



SUN-DIC

SUN-DIC User Manual

Stellenbosch University

Version 0.0.29

<https://github.com/gventer/SUN-DIC>



Comment

The primary focus of the developers is the continued development and maintenance of the SUN-DIC code. As a result, this manual may occasionally lag behind the latest software updates. If you encounter any inaccuracies, omissions, or outdated information, please report them so that they can be addressed as a priority. This is particularly relevant for version numbers, default settings, and algorithm parameters, which may change over time.

Contents

Comment	i
1 About SUN-DIC	1
1.1 What is Digital Image Correlation?	1
1.2 Key Features of SUN-DIC	1
1.3 Citation	2
1.4 What This Manual Covers	2
2 Installation	3
2.1 Requirements	3
2.2 Creating a Virtual Environment	4
2.2.1 Using conda	4
2.2.2 Using venv	4
2.3 Installing SUN-DIC from PyPI	4
2.4 Installing from GitHub (Advanced)	4
2.5 Copying the Example Files	5
2.6 Launching the GUI	5
3 Parameters	6
3.1 [General]	7
3.2 [DICSettings]	8
3.3 [PreProcess]	9
3.4 [ImageSetDefinition]	10
3.5 [Optimisation]	11
4 GUI Workflow and Example	13
4.1 File Menu	13
4.2 Settings Tab	14
4.3 Image Set Tab	14
4.4 ROI Definition Tab	15
4.5 Analysis Tab	16
4.6 Results Tab	17
4.6.1 Contour Graph	19
4.6.2 Line Cut Graph	19
4.6.3 Text Output	20
4.7 Example Problem	20

5	Python API Workflow and Example	21
5.1	Setup	21
5.2	API Module Overview	21
5.3	Running an Analysis	22
5.4	Post-Processing	23
5.4.1	Smoothing	23
5.4.2	Contour Plots	23
5.4.3	Correlation Quality Plot	25
5.4.4	Line Cuts	25
5.4.5	Time History Plots	27
5.4.6	Accessing Raw Data	28
5.4.7	Accessing Raw Time History Data	30
6	Theory	32
6.1	The DIC Problem	32
6.2	Correlation Criterion	32
6.3	Subpixel Interpolation	33
6.4	Shape Functions	33
6.4.1	Affine Shape Functions (6 DOF)	33
6.4.2	Quadratic Shape Functions (12 DOF)	34
6.5	Optimisation Algorithms	34
6.5.1	IC-GN: Inverse Compositional Gauss–Newton	34
6.5.2	IC-LM: Inverse Compositional Levenberg–Marquardt	35
6.6	Starting Strategy – AKAZE Initialisation	35
6.7	Strain Calculation	35

Chapter 1

About SUN-DIC

1.1 What is Digital Image Correlation?

Digital Image Correlation (DIC) is a non-contact optical technique used to measure full-field displacement and strain on the surface of a specimen under load. A random speckle pattern is applied to the specimen surface (or may occur naturally on the surface). One or more cameras (for stereo DIC) record images as the specimen deforms.

DIC software, such as SUN-DIC, tracks small image regions called subsets between a reference image and one or more deformed images. This process, known as correlation, determines how each subset has translated and deformed during loading. The result is a dense displacement field (u, v) over the selected region of interest (ROI), from which surface strains can be computed.

1.2 Key Features of SUN-DIC

SUN-DIC (Stellenbosch University Digital Image Correlation) is an open-source Python software platform for two-dimensional planar DIC analysis. It was developed at Stellenbosch University to provide a freely available, transparent, and extensible DIC solution that incorporates modern, widely used DIC algorithms.

Key features of SUN-DIC include:

- **Correlation criterion:** Zero-Mean Normalised Sum of Squared Differences (ZNSSD)
- **Optimisation algorithms:** Inverse Compositional Gauss–Newton (IC-GN) and Inverse Compositional Levenberg–Marquardt (IC-LM)
- **Shape functions:** Affine (6 DOF) and Quadratic (12 DOF)
- **Robust initialisation:** AKAZE-based initial guess strategy to improve robustness and overcome the limited convergence radius of the IC-GN and IC-LM optimisers
- **Flexible ROI definition:** Support for rectangular regions of interest, binary masks, and automatic exclusion of near-zero intensity subsets (typically corresponding to a black background)

- **Parallel execution:** Seamless parallel processing using the ray framework
- **Broad image-format support:** Compatible with all image formats supported by the opencv library
- **Multiple interfaces:** Includes both a Python API and an easy-to-use PyQt6 graphical user interface (GUI)
- **Simple installation:** Distributed through pypi

1.3 Citation

The software is described in detail in the following publication. Please cite this article when referencing SUN-DIC in academic publications:

G. Venter and M. Neaves, “SUN-DIC: A Python-Based Open-Source Software Tool for Digital Image Correlation,” *Advances in Engineering Software*, vol. 211, 2025.
[DOI: 10.1016/j.advengsoft.2025.104043](https://doi.org/10.1016/j.advengsoft.2025.104043)

1.4 What This Manual Covers

This manual provides:

1. Installation instructions for SUN-DIC and its dependencies
2. A complete reference for all SUN-DIC algorithm parameters
3. A step-by-step guide to the graphical user interface (GUI)
4. A worked example using the Python API
5. A concise theoretical background covering the ZNSSD criterion, subpixel interpolation, affine and quadratic shape functions, AKAZE initialisation, and the IC-GN and IC-LM optimisation algorithms

Chapter 2

Installation

2.1 Requirements

SUN-DIC has been tested on Windows, macOS, and Linux, with Linux serving as the primary development platform. The examples in this manual assume **Python 3.11**. Before proceeding, verify your Python version by running the following command from a terminal or command prompt:

```
python --version
```

Special Installation Notes

- On Windows, the ray package may not always support the latest Python release. For example, at the time of writing, Python 3.14 was available, while ray was only supported up to Python 3.12. If installation fails because of the ray dependency, install an earlier supported version of Python.
- On Linux, the GUI requires Qt6 system libraries (for example, libxcb and xcb-util). If the GUI fails to start, these libraries may need to be installed using your system package manager.
- On a Mac equipped with an Apple Silicon processor (Mac M1/M2/M3), using the standard Anaconda distribution may cause version conflicts or C++ compilation errors (eg, with llvmlite or ray). This is typically due to Anaconda defaulting to x86_64 emulation. To ensure a seamless, native ARM64 installation without needing to modify the requirements.txt file, it is recommended to use [Miniforge](#) instead of Anaconda.

To avoid dependency conflicts, it is strongly recommended that SUN-DIC be installed in a dedicated virtual environment. The instructions below assume that such an environment is used. Installation can be performed either with conda in an Anaconda environment or with pip in a standard venv environment.

2.2 Creating a Virtual Environment

2.2.1 Using conda

Create and activate a dedicated virtual environment named `sundic`:

```
conda create -n sundic python=3.11
conda activate sundic
```

2.2.2 Using venv

Create a virtual environment named `sundic`:

```
python3.11 -m venv sundic
```

Activate the environment:

```
# macOS / Linux
source sundic/bin/activate

# Windows
sundic\Scripts\activate
```

2.3 Installing SUN-DIC from PyPI

Once the virtual environment has been created and activated, install SUN-DIC using:

```
pip install SUN-DIC
```

To install the optional Jupyter notebook dependencies (required for the example notebook `test_sundic.ipynb`), use:

```
pip install SUN-DIC[jupyter]
```

2.4 Installing from GitHub (Advanced)

Users who wish to access the latest release, experiment with development versions, or contribute to SUN-DIC can install the software directly from the GitHub repository. To clone and install the latest release version:

```
git clone https://github.com/gventer/SUN-DIC.git
pip install ./SUN-DIC
```

To include optional Jupyter notebook support:

```
pip install ./SUN-DIC[jupyter]
```

To obtain the latest development version, clone the dev branch instead:

```
git clone --branch dev https://github.com/gventer/SUN-DIC.git
```

2.5 Copying the Example Files

The `copy-examples` console script is installed alongside SUN-DIC. Run it once after installation to copy the bundled example files to the current working directory:

```
copy-examples
```

This command creates the following files and directories:

- `test_sundic.ipynb` – a complete worked example demonstrating the Python API within a Jupyter notebook
- `settings.ini` — a fully documented reference configuration file for the API example, but that can also be loaded into the GUI
- `planar_images/` — five sample TIFF images of a planar tensile specimen for use with both the API example and the GUI

These files provide a convenient starting point for both API and GUI users.

