Superposition des champs de déformation

Mohamad SAMMAN et Qinyan YANG

24 février 2025

Contents

1 Introduction

La superposition des champs de déformation est une approche clé en mécanique des matériaux, permettant d'analyser le comportement des structures sous différentes sollicitations mécaniques. Cette méthode est particulièrement utile dans l'étude des matériaux complexes, où les déformations résultent de multiples sources de chargement.

Dans ce projet, nous nous intéressons à la caractérisation mécanique des matériaux en combinant des approches expérimentales et numériques. Plus précisément, nous exploitons l'essai de traction, couplé à la corrélation d'images numériques (DIC, Digital Image Correlation), afin d'obtenir des champs de déformation précis. Ces données sont ensuite comparées et combinées avec des simulations par la méthode des éléments finis (FEM, Finite Element Method) pour affiner l'identification des paramètres mécaniques.

L'objectif principal est d'appliquer une méthode d'analyse inverse pour déterminer les propriétés mécaniques des matériaux. Cette approche permet d'extraire des paramètres constitutifs en confrontant les mesures expérimentales aux modèles numériques, optimisant ainsi la prédiction du comportement des matériaux sous charge.

2 Concepts de base

2.1 Mécanique

L'étude du comportement mécanique des matériaux repose sur l'analyse des contraintes et des déformations qu'ils subissent sous l'action de forces externes. Différentes mesures de la déformation et de la contrainte sont utilisées pour caractériser ces comportements, notamment la **déformation vraie**, la **déformation ingénieur**, la **contrainte vraie** et la **contrainte ingénieur**.

2.1.1 Déformation vraie et déformation ingénieur

La **déformation** est une mesure du changement dimensionnel d'un matériau sous contrainte. Elle peut être exprimée de différentes manières en fonction du cadre d'analyse :

- **Déformation vraie** (ou logarithmique) : Elle prend en compte la variation continue de la longueur et s'exprime sous la forme :

$$\varepsilon_{\text{vraie}} = \ln\left(\frac{L}{L_0}\right) \tag{1}$$

  Contrairement à la déformation ingénieur, elle est plus adaptée aux grandes déformations car elle considère l'accumulation progressive du changement de longueur.

- **Déformation ingénieur** (ou déformation conventionnelle) : Elle est définie comme le rapport entre la variation de longueur et la longueur initiale d'un échantillon soumis à un chargement uniaxial :

$$\varepsilon = \frac{L - L_0}{L_0} = \frac{\Delta L}{L_0} \tag{2}$$

  où L_0 est la longueur initiale et L est la longueur actuelle après déformation.

2.1.2 Contrainte vraie et contrainte ingénieur

La **contrainte** est définie comme la force appliquée par unité de surface. Deux approches sont couramment utilisées :

- **Contrainte vraie** :
$$\sigma_{\text{vraie}} = \frac{F}{A} \tag{3}$$
où A est la section instantanée de l'échantillon sous charge. Elle est plus précise pour les matériaux subissant des déformations importantes.

- **Contrainte ingénieur** (ou contrainte nominale) :
$$\sigma = \frac{F}{A_0} \tag{4}$$
où F est la force appliquée et A_0 est la section initiale de l'échantillon. Cette contrainte ne prend pas en compte la variation de section pendant la déformation.

2.1.3 Courbe force-déplacement

La courbe force-déplacement est un outil fondamental pour analyser le comportement mécanique d'un matériau. Elle est obtenue lors d'un essai mécanique, tel que l'essai de traction, et représente l'évolution de la force appliquée en fonction du déplacement de l'éprouvette Figure (1).

Cette courbe peut être divisée en plusieurs régions caractéristiques :

- Une phase **élastique**, où le matériau revient à sa forme initiale après déchargement.
- Une phase **plastique**, où le matériau subit des déformations permanentes.
- Une phase de **striction** suivie d'une **rupture**, indiquant la fin de la résistance mécanique du matériau.

