muDIC Documentation

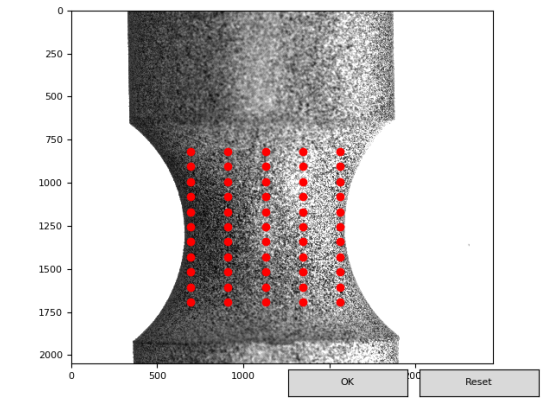
Release 1.0

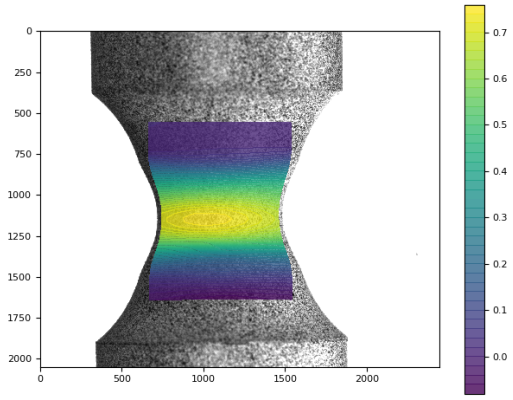
Sindre Olufsen

Apr 15, 2021

First Steps

This project aims at providing a “batteries included” toolktit for digital image correlation in Python. The functionality you need to perform digital image correaltion on experimental data as well as for doing vitrual experiments are included.





CHAPTER 1

The main components:

We have contructed the toolkit as a set of packages where each package provides a isolated set of tools:

- IO-Tools
- **Virtual Lab**
  - Speckle generation tools
  - Image deformation tools
  - Noise injection tools
  - Image downsampling tools
- **Mesh generation**
  - A light weight GUI for structured meshing
  - **B-spline elements**
    - Arbitrary polynomial order
    - Knot vectors can be manipulated
- **Image correlation routines:**
  - Non linear least squares solver
- **Post processor**
  - Calculates most popular strain measures
  - Light weight visualization
- **Test suites**
  - Automated function tests

CHAPTER 2

Our motivation

The motivation for this work was the need for a transparent code which could be modified and extended easly, without digging deep into C or C++ source code. The implementation is pure python with the exception of third-party packages such as Scipy, Numy etc.

CHAPTER 3

Contributing

Clone the repository, add your changes, add new tests and you are ready for a pull request

CHAPTER 4

Authors

- **Sindre Olufsen** - *Implementation* - [PolymerGuy](https://github.com/polymerguy)
- **Marius Endre Andersen** - *Wrote the Matlab code on which this is based*

CHAPTER 5

License

This project is licensed under the MIT License - see the LICENSE.MD.

CHAPTER 6

Citing this project

This project is described in the following paper and citation is highly appreciated

[THE AWESOME PAPER TO BE WRITTEN, WHICH WILL PRODUCE MILLIONS OF CITATIONS]

6.1 Getting started with µDIC

In order to get started with µDIC, you need to install it on your computer. There are two main ways to to this:

- You can install it via a package manager like PIP or Conda
- You can clone the repo

6.1.1 Installing via a package manager:

Prerequisites: This toolkit is tested on Python 2.7x and Python 3.7

On the command line, check if python is available:

```
$ python --version
```

If this command does not return the version of you python installation, you need to fix this first.

If everything seems to work, you install the package in your global python environment (Not recommend) via pip:

```
$ pip install muDIC
```

and you are good to go!

We recommend that you always use virtual environments by virtualenv or by Conda env.

Virtual env:

```
$ cd /path/to/your/project
$ python -m virtualenv env
$ source ./env/bin/activate #On Linux and Mac OS
$ env\Scripts\activate.bat #On Windows
$ pip install muDIC
```

6.1.2 By cloning the repo:

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites: This toolkit is tested on Python 2.7x and Python 3.7

Installing: Start to clone this repo to your preferred location:

```
$ git init
$ git clone https://github.com/PolymerGuy/myDIC.git
```

We recommend that you always use virtual environments, either by virtualenv or by Conda env

Virtual env:

```
$ python -m virtualenv env
$ source ./env/bin/activate #On Linux and Mac OS
$ env\Scripts\activate.bat #On Windows
$ pip install -r requirements.txt
```

You can now run an example:: $ python path_to_muDIC/Examples/quick_start.py

6.1.3 Running the tests

The tests should always be launched to check your installation. These tests are integration and unit tests

If you installed via a package manger:

```
$ nosetests muDIC
```

If you cloned the repo, you have to call nosetests from within the folder:

```
$ nosetests muDIC
```

6.2 A crash-course in Digital Image Correlation

6.2.1 What is Digital Image Correlation?

Digital Image Correlation (DIC) is a method which can be used to measure the deformation of an object based on a set of images of the object during deformation.

An ideal case might look like this:

