware to make flow predictions in real-time, parallel to experimental measurements.

The successes of RAFT-PIV and LIMA have been demonstrated on a number of classic experimental fluid dynamics settings such as flow behind a cylinder, boundary layer flow, or convective flow of a hot air plume [?]. However, to advance the accuracy of training CNNs for more complex PIV applications, a number of research questions will have to be addressed next:

1. How rich does the training dataset need to be to remain applicable in a given experimental setting?
2. What degree of data augmentation is sufficient to transfer knowledge from one trained CNN to the next when changing experimental settings?
3. At what time separation, Δt, would the current CNN architectures fail?

To help researchers answer those questions, we propose the present Python library `pykitPIV` – **Py**thon **ki**nematic **t**raining for **PIV**. Our library is compatible with PyTorch [? ?] to allow for easier porting with machine learning algorithms. The format for generated image tensors is consistent with the input required by the classic convolutional layers available in PyTorch (`torch.nn.Conv2d`). The high flexibility in the types of images to create can also port very well with reinforcement learning (RL) algorithms, where an agent may learn to augment the training dataset in real-time to account for changes in experimental settings.

2. Software description

2.1. Software architecture

All functionalities of `pykitPIV` are organized in five classes: `Particle`, `FlowField`, `Motion`, `Image`, and `Postprocess`, each achieving its own role in generating PIV image pairs and the corresponding flow targets. Fig. 1 illustrates the hierarchy of using `pykitPIV` classes and briefly describes what can be achieved with each class. The user selects the number of image pairs to generate (batch size) and their dimensions (height and width). At each stage of image generation, the user can fix random seeds to assure that data generation is reproducible.

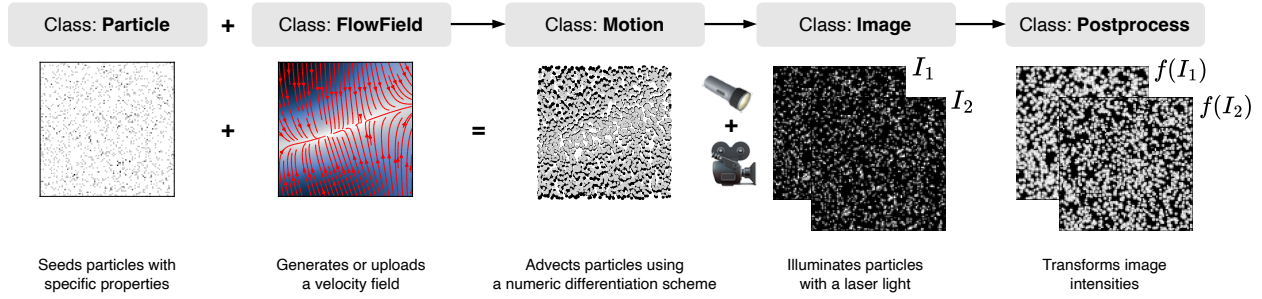✉ kamila.zdybal@gmail.com (K. Zdybał*)
ORCID(s):

Figure 1: Order of using `pykitPIV` classes. At each stage of synthetic image generation, the user has freedom in selecting various parameters that would normally be available in an experimental setting such as particle seeding density, thickness of the laser plane, camera exposure, particle loss due to out-of-plane movement, or time separation between images.

2.2. Software functionalities

2.2.1. Class: `Particle`

The `Particle` class seeds the two-dimensional flow domain with tracer particles. The user can steer the range of particle diameters and their standard deviation, the seeding density, or the average distances between particles. The initial particle positions are those appearing on snapshots I_1.

2.2.2. Class: `FlowField`

The `FlowField` class allows to generate the velocity field to be applied on the two-dimensional domain. We implemented several methods to generate velocity fields, such as random smooth field, checkered field, Chebyshev polynomial field, or spherical harmonics field. Those are illustratively visualized in Fig. 2a-d. This variety of velocity fields span cases with smooth and sharp velocity gradients and can help put machine learning algorithms to test.

The user also has the option of uploading an external velocity field, *e.g.*, coming from a numerical simulation of Navier-Stokes equations (*cf.* Fig. 2e), or coming from a synthetic turbulence generator (*cf.* Fig. 2f) [**? ?**].

2.2.3. Class: `Motion`

The `Motion` class applies the flow field to the particles. It uses a forward Euler or the Runge-Kutta 4th order numeric scheme to advect particles by a user-specified time separation, Δt. Velocity components in-between the grid points are interpolated with a regular grid interpolation. The main output of this class are particle positions that will appear on snapshots I_2, each paired with a respective snapshot I_1.

2.2.4. Class: `Image`

The `Image` class generates image intensities. It adds the reflected laser light to the generated PIV image pairs. The core functionality is to add a Gaussian intensity to each particle [**? ?**]. The user has a lot of flexibility in setting up the laser plane and camera properties. The user can also steer the amount of particles lost between frame I_1 and I_2 due to out-of-plane movement.
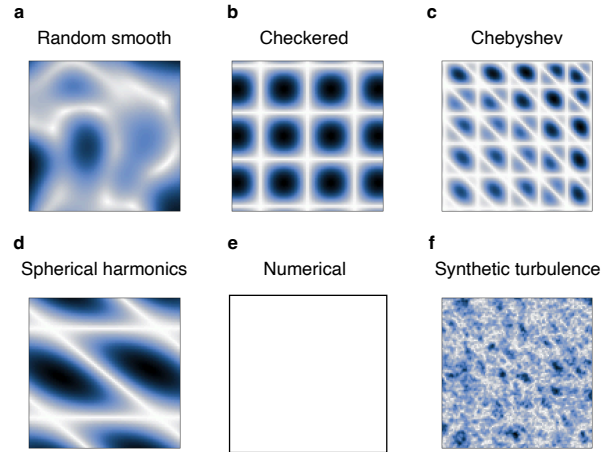

Figure 2: a-d: Types of two-dimensional velocity fields that can be generated with the `FlowField` class. **e-f**: The user also has the option to upload an external velocity field.