2.1.4 Champs de déplacement et champs de déformation

Les champs de déplacement et de déformation permettent une analyse plus fine du comportement mécanique d'un matériau.

- **Champs de déplacement** : Ils décrivent le mouvement de chaque point du matériau sous l'effet des forces appliquées. Mathématiquement, un champ de déplacement en 2D est défini par un vecteur $\mathbf{u}(x, y) = (u_x, u_y)$, où u_x et u_y sont les composantes du déplacement selon les axes x et y.

- **Champs de déformation** : Ils dérivent des champs de déplacement et quantifient les variations locales de forme et de volume. Ils sont définis à partir du tenseur des déformations :
$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{5}$$
où ε_{ij} est un terme du tenseur de déformation décrivant la variation relative des distances entre points voisins.

L'analyse de ces champs est cruciale pour mieux comprendre la distribution des efforts mécaniques au sein du matériau et pour valider les modèles numériques basés sur la méthode des éléments finis.

2.2 DIC (Digital Image Correlation)

La **corrélation d'images numériques** (DIC - Digital Image Correlation) est une technique expérimentale de mesure des champs de déplacement et de déformation basée sur l'analyse d'images successives d'un échantillon soumis à une sollicitation mécanique.

2.2.1 Principe de la méthode

La DIC repose sur la comparaison d'images d'un échantillon avant et après déformation. Pour cela, la surface du matériau est recouverte d'un motif aléatoire (souvent sous forme de speckle) qui permet de suivre les déplacements locaux de différentes zones. L'algorithme de corrélation d'images identifie ces déplacements en subdivisant l'image en petites régions appelées **zones d'intérêt** (ou subset). Le déplacement de chaque subset est ensuite déterminé en maximisant une fonction de corrélation entre les images initiale et déformée.

2.2.2 Acquisition et traitement des images

Le processus de mesure par DIC comprend plusieurs étapes :

1. **Préparation de l'échantillon** : Un motif aléatoire est appliqué sur la surface pour assurer un suivi précis des déplacements.
2. **Acquisition des images** : Une caméra haute résolution capture des images avant, pendant et après la sollicitation mécanique Figure (2) (a).
3. **Corrélation des images** : Un logiciel de traitement d'images identifie le déplacement des motifs et calcule les champs de déplacement Figure (2) (b) et (c).
4. **Post-traitement** : Les champs de déplacement sont différenciés pour obtenir les champs de déformation et analyser les comportements mécaniques.

2.2.3 Extraction de la courbe force-déplacement et des champs de déformation

Dans le cadre de l'essai de traction couplé à la DIC, il est possible d'extraire plusieurs informations clés à partir des données collectées :

Courbe force-déplacement La courbe force-déplacement est obtenue en enregistrant la force appliquée sur l'échantillon et en mesurant les déplacements correspondants, à chaque étape de la sollicitation. À chaque instant, le déplacement mesuré par DIC permet de calculer la force correspondante, et ainsi d'obtenir cette courbe. Cette courbe est essentielle pour déterminer le comportement du matériau, en particulier pour observer les zones de plasticité et de rupture.

Champs de déformation Les champs de déformation sont calculés à partir des champs de déplacement obtenus par DIC. En différenciant les déplacements mesurés par rapport aux coordonnées spatiales, on peut obtenir les déformations locales de l'échantillon. Ces déformations peuvent être présentées sous forme de cartes de déformation (ex. : déformation normale, déformation de cisaillement) qui permettent de visualiser et d'analyser les zones de concentration de déformation.

2.2.4 Avantages et limitations

Avantages :

- Technique **sans contact**, évitant toute perturbation de l'échantillon.
- Fournit des **champs de déplacement et de déformation** avec une haute résolution spatiale.
- Applicabilité à une large gamme de matériaux et de sollicitations (traction, compression, flexion...).
- Facilité d'intégration avec la modélisation par éléments finis pour la validation des modèles numériques.