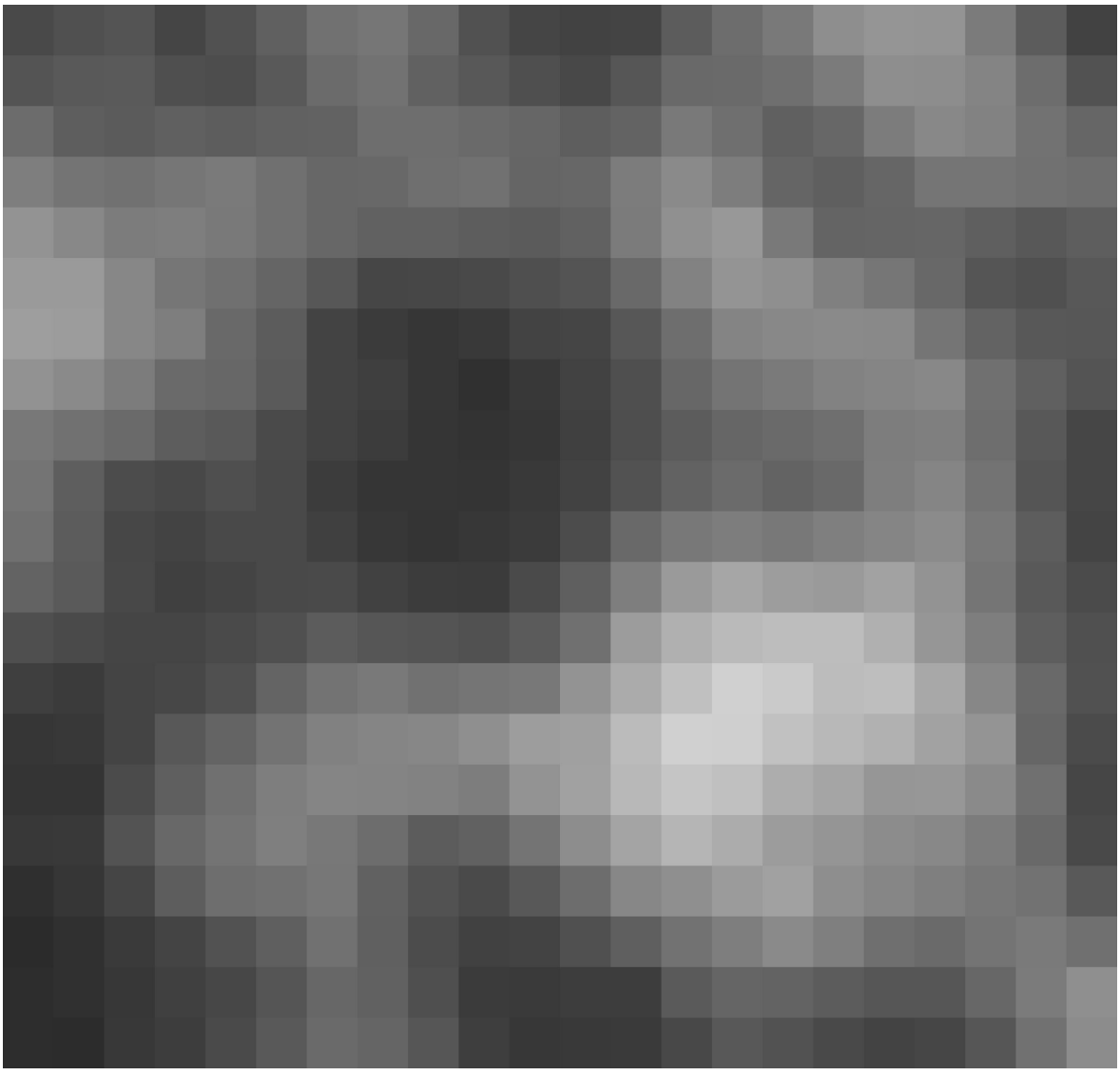
Here we could by eyesight track every individual point and calculate their displacement.

However, in reality, the data we get tends to look like this:

Here we have painted a speckle pattern on the surface of a specimen and used a camera to capture images of the surface while it was deformed. When we are working with such data, tracking a point on the surface of the specimen is not easily done.

First of all, there appears not be any distinct points?

If we look closer on the first image, we see the individual pixels of the image



We now assume that the grey scales of every pixel in the first image represents points on the surface, and that they are convected by the deformation of the surface. Our objective is then to find the same greyscale values in subsequent frames.

During deformation, the greyscales are convected like we see below:

The concept that the grey scales are shifted but their value is preserved is called "conservation of optical flow" and is the fundamental assumption for DIC. In terms of equations, this can be written as:

$$g(\boldsymbol{x}_0) = f(\boldsymbol{x})$$

here, g and f represents the grey scale values of the first image and a subsequent image respectively. The coordinates of a set of grey scales in g are denoted $\boldsymbol{x}_0$ and $\boldsymbol{x}$ in f.

So, the job of the correlation routine is to find $\boldsymbol{x}$.

Note: Put in simple words, we assume that the grey scale values g at position $\boldsymbol{x}_0$ can be found in the other image f at new positions $\boldsymbol{x}$.

Note: The new positions $\boldsymbol{x}$ are not necessarily in the center of a pixel and we therefore need to **interpolate** the grey scale values of f to determine $f(\boldsymbol{x})$

Let us look at a very common misconception...

6.2.2 Misconception: We track single pixels explicitly?

TLDR; No, we dont...

We could imagine that we could determine the position $\boldsymbol{x}$ of every individual pixel directly.

However, there are numerous problems with this direct approach. First of all, for every point in $\boldsymbol{x}$ we have two unknowns, namely the two components of the coordinate, but only have one equation (the conservation of the grey scale). In practise, tracking individual pixels in this way is not a feasible approach, and will in most cases yield a noisy and inaccurate measurement.

6.2.3 Finite element discretization

Ok, so we need to somehow decrease the number of unknowns. What we do then is to assume that the pixels are shifted collectively according to an assumed kinematic. As an illustration, see the figure below.