Optionally, the `copy-examples` script can also be used to copy this user manual by making use of the `--manual` command line flag:

```
copy-examples --manual
```

In addition to the example problem files and directories listed above, this command will also create:

- `docs/SUN-DIC_Manual.pdf` – this user manual

2.6 Launching the GUI

The GUI can be launched using the following console command:

```
sundic
```

If the installation was successful, the SUN-DIC graphical interface should now open.

Chapter 3

Parameters

All parameters are stored in an INI-format configuration file (typically `settings.ini`). This file can be edited manually and used directly with the API. When using the GUI, an existing `settings.ini` file can be loaded to populate the interface. Alternatively, the GUI can be used to define settings and export a `settings.ini` file for use with the API.

The `settings.ini` file is divided into five sections, which are described in detail below. After each section, general notes are provided to guide parameter selection.

3.1 [General]

Key	Type	Default	Description
DebugLevel	int	0	Verbosity level 0 – silent 1 – top-level output 2 – per-iteration output Values outside 0–2 are silently clamped
ImageFolder	str	images	Path to image directory (absolute or relative to working directory)
CPUCount	int or auto	1	Number of CPU cores to use auto uses all available cores
DICType	str	Planar	Analysis type (currently only Planar is implemented)

Notes:

1. **CPUCount** If set to 1, the ray library is not used and computation runs on a single core. This can be useful for debugging, as parallel execution may reduce the amount of detailed output available.
2. **CPUCount** Setting auto may not always provide optimal performance on many-core systems. In practice, specifying approximately half the available cores often yields better performance for small to medium workloads.

3.2 [DICSettings]

Key	Type	Default	Description
SubsetSize	int (odd)	33	Subset side length in pixels. Must be odd so that the subset is centred on a single pixel.
StepSize	int	5	Grid spacing between subset centres in pixels
ShapeFunctions	str	Affine	Affine (6 DOF) or Quadratic (12 DOF). Quadratic is more accurate in high-gradient regions but computationally more expensive.
StartingPoints	int	4	Defines an $n \times n$ grid of candidate starting points across the ROI to initialise the correlation process.
ReferenceStrategy	str	Relative	Relative: each frame is compared to the previous frame (ROI updated per step) — recommended for large deformations Absolute: all frames are compared to the first image (fixed ROI) — recommended for small deformations

Notes:

1. **SubsetSize** Larger subsets reduce noise but increase spatial averaging and potential bias. Quadratic shape functions generally require larger subset sizes than affine functions. Increasing this value can improve robustness if correlation fails.
2. **StepSize** Controls the density of evaluated points within the ROI and therefore has a strong impact on runtime. It does not directly affect accuracy, only spatial sampling density.
3. **StartingPoints** Increasing the number of starting points improves robustness of the initialisation but increases computational cost. Values above 4 or 5 rarely provide significant benefit. When running in parallel, the ROI is split across cores, and the starting-point grid is applied independently to each sub-region.

3.3 [PreProcess]

Key	Type	Default	Description
GaussianBlurSize	int	5	Gaussian blur kernel size (must be odd).
GaussianBlurStdDev	float	0.0	0 – disables filtering Gaussian standard deviation. 0.0 – uses an automatically determined value based on kernel size

3.4 [ImageSetDefinition]

Key	Type	Default	Description
DatumImage	int	0	Index of reference image in the naturally sorted image list
TargetImage	int	-1	Index of last target image (-1 indicates the last image in the folder)
Increment	int	1	Step size between frames in the datum-to-target range
ROI	int[4]	0,0,0,0	Region of interest defined as x, y, w, h . 0,0,0,0 selects the full image
BackgroundCutoff	int	25	Subsets with mean intensity below this threshold are excluded. This enables automatic handling of holes, notches, and irregular boundaries without manual masking
MaskFile	str	(empty)	Optional binary mask file (absolute or relative to working directory), combined with the ROI. White pixels (255) are included; black pixels (0) are excluded

Notes:

1. **ROI** When set to 0,0,0,0, SUN-DIC automatically defines an internal ROI that fits the subsets within the image bounds.
2. **BackgroundCutoff** Set to 0 to disable automatic exclusion based on intensity. Note that this feature may occasionally exclude valid regions within the ROI.
3. **MaskFile** The mask must have the same dimensions as the first image in the dataset and contain only black (0) and white (255) pixels. It is applied in combination with the ROI. The mask can be created using external tools such as GIMP.

3.5 [Optimisation]

Key	Type	Default	Description
OptimizationAlgorithm	str	IC-GN	IC-GN: fastest, suitable for most cases IC-LM: more robust for difficult cases, but computationally more expensive Fast-IC-LM: reduced-accuracy but faster variant of IC-LM
MaxIterations	int	50	Maximum number of iterations per subset
InterpolationOrder	int	5	3 – cubic interpolation (more stable, usually sufficient) 5 – quintic interpolation (potentially more accurate, higher cost)
ConvergenceThreshold	float	0.0001	Stop when displacement change is below this threshold 0 disables this criterion
NZCCThreshold	float	0.999	Stop when ZNCC exceeds this value Values ≥ 1.0 disable this criterion

Notes:

1. **OptimizationAlgorithm** IC-LM is generally more robust but slightly slower than IC-GN. With the initialisation strategy implemented in SUN-DIC, IC-GN is sufficient in most cases. IC-LM should be used only when convergence issues occur with IC-GN. The Fast-IC-LM option is experimental and should be used with caution.
2. **MaxIterations** This value is intentionally conservative and should rarely need adjustment. In most cases, convergence occurs in fewer than 5 iterations.
3. **InterpolationOrder** The cubic option (3) is generally sufficient. The quintic option (5) may provide slightly higher accuracy but at increased computational cost. Cubic interpolation is typically more robust.
4. **ConvergenceThreshold** The default value is appropriate for most applications. Can be disabled by setting this value to 0.
5. **NZCCThreshold** This provides an additional stopping criterion. If set below 1.0, optimisation stops when either convergence criterion is satisfied. To disable this criterion, set to ≥ 1.0 .

6. If both `ConvergenceThreshold` and `NZCCThreshold` are disabled, `MaxIterations` will be reached for all subsets. This is not advised. Generally, only `NZCCThreshold` should be disabled if a more stringent convergence is required.

Chapter 4

GUI Workflow and Example

Launch the GUI from the terminal:

```
sundic
```

The GUI presents a five-tab workflow (Figure 4.1), where each tab must be completed in order before proceeding to the next. Hovering over any input field displays a tooltip explaining the corresponding parameter.

The sections below describe each menu item and tab, followed by instructions for running the included example problem.

4.1 File Menu

The **File** menu provides session management functionality as follows:

- **New** – Clear all settings and start a new session.
- **Open** – Load a previously saved `.sdic` file. This is a binary file that may contain settings and/or results. It is generated either by the GUI or the API, in which case it can be opened for post-processing in the GUI.
- **Import Settings File** – Populate all GUI fields from an existing `settings.ini` file.
- **Export Settings File** – Save the current GUI settings to a `settings.ini` file for use with the API.
- **Save / Save As** – Save the current session under a specified name. This will create a `.sdic` binary file.
- **Exit** – Close the application.

Importing and exporting configuration files is the fastest way to share or reuse configurations. The exported file is identical to the INI format described in Section 3.

4.2 Settings Tab

The **Settings** tab is used to configure all global DIC parameters before loading images and defining the ROI. The fields are arranged in two columns and correspond directly to the parameters defined in Section 3.

The **Set Defaults (For this Panel Only)** button may be used to restore factory defaults for this tab only without affecting other settings.

