pysdic

Stéréo-corrélation d'images numériques en Python

Artezaru

November 26, 2026

À propos de cette présentation

- une présentation des principes et de l'organisation de `pysdic`, avec quelques exemples de code ;
- pas une documentation : les arguments, les options et les cas limites ne sont pas détaillés ;
- la documentation complète de l'API est en ligne, en HTML.

Plan

Introduction

Géométrie

Imagerie

SDIC

Outils complémentaires

Introduction

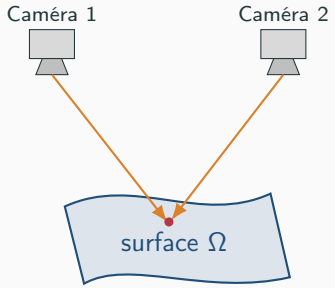
La stéréo-corrélation d'images (SDIC)

Objectif : mesurer le champ de déplacement 3D d'une surface à partir d'images prises par plusieurs caméras, à plusieurs instants.

Trois ingrédients :

- une géométrie : la surface, un maillage éléments finis ;
- une imagerie : les caméras, les images, les vues ;
- une analyse : quelles vues comparer, quelles inconnues optimiser (déplacements et/ou paramètres des caméras).

La mesure compare les niveaux de gris d'un même point matériel vu par deux vues.



Dépendances et conventions

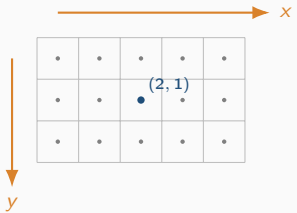
Dépendances

- obligatoires : `numpy`, `scipy`, `matplotlib`, `cv2`, `pysolvegn` ;
- optionnelles, importées à l'usage : `open3d`, `pyvista`, `meshio`, `h5py`...

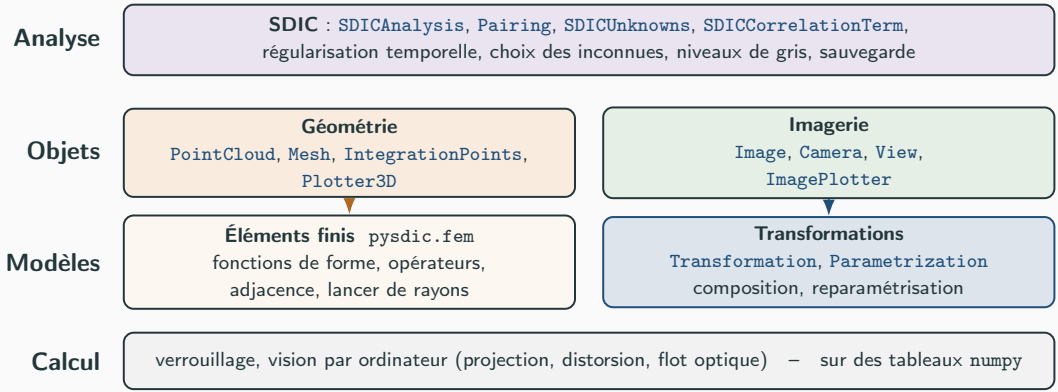
```
mesh.cast_rays(rays)
# ImportError : cast_rays requires 'open3d'
#   pip install open3d
```

Coordonnées image

- (x, y) = (colonne, ligne), y vers le bas ;
- pixel (u, v) = (ligne, colonne) ;
- centres des pixels aux entiers.



Organisation de `pysdic` : trois couches



Partager et verrouiller

Objets partagés

Les objets ne sont jamais copiés : deux caméras construites avec la même extrinsèque utilisent le même objet.

```
ext = PerspectiveExtrinsic(...)
cam_1 = Camera(..., extrinsic=ext, ...)
cam_2 = Camera(..., extrinsic=ext, ...)

cam_1.extrinsic.parameters = p
cam_2.extrinsic.parameters   # p aussi !
```

Attention : modifier l'un modifie tous ceux qui le partagent.

Objets verrouillés

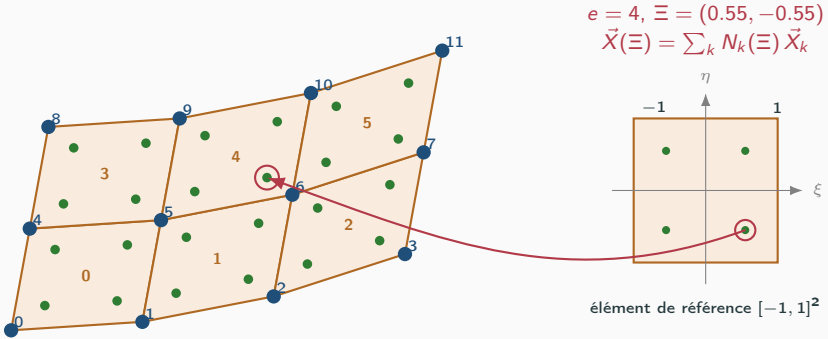
Une analyse (`SDICAnalysis`) a besoin d'un état invariant : elle verrouille son maillage.

```
mesh = Mesh(...)
mesh.is_locked          # False

analysis = SDICAnalysis(mesh, views)
mesh.is_locked          # True : verrouillé par l'analyse

mesh.coordinates = X
# RuntimeError: Cannot set the coordinates: this Mesh
#   is locked. It is used by an SDIC analysis, whose
#   operators [...] assume that its vertices and
#   elements never change. [...] To use a modified
#   mesh, work on a copy (mesh.copy()) and create a
#   new analysis with it (analysis.with_mesh(new_mesh)).

new_mesh = mesh.copy()  # déverrouillé
```

Messages détaillés : qui verrouille, pourquoi, et quoi faire.

Géométrie

Maillage et points d'intégration : ce qu'on définit

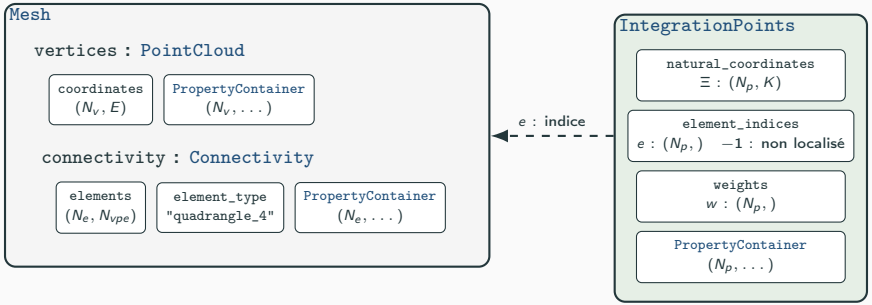


Maillage ● sommets `mesh.vertices` ■ éléments `mesh.connectivity`

Points d'intégration ● `IntegrationPoints` : (e, Ξ, w)

Un point d'intégration est repéré dans un élément, pas dans l'espace : il suit les sommets quand le maillage se déforme.