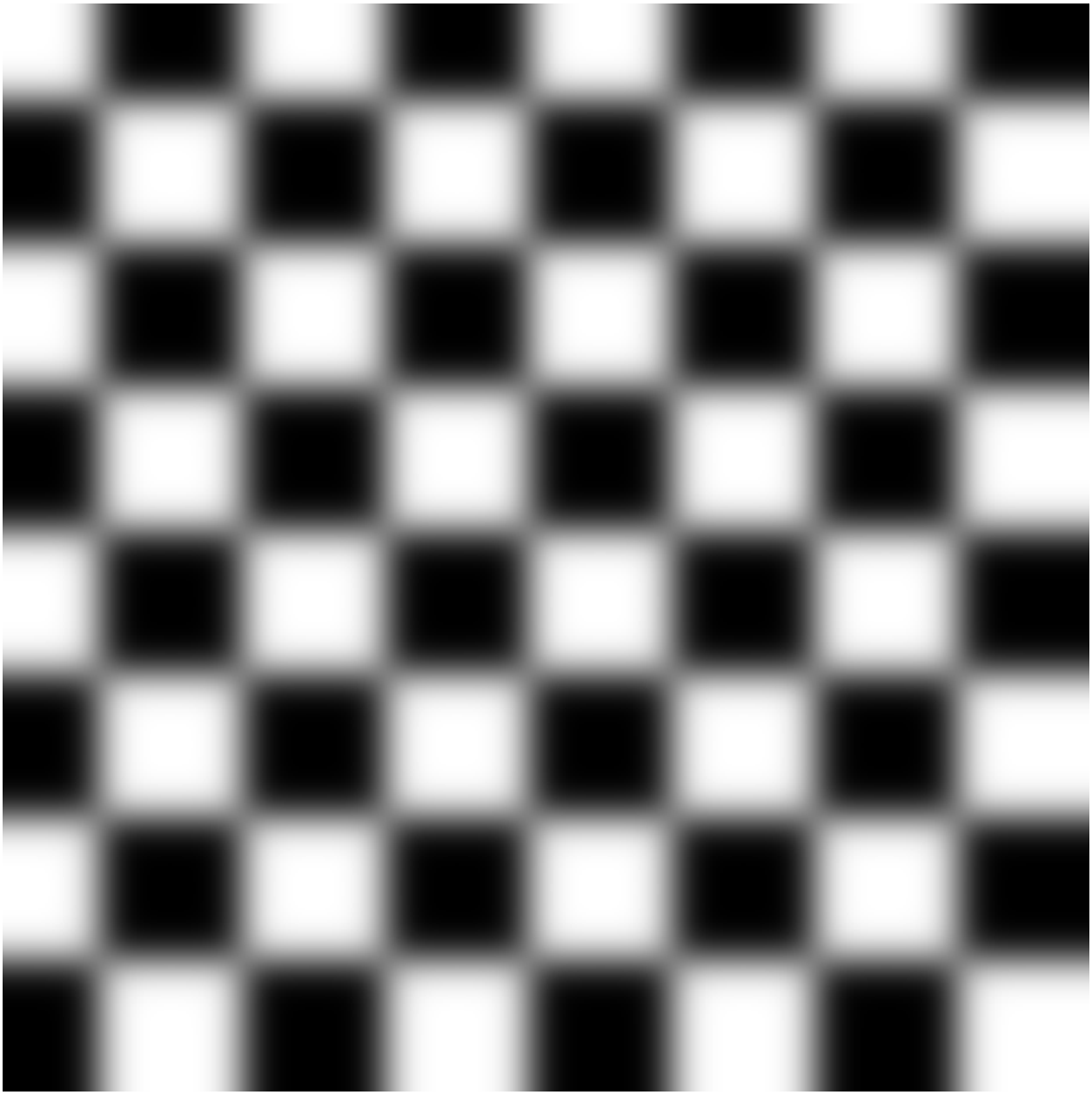
In the figure, the points (blue dots) are shifted according to the movement of a node (red dots). The positions of the points are determined by interpolation (aka. shape functions) of the position of the nodes. This approach is called finite element discretization.

If we now say that the points (blue dots) are the new positions $\boldsymbol{x}$, the objective of the solver is now reduced to find the nodal positions (red dots) which makes the grey scales found at $\boldsymbol{x}$ in f equal to the grey scales at $\boldsymbol{x_0}$ in g.

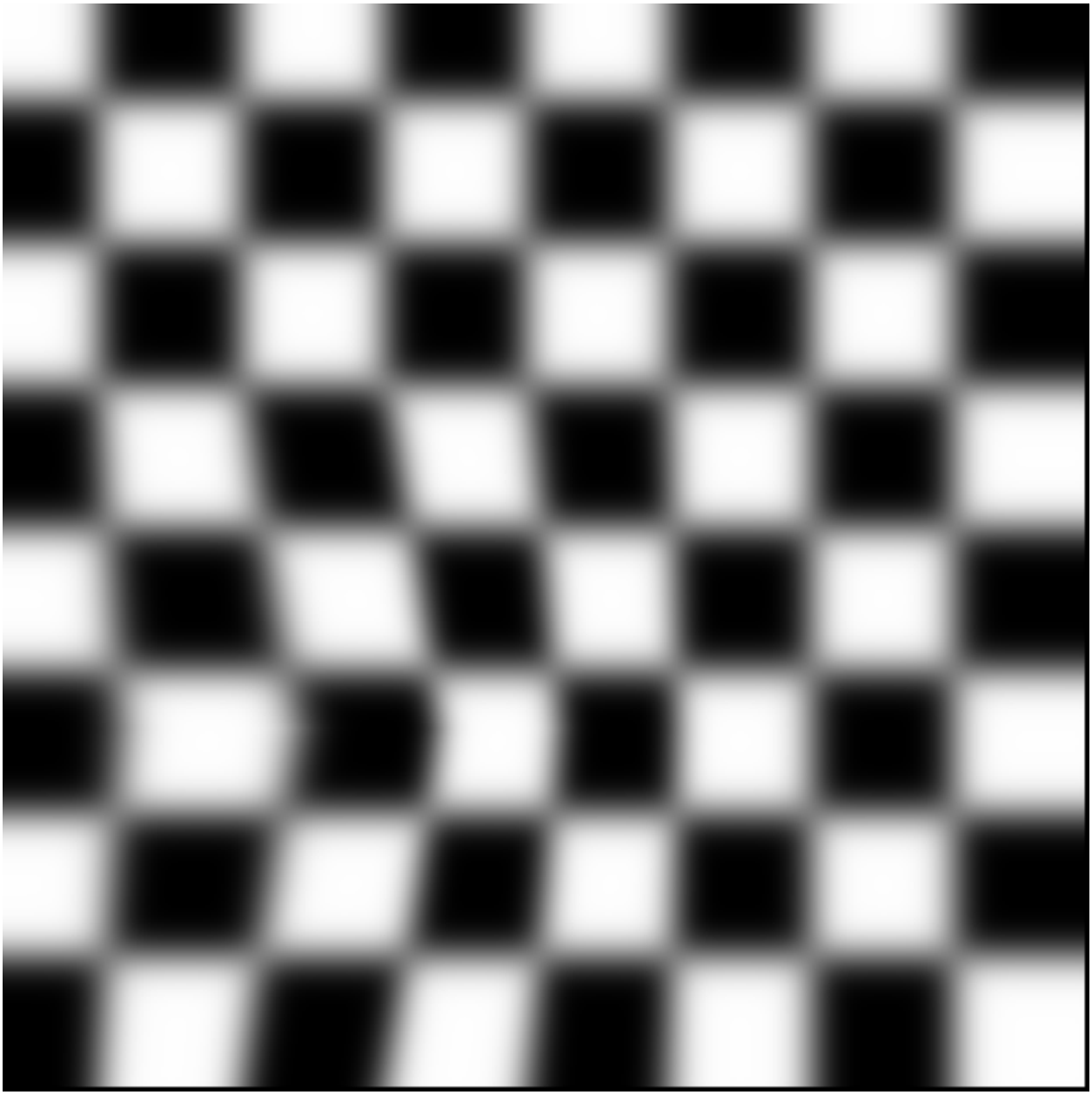
Note: We have now reduced our problem by having many equations (grey scale conservation of every pixel) but only a few unknowns (the nodal positions).