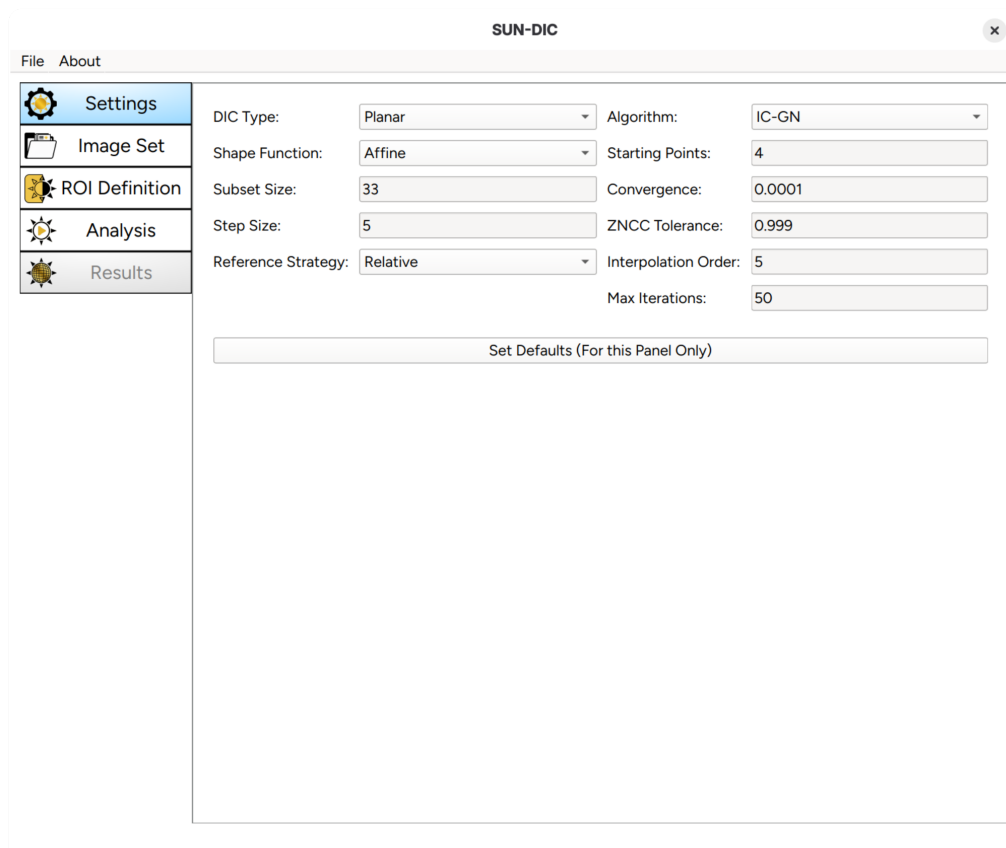


Figure 4.1: Settings Tab. General SUN-DIC parameters are configured here.

4.3 Image Set Tab

The **Image Set** tab is used to define the image set that will be used during the DIC analysis. Click the **Select Directory** button to choose the folder containing the image sequence. This folder should contain only the image sequence, as all files in the directory will be considered. The selected folder path is displayed in the **Folder** field. Images are considered in natural sort order.

To define the image set, the first and last images in the range are specified using the **Start** and **End** fields, while the **Increment** field defines the spacing between images in the sequence. The **Set Max** button can be used to automatically set the **End** field to the last image in the folder. The default behaviour is to consider all images in the selected image folder. The selected images are displayed in the **Images** list.

The **Advanced Settings** area exposes two pre-processing parameters: Gaussian blur kernel size (for noise reduction) and the background cut-off value. The background cut-off is used to automatically detect dark regions (e.g. holes) that are treated as background and automatically excluded from the ROI. Subsets with mean intensity below the specified threshold value are ignored.

The **Set Defaults (For this Panel Only)** button may be used to restore factory defaults for this tab only without affecting other settings.

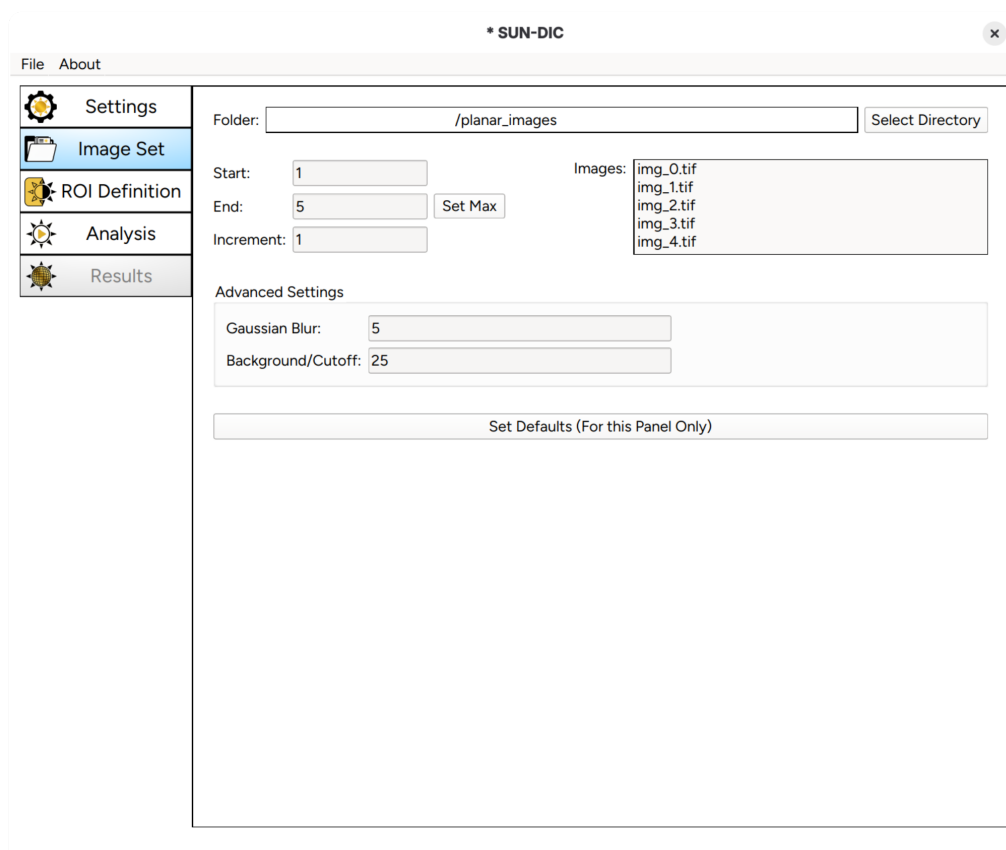


Figure 4.2: Image Set Tab. Select image folder, datum image, and target range.

4.4 ROI Definition Tab

The **ROI Definition** tab is used to define and visualise the ROI. The datum image selected in the **Image Set** tab is displayed in an interactive viewer where the ROI is defined. The ROI is defined by clicking and dragging the mouse over the image to draw a rectangular region. It is shown as a red outline, and green markers indicate active subset centres so that coverage can be verified before running the analysis.

The four numeric fields (**Top Left x**, **Top Left y**, **Width**, **Height**) update automatically but may also be edited manually for precise control.

Navigation controls:

- **Mouse wheel** – Zoom in/out centered on the cursor.
- **Shift + left-drag** – Pan the image.
- **Status bar** – Displays the current cursor coordinates.

Specifying a binary mask is optional. Enable the **Use ROI binary mask** checkbox and select a mask file to exclude irregular regions from the analysis. The mask must be a single-channel binary image of the same size as the input images, where white indicates inclusion and black indicates exclusion.