Géométrie : comment les objets s'emboîtent



element_type	N_{vpe}	K
segment_2/3	2 / 3	1
triangle_3/6	3 / 6	2
quadrangle_4/8	4 / 8	2

Le type d'élément fixe le nombre de sommets par élément N_{vpe}, la dimension K des coordonnées naturelles et les fonctions de forme :

$$\vec{X}(\Xi) = \sum_k N_k(\Xi)\, \vec{X}_k$$

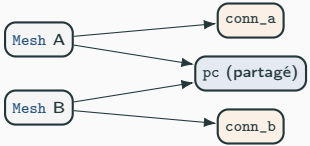
Mesh dans le code

```python
from pysdic import Mesh

# coords (N_v, 3), elements (N_e, 4)
coords = [[0, 0, 0], [1, 0, 0], [2, 0, 0],
          [0, 1, 0], [1, 1, 0], [2, 1, 0]]
elements = [[0, 1, 4, 3], [1, 2, 5, 4]]
mesh = Mesh(coords, elements,
            element_type="quadrangle_4")

mesh.vertices.coordinates   # (N_v, 3)
mesh.connectivity.elements  # (N_e, 4)

# Des propriétés sur les sommets et les éléments
mesh.vertices["temperature"] = T   # (N_v,)
mesh.connectivity["material"] = [0, 1]
```

```python
mesh.to_vtk_file("surface.vtk")
mesh = Mesh.from_vtk_file("surface.vtk")

mesh.to_json_file("surface.json")
mesh = Mesh.from_json_file(
    "surface.json")
```

Deux maillages peuvent partager les mêmes sommets ou la même connectivité :

```python
mesh_a = Mesh(pc, conn_a)
mesh_b = Mesh(pc, conn_b)  # mêmes sommets
mesh_b.vertices is mesh_a.vertices  # True
```



`IntegrationPoints` : deux façons de les créer

Par coordonnées

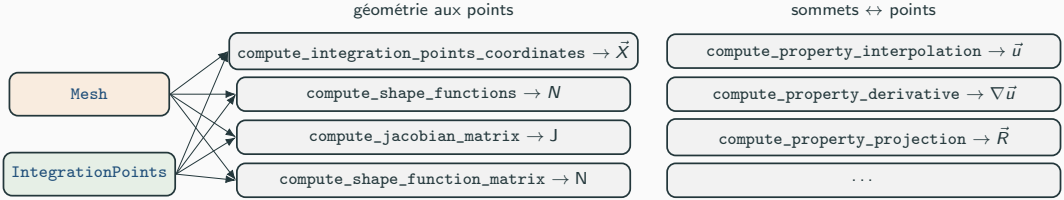
```python
from pysdic import IntegrationPoints

ip = IntegrationPoints(
    natural_coordinates=xi_p,  # Xi (N_p, K)
    element_indices=e_p,       # e  (N_p,)
    weights=w_p)               # w  (N_p,)
```

Ξ : position dans l'élément de référence ; e : élément qui contient le point ; w : poids.

Par lancer de rayons (un point par pixel)

```python
rays = camera.get_camera_rays()  # (H, W, 6)
ip = mesh.cast_rays(rays)        # H*W
     points
ip.is_located()                  # e != -1
ip = ip.remove_unlocated()
```

Maillage × points d'intégration




```
X = mesh.compute_integration_points_coordinates(ip)  # (N_p, E)       positions
N = mesh.compute_shape_functions(ip)                 # (N_p, N_vpe)   N_k aux points
J = mesh.compute_jacobian_matrix(ip)                 # (N_p, E, K)    dX / dXi
N_mat = mesh.compute_shape_function_matrix(ip, sparse=True)  # (N_p, N_v)
u = mesh.compute_property_interpolation(U, ip)       # (N_p, 3)       sommets -> points
grad_u = mesh.compute_property_derivative(U, ip)     # (N_p, 3, 3)
R = mesh.compute_property_projection(r, ip)          # (N_v, ...)     points -> sommets
```

Visualisation : `Plotter3D`

Basé sur PyVista :

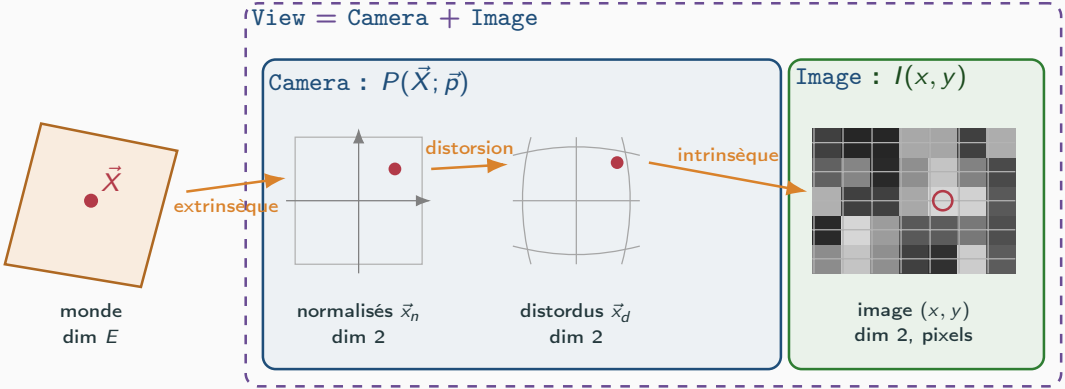
- `add_mesh` : faces, arêtes, sommets, propriétés, texture (UV map), `warp_by` ;
- `add_point_cloud` ;
- `add_integration_points` : positions dans le maillage (déformé) ;
- champs de vecteurs (flèches).