6.2.4 Let us now run through a correlation step

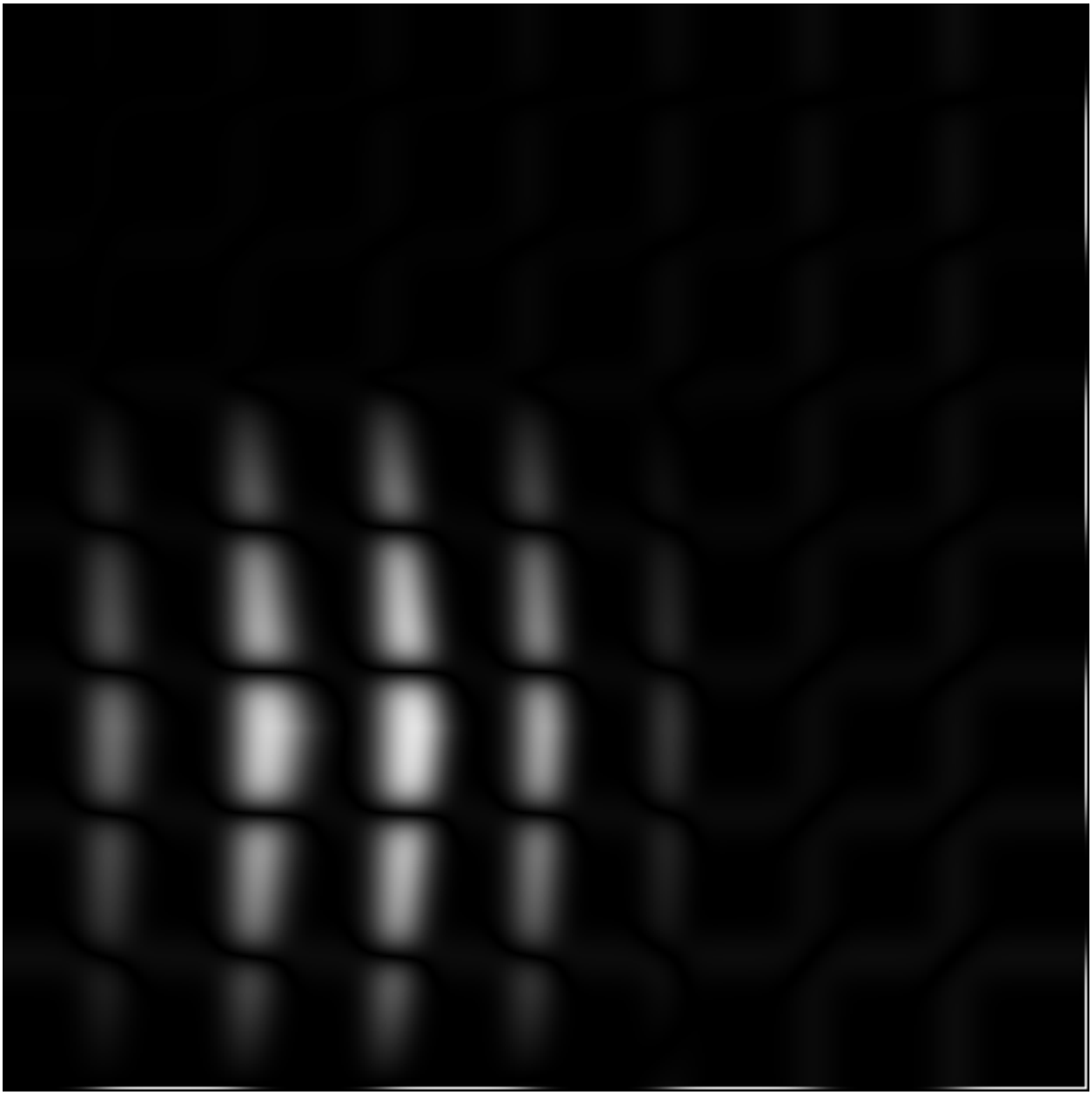
First, let us make an image of something, and let us call it g. If we now set $\boldsymbol{x_0}$ to be the coordinates of every pixel, we can plot $g(\boldsymbol{x_0})$:



If this something has been deformed in the next image, let us call this image f, we can now plot $f(\boldsymbol{x_0})$:



If we now just subtract one image from the other ($g(\boldsymbol{x_0}) - f(\boldsymbol{x_0})$) we see the difference between the images:



We now clearly see that the grey scales are not conserved and that $g(\boldsymbol{x_0}) \neq f(\boldsymbol{x_0})$. Our job is now to figure out where the grey scales found at $\boldsymbol{x_0}$ in g have moved. This means that is we need to find $\boldsymbol{x}$ such that $g(\boldsymbol{x_0}) = f(\boldsymbol{x})$

If a node is moved, the points $\boldsymbol{x}$ are moved like shown on the left below. On the right side, the coordinates $\boldsymbol{x}$ have been moved back to their initial positions $\boldsymbol{x_0}$.

Let us now sample the grey scale values of the deformed image f at the positions $\boldsymbol{x}$, and plot the grey scales in the figure on the right at the positions where they used to be, namely at $\boldsymbol{x_0}$. This operation can be thought of as "un-warping" the image. The "un-warped" image should be equal to the undeformed image g.

We can now see the whole operation below

Ok, so we see that we are able to find the position of the node such that the grey scales that used to be at $\boldsymbol{x_0}$ in the first picture f are the same as found at $\boldsymbol{x}$ in f.

But, how do we know that we have found the best possible match? And how do we make a routine which does this with sub-pixel accuracy?

6.3 Quick start

Let's now go through the neccessary steps for doing a DIC analysis on a set of images. First, we need to import the tools:

```
import muDIC as dic
```

Assuming that you have all the pictures you need in a folder, we can import them all into an image stack:

```
path = r"/the/path/to/the/images/"
image_stack = dic.image_stack_from_folder(path,file_type=".tif")
```

We are now ready to generate a mesh, and to do that, you first need to instantiate a mesher object:

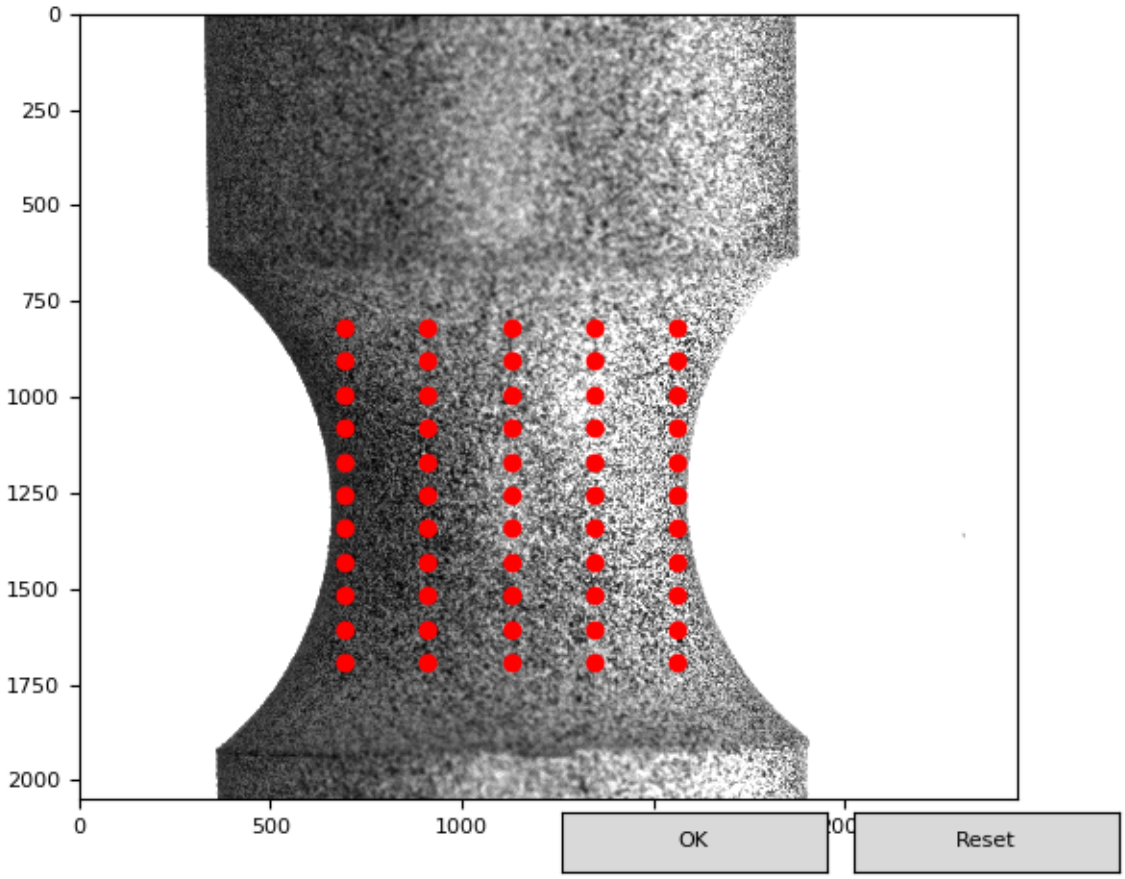
```
mesher = dic.Mesher()
```

Mesher can take a set of settings such as polynomial order and pre-defined knot vectors. If none are given, it uses the default first order polynomials.

Now, let us make a mesh on the first image in the image_stack object we have made earlier:

```
mesh = mesher.mesh(image_stack)
```

A GUI will now pop up, looking something like this:



You can now drag a rectangle over the region you want to cover by the mesh. To manipulate the mesh, you can use:

- A,D: add or remove element in the horizontal direction
- W,X: add or remove element in the vertical direction
- arrow keys: move the mesh one pixel in the direction of the arrow

A good initial guess on element size is in the order of 40x40 pixels in each direction.

In order for us to run a DIC analysis, we have to prepare the inputs by generating a settings object:

```
inputs = dic.DICInput(mesh,image_stack)
```

We are now ready for running a DIC-analysis. We now make a DIC-job object, and call the .run() method:

```
dic_job = dic.DICAnalysis(inputs)
results = dic_job.run()
```

We can now calculate the fields such as deformation gradients and strains:

```
fields = dic.Fields(results)
```

If you want to extract a field for use somewhere else, you can do this by calling the method with the same name as the field variable you want:

```
true_strain = fields.true_strain()
```

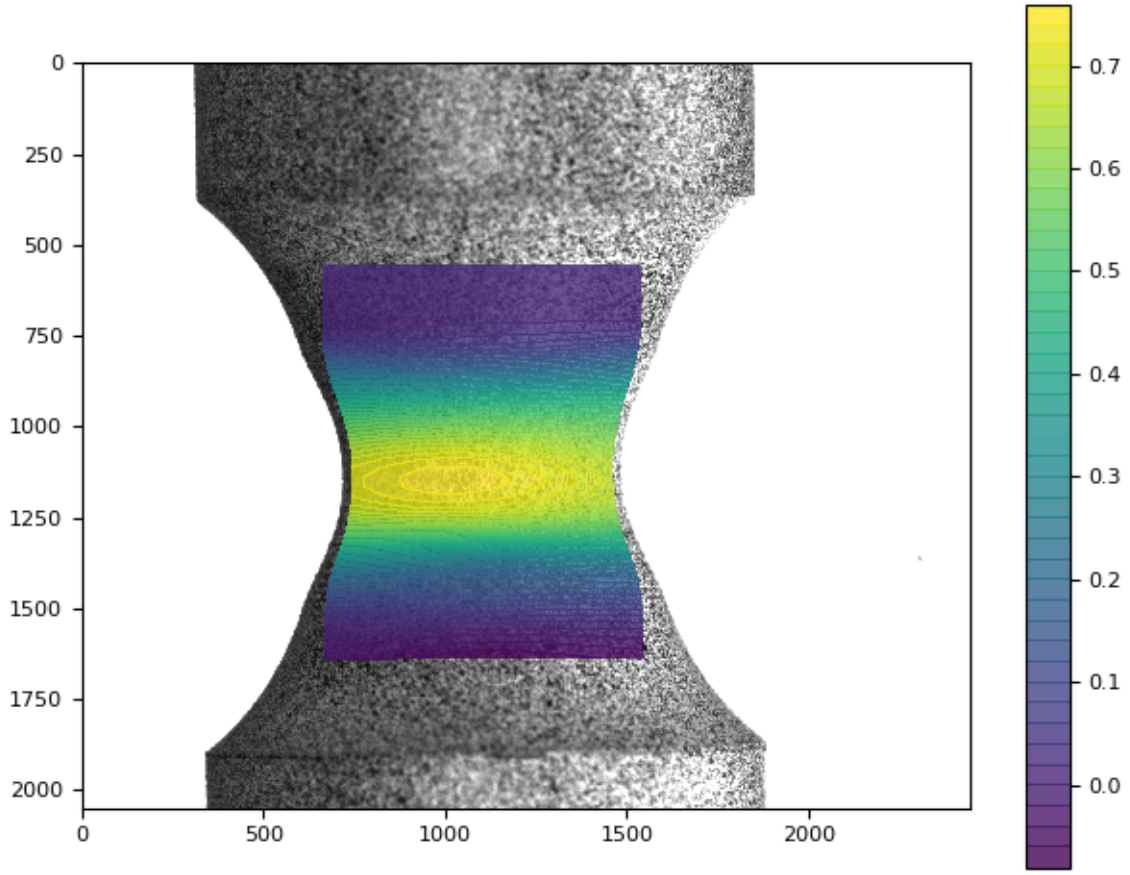
If you want to visualise the results, correlate_img_to_refthere are tools made for this purpose. First, we need to instanciate it:

```
viz = dic.Visualizer(fields,images=image_stack)
```

If we provide the images argument, the fields will be overlayed on the images. Then, we can use the .show method to look at a field for a given frame:

```
viz.show(field="True strain", component = (1,1), frame = 39)
```

which will show the figure below:



6.4 Virtual experiment

Let's now go through how you perform virtual experiments using the virtual lab package.