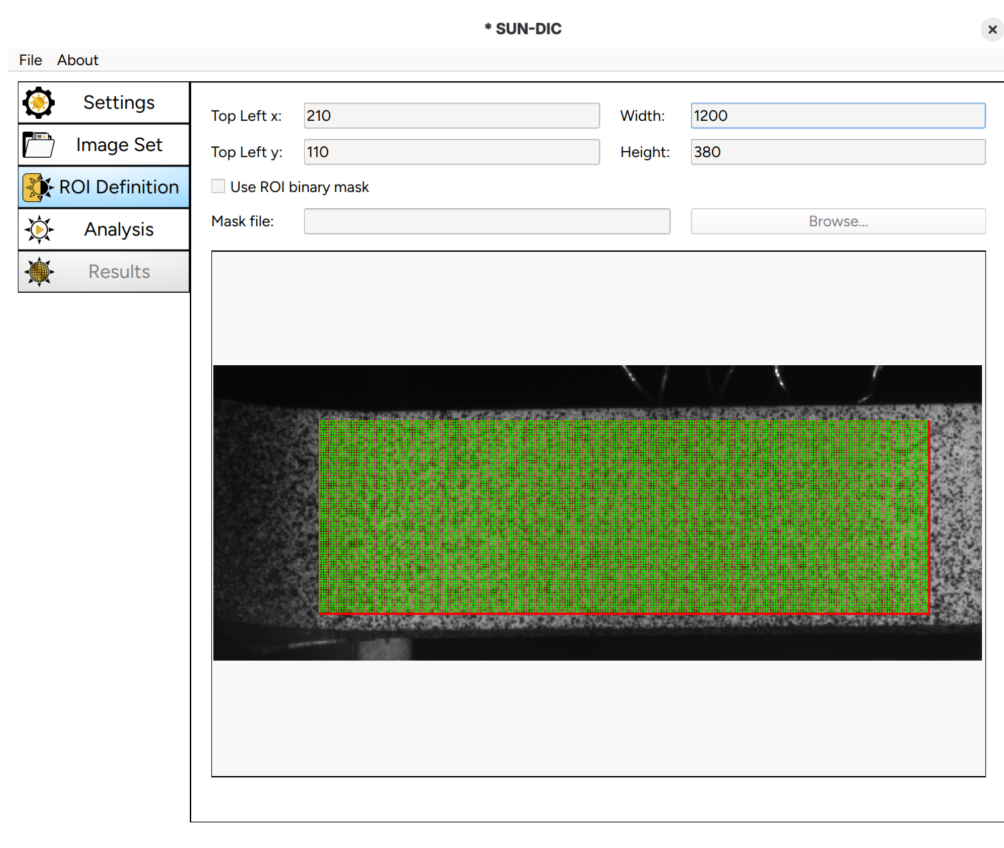


Figure 4.3: ROI Definition Tab. Define the region of interest on the datum image.

4.5 Analysis Tab

The **Analysis** tab is used to perform the DIC analysis. Two runtime parameters can be configured in this tab:

- **Debug Level** – Controls verbosity of console output where 0 is no debug output, 1 corresponds to top-level output and 2 corresponds to per-iteration level output. Note that debug output is limited when running the analysis in parallel.
- **CPU Count** – The number of parallel worker processes to use during the analysis.

Click the **Start** button to begin the analysis. Live output is displayed in the output panel. The status indicator turns blue while running, green when complete, and red if the run is interrupted. Click the **Stop** button to abort execution.

Results are written to a binary `.sdic` file in the working directory. This file should not be saved inside the image directory, since all files in that directory are assumed to be image files during analysis.

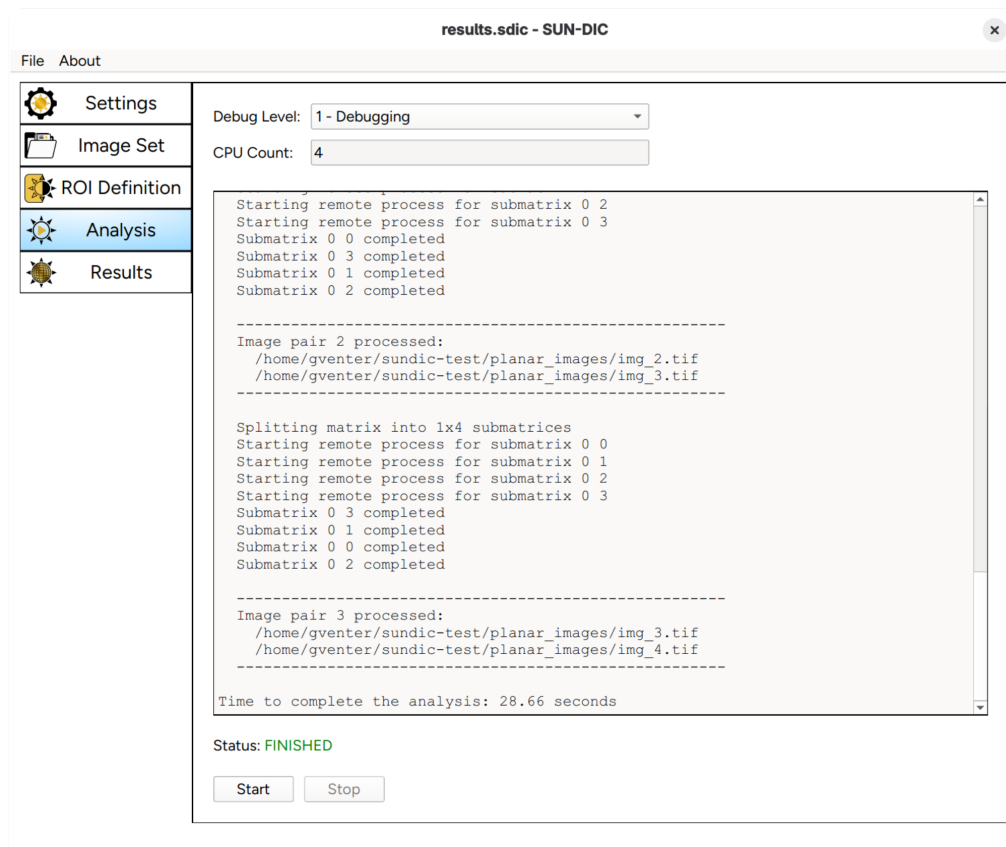


Figure 4.4: Analysis Tab. Run the DIC engine and monitor progress.

4.6 Results Tab

The **Results** tab is used for post-processing the analysis results. The **Results** tab becomes active when there are results in the `.sdic` file, typically after the analysis is complete. Three view modes are available via the buttons at the top of the panel.

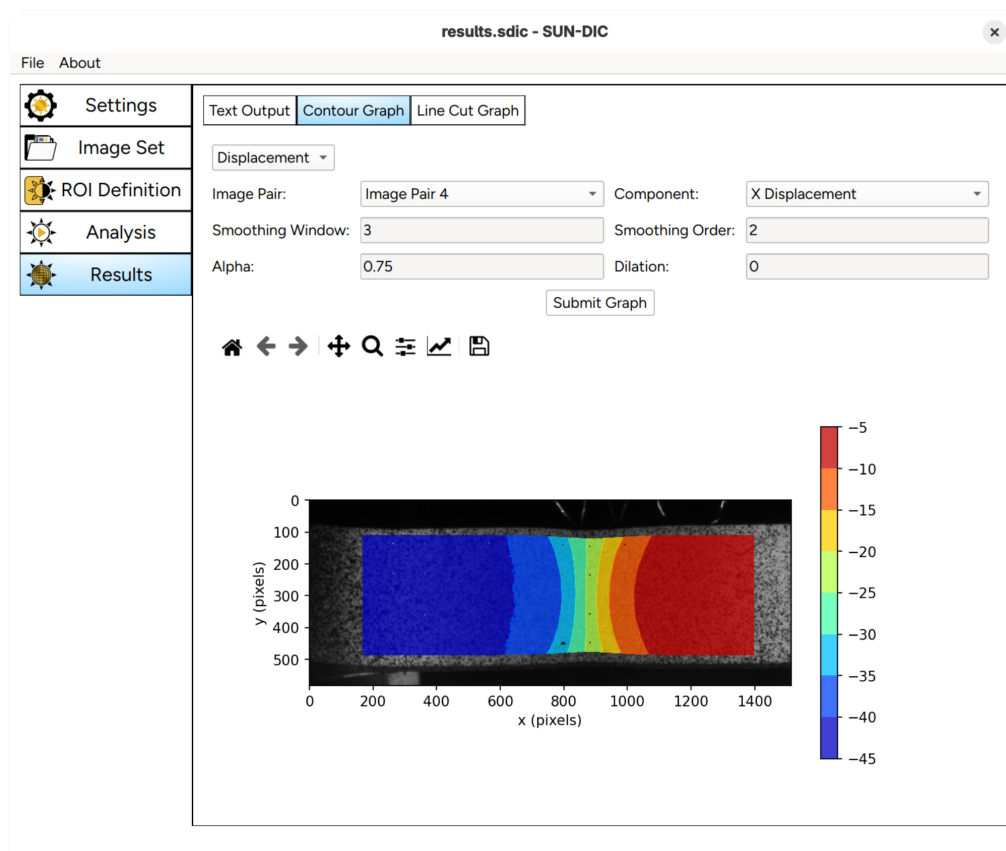


Figure 4.5: Results Tab. Contour plots, line cuts, and data export views.