`warp_by` : un tableau (N_v, E) ou le nom d'une propriété des sommets.

```python
from pysdic import Plotter3D

plotter = Plotter3D()
plotter.add_mesh(
    mesh,
    warp_by=displacement,
    vertex_property=displacement,
    show_bar=True,
)
plotter.add_integration_points(
    ip, mesh, warp_by=displacement,
    color="red", size=4,
)
plotter.show()
```

Imagerie

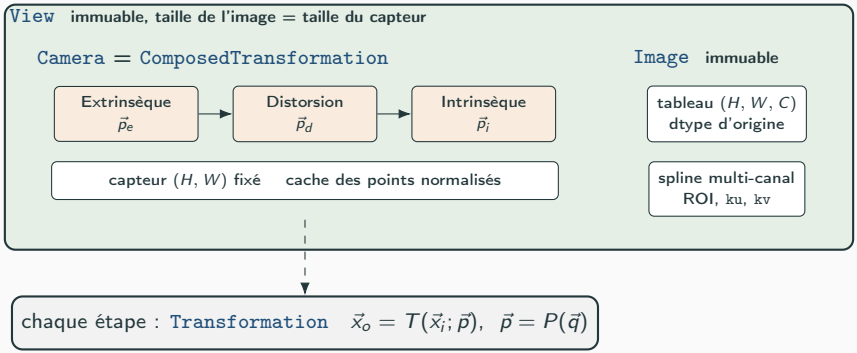
Imagerie : ce qu'on définit



$g = I\big(P(\vec{X};\vec{p})\big)$ niveau de gris du point matériel $\vec{X}$ vu par la vue

Les dérivées par rapport à $\vec{X}$ (mesure) et à $\vec{p}$ (étalonnage) sont fournies.

Imagerie : comment les objets s'emboîtent



La nomenclature `View` / `Camera` / `Image` reprend celle de uCheckmate (H. Matar, 2025). Les `Transformation` et `Parametrization` sont propres à `pysdic`. Transformations et capteur sont fixes : seuls les paramètres changent.

Transformation : $\vec{x}_o = T(\vec{x}_i; \vec{p})$

Classe abstraite d'une application paramétrée $\mathbb{R}^{n_i} \to \mathbb{R}^{n_o}$, point par point :

- `apply` : $\vec{x}_o$, $\mathrm{J}_x = \partial T / \partial \vec{x}_i$, $\mathrm{J}_p = \partial T / \partial \vec{p}$;
- `invert` : T^{-1} et ses jacobiennes ;
- paramètres (dérivables, optimisés) vs constantes ;
- `has_apply_dx`, `has_apply_dp`… : indiquent si une quantité est définie ;
- verrouillable : une transformation verrouillée refuse `set_parameters`.

```
res = T.apply(x, dx=True, dp=True)
res.x             # (..., n_o)
res.jacobian_dx   # (..., n_o, n_i)
res.jacobian_dp   # (..., n_o, n_p)

T.parameters = p
T.to_json_file("extrinsic.json")
```

Les modèles implémentés

modèle	dim.	en une ligne
Extrinsèques monde → normalisés		
`NoExtrinsic`	3D/2D → 2D	identité ; en 3D, z ignoré (orthographique)
`PerspectiveExtrinsic`	3D → 2D	pose $[R\|T]$ puis division par Z_c (OpenCV)
`OrthographicExtrinsic`	3D → 2D	pose $[R\|T]$ sans division par Z_c
`WorldPlanePerspectiveExtrinsic`	2D → 2D	perspective 3D sur le plan $Z_w = 0$: $H = [\vec{r}_1, \vec{r}_2, T]$
`WorldPlaneOrthographicExtrinsic`	2D → 2D	orthographique 3D sur le plan $Z_w = 0$ (affine)
`PlanarRigidExtrinsic`	2D → 2D	rigide plane : θ et $\vec{t}$ (3 paramètres)
`PlanarHomographicExtrinsic`	2D → 2D	homographie plane H (8 paramètres, $H_{22} = 1$)
Distorsions normalisés → distordus		(inverse par Newton)
`NoDistortion`	2D → 2D	identité
`BrownDistortion`	2D → 2D	Brown–Conrady d'OpenCV (0, 4, 5, 8, 12 ou 14 paramètres)
`ZernikeDistortion`	2D → 2D	déplacement sur les polynômes de Zernike, sur une ellipse
Intrinsèques distordus → image (pixels)		
`NoIntrinsic`	2D → 2D	identité
`PinholeIntrinsic`	2D → 2D	matrice K : f_x, f_y, c_x, c_y, biais s optionnel

Composition et reparamétrisation

Composition `T2 @ T1`

$$T(\vec{x};\vec{p}) = T_2\big(T_1(\vec{x};\vec{p}_1);\vec{p}_2\big), \quad \vec{p} = [\vec{p}_1, \vec{p}_2]$$

$$\mathrm{J}_p = \left[\begin{array}{c|c} \frac{\partial T_2}{\partial \vec{x}}\,\frac{\partial T_1}{\partial \vec{p}_1} & \frac{\partial T_2}{\partial \vec{p}_2} \end{array}\right]$$

Les transformations ne sont pas copiées : modifier $\vec{p}$ de la composition modifie chaque étape. Le verrou se propage aux étapes.

Reparamétrisation

`ReparametrizedTransformation`

$$\tilde{T}(\vec{x};\vec{q}) = T\big(\vec{x};P(\vec{q})\big)$$

$$\mathrm{J}_q = \frac{\partial T}{\partial \vec{p}} \cdot \frac{\partial P}{\partial \vec{q}}$$

Une `Parametrization` P change la modélisation de la caméra : sous-ensemble de paramètres, modes PCA, etc.

`Camera` : projection et jacobiennes




```python
from pysdic import (Camera, BrownDistortion,
    PerspectiveExtrinsic, PinholeIntrinsic)

camera = Camera(
    sensor_height=480, sensor_width=640,  # pixels
    extrinsic=PerspectiveExtrinsic.from_rt(
        rvec=[0.1, 0.2, 0.3], tvec=[12, 34, 56]),
    distortion=BrownDistortion(dist),
    intrinsic=PinholeIntrinsic.from_matrix(K))

res = camera.project(X, dx=True, dextrinsic=True)
res.image_points          # (..., 2)
res.jacobian_dx           # (..., 2, 3)
res.jacobian_dextrinsic   # (..., 2, N_e)
```

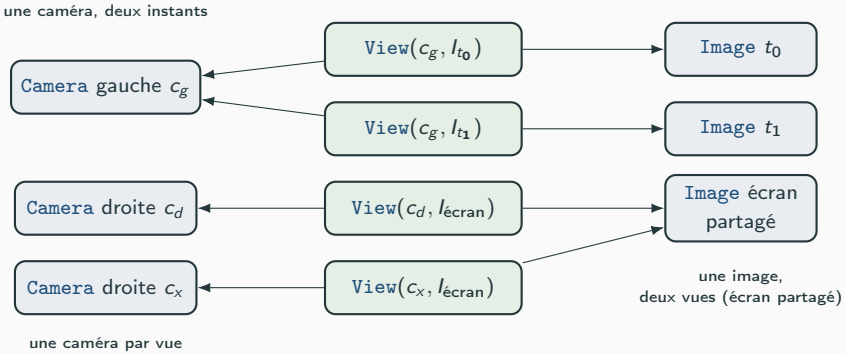
- J_x : sensibilité au déplacement ;
- J_p par blocs, paramètres $[\vec{p}_e, \vec{p}_d, \vec{p}_i]$: étalonnage ; les blocs non demandés valent NaN ;
- accepte aussi un `PointCloud` ou un `Mesh`.