Limitations :

- Sensible aux **conditions d'éclairage** et à la qualité du motif speckle.
- Nécessite un traitement d'images avancé pouvant être **computationnellement coûteux**.
- Précision dépendante de la résolution de la caméra et du niveau de bruit dans les images.

2.2.5 Application à la superposition des champs de déformation

Dans ce projet, la DIC est utilisée pour mesurer les champs de déplacement et de déformation d'échantillons soumis à un essai de traction. Les résultats obtenus sont ensuite comparés aux simulations numériques réalisées par la méthode des éléments finis (FEM). Cette approche permet de valider les modèles numériques et d'optimiser l'identification des paramètres mécaniques des matériaux.

2.3 Éléments finis

La méthode des éléments finis (FEM) est une technique numérique largement utilisée pour résoudre des problèmes complexes en mécanique des matériaux, tels que les déformations et les contraintes sous différentes sollicitations. Dans ce projet, nous utilisons le logiciel **Abaqus**, un outil puissant pour effectuer des simulations numériques de déformations et de contraintes dans des structures soumises à des chargements variés.

2.3.1 Principe de la méthode

La méthode des éléments finis consiste à diviser une structure ou un matériau en un grand nombre d'éléments de forme simple (triangulaires, quadrilatéraux, tétraédriques, etc.). Chaque élément est représenté par un ensemble de variables (déplacements, contraintes, etc.), et les équations qui régissent le comportement de chaque élément sont résolues de manière itérative pour toute la structure. Les solutions de chaque élément sont ensuite assemblées pour obtenir la solution globale du problème.

2.3.2 Simulations avec Abaqus

Dans Abaqus, les matériaux sont définis par leurs propriétés mécaniques (module d'élasticité, coefficient de Poisson, etc.), et des conditions aux limites sont appliquées pour simuler des sollicitations réelles. Abaqus permet de réaliser des simulations de **traction**, de **compression**, ainsi que des analyses thermomécaniques et dynamiques.

2.3.3 Extraction de la courbe force-déplacement et des champs de déformation

À partir des simulations effectuées dans Abaqus, plusieurs informations clés sont extraites :

- **Courbe force-déplacement** : Tout comme dans l'analyse expérimentale, la courbe force-déplacement est générée à partir des résultats numériques en enregistrant la force appliquée et les déplacements correspondants dans le modèle. Cela permet de suivre l'évolution de la déformation du matériau et de prédire son comportement sous différentes charges.
- **Champs de déformation** : Abaqus permet également de générer des cartes de déformation pour chaque étape de la simulation. Ces cartes montrent comment les déformations sont distribuées dans la structure sous l'effet des charges appliquées. Elles permettent de visualiser les zones de concentration de déformation et d'analyser la réponse locale du matériau.

2.3.4 Avantages et limitations

Avantages :

- Précision des résultats grâce à une modélisation fine du problème.
- Flexibilité pour simuler des géométries complexes et des matériaux non homogènes.
- Possibilité d'intégrer des **modèles constitutifs avancés** pour mieux reproduire le comportement du matériau sous différentes conditions.

Limitations :

- Les résultats sont fortement dépendants de la qualité de la **maillage** et de la modélisation.
- La simulation de matériaux complexes (comme les matériaux non linéaires ou hétérogènes) peut nécessiter des modèles constitutifs spécifiques et une calibration précise des paramètres.
- Le coût computationnel peut être élevé, en particulier pour des simulations 3D complexes.

2.3.5 Application à la superposition des champs de déformation

Dans ce projet, les résultats obtenus par la méthode des éléments finis avec Abaqus sont utilisés pour modéliser les comportements mécaniques des matériaux sous sollicitation. Ces résultats sont ensuite comparés avec les données expérimentales obtenues par la technique de corrélation d'images numériques (DIC). L'objectif est de superposer les champs de déformation issus des deux approches pour affiner les paramètres mécaniques du matériau et améliorer la précision de la modélisation numérique.