4.6.1 Contour Graph

The **Contour Graph** panel is used to create contour plots of the computed field data over the specimen. This panel provides the following options:

- **Results** – Choose Displacement, Strain, or Correlation data to plot.
- **Image Pair** – Select the image pair for post-processing.
- **Component** – Select the field component: u (X displacement), v (Y displacement), displacement magnitude, ϵ_x , ϵ_y , ϵ_{xy} , or Von Mises strain ϵ_{VM} .
- **Smoothing Window / Order** – Window size and polynomial order for Savitzky–Golay smoothing.
- **Alpha** – Transparency of the contour overlay.
- **Dilation** – Number of subsets to expand the exclusion mask, useful for edge cleaning.

Click the **Submit Graph** button to generate the plot.

4.6.2 Line Cut Graph

The **Line Cut Graph** panel is used to create two-dimensional line plots through the computed field data. A line-cut graph plots field values along one or more straight lines through the domain. This panel provides the following options:

- **Results** – Choose Displacement or Strain data to plot.
- **Image Pair** – Select the image pair for post-processing.
- **Display Component** – Select the field component: u (X displacement), v (Y displacement), displacement magnitude, ϵ_x , ϵ_y , ϵ_{xy} , or Von Mises strain ϵ_{VM} .
- **Cut Component** – Axis along which cuts are taken (X or Y coordinate).
- **Cut Values** – Comma-separated list of positions at which to make the cuts, e.g. 250, 300, 350.
- **Smoothing Window / Order** – Window size and polynomial order for Savitzky–Golay smoothing.
- **Dilation** – Number of subsets to expand the exclusion mask, useful for edge cleaning.
- **Interpolate** – Interpolate data onto a uniform grid before plotting. When this option is selected, the exact cut values are used. Otherwise, the nearest row or column of subset centres is used.
- **Grid Lines** – Display grid lines on the graph.

Click the **Submit Graph** button to generate the plot.

4.6.3 Text Output

The **Text Output** panel is used to export the raw computed field data for external processing and provides the following options:

- **Image Pair** – Image pair to export.
- **Include Displacements / Strains** – Select output fields to export.
- **Remove NaNs** – Exclude failed or invalid subsets.
- **Smoothing Window / Order** – Optional preprocessing before export. Uses a Savitzky–Golay filter to reduce noise in the exported data.
- **Dilation** – Number of subsets to expand the exclusion mask, useful for edge cleaning.

Click the **Export Data** button to write the data to a `.csv` output file. The column structure matches the raw data format described in Section 5.

4.7 Example Problem

To run the example problem in the GUI, proceed as follows:

1. Copy the example files to your working directory:

```
copy - examples
```

This creates:

- `test_sundic.ipynb` – Worked Python API example in Jupyter notebook format
- `settings.ini` – Fully documented configuration file usable in both the GUI and API
- `planar_images/` – Sample TIFF image sequence for testing

2. Launch the GUI:

```
sundic
```

3. Import the `settings.ini` configuration via **File** → **Import Settings File**.
4. Verify that settings, images, and ROI are correctly loaded. If the image files are not found, manually browse to the `planar_images` folder.
5. Run the analysis from the **Analysis** tab.
6. Explore the results in the **Results** tab.

You are encouraged to modify settings and rerun the analysis to explore different configurations and effects.

Chapter 5

Python API Workflow and Example

5.1 Setup

Start by copying the example problem to your working directory by issuing the `copy-examples` command from a terminal after activating your virtual environment. After running `copy-examples`, the working directory will contain:

- `settings.ini` – Example configuration file
- `planar_images/` – Five sample TIFF images
- `test_sundic.ipynb` – Jupyter notebook containing a complete worked example

The Jupyter notebook is the recommended starting point for new users because it demonstrates every step described in this chapter with live output and visualisation. The API can also be used directly from a standard Python `.py` script.

If you intend to use the provided notebook, remember to install the optional Jupyter dependencies using:

```
pip install ./SUN-DIC[jupyter]
```

5.2 API Module Overview

Most users will interact with only three modules:

- `sundic.settings` – Reading, writing, and managing analysis settings.
- `sundic.sundic` – Running the DIC analysis.
- `sundic.post_process` – Visualisation, export, and access to computed results.

The typical workflow is:

1. Load a `settings.ini` file.
2. Run the DIC analysis.

3. Open the resulting `.sdic` file for post-processing.
4. Generate plots, export data, or access NumPy arrays directly.

The API follows a two-phase workflow. During the first phase, the DIC analysis reads the configuration and image sequence and writes all results to a binary `.sdic` file.

During the second phase, post-processing functions read the `.sdic` file on demand to generate displacement fields, strain fields, contour plots, line-cut graphs, correlation-quality plots, and exported data files.

Because all results are stored in the `.sdic` file, the computationally expensive DIC analysis only needs to be performed once. Post-processing operations can then be repeated with different settings without rerunning the analysis.

5.3 Running an Analysis

Settings are loaded from a `settings.ini` file and passed directly to the analysis function along with a name for the output file. The fully documented `settings.ini` provided with the example problem is a good starting point for creating a custom configuration file for your own analysis.

The `Settings` class implements a custom string representation, allowing the current configuration to be printed directly. The following listing reads the provided `settings.ini` file, prints the configuration to the screen, and then submits the DIC analysis using the `planarDICLocal` function.

```
import sundic.sundic as sdic
import sundic.settings as sdset

settings = sdset.Settings.fromSettingsFile('settings.ini')
print(settings)

sdic.planarDICLocal(settings, 'results.sdic')
```

Debug output is printed to the console as each image pair is processed. All raw results and the settings snapshot are stored in the specified `results.sdic` file. This is a SUN-DIC specific binary file that can also be opened in the GUI for post-processing.

The `planarDICLocal` function has an optional argument named `externalRay` that can be set to `True` if a Ray server was started separately from the API run. This is useful when running a number of DIC analyses in a loop where an external Ray server reduces the overhead of starting and stopping the server each time an analysis is performed. The call to `planarDICLocal` would then look as follows:

```
sdic.planarDICLocal(settings, 'results.sdic', externalRay=True)
```

The external Ray server is typically started from the terminal using the following command. There are however more advanced options available for starting a Ray server, please refer to the Ray documentation.

A complete list of arguments for these contour plot functions are summarized below:

```
# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# imgPair (int)    : the zero based image pair ID to process
#
# Optional Parameters:
# -----
# alpha (float)    : the transparency of the contour plot
#                   (Default is 0.75 -> 75% opaque)
# plotImage (bool) : whether to plot the underlying image
#                   (Default is True -> plot the underlying image)
# showPlot (bool)  : whether to show the plot here
#                   (Default is True -> show the plot)
# fileName (str)   : the file name to save the plot to if
#                   required
#                   (Default is '' -> No file saved)
# smoothWindow (int): the size of the smoothing window
#                   this is the number of surrounding subSets to
#                   consider in the smoothing
#                   (Default=0 -> no smoothing)
# smoothOrder (int) : the order of the smoothing polynomial
#                   (Default=2 -> quadratic smoothing)
# dilation (int)    : the number of subsets to dilate the mask
#                   around automatically detected features (eg holes)
#                   where the results may be of poor quality
#                   (Default = 0 -> no dilation)
# maxValue (float)  : the maximum value to plot
#                   (Default is None -> auto scale)
# minValue (float)  : the minimum value to plot
#                   (Default is None -> auto scale)
```