`Image` : niveaux de gris continus

- tableau (H, W) ou (H, W, C), entier ou flottant, `dtype` d'origine conservé ;
- une spline multi-canal, degrés `ku`, `kv` $\in [1, 5]$ (`scipy`) ;
- `roi` $= (x, y, w, h)$: la vraie taille en pixels de la zone interpolée ;
- hors ROI : NaN ;
- valeurs et dérivées en un seul passage ;
- immuable : données copiées en lecture seule, aucun setter ; `with_settings` rend une nouvelle image qui partage les données.

```python
from pysdic import Image

image = Image.from_file(
    "ref.tif", roi=(100, 50, 400, 300))
image = Image.from_array(array)

# ordres (x, y) -> un tuple
I, Ix, Iy = image.evaluate_at_image_points(
    xy, derivatives=((0, 0), (1, 0), (0, 1)))
(I,) = image.evaluate_at_image_points(xy)
(I,) = image.evaluate_at_pixel_points(uv)

zoom = image.with_settings(
    roi=(0, 0, 200, 200), ku=5)

image.to_file("copy.png")      # dtype conservé
image.to_npz_file("ref.npz")   # + ROI, ku, kv
```

`View` : partager caméras et images



$$g = I\big(P(\vec{X})\big), \qquad \nabla_\gamma \big[I \circ P\big] = \nabla_x I\big(P(\vec{X})\big) \cdot \nabla_\gamma P(\vec{X}), \quad \gamma \in \{\vec{X}, \vec{p}\}$$

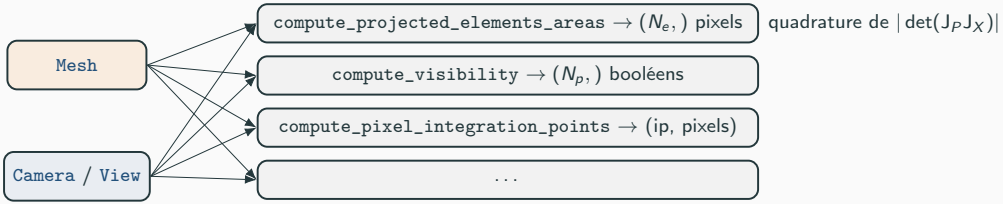
`View` dans le code

```python
from pysdic import View

view = View(camera, image)
res = view.image_project(X, dx=True,
                         dextrinsic=True)
res.gray_levels            # (..., C)
res.image_points           # (..., 2)
res.jacobian_dx            # (..., C, E)
res.jacobian_dextrinsic    # (..., C, N_e)

view.quick_visualize(mesh)
```

- `View` immuable ;
- C canaux par point ;
- hors ROI : NaN.

`jacobian_dx` est exactement le gradient attendu par `assemble_displacement_jacobian_matrix`.

Maillage × caméra




```
from pysdic import (compute_projected_elements_areas, compute_visibility,
                    compute_pixel_integration_points)

areas = compute_projected_elements_areas(mesh, camera)
seen = compute_visibility(mesh, ip, view, warp_by=displacement)
ip_px, pixels = compute_pixel_integration_points(mesh, view)
```

Visualisation : `ImagePlotter`

Basé sur `matplotlib`, dans le repère image :

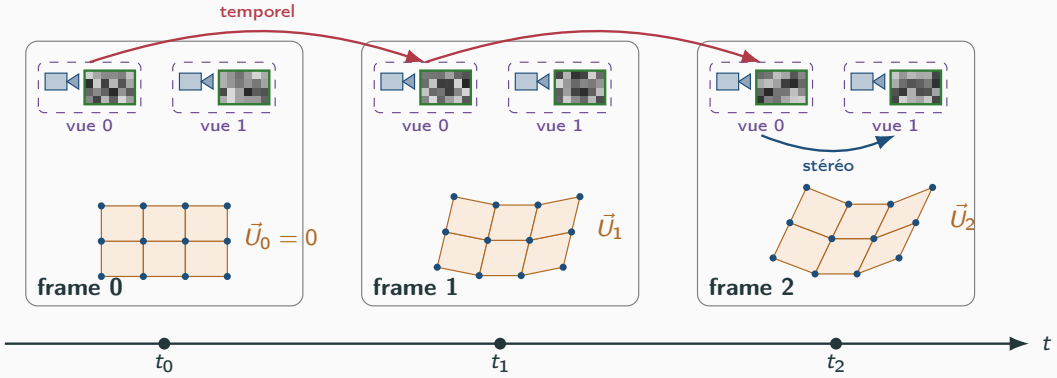
- source : une `Camera`, une `View` (avec son image) ou une taille (H, W) ;
- `add_image`, `add_sensor` ;
- `add_mesh`, `add_point_cloud`, `add_integration_points` : objets 3D projetés par la caméra ;
- `add_rectangle`.

Même API que `Plotter3D`.

```python
from pysdic import ImagePlotter

plotter = ImagePlotter(view)
plotter.add_image()
plotter.add_mesh(
    mesh,
    warp_by=displacement,
    vertex_property=displacement,
    face_opacity=0.6,
    show_bar=True,
)
plotter.add_integration_points(
    ip, mesh, warp_by=displacement,
    color="red", size=3,
)
plotter.show()
```

SDIC

SDIC : ce qu'on définit



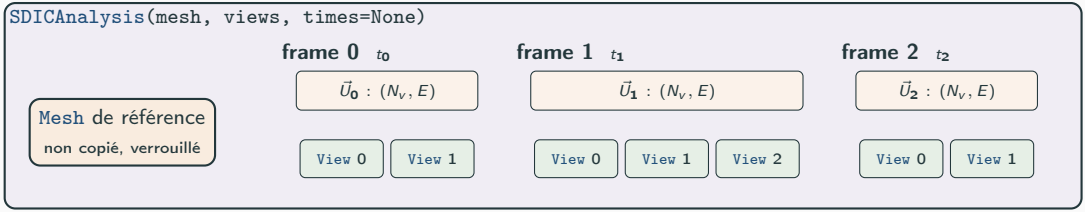
Maillage ■ un seul maillage de référence, déformé par un déplacement $\vec{U}_f$ par frame

Vues □ par frame, des vues : caméra + image

Appariements $\rightarrow$ temporel, $\rightarrow$ stéréo : les vues comparées

On cherche les $\vec{U}_f$ (et des paramètres de caméras) qui rendent les vues cohérentes.