2.4 Analyse inverse

L'analyse inverse est une méthode puissante qui permet d'identifier des paramètres mécaniques à partir de données expérimentales en utilisant un modèle numérique. Cette approche est particulièrement utile lorsque les propriétés mécaniques d'un matériau ou d'une structure sont inconnues ou difficiles à mesurer directement. L'idée principale de l'analyse inverse est de comparer les résultats expérimentaux avec ceux obtenus par simulation numérique, puis d'ajuster les paramètres du modèle pour minimiser l'écart entre les deux [4].

2.4.1 Principe de l'analyse inverse

Le principe de l'analyse inverse repose sur l'utilisation de modèles numériques (comme ceux obtenus par la méthode des éléments finis) et de données expérimentales (comme celles provenant de la corrélation d'images numériques). L'objectif est de déterminer les paramètres du modèle qui permettent de reproduire les résultats expérimentaux avec un maximum de précision.

L'idée est de résoudre un problème d'optimisation dans lequel les paramètres du modèle sont ajustés jusqu'à ce que la différence entre les résultats numériques et expérimentaux soit minimisée. Cette méthode est souvent appliquée pour estimer des paramètres tels que le module d'élasticité, le coefficient de Poisson, les lois de durcissement plastique, etc.

2.4.2 Méthodologie

L'analyse inverse suit généralement les étapes suivantes :

- **Collecte des données expérimentales** : Les données expérimentales nécessaires sont obtenues à partir des mesures de déformation et de force (par exemple, via la corrélation d'images numériques (DIC) et la courbe force-déplacement).
- **Modélisation numérique** : Un modèle numérique est construit, souvent à l'aide de la méthode des éléments finis (FEM) pour simuler le comportement du matériau sous les mêmes conditions que celles expérimentées.
- **Optimisation des paramètres** : Un algorithme d'optimisation est utilisé pour ajuster les paramètres du modèle numérique afin de minimiser la différence entre les résultats expérimentaux et numériques.
- **Validation du modèle** : Une fois les paramètres optimisés, le modèle est validé en comparant les prédictions avec des données supplémentaires ou en effectuant des tests expérimentaux supplémentaires.

2.4.3 Algorithmes utilisés

Dans ce projet, l'algorithme de Nelder-Mead est utilisé pour effectuer l'optimisation dans le cadre de l'analyse inverse. Il s'agit d'une méthode d'optimisation sans dérivées, particulièrement adaptée aux problèmes non linéaires et lorsque les gradients des fonctions à optimiser ne sont pas disponibles ou sont difficiles à calculer. Cet algorithme est basé sur l'idée de faire évoluer un simplex (un ensemble de points dans l'espace des paramètres) en fonction des évaluations des valeurs de la fonction objectif pour les points du simplex.

L'algorithme de Nelder-Mead est particulièrement utile dans ce contexte car il est relativement simple à mettre en œuvre et peut être efficace même pour des problèmes d'optimisation complexes, comme ceux rencontrés dans l'analyse inverse des champs de déformation.

2.4.4 Application à la superposition des champs de déformation

Dans ce projet, l'analyse inverse est utilisée pour déterminer les paramètres mécaniques des matériaux à partir des données expérimentales obtenues par DIC et des simulations réalisées avec la méthode des éléments finis (FEM). Plus précisément, les champs de déformation obtenus par DIC et FEM sont comparés et superposés pour identifier les paramètres qui permettent de reproduire les observations expérimentales.

Une fois les paramètres ajustés par l'analyse inverse, le modèle numérique est utilisé pour simuler le comportement du matériau dans des conditions différentes de celles expérimentées, permettant ainsi de prédire son comportement sous diverses sollicitations. Cette méthode permet non seulement de valider les résultats des simulations, mais aussi d'affiner les modèles constitutifs utilisés dans les simulations pour obtenir des prédictions plus précises.