For plotDispContour there is an additional optional parameter dispComp to identify the displacement component to plot. Available dispComp values are X_DISP, Y_DISP, and DISP_MAG.

```
# dispComp (int)    : the displacement component to plot
#                   (Default is sdpp.DispComp.MAG -> Magnitude)
```

For plotStrainContour the corresponding parameter to identify the strain component to plot is strainComp and available strainComp values are X_STRAIN, Y_STRAIN, SHEAR_STRAIN, and VM_STRAIN.

```
# strainComp (int)  : the strain component to plot
#                   (Default is sdpp.StrainComp.VM_STRAIN)
```

5.4.3 Correlation Quality Plot

A filled contour map of the ZNCC correlation quality field can be plotted to identify subsets that converged poorly:

```
import sundic.post_process as sdpp

sdpp.plotZNCCContour('results.sdic', -1)
```

ZNCC ranges from -1 to 1 , with 1 indicating a perfect match. Values close to 1 across the domain indicate a reliable analysis. Low or negative values flag problematic regions (poor speckle, occlusion, large out-of-plane motion). A complete list of parameters for the `plotZNCCContour` function are summarized below.

```
# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# imgPair (int)    : the zero based image pair ID to process
#
# Optional Parameters:
# -----
# alpha (float)    : the transparency of the contour plot
#                   (Default is 0.75 -> 75% opaque)
# plotImage (bool) : whether to plot the underlying image
#                   (Default is True -> Plot the underlying image)
# showPlot (bool)  : whether to show the plot here
#                   (Default is True -> Show the plot)
# fileName (str)   : the file name to save the plot to if
#                   required
#                   (Default is '' -> No file saved)
# maxValue (float) : the maximum value to plot
#                   (Default is None -> auto scale)
# minValue (float) : the minimum value to plot
#                   (Default is None -> auto scale)
```

5.4.4 Line Cuts

A line-cut graph plots field values along one or more straight lines through the domain. The `plotDispCutLine` and `plotStrainCutLine` functions extract displacement or strain values along one or more straight cuts through the image. The listing below shows how to generate a displacement line-cut graph for the X displacement with cuts at 250, 300 and 350 pixels along the Y coordinate. This results in a 2D graph with three lines. The second function call in the listing generates a line-cut graph for the Von Mises strain with cuts at 500 and 700 pixels along the X coordinate. A 2D graph with two lines will be produced.

```
import sundic.post_process as sdpp

# Line cut through the displacement field at fixed Y positions
sdpp.plotDispCutLine('results.sdic', -1,
                    dispComp=sdpp.DispComp.X_DISP,
                    cutComp=sdpp.CompID.YCoordID,
                    cutValues=[250, 300, 350])

# Von Mises strain along fixed X positions
sdpp.plotStrainCutLine('results.sdic', -1,
                     strainComp=sdpp.StrainComp.VM_STRAIN,
                     cutComp=sdpp.CompID.XCoordID,
                     cutValues=[500, 700])
```

A complete list of parameters for the `plotDispCutLine` and `plotStrainCutLine` functions are summarized below.

```
# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# imgPair (int)      : the zero based image pair ID to process
#
# Optional Parameters:
# -----
# cutComp (int)      : the component to cut through
#                      (Default is sdpp.CompID.XCoordID -> X coordinate)
# cutValues (List)   : the values to cut through
#                      (Default is None -> No cuts)
# gridLines (bool)   : whether to plot grid lines
#                      (Default is True -> Plot grid lines)
# showPlot (bool)    : whether to show the plot here
#                      (Default is True -> Show the plot)
# fileName (str)     : the file name to save the plot to
#                      (Default is '' -> No file saved)
# smoothWindow (int) : the size of the smoothing window
#                      this is the number of surrounding subSets to
#                      consider in the smoothing
#                      (Default is 0 -> no smoothing)
# smoothOrder (int)  : the order of the smoothing polynomial
#                      (Default is 2 -> quadratic smoothing)
# dilation (int)     : the number of subsets to dilate the mask
#                      around automatically detected features (eg holes)
#                      where the results may be of poor quality
#                      (Default = 0 -> no dilation)
# interpolate (bool) : whether to interpolate the data
#                      (Default is False -> no interpolation)
# -----
```

As for the contour plots, there is an additional optional parameter `dispComp` to identify the displacement component to plot when calling `plotDispCutLine`. Available `dispComp` values are `X_DISP`, `Y_DISP`, and `DISP_MAG`.

```
# dispComp (int)      : the displacement component to plot
#                      (Default is sdpp.DispComp.MAG -> Magnitude)
```

For `plotStrainCutLine` the corresponding parameter to identify the strain component to plot is `strainComp` and available `strainComp` values are `X_STRAIN`, `Y_STRAIN`, `SHEAR_STRAIN`, and `VM_STRAIN`.

```
# strainComp (int)    : the strain component to plot
#                      (Default is sdpp.StrainComp.VM_STRAIN)
```

5.4.5 Time History Plots

When the analysis contains more than one image pair (thus more than two images), time history plots can be created to investigate how a displacement or strain component develops over time for one more points in the ROI. The `plotDispTimeHistory` and `plotStrainTimeHistory` functions can be used to create these plots. The listing below shows how to generate time history graphs for the displacement magnitude for points located at (300,300), (800,300) and (900,300) pixels. This results in a 2D graph with three lines. The second function call in the listing generates a similar plot for the Von Mises strain for the same three points, also resulting in a graph with three lines. These functions will use all image pairs available in the results file.

```
import sundic.post_process as sdpp

# Setup the list of points to track
points=[(300,300), (800,300), (900,300)]

# Create the displacement time history graph
sdpp.plotDispTimeHistory('results.sdic', points,
                        dispComp=sdpp.DispComp.DISP_MAG)

# Create the strain time history graph
sdpp.plotStrainTimeHistory('results.sdic', points,
                          strainComp=sdpp.StrainComp.VM_STRAIN)
```

A complete list of parameters for the `plotDispTimeHistory` and `plotStrainTimeHistory` functions are summarized below.

```

# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# points (list)      : list of (x, y) tuples in pixel coordinates
#
# Optional Parameters:
# -----
# gridLines (bool)  : whether to plot grid lines
#                    (Default is True -> Plot grid lines)
# showPlot (bool)   : whether to show the plot here
#                    (Default is True -> Show the plot)
# fileName (str)    : the file name to save the plot to
#                    (Default is '' -> No file saved)
# smoothWindow (int): the size of the smoothing window
#                    this is the number of surrounding subSets to
#                    consider in the smoothing
#                    (Default is 0 -> no smoothing for displacements
#                    9 -> smoothing for strains)
# smoothOrder (int) : the order of the smoothing polynomial
#                    (Default is 2 -> quadratic smoothing)
# interpolate (bool): flag to interpolate values at the specified
#                    points
#                    (Default is False -> Use nearest subset value)
# -----

```

If the interpolate flag is set to False, the nearest subset point to the specified point will be used to create the time history plots. If this flag is set to True, the data will be interpolated to the specified point.