First, we need to import the tools:

```
import muDIC as dic
from muDIC import vlab
```

6.4.1 Speckle image

We will first make a high resolution speckle image, which we later will deform and downsample. First we declare variables for later use:

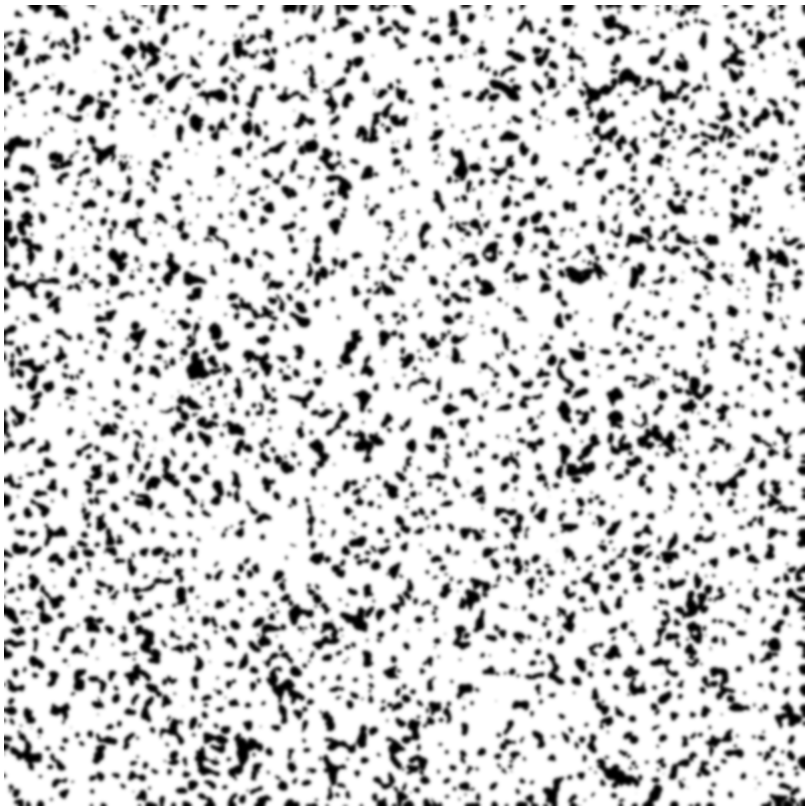
```
image_shape = (2000,2000)
```

For making speckle images, the toolkit comes with several algorithms and we will use one called "Rosta":

```
speckle_image = vlab.rosta_speckle(
    image_shape,
    dot_size=4,
    density=0.32,
    smoothness=2.0)
```

If you want further explaination on the arguments, you can look in the theory section or in the API docs.

The speckle image now looks like this:



6.4.2 Image deformer

Let's now make an image deformer which stretches the image according to a deformation gradient. Firts, we define the deformation gradient:

```
F = np.array([[1.1,.0], [0., 1.0]], dtype=np.float64)
```

We then make an image deformer which uses this deformation gradient:

```
image_deformer = vlab.imageDeformer_from_defGrad(F)
```

image deformer can now take an image as argument and returns a list of deformed images::
deformed_speckles = image_deformer(speckle_image)

6.4.3 Downsampler

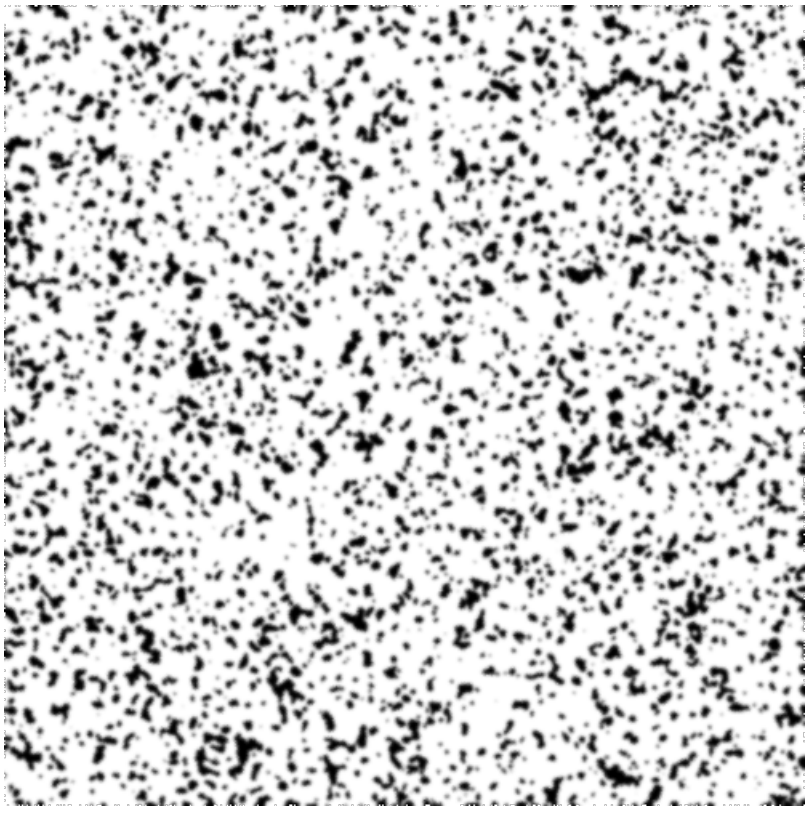
If we want to mimic the artefacts caused by camera sensors, we can downsample the images. I order to do this, we instantiate a downsampler:

```
downsampler = vlab.Downsampler(image_shape=image_shape,
    factor=4,
    fill=0.8,
    pixel_offset_stddev=0.1)
```

This object can now be called, returning a downsampled image:

```
downsampled_speckle = downsampler(speckle_image)
```