SDICAnalysis : comment les données s'emboîtent



- les déplacements sont stockés hors du maillage, un tableau par frame (lecture seule, jamais remis à `None`) ;
- instants d'acquisition `times` optionnels, strictement croissants ;
- le nombre de vues peut changer d'une frame à l'autre (même zéro) ;
- des vues peuvent partager, ou non, la même `Camera`, la même `Image`, ou seulement une transformation (extrinsèque, distorsion, intrinsèque) ;
- chaque transformation distincte est repérée par `(kind, index)` : `analysis.display_structure("distortion")`, `analysis.get_transformation("distortion", 1)`.

`SDICAnalysis` dans le code

```python
from pysdic import SDICAnalysis, View

# views[f] : les vues de la frame f
views = [[View(left, images[f, 0]), View(right, images[f, 1])]
         for f in range(2)]
analysis = SDICAnalysis(mesh, views, times=[0.0, 0.1])
analysis.set_zero_displacement(0)
analysis.set_displacement(1, first_guess)

analysis.get_view(1, 0)         # frame 1, vue 0
analysis.get_displacement(1)    # (N_v, 3), lecture seule
analysis.get_deformed_mesh(1)   # Mesh déformé

analysis.display_structure("extrinsic")
#   Frame 0  : [0, 1]
#   Frame 1  : [0, 1]
p = analysis.get_transformation_parameters("extrinsic", 1)
analysis.set_transformation_parameters("extrinsic", 1, p)

# Une autre structure = une nouvelle analyse
other = analysis.with_frame_views(1, [View(left, image)])
other = analysis.with_mesh(finer_mesh, keep_displacements=False)
```

- vues données à la construction : la structure est figée, calculée une fois ;
- `with_mesh`, `with_views`, `with_frame_views`, `with_times` : nouvelle `SDICAnalysis` qui partage le reste (images, caméras, déplacements) ;
- `compute_structure(kind)`, par identité : vue, image, caméra, extrinsèque, distorsion, intrinsèque ;
- `get/set_transformation_parameters` : les paramètres restent modifiables, verrous respectés ;
- `check_consistency` (à la construction) : extrinsèques, distorsions et intrinsèques disjoints, et chaque transformation unitaire présente une seule fois (même dans une composition ou une reparamétrisation).

Montages types

```python
from pysdic import (create_multicamera_analysis,
      create_bicamera_analysis,
                  create_splitscreen_analysis,
      create_cordin_analysis)

# images[i][f] : caméra i, frame f
analysis = create_multicamera_analysis(mesh, cameras, images,
                                       times=times)  # optionnel
analysis = create_bicamera_analysis(mesh, left, right, imgs_g,
      imgs_d)

# Une image par frame, partagée par les vues
analysis = create_splitscreen_analysis(mesh, [left, right], images
      )

# Une distorsion par frame (miroir tournant)
analysis = create_cordin_analysis(mesh,
    extrinsics=[ext_g, ext_d],
    intrinsic=PinholeIntrinsic.from_matrix(K),
    distortions=[BrownDistortion(d) for d in ds],
    images=images)
```

Cordin :

$$\text{Cam.}[i][f] = \text{Intr.} \circ \text{Dist.}[f] \circ \text{Extr.}[i]$$

- transformations non copiées : modifier une distorsion modifie toutes les caméras de sa frame ;
- déplacements non définis à la création.

Sauvegarder une analyse

```
essai/
  metadata.json   # format, version, date, titre
  README.txt
  manifest.json   # SHA-256 de chaque fichier
  structure.json  # caméras, vues, frames
  mesh/mesh.npz   # (+ mesh.vtk, export)
  images/image_00.npz ...
  extrinsics/extrinsic_0.json ...
  distortions/distortion_0.json ...
  intrinsics/intrinsic_0.json ...
  displacements.npz   # frame_00, ...
  times.npz
```

```
analysis.to_folder("essai", title="Essai 3",
                   export_vtk=True)
analysis = SDICAnalysis.from_folder("essai")

analysis.to_archive("essai.zip", overwrite=True)
analysis = SDICAnalysis.from_archive("essai.zip")
```

- chaque atome (image, transformation) écrit une seule fois, même partagé ; `structure.json` relie caméras, vues et frames ;
- numéros sur autant de chiffres que le plus grand indice : 1543 images → `image_0000.npz` ;
- aucun pickle : JSON et tableaux numériques (`allow_pickle=False`) ;
- écriture atomique : dossier temporaire horodaté, puis renommé ; `overwrite=True` ne remplace qu'une sauvegarde `SDICAnalysis` ;
- lecture : format, version, SHA-256 (`check_integrity`), puis structures comparées → erreur « Inconsistent loading » ;
- archive : les mêmes fichiers dans un zip (`compress=False` par défaut) ; chemins dangereux refusés.

Non sauvegardés : points d'intégration et appariements.

SDICUnknowns : les inconnues $[\vec{p}_U,\ \vec{p}_T]$

problème	frames	transformations
mesure	`[1]`	`[]`
étalonnage	`[]`	`[("extrinsic", 1)]`
conjoint	`[1]`	`[("distortion", 1)]`

- $\vec{p}_U$: déplacements des frames optimisées, en ordre Fortran ($[U^x_{1..N_v}, U^y_{1..N_v}, U^z_{1..N_v}]$) ;
- $\vec{p}_T$: paramètres des transformations optimisées ;
- simple description (comme `Pairing`) : ne connaît pas l'analyse, validée par `analysis.validate_unknowns` ;
- le vecteur est construit par l'analyse : `analysis.pack/unpack/apply_unknowns(unknowns, ...)` ;
- avec paramétrisation : **il FAUT** `param.reparametrized_pack/unpack/apply`.

```python
from pysdic import SDICUnknowns

unknowns = SDICUnknowns([1], [("distortion", 1)])
analysis.validate_unknowns(unknowns)
analysis.n_unknowns(unknowns)
p0 = analysis.pack_unknowns(unknowns)          # état -> p
U, T = analysis.unpack_unknowns(unknowns, p0) # p -> copies
U[1]                                           # (N_v, E)
T[("distortion", 1)]                           # (n_p,)
analysis.apply_unknowns(unknowns, p0)          # p -> analyse

analysis.get_unknowns_frame_slice(unknowns, 1)
analysis.get_unknowns_transformation_slice(
    unknowns, "distortion", 1)
analysis.compute_unknowns_indices(
    unknowns, vertices=v, components=[0, 1])
```

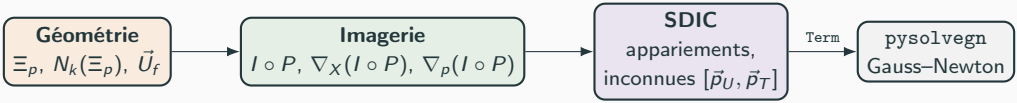
Attention : déplacements et transformations ne sont pas toujours séparables (un mouvement rigide de la surface équivaut à un mouvement rigide de toutes les caméras). N'optimiser que ce que les appariements déterminent.