As for the line cut plots, there is an additional optional parameter `dispComp` to identify the displacement component to plot when calling `plotDispTimeHistory`. Available `dispComp` values are `X_DISP`, `Y_DISP`, and `DISP_MAG`.

```

# dispComp (int)      : the displacement component to plot
#                    (Default is sdpp.DispComp.MAG -> Magnitude)

```

For `plotStrainTimeHistory` the corresponding parameter to identify the strain component to plot is `strainComp` and available `strainComp` values are `X_STRAIN`, `Y_STRAIN`, `SHEAR_STRAIN`, and `VM_STRAIN`.

```

# strainComp (int)    : the strain component to plot
#                    (Default is sdpp.StrainComp.VM_STRAIN)

```

5.4.6 Accessing Raw Data

For custom analysis or export, the displacement and strain arrays can be retrieved directly as NumPy arrays. Each row corresponds to one subset point and missing points carry NaN values

that can be filtered out with a standard NumPy mask. The following listing first retrieves the displacement data and then the strain data.

```
import sundic.post_process as sdpp

# Get raw displacement data for the last image pair
disp, nRows, nCols = sdpp.getDisplacements('results.sdic', -1)

# Get raw strain data for the last image pair
strains, nRows, nCols = sdpp.getStrains('results.sdic', -1)
```

The column layout for the displacement data is as follows:

X Coord	Y Coord	Z Coord	X Disp	Y Disp	Z Disp	Magnitude	
---------	---------	---------	--------	--------	--------	-----------	--

The column layout for the strain data is as follows:

X Coord	Y Coord	Z Coord	X Strain	Y Strain	XY Strain	VM Strain

A complete list of parameters for the `getDisplacements` and `getStrains` functions are summarized below.

```
# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# imgPair (int)      : the zero based image pair ID to process
#
# Optional Parameters:
# -----
# smoothWindow (int): the size of the smoothing window
#                      this is the number of surrounding subSets to
#                      consider in the smoothing
#                      (Default is 0 -> no smoothing for displacements
#                      9 -> smoothing for strains)
# smoothOrder (int) : the order of the smoothing polynomial
#                      (Default is 2 -> quadratic smoothing)
# dilation (int)      : the number of subsets to dilate the mask
#                      around automatically detected features (eg holes)
#                      where the results may be of poor quality
#                      (Default is 0 -> no dilation)
#
# Returns:
# -----
# results: the displacement/strain results with the columns as outlined
#          above
# nRows   : the number of subset rows in the image
# nCols   : the number of subset columns in the image
```

5.4.7 Accessing Raw Time History Data

For custom analysis or export, the displacement and strain time history data can be retrieved directly as NumPy arrays using the `getDispTimeHistory` and `getStrainTimeHistory` functions. The following listing retrieves the displacement and strain time history.

```
import sundic.post_process as sdpp

# Setup some points to track
points = [(300,300), (800,300), (900,300)]

# Get the displacement time history data for these points
imgPairs, dispHist, usedPnts = sdpp.getDispTimeHistory('results.sdic',
                                                       points, dispComp=sdpp.DispComp.X_DISP)

# Get the strain time history data for these points
imgPairs, strnHist, usedPnts = sdpp.getStrainTimeHistory(
    'results.sdic', points,
    strainComp=sdpp.StrainComp.VM_STRAIN)
```

A complete list of parameters for the `getDispTimeHistory` and `getStrainTimeHistory` functions are summarized below.

```
# Required Parameters:
# -----
# resultsFile (str) : the binary results file name to read
# points (list)      : list of (x, y) tuples in pixel coordinates
#
# Optional Parameters:
# -----
# smoothWindow (int): the size of the smoothing window
#                    this is the number of surrounding subSets to
#                    consider in the smoothing
#                    (Default is 0 -> no smoothing for displacements
#                    9 -> smoothing for strains)
# smoothOrder (int) : the order of the smoothing polynomial
#                    (Default is 2 -> quadratic smoothing)
# interpolate (bool): flag to interpolate values at the specified
#                    points
#                    (Default is False -> Use nearest subset value)
#
# Returns:
# -----
# imgPairs (ndarray): the image pair indices, shape (nPairs,)
# hist (ndarray)      : the time history values, shape (nPoints, nPairs)
# usedPoints (list)   : List of (x, y) points used to sample data
#                    If interpolate=False -> nearest subset
#                    If interpolate=True -> the requested input points
```

As for the time history plots, there is an additional optional parameter `dispComp` to identify the displacement component to consider when calling `getDispTimeHistory`. Available `dispComp` values are `X_DISP`, `Y_DISP`, and `DISP_MAG`.

```
# dispComp (int)      : the displacement component to plot
#                    (Default is sdpp.DispComp.MAG -> Magnitude)
```

For `getStrainTimeHistory` the corresponding parameter to identify the strain component to consider is `strainComp` and available `strainComp` values are `X_STRAIN`, `Y_STRAIN`, `SHEAR_STRAIN`, and `VM_STRAIN`.

```
# strainComp (int)    : the strain component to plot
#                    (Default is sdpp.StrainComp.VM_STRAIN)
```

Chapter 6

Theory

SUN-DIC implements local (subset-based) Digital Image Correlation [Sutton et al., 2009, Pan, 2018], in which the image is divided into small overlapping subsets that are independently tracked from a reference image to a deformed image. The sections below provide a short summary of some of the key algorithmic components implemented in SUN-DIC. A good high level overview of DIC is provided by [A practical guide to DIC](#), while many excellent textbooks and academic papers are available for further reading.

The purpose of this chapter is to provide a brief overview of the algorithms implemented in SUN-DIC. It is not intended to be a comprehensive introduction to Digital Image Correlation. Readers interested in detailed derivations and theoretical background are encouraged to consult the references cited throughout this chapter.

6.1 The DIC Problem

Let $f(\mathbf{x})$ be the intensity at pixel $\mathbf{x} = (x, y)$ in the reference image and $g(\mathbf{x}')$ the intensity at the corresponding location in the deformed image. A subset S centred at $\mathbf{x}_0$ in the reference image is mapped to the deformed image by a warp function $\mathcal{W}(\xi; \mathbf{p})$, where $\xi = (\xi, \eta)$ are local coordinates measured relative to the subset centre and $\mathbf{p}$ is the vector of deformation parameters (see Section 6.4).

The goal of the optimisation is to determine the deformation parameters $\mathbf{p}$ that best align the reference subset with its counterpart in the deformed image by minimising a correlation cost function.

6.2 Correlation Criterion

Many correlation criteria have been proposed in the DIC literature. SUN-DIC uses the Zero-Mean Normalised Sum of Squared Differences (ZNSSD) criterion to determine the optimal deformation parameters $\mathbf{p}$. The ZNSSD criterion provides robustness to illumination changes by normalising intensity values within each subset:

$$C_{\text{ZNSSD}}(\mathbf{p}) = \sum_{\xi \in S} \left[\frac{f(\mathbf{x}_0 + \xi) - \bar{f}}{\Delta f} - \frac{g(\mathcal{W}(\xi; \mathbf{p})) - \bar{g}}{\Delta g} \right]^2 \quad (6.1)$$

where $\bar{f}$ and $\bar{g}$ are the mean intensities of the reference and deformed subsets, and

$$\Delta f = \sqrt{\sum_{\xi \in S} (f(\mathbf{x}_0 + \xi) - \bar{f})^2}, \quad \Delta g = \sqrt{\sum_{\xi \in S} (g(\mathcal{W}(\xi; \mathbf{p})) - \bar{g})^2} \quad (6.2)$$

are the L_2 norms of the zero-mean subsets. ZNSSD is insensitive to uniform changes in illumination offset and scale.

The closely related Zero-Mean Normalised Cross-Correlation (ZNCC) is related to ZNSSD by [Pan et al., 2013]:

$$\text{ZNCC} = 1 - \frac{1}{2} C_{\text{ZNSSD}} \quad (6.3)$$

The ZNCC value ranges from -1 to 1 , where a value of 1 indicates a perfect match between the reference and deformed subsets. SUN-DIC uses ZNCC as an optional convergence criterion via the `NZCCThreshold` parameter.

6.3 Subpixel Interpolation

Deformation rarely produces integer-pixel shifts, so the deformed subset intensities must be evaluated at non-integer pixel locations at every optimisation iteration. Accurate subpixel interpolation is essential for measurement accuracy [Schreier, 2000].

SUN-DIC uses bicubic (`InterpolationOrder=3`) or biquintic (`InterpolationOrder=5`) interpolation [Stein, 2024]. Fifth-order interpolation is marginally more accurate, while third-order interpolation is generally more numerically robust and is recommended for most applications.

6.4 Shape Functions

The mapping between the reference and deformed subsets is described by a shape function that approximates the local deformation within each subset. SUN-DIC supports both affine and quadratic shape functions.

6.4.1 Affine Shape Functions (6 DOF)

The affine warp uses six deformation parameters and is written in homogeneous matrix form [Gao et al., 2015]:

$$\mathcal{W}(\xi; \mathbf{p}) = \begin{bmatrix} 1 + u_x & u_y & u \\ v_x & 1 + v_y & v \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} \xi \\ \eta \\ 1 \end{pmatrix} \quad (6.4)$$

with $\mathbf{p} = (u, u_x, u_y, v, v_x, v_y)^T$, where (u, v) are the in-plane translations and (u_x, u_y, v_x, v_y) are first-order displacement gradients that capture uniform stretching, rotation, and shear within the subset. Figure 6.1 illustrates the transformation.