A downsampled speckle would look like this:

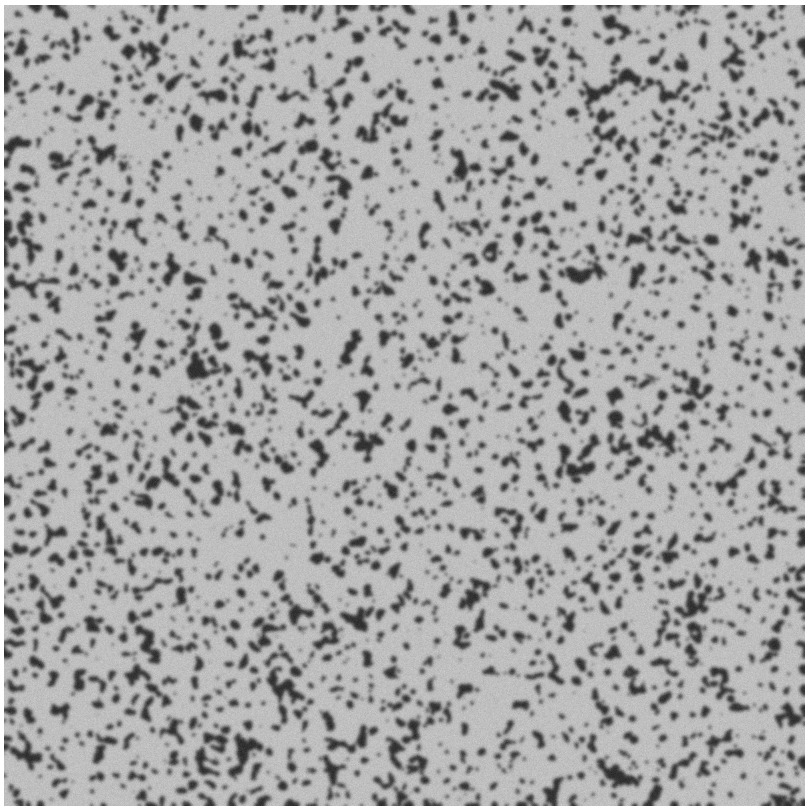


6.4.4 Noise injection

We can now add noise according to noise model of our choice like this:

```
noise_injector = vlab.noise_injector("gaussian", sigma=.1)
```

By passing an image to this function, noise is added. By using the extreme value of sigma = 0.1, the resulting image would look like this:



6.4.5 Virtual experiment

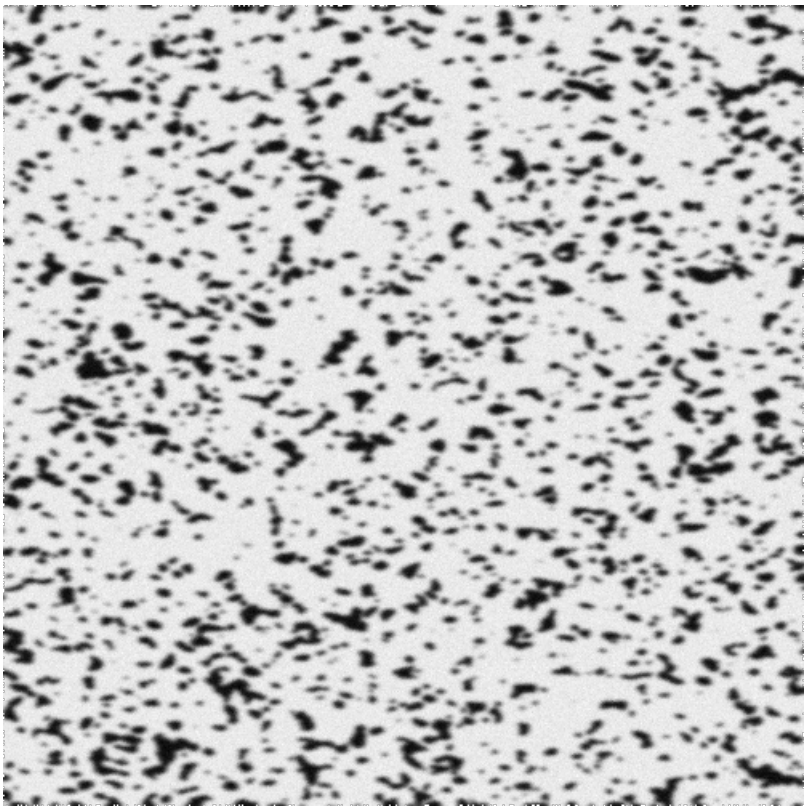
These components can now be composed in a script, or we can use the virtual-experiment fascility to make an image stack directly:

```
image_stack = vlab.SyntheticImageGenerator(speckle_image=speckle_image,
    image_deformer=image_deformer,
    downsampler=downsampler,
    noise_injector=noise_injector,
    n=n)
```

If we as for a given image in the stack, the results from the whole pipeline will be returned:

```
image_five = image_stack[5]
```

and would look like this:



6.5 IO tools

This package contains the tools you need for importing images. Lets import all tools first:

```
import muDIC as dic
```

6.5.1 Importing images from a folder

Say you have a folder with a set of .png images which you want to use for your DIC analysis. We can then import them all into an image stack:

```
path = r"/the/path/to/the/images/"
image_stack = dic.image_stack_from_folder(path,filetype=".png")
```

6.5.2 Creating an image stack from a list of images

Say you have imported a list of images from somewhere which you want to use for your DIC analysis. We can then import them all into an image stack:

```
path = r"/the/path/to/the/images/"
image_stack = dic.image_stack_from_list(list_of_images)
```

6.5.3 Manipulating the image stack

The image_stack object has a set of methods for manipulating the behaviour of the stack. Let us skip the first 10 images:

```
image_stack.skip_frames(range(10))
```

and for some strange reason reverse the order of the images:

```
image_stack.reverse()
```

6.5.4 Adding a filter to the image stack

In many applications, filtering of the images prior to the DIC analysis ca be attractive.