3 Protocole expérimental

3.1 Matériau

Dans cette étude, le matériau utilisé est l'Aluminium 7075-T6, un alliage d'aluminium couramment employé dans des applications où la résistance mécanique est primordiale, comme dans l'aéronautique et l'automobile. Les propriétés mécaniques de ce matériau sont les suivantes :

- **Module de Young** $E = 71,7$ GPa
- **Limite élastique** $\sigma_y = 503$ MPa
- **Module de Poisson** $\nu = 0,33$

Ces propriétés caractérisent le comportement mécanique de l'Aluminium 7075-T6 dans la zone élastique et sont essentielles pour modéliser les déformations et les contraintes dans le cadre de l'analyse des champs de déformation.

3.2 Loi de comportement : modèle de Johnson-Cook

Pour modéliser le comportement plastique du matériau, nous utilisons la loi de Johnson-Cook, qui est une loi de comportement largement employée pour les matériaux métalliques soumis à des sollicitations dynamiques. Elle s'exprime sous la forme générale :

$$\sigma = (A + B\epsilon^n)\,(1 + C\ln\dot{\epsilon})\left(1 - \left(\frac{T - T_0}{T_m - T_0}\right)^m\right) \tag{6}$$

où :

- σ est la contrainte d'écoulement,
- A est la limite d'élasticité à froid,
- B et n sont des paramètres liés à l'écrouissage,
- C est un paramètre de sensibilité à la vitesse de déformation,
- m est un paramètre lié à la dépendance thermique du matériau,
- ϵ est la déformation plastique,
- $\dot{\epsilon}$ est la vitesse de déformation normalisée,
- T est la température instantanée, T_0 est la température ambiante et T_m est la température de fusion du matériau.

Dans cette étude, nous adoptons une version simplifiée de la loi de Johnson-Cook en ne considérant que la contribution de l'écrouissage, c'est-à-dire :

$$\sigma = A + B\epsilon^n \tag{7}$$

Cette simplification est justifiée par le fait que nous travaillons à des vitesses de déformation faibles et à température ambiante, rendant négligeables les effets de sensibilité à la vitesse de déformation et de dépendance thermique. Cette approximation permet de réduire la complexité du modèle tout en conservant une bonne précision pour les applications envisagées.

3.3 Dispositif expérimental

L'expérimentation repose sur un ensemble d'équipements permettant de caractériser le comportement mécanique du matériau étudié. Le dispositif comprend :

- **Machine de traction** : Une machine de traction équipée d'un capteur de force est utilisée pour appliquer un chargement contrôlé sur l'éprouvette. Elle permet de mesurer la courbe force-déplacement et de déterminer les propriétés mécaniques du matériau, notamment la limite d'élasticité et l'écrouissage.
- **Éprouvette** : L'éprouvette est conçue selon les normes de traction afin de garantir des résultats reproductibles. Sa géométrie est adaptée pour assurer une distribution homogène des contraintes et minimiser les effets de concentration.
- **Capteurs de force et de déplacement** : Des capteurs de force haute précision sont intégrés à la machine de traction pour enregistrer l'évolution de la charge appliquée. Un extensomètre ou un capteur de déplacement est utilisé pour mesurer l'allongement de l'éprouvette.

- **Caméra haute résolution** : Une caméra numérique haute résolution capture des images de la surface de l'éprouvette avant et après déformation. Elle est utilisée dans le cadre de la corrélation d'images numériques (DIC) pour mesurer les champs de déplacement et de déformation.

- **Système d'éclairage** : Un éclairage homogène est mis en place pour optimiser la qualité des images acquises par la caméra et éviter les ombres ou reflets pouvant perturber l'analyse DIC.

- **Système DIC (Digital Image Correlation)** : Un logiciel de corrélation d'images numériques est utilisé pour analyser les images enregistrées par la caméra. Cette technique permet d'obtenir les champs de déplacement 2D ou 3D avec une grande précision.