Le problème : comparer des vues par paires

Un appariement relie la vue (f_1, v_1) à la vue (f_2, v_2). Pour un point matériel $\vec{X}_p$ de poids w_p :

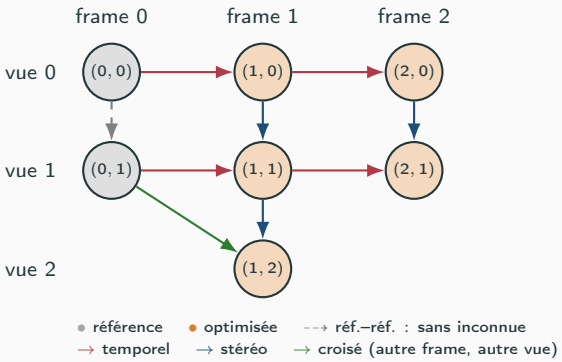
$$r_p = \sqrt{w_p}\left([I_2 \circ P_2^{\vec{p}_T}](\vec{X}_p + \vec{u}_{f_2}(\vec{X}_p)) - [I_1 \circ P_1^{\vec{p}_T}](\vec{X}_p + \vec{u}_{f_1}(\vec{X}_p))\right), \qquad \vec{u}_f(\vec{X}_p) = \sum_k N_k(\Xi_p)\,\vec{U}_{f,k}$$

$$\Phi(\vec{p}_U, \vec{p}_T) = \sum_{\text{appariements}} \sum_p r_p^2, \qquad \vec{p}_U : \text{déplacements des sommets}, \quad \vec{p}_T : \text{paramètres de caméras}$$



Un seul terme, `SDICCorrelationTerm`, optimise les déplacements, les paramètres des caméras, ou les deux à la fois.

Pairing : quoi comparer




```python
from pysdic import Pairing

Pairing(0, 0, 1, 0)   # f1, v1, f2, v2
Pairing.temporal(0, 0, 1)  # v0: f0->f1
Pairing.stereo(1, 0, 1)    # f1: v0->v1
Pairing(0, 1, 1, 2)        # croisé : accepté
pairing.is_stereo, pairing.reversed()
analysis.validate_pairing(pairing)  # vues, canaux
```

Règles du terme de corrélation :

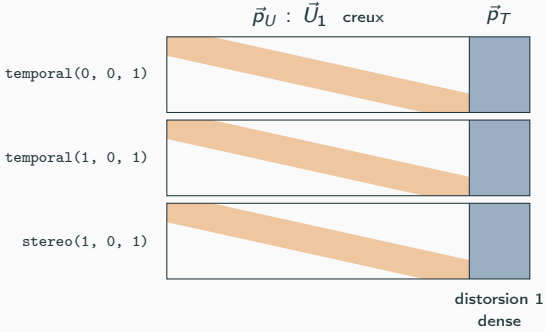
- frame non optimisée = référence (déplacement fixé, défini) ;
- appariement sans inconnue ignoré ; réf.–réf. utile en étalonnage (transformation optimisée) ;
- frame ou transformation optimisée sans appariement → erreur.

SDICCorrelationTerm : un terme, trois problèmes

```python
from pysdic import SDICCorrelationTerm, SDICUnknowns

unknowns = SDICUnknowns([1], [("distortion", 1)])
correlation = SDICCorrelationTerm(
    analysis, unknowns, ip, pairings,
    mode="rJ",                  # ou "gH"
    gl_correction={"mode": "ZNSSD"},
    use_cache=True)

result = pysolvegn.solve(terms=[correlation.term],
                         p0=correlation.pack())
correlation.apply(result.parameters)

# Frame suivante : nouvelles inconnues -> nouveau terme
unknowns_2 = SDICUnknowns([2], [("distortion", 2)])
correlation_2 = correlation.with_unknowns(
    unknowns_2, pairings=pairings_2)
heavier = correlation.with_options(weight=10.0)
```

- `.term` : le `pysolvegn.Term` ($\vec{r}$, J ou $\vec{g}$, H, c) ;
- construit sur une analyse et des inconnues données ; `pack`, `unpack`, `apply` → `analysis.*_unknowns(unknowns, ...)` ;
- + points d'intégration + appariements (`Pairing`) ;
- lit l'analyse à chaque évaluation : déplacements des frames de référence, paramètres des transformations ;
- immuable : `with_unknowns`, `with_pairings`, `with_integration_points`, `with_options(**changes)` créent un nouveau terme (caches neufs) ;
- l'ancien terme reste valide sur ses propres inconnues.

Structure de la jacobienne



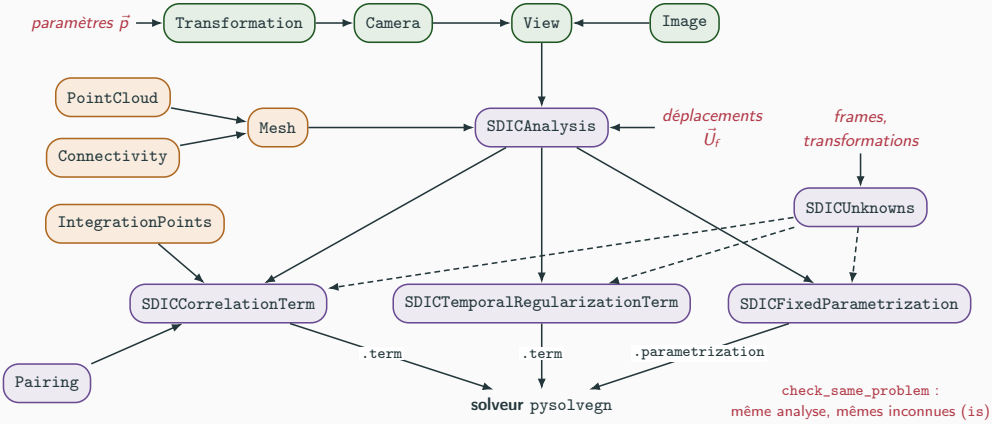
Pour chaque vue d'un appariement :

- déplacement de sa frame (si optimisée) : `jacobian_dx` × fonctions de forme : bloc creux (sommets de l'élément du point) ;
- chaque transformation optimisée de sa caméra : `jacobian_d{kind}`, bloc dense de n_p colonnes ;
- − vue 1, + vue 2 ; transformation partagée ou même frame en stéréo : sommées dans les mêmes colonnes ;
- structure creuse fixe (calculée une fois, CSR triée) : un point hors de l'image donne des zéros explicites.