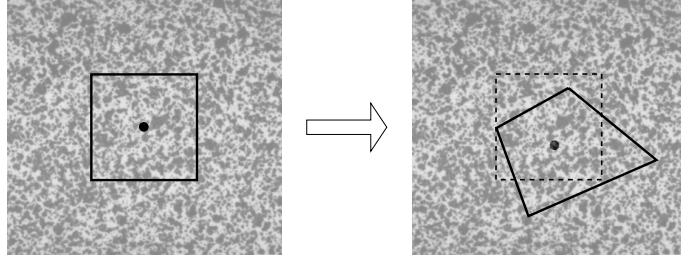


Figure 6.1: Affine transformation from reference (left) to deformed (right) subset.

6.4.2 Quadratic Shape Functions (12 DOF)

The quadratic warp adds second-order terms, increasing the parameter vector to

$$\mathbf{p} = (u, u_x, u_y, u_{xx}, u_{xy}, u_{yy}, v, v_x, v_y, v_{xx}, v_{xy}, v_{yy})^T.$$

This model can better represent non-uniform deformation, bending, and large strain gradients. However, the additional degrees of freedom increase sensitivity to noise and generally require larger subset sizes for stable results.

6.5 Optimisation Algorithms

SUN-DIC provides two optimisation algorithms for determining the deformation parameters $\mathbf{p}$. Both are based on the inverse compositional framework [Baker and Matthews, 2001, Pan et al., 2013], where image gradients and the Hessian are computed once from the reference image and reused at every iteration, providing significant computational savings.

The approximate Hessian is

$$\mathbf{H} = \sum_{\xi \in S} \mathbf{s}(\xi)^T \mathbf{s}(\xi) \quad (6.5)$$

with

$$\mathbf{s}(\xi) = \frac{1}{\Delta f} \nabla f(\mathbf{x}_0 + \xi) \cdot \left. \frac{\partial \mathcal{W}}{\partial \mathbf{p}} \right|_{\mathbf{p}=0} \quad (6.6)$$

where $\nabla f = (\partial f / \partial x, \partial f / \partial y)$ is the reference-image gradient and $\partial \mathcal{W} / \partial \mathbf{p}$ is the warp Jacobian.

6.5.1 IC-GN: Inverse Compositional Gauss-Newton

The IC-GN algorithm computes the residual:

$$r(\xi; \mathbf{p}) = \frac{f(\mathbf{x}_0 + \xi) - \bar{f}}{\Delta f} - \frac{g(\mathcal{W}(\xi; \mathbf{p})) - \bar{g}}{\Delta g} \quad (6.7)$$

The parameter update is

$$\Delta \mathbf{p} = -\mathbf{H}^{-1} \sum_{\xi \in S} \mathbf{s}(\xi)^T r(\xi; \mathbf{p}) \quad (6.8)$$

and the warp update is

$$\mathcal{W}(\cdot; \mathbf{p}) \leftarrow \mathcal{W}(\cdot; \mathbf{p}) \circ \mathcal{W}(\cdot; \Delta \mathbf{p})^{-1} \quad (6.9)$$

IC-GN exhibits second-order convergence near the solution but has a relatively limited convergence radius. Reliable initial guesses (Section 6.6) are therefore important for robust performance.

6.5.2 IC-LM: Inverse Compositional Levenberg–Marquardt

IC-LM replaces the Hessian with a damped form:

$$\Delta \mathbf{p} = -(\mathbf{H} + \lambda \mathbf{I})^{-1} \sum_{\xi \in S} \mathbf{s}(\xi)^T r(\xi; \mathbf{p}) \quad (6.10)$$

The damping parameter λ is adapted dynamically to balance stability and speed. This gives IC-LM a larger convergence radius than IC-GN, although each iteration is computationally more expensive.

SUN-DIC also provides an experimental Fast-IC-LM variant that improves computational efficiency while introducing only a negligible loss in accuracy.

Guidance: IC-GN with AKAZE initialisation (Section 6.6) is sufficient for most cases. IC-LM is recommended for difficult cases involving large deformation gradients or poor initial guesses.

6.6 Starting Strategy – AKAZE Initialisation

Both IC-GN and IC-LM require good initial estimates for $\mathbf{p}$ to converge reliably. SUN-DIC uses a two-phase strategy to generate robust initial guesses and reduce the risk of error propagation between neighbouring subsets.

Phase 1 – Initialisation: An $n \times n$ grid of seed points (controlled by `StartingPoints`) is distributed across the ROI. For each seed, AKAZE features [Alcantarilla et al., 2013] are used to estimate an initial affine transformation and compute C_{ZNSSD} .

Phase 2 – Propagation: The best seed is optimised first. Its solution is then propagated to neighbouring subsets, and optimisation proceeds in order of increasing initial cost function values. This reduces the accumulation of errors that can occur in purely sequential approaches.

AKAZE is preferred for its robustness to blur, rotation, scaling, and affine distortions [Isik, 2024].

6.7 Strain Calculation

Strains are computed from the displacement field (u, v) produced by the optimiser and are evaluated as a post-processing step. Because DIC displacement fields contain noise, direct numerical differentiation would amplify errors. Instead, SUN-DIC applies 2D Savitzky–Golay smoothing [Pan, 2007]. The displacement field is locally approximated by a polynomial, and derivatives are computed analytically from the fitted coefficients.

The in-plane engineering strains are:

$$\varepsilon_{xx} = \frac{\partial u}{\partial x}, \quad \varepsilon_{yy} = \frac{\partial v}{\partial y}, \quad \varepsilon_{xy} = \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \quad (6.11)$$

The Von Mises equivalent strain is:

$$\varepsilon_{VM} = \sqrt{\varepsilon_{xx}^2 + \varepsilon_{yy}^2 - \varepsilon_{xx} \varepsilon_{yy} + 3 \varepsilon_{xy}^2} \quad (6.12)$$

Bibliography

- P. Alcantarilla, J. Nuevo, and A. Bartoli. Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *Proceedings of the British Machine Vision Conference*. BMVA Press, 2013.
- S. Baker and I. Matthews. Equivalence and efficiency of image alignment algorithms. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–1090–I–1097. IEEE Comput. Soc, 2001. doi: 10.1109/cvpr.2001.990652.
- T. Gao, Y. and Cheng, Y. Su, X. Xu, Y. Zhang, and Q. Zhang. High-efficiency and high-accuracy digital image correlation for three-dimensional measurement. *Optics and Lasers in Engineering*, 65:73–80, 2015. doi: 10.1016/j.optlaseng.2014.05.013.
- M. Isik. Comprehensive empirical evaluation of feature extractors in computer vision. *PeerJ Computer Science*, 10:e2415, 11 2024. doi: 10.7717/peerj-cs.2415.
- B Pan. Full-field strain measurement using a two-dimensional savitzky-golay digital differentiator in digital image correlation. *Optical Engineering*, 46(3):033601, 3 2007. doi: 10.1117/1.2714926.
- B. Pan. Digital image correlation for surface deformation measurement: historical developments, recent advances and future goals. *Measurement Science and Technology*, 29(8): 082001, 6 2018. doi: 10.1088/1361-6501/aac55b.
- B. Pan, K. Li, and W. Tong. Fast, robust and accurate digital image correlation calculation without redundant computations. *Experimental Mechanics*, 53(7):1277–1289, 2 2013. doi: 10.1007/s11340-013-9717-6.
- H. Schreier. Systematic errors in digital image correlation caused by intensity interpolation. *Optical Engineering*, 39(11):2915, 11 2000. doi: 10.1117/1.1314593.
- D. B. Stein. fast_interp: Fast n-dimensional linear interpolation in numpy. https://github.com/dbstein/fast_interp, 2024. Accessed: 2025-05-30.
- M. A. Sutton, J. J. Orteu, and H. Schreier. *Image correlation for shape, motion and deformation measurements: basic concepts, theory and applications*. Springer Science & Business Media, 2009.