The PIV image pair tensor has shape $(N, 2, H, W)$, where N is the batch size, H is image height and W is image width. The second dimension can be thought of as the number of channels and those correspond to I_1 and I_2, respectively. This is compatible with tensor shape accepted by convolutional layers implemented in PyTorch. Note that the whole batch of N images is generated all at once. The `Image` class contains convenient functions for saving images to `.h5` files and for plotting or animating image pairs. `pykitPIV` uses sequential colormaps by Crameri et al. [**?**].

2.2.5. Class: `Postprocess`

The `Postprocess` class contains functions that apply transformations to generated images. It can be especially useful for data augmentation, where the training dataset is extended with images with various levels of noise or illumination levels.
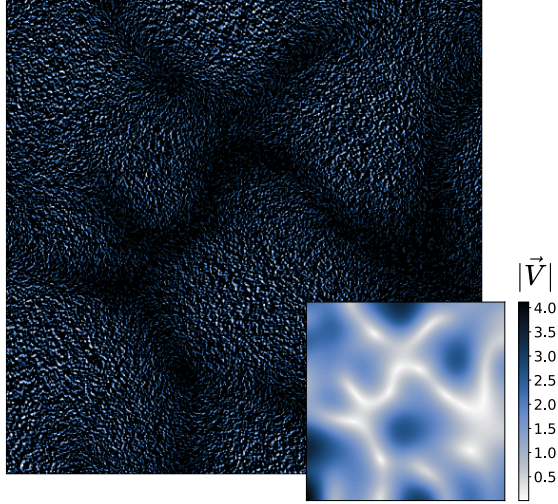
Figure 3: Example visualization of particle motion resulting from superimposing I_1 and $-I_2$ on one image and given the velocity magnitude, $|\vec{V}|$.

3. Illustrative examples

Below, we present the most straightforward workflow for generating a batch of $N = 100$ PIV image pairs and their associated targets and we save the resulting tensors for later use in PyTorch. Each image is 256px × 256px. We create a 10px boundary buffer during image generation to allow for new particles to enter the image area and hence to prevent spurious disappearance of particles near image boundaries.

We import pykitPIV's classes and specify the global parameters:

```
from pykitPIV import Particle, FlowField, Motion, Image

n_images = 100
image_size = (256, 256)
size_buffer = 10
random_seed = 100
```

We instantiate an object of the Particle class:

```
particles = Particle(n_images,
                     size=image_size,
                     size_buffer=size_buffer,
                     diameters=(4,4.1),
                     distances=(1,2),
                     densities=(0.05,0.1),
                     signal_to_noise=(5,20),
                     diameter_std=0.2,
                     seeding_mode='random',
                     random_seed=random_seed)
```

We instantiate an object of the FlowField class and we generate random smooth velocity fields for each of the $N = 100$ image pairs:

```
flowfield = FlowField(n_images,
                      size=image_size,
                      size_buffer=size_buffer,
                      random_seed=random_seed)

flowfield.generate_random_velocity_field(
        gaussian_filters=(10,11),
        n_gaussian_filter_iter=20,
        displacement=(0,10))
```

We instantiate an object of the Motion class and we run a forward Euler numerical scheme to advect the particles:

```
motion = Motion(particles,
                flowfield,
```

```
                time_separation=0.1)

motion.forward_euler(n_steps=10)
```

Finally, we instantiate an object of the Image class, add particles, flow field, and motion objects and add reflected light with the specific laser plane properties:

```
image = Image(random_seed=random_seed)

# Add particles to images:
image.add_particles(particles)

# Add flow field to images:
image.add_flowfield(flowfield)

# Add motion to images:
image.add_motion(motion)

# Add reflected light to images:
image.add_reflected_light(exposures=(0.7,0.8),
                          maximum_intensity=2**16-1,
                          laser_beam_thickness=1,
                          laser_over_exposure=1,
                          laser_beam_shape=0.95,
                          alpha=1/10)
```

Once all images are created, we can remove buffers from the images, convert image pairs and the flow targets to tensors, and save tensors to .h5 files for future use:

```
# Remove image buffers:
image.remove_buffers()

# Prepare image tensors:
image_pairs = image.image_pairs_to_tensor()
targets = image.targets_to_tensor()

# Save images to .h5:
image.save_to_h5({'I': image_pairs,
                  'targets': targets},
                 filename='PIV-dataset.h5')
```

4. Impact

The kinematic training methodology [?], which is the core concept behind generating PIV image pairs with pykitPIV, has been used in our group to train LIMA [? ?]. The approach proved successful, which suggests that for small Δt, learning the kinematic relationship between two consecutive PIV snapshots, as opposed to knowing the full dynamic relationship, is sufficient to train CNNs.

Kamila: It would be great if we could extend image generation to synthetic event-based camera datasets. This would make the software truly novel.

Kamila: Perhaps a nice novelty would be to allow the user to add solid boundaries into the image?

4.1. Porting with convolutional neural networks

Kamila: Here we can describe what can be achieved in terms of training a CNN.

4.2. Porting with reinforcement learning

Kamila: Here we can describe what can be achieved in terms of training an RL agent, e.g. in the context of autonomous experimentation. Maybe the agent will learn to augment the dataset in real time to account for changing experimental settings.

5. Conclusions

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have

appeared to influence the work reported in this paper.

Author contributions

Acknowledgments