Exemple complet : une mesure

```python
import numpy, pysolvegn
from pysdic import (IntegrationPoints, Pairing, create_bicamera_analysis, SDICCorrelationTerm, SDICUnknowns,
                    compute_gray_level_groups, compute_visibility, display_residual_maps)

analysis = create_bicamera_analysis(mesh, left, right, left_images, right_images)
analysis.set_zero_displacement(0)
analysis.set_displacement(1, first_guess)

ip = IntegrationPoints.from_gauss_points("quadrangle_4", mesh.n_elements)
for view in analysis.get_views(1):                  # garde-fou : points vus ?
    print(compute_visibility(mesh, ip, view, warp_by=first_guess).mean())

pairings = [Pairing.temporal(0, 0, 1), Pairing.temporal(1, 0, 1), Pairing.stereo(1, 0, 1)]
groups = compute_gray_level_groups(analysis.mesh, ip, neighborhood=1)  # une fois
unknowns = SDICUnknowns([1], [])                    # U_1 inconnu, caméras fixes
correlation = SDICCorrelationTerm(analysis, unknowns, ip, pairings,
                                  gl_correction={"mode": "ZNSSD", "scale": 255.0},
                                  gray_level_groups=groups, use_cache=True)

result = pysolvegn.solve(terms=[correlation.term], p0=analysis.pack_unknowns(unknowns), max_iteration=30)
analysis.apply_unknowns(unknowns, result.parameters)  # U_1 écrit dans l'analyse
display_residual_maps(analysis, correlation.compute_residuals()[2])  # stéréo, ZNSSD
analysis.to_archive("mesure.zip", title="Mesure bicaméra")
```

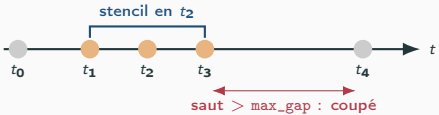
Qui dépend de quoi



$A \to B$: B utilise A, sans copie : modifier *les paramètres* ou *les déplacements* agit sur tout ce qui est en aval. Les bulles sont figées : on en crée une nouvelle (`with_...`). `SDICUnknowns` ne connaît pas l'analyse (pointillés) : l'analyse la valide.

Outils complémentaires

Régularisation temporelle



$$r_{i,v,e} = \sqrt{w_e\,\omega_i}\left(\sum_k c_{i,k}\,u_{f_k,v,e} - a_{f_i,v,e}\right)$$

- $c_{i,k}$: différences finies d'ordre n aux instants `analysis.times` (obligatoires) ;
- `row_weights="time"` : coût $\approx \int \|\vec{u}^{(n)} - \vec{a}\|^2\,dt$, indépendant de l'échantillonnage.

```
from pysdic import (SDICTemporalRegularizationTerm,
                    check_same_problem)

analysis = create_bicamera_analysis(mesh, left, right,
    imgs_g, imgs_d, times=times)
unknowns = SDICUnknowns([1, 2, 3], [])
correlation = SDICCorrelationTerm(analysis, unknowns, ip,
      pairings)
reg = SDICTemporalRegularizationTerm(
    analysis, unknowns, order=2,
    accuracy=2,              # ordre des stencils
    boundary="one_sided",    # ou "valid"
    max_gap=None,            # coupe les grands sauts
    row_weights="time",      # ou "uniform"
    weight=1e-2)

check_same_problem([correlation, reg])
result = pysolvegn.solve([correlation.term, reg.term],
                         analysis.pack_unknowns(unknowns))
analysis.apply_unknowns(unknowns, result.parameters)
softer = reg.with_options(order=1, weight=1e-1)
```

Mêmes objets analyse et `SDICUnknowns` que la corrélation (`check_same_problem`) : les termes s'additionnent. Nouvelles inconnues : `reg.with_unknowns(unknowns)`. Aussi : `target`, `component_weights`.

Choisir les inconnues : paramétrisation fixe

Les termes restent sur le vecteur complet $\vec{p}$. Le solveur optimise $\vec{q}$, une sélection de $\vec{p}$:

$$\vec{p} = P(\vec{q}) = S\,\vec{q} + \vec{p}_{\text{fixe}}, \qquad \mathrm{J}_q = \mathrm{J}_p\, S$$

- `vertices` et `components` choisis ; transformations optimisées : toutes ;
- `compute_supported_vertices` : sommets portant des points ;
- sur l'analyse et les `SDICUnknowns` du problème ; `.parametrization` pour le solveur ;
- $\vec{p}_{\text{fixe}}$ relu à chaque appel de `p_func` : rien à reconstruire après une convergence ;

Attention : `pysolvegn.Parametrization` ($\vec{q} \mapsto \vec{p}$ du solveur) $\neq$ `Parametrization` (paramètres d'une transformation).

```
# 1. Inconnues et termes sur le p complet
unknowns = SDICUnknowns(frames, [])
corr = SDICCorrelationTerm(analysis, unknowns, ip, pairings)
reg = SDICTemporalRegularizationTerm(
    analysis, unknowns, order=2, weight=1e-2)
# 2. Paramétrisation : même analyse, mêmes inconnues
vertices = compute_supported_vertices(analysis.mesh, ip)
param = SDICFixedParametrization(
    analysis, unknowns, vertices=vertices, components=[0, 1])
check_same_problem([corr, reg, param])
# 3. Résolution en q, retour à p
result = pysolvegn.solve([corr.term, reg.term],
    param.reparametrized_pack(),
    parametrization=param.parametrization)
param.reparametrized_apply(result.parameters)
```

Avec une paramétrisation, il FAUT utiliser `param.reparametrized_pack/unpack/apply` (vecteur réduit $\vec{q}$), **jamais** `analysis.*_unknowns` (vecteur complet $\vec{p}$).