- **Marquage de la surface** : Pour améliorer la précision de la DIC, la surface de l'éprouvette est recouverte d'un motif aléatoire (spray noir et blanc) afin de faciliter le suivi des déplacements de chaque point de l'échantillon.

- **Ordinateur et logiciel d'acquisition** : Un ordinateur équipé d'un logiciel de pilotage de la machine de traction et d'un système d'acquisition de données enregistre et synchronise les mesures de force, de déplacement et les images capturées.

L'ensemble de ces équipements permet une analyse complète du comportement mécanique du matériau en combinant des mesures globales (courbe force-déplacement) et locales (champs de déformation obtenus via DIC).

4 Superposition des données DIC / FEM

4.1 Données FEM / DIC

Les données utilisées dans cette étude proviennent de deux sources complémentaires : la corrélation d'images numériques (DIC) et la simulation par éléments finis (FEM). Cependant, ces deux méthodes produisent des données dans des référentiels et avec des résolutions différentes, ce qui nécessite un traitement particulier pour assurer leur compatibilité.

Nous avons les courbes force-déplacement DIC Figure (6) et FEM Figure (4) ainsi que les champs de déformation DIC Figure (5) et FEM Figure (3) à traiter.

Différences d'échelle et d'unités : Les données issues du DIC sont exprimées en pixels, tandis que les résultats des simulations FEM sont directement en millimètres. Une conversion entre ces deux référentiels est donc nécessaire afin de superposer correctement les champs de déplacement et de déformation. Cette étape implique un recalage spatial basé sur les dimensions physiques de l'éprouvette et la résolution des images acquises.

Perte progressive des données DIC : Au fur et à mesure de l'essai, des zones de l'échantillon peuvent sortir du champ de vision ou perdre leur texture détectable, entraînant une diminution progressive de la quantité de données exploitables par le DIC. Ce manque d'informations peut être problématique pour l'analyse des champs de déformation, en particulier dans les zones où les concentrations de contrainte sont importantes.

Nécessité d'un traitement adapté : Pour pallier ces limitations, un ensemble de traitements spécifiques est mis en place afin de reconstituer les données manquantes et d'assurer une comparaison cohérente entre FEM et DIC. Ces aspects sont détaillés dans la section suivante dédiée au protocole de traitement des données.

4.2 Prétraitement des données DIC

Les données DIC sont initialement exprimées en pixels avec une origine située en haut à gauche et un axe y orienté vers le bas. En revanche, les données FEM sont exprimées en millimètres avec une origine en bas à gauche et un axe y orienté vers le haut. Afin d'assurer la cohérence entre ces deux référentiels, plusieurs transformations sont appliquées :

- **Conversion des unités** : Les déplacements et positions DIC sont convertis en millimètres en utilisant un facteur d'échelle basé sur la résolution de l'image et les dimensions connues de l'éprouvette.
- **Recentrage de l'origine** : L'éprouvette étant initialement située au centre de l'image, une translation est appliquée pour repositionner son origine en bas à gauche.
- **Changement d'orientation des axes** : L'axe y du DIC est inversé pour correspondre à la convention utilisée dans les simulations FEM.

Ces transformations garantissent que les champs de déplacement et de déformation DIC sont correctement alignés avec ceux obtenus via FEM.

4.3 Interpolation spatiale des données DIC

Les données DIC peuvent présenter des zones manquantes au fur et à mesure de l'essai en raison d'un suivi imparfait du motif ou de la sortie de certaines zones du champ de vision. Pour pallier cette perte d'information, une interpolation spatiale est réalisée afin de reconstruire les champs de déplacement et de déformation sur l'ensemble de la surface de l'éprouvette.

4.4 Interpolation temporelle sur la grille de temps FEM

Les simulations Abaqus fournissent les résultats à des instants spécifiques définis par une grille de temps discrète uniforme. En revanche, les mesures DIC sont généralement acquises à une fréquence constante indépendante de cette grille. Pour assurer une comparaison cohérente entre les deux ensembles de données, les valeurs DIC sont interpolées temporellement sur les instants de sortie des résultats FEM.