Let us now a add gaussian blur filter with a standard deviation of one pixel to all images:

```
image_stack.add_filter(dic.filters.gaussian_lowpass,sigma=1.0)
```

6.6 Mesher

The mesher package contains everything you need for generation of simplistic structured meshes. Let us first import the tools:

```
import muDIC as dic
```

6.6.1 Generating a mesh from an image stack

In order to generate a mesh, you first need to instanciate a mesher object:

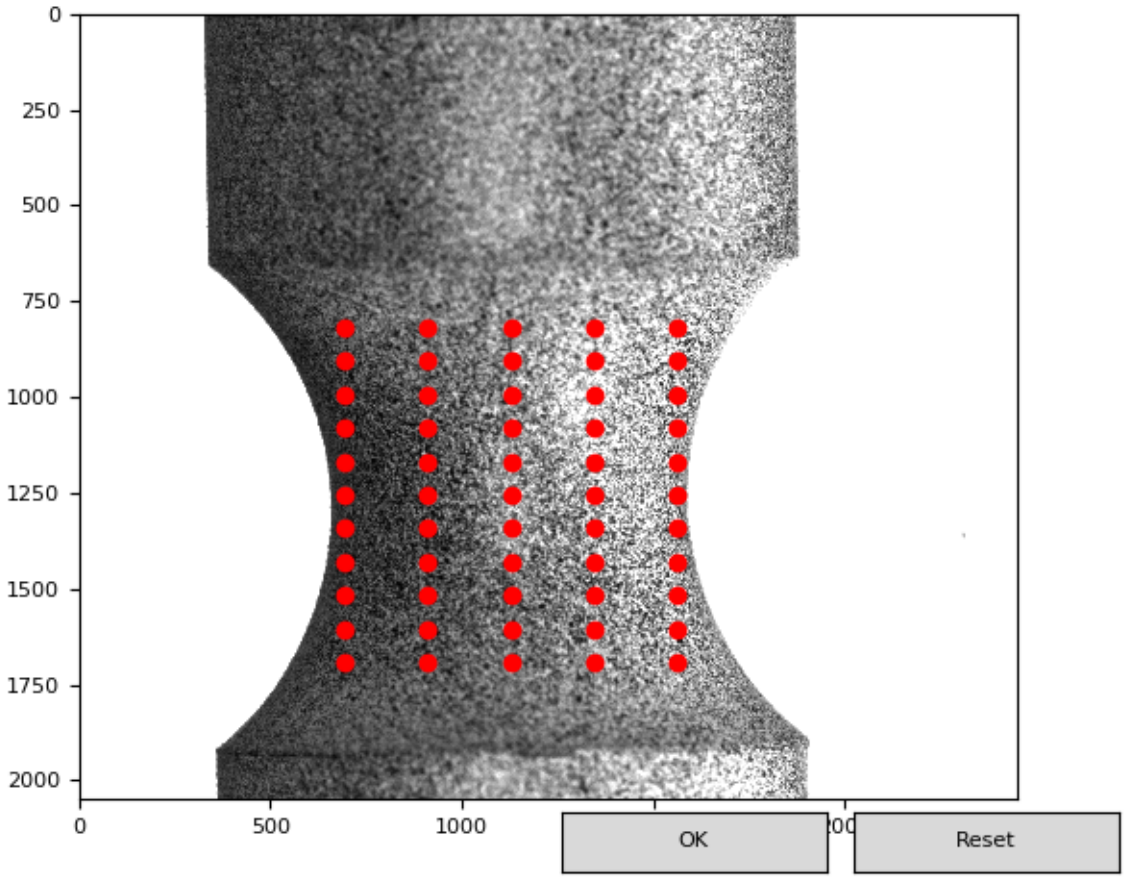
```
mesher = dic.Mesher()
```

Mesher can take a set of settings such as polynomial order and pre-defined knot vectors. If none are given, it uses the default fisrt order polynomials.

Now, let us make a mesh on the image_stack object we have made earlier:

```
mesh = mesher.mesh(image_stack)
```

A GUI will now pop up, looking something like this:



You can now drag a rectangle over the region you want to cover by the mesh. To manipulate the mesh, you can use:

- A,D: add or remove element in the horizontal direction
- W,X: add or remove element in the vertical direction
- arrow keys: move the mesh one pixel in the direction of the arrow

If everything is working properly, a small matplotlib-GUI will pop up. If you dont want to use a GUI, you can set GUI=False and give the numeber of control points and the coordinates of the corners manually.

The mesh object is now ready for use.

6.6.2 Manipulating a mesh

The mesh-object has a set of methods for manipulating the mesh after it has been created. We can use them to translate the mesh five pixels in the X-direction:

```
mesh.move((5,0))
```

or scale the mesh by a factor of 1.2:

```
mesh.scale(1.2)
```

6.7 Correlator

The Correlator package contains the main image correlation routines.

First we need to import it:

```
import muDIC as dic
```

6.7.1 Solver settings

In order for us to run a DIC analysis, we have to prepare the inputs by generatin a settings object:

```
settings = dic.DIC_settings(image_stack,mesh)
```

A image stack and a mesh has to be passed to the DIC_settings class. The DIC_settings class contains all the settings the image correaltion routines need for performing the analysis. Default values are used when the settings object is instanciated.

If we want to alter any settings, for instance set the frames at which to to a reference update, we can do:

```
settings.update_ref_frames = [50,124,197]
```

or, if we want to set the increment size used as convergenve criterion by the solver:

```
settings.convergence_inc = 1e-5
```

6.7.2 Running an analysis

We are now ready for running a DIC-analysis. We now make a DIC-job object, and call the .run() method:

```
dic_job = dic.DIC_job(settings)
results = dic_job.run()
```

Note that the results of the analysis are returned by the .run() method.

6.8 Post processing

After doing a DIC analysis, we are ready to calculate the field variables and to visualize the results.

Import the toolkit:

```
import muDIC as dic
```

6.8.1 Calculate fields

Let's assume we have some dic_results available.

First, we calculate the fields such as deformation gradients and strains:

```
fields = dic.Fields(dic_results)
```

The fields object is lazy and will not calculate anything before the field is quired.

6.8.2 Extract a field variable

If you want to extract a fields for use somewhere else, you can do this by:

```
true_strain = fields.true_strain()
```

the true_strain varialble is now a ndarray with the following shape:

```
true_strain.shape
(100,2,2,21,21)
```

in our example, this shape corresponds to the formatting: (img_frames,i,j,e,n) where img_frames is the number of processed images, i and j are the components of the true strain tensor, and e,n are the iso-parametric element coordinates.

6.8.3 Visualize fields

We can visualize fields manually by using matplotlib or you could use the visualizer included in the toolkit.

Now, lets have a look at the results by using the visualizer:: First, we need to instanciate it:

```
viz = dic.visualizer(fields,images=image_stack)
```

If we provide the images argument, the fields will be overlayed on the images. Then, we can use the .show method to look at a field for a given frame:

```
viz.show(field="True strain", component = (1,1), frame = 45)
```

This will show us the 11 component of the true strain field at frame 45

CHAPTER 7

Indices and tables

- genindex
- modindex
- search