Corrections des niveaux de gris

ZNSSD (mode par défaut) : les deux vues normalisées

$$g' = \text{scale} \cdot \frac{g - m}{s}$$

Matching : la vue 2 ramenée sur la vue 1

$$g_2' = \text{scale} \cdot \left(\frac{(g_2 - m_2)\, s_1}{s_2} + m_1 \right)$$

Attention : avec plusieurs appariements, ils n'ont plus le même poids.

Statistiques globales ou par voisinage d'éléments (groupes qui se recouvrent). Points hors ROI ignorés. Les groupes ne dépendent que du maillage et des points : `compute_gray_level_groups` une fois, puis `gray_level_groups=groups` pour tous les termes.

```
gl_correction = {            # None : aucune
    "mode": "ZNSSD",         # "matching"
    "jacobian": "approximate",  # "exact"
    "neighborhood": None,    # ou n >= 0
    "scale": 255.0,          # défaut : 1.0
}
```

- `"approximate"` : m, s constants, $\partial g' \approx \frac{\text{scale}}{s}\, \partial g$;
- `"exact"` : matrice creuse D appliquée aux jacobiennes brutes, $\mathrm{J} = (D_{2\cdot} - D_{1\cdot})\, \mathrm{J}_{\text{brut}}$; dense pour un groupe global ;
- s'applique aussi aux colonnes des transformations ;
- fonctions séparées : `znssd_correction(_exact)`, `matching_correction(_exact)`.

Caches : explicites, jamais par défaut

Terme de corrélation `use_cache=True`

- opérateurs ($\vec{r}$, J ou $\vec{g}$, H, c) rendus pour le même $\vec{p}$ si rien n'a bougé ;
- vues de référence projetées une fois, refaites si leur déplacement ou leur caméra change ;
- un nouveau terme (`with_...`) part de caches vides.

Polynômes de Zernike `ZernikePolynomialsCache`

- les $Z_j(x, y)$ pour des points et une ellipse (R_x, R_y, x_0, y_0) ;
- utile si les points normalisés ne bougent pas (étalonnage de la distorsion seule) ;
- partageable, LRU (`max_entries`).

```
import pysdic

# Une distorsion partagée par les vues L et R
cache = pysdic.ZernikePolynomialsCache(max_entries=2)
distortion.polynomials_cache = cache

unknowns = pysdic.SDICUnknowns([], [("distortion", d)])
correlation = pysdic.SDICCorrelationTerm(
    analysis, unknowns, ip, [pysdic.Pairing.stereo(f, 0, 1)
      ],
    sparse=False, use_cache=True)
result = pysolvegn.solve(terms=[correlation.term],
                         p0=analysis.pack_unknowns(unknowns
      ))
analysis.apply_unknowns(unknowns, result.parameters)

print(cache)  # n_entries, nbytes, hits, misses
distortion.polynomials_cache = None   # retiré
cache.clear()                         # mémoire libérée
```

Attention : environ $8\,N_p(2+P)$ octets par entrée (P polynômes) ; inutile en mesure (les points bougent).

Diagnostic : les résidus

```python
from pysdic import (Pairing, compute_residuals,
    compute_pixel_residual_maps, display_residual_maps)

# Aux points d'intégration : un PairingResiduals
pairing = Pairing.stereo(1, 0, 1)
gl = {"mode": "ZNSSD", "scale": 255.0}
res = compute_residuals(analysis, pairing, ip, gl_correction=gl)
res.gray_levels_1, res.gray_levels_2   # (N_p, C)
res.residuals                          # g'_2 - g'_1

# Au pixel, tracé dans les deux vues
map_1, map_2 = compute_pixel_residual_maps(analysis, res,
                                           fill_radius=1.0)
display_residual_maps(analysis, res)

# Ce que compare un terme, pour un vecteur p
residuals = correlation.compute_residuals(p)  # un par appariement

# Contrôle qualité : RMS par élément, éléments suspects
from pysdic.workflows import compute_elements_residual_rms
quality = compute_elements_residual_rms(analysis, ip, residuals)
quality.bad_elements
```

- les niveaux de gris des deux vues d'un appariement et leur différence, avec la même correction que le terme, sans les poids ;
- le résidu appartient au point matériel : tracé à sa projection dans chaque vue ;
- la carte au pixel montre où la corrélation est mauvaise (reflets, zones cachées…) ;
- `pysdic.workflows` : RMS par élément et par appariement, incertitude des déplacements.

Images synthétiques

Lancer de rayons

vérité exacte

```
from pysdic import (compute_wave_speckle,
    generate_synthetic_image, add_gaussian_noise,
    quantize_image)

texture = compute_wave_speckle(uv_points, wavelength=0.15)
image = generate_synthetic_image(camera, mesh, texture,
    displacement=U, n_samples=3)        # (H, W, 1)
image = quantize_image(add_gaussian_noise(image, std=1.0),
                       bit_depth=8)
```

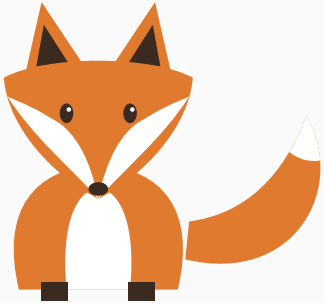
- un rayon par sous-pixel, intersecté avec le maillage déformé ;
- la texture suit le point matériel (UV map) : niveaux de gris conservés, sans éclairage ;
- bruit, flou, quantification : fonctions séparées.

Blender

`pysdic-blender`

```
from pysdic_blender import (BlenderCamera, BlenderMesh,
    BlenderExperiment, BlenderSpotLight, get_steel_material)

experiment = BlenderExperiment(background_strength=0.1)
experiment.add_camera("left", BlenderCamera(camera))
experiment.add_spotlight("spot", BlenderSpotLight.looking_at(
    [300, 0, 600], [0, 0, 0], energy=5e5))
experiment.add_mesh("plate", BlenderMesh(mesh, texture=
      texture),
                    material=get_steel_material())
intensity = experiment.render("left", n_samples=128)
```

- Cycles : ombres, reflets, matériaux (Principled BSDF) ;
- caméra OpenCV sténopé vérifiée, maillage `triangle_3` ;
- package séparé : `bpy` impose la version de Python de Blender (3.13).



Merci !

Questions ?

Références

- H. Matar. *Développement d'un outil de corrélation d'images stéréo : formulation d'un potentiel unique.* Thèse de doctorat, Matériaux, Université Clermont Auvergne, 2025. NNT : 2025UCFA0186. HAL : tel-05612939. (uCheckmate)
- Modèles de caméra et de distorsion : documentation OpenCV `calib3d`.
- `pysolvegn` : solveur de moindres carrés de Gauss–Newton (termes `rJ` / `gH`).