4.5 Interpolation spatiale des données FEM sur la grille expérimentale

Les simulations FEM sont effectuées sur une grille relativement grossière afin de réduire les temps de calcul et tester la faisabilité de la méthode. En revanche, la grille expérimentale DIC est plus fine, offrant une meilleure résolution spatiale des champs mesurés. Afin de comparer correctement les deux ensembles de données, les champs issus des simulations FEM sont interpolés spatialement sur la grille déformée DIC.

Cette étape permet d'obtenir une correspondance point par point entre les champs DIC et FEM, facilitant ainsi l'analyse et l'identification des paramètres mécaniques.

4.6 Calcul de la fonction coût

La fonction coût permet de quantifier l'écart entre les champs de déformation obtenus expérimentalement par la DIC et ceux issus des simulations par éléments finis (FEM). Nous adoptons l'approche proposée par E. Roux [1], où la fonction coût est définie comme suit :

$$f_c = \sqrt{\frac{\sum_{i=1}^{N} [(u_i^{\mathrm{DIC}} - u_i^{\mathrm{FEM}})^2 + (v_i^{\mathrm{DIC}} - v_i^{\mathrm{FEM}})^2]}{\sum_{i=1}^{N} [(u_i^{\mathrm{DIC}})^2 + (v_i^{\mathrm{DIC}})^2]}} \tag{8}$$

où :

- N est le nombre de points de mesure.
- $u_i^{\mathrm{DIC}}, v_i^{\mathrm{DIC}}$ sont les composantes du champ de déplacement mesurées expérimentalement.
- $u_i^{\mathrm{FEM}}, v_i^{\mathrm{FEM}}$ sont les composantes du champ de déplacement issues de la simulation numérique.

Cette fonction normalisée permet d'évaluer la qualité de la correspondance entre les deux sources de données. Une valeur faible de f_c indique une bonne adéquation entre l'expérience et la simulation, tandis qu'une valeur élevée suggère un ajustement nécessaire des paramètres du modèle.

5 Résultats et conclusions

5.1 Résultats

L'analyse des champs de déplacement obtenus expérimentalement et numériquement met en évidence une bonne corrélation entre les deux méthodes. La superposition des champs DIC et FEM Figure (7) montre que les tendances générales des déformations sont bien capturées par la simulation, bien que des écarts subsistent localement.

En affinant l'analyse sur la zone utile de l'éprouvette Figure (8), on observe que les écarts sont principalement localisés aux bords, où les effets de bord et les incertitudes expérimentales influencent les résultats. Ces différences peuvent être attribuées aux limites de la résolution DIC et à la simplification des conditions aux limites dans la simulation par éléments finis.

Malgré ces écarts, la méthode proposée permet de comparer efficacement les champs expérimentaux et numériques, et pourra être améliorée en ajustant les paramètres du modèle ou en affinant le maillage FEM.

5.2 Conclusions

5.2.1 Résumé des Travaux

Les travaux réalisés dans ce projet ont permis d'exploiter différentes approches expérimentales et numériques pour l'analyse des champs de déformation :

- Utilisation de la corrélation d'images numériques (DIC) pour mesurer les champs de déplacement et de déformation.

- Superposition des champs de déformation issus des mesures expérimentales et des simulations par éléments finis (FEM).
- Développement d'un code Python basé sur la méthode de Nelder-Mead pour l'identification des paramètres mécaniques à partir d'une approche d'analyse inverse.

5.2.2 Principaux Résultats

Les résultats obtenus ont permis de mettre en évidence la pertinence de l'approche adoptée :

- Mise en place d'un dialogue entre les données expérimentales (DIC) et les simulations numériques (FEM).
- Calcul de la fonction coût pour quantifier les écarts entre les résultats expérimentaux et les modèles numériques.
- Validation du code Python d'optimisation avec trois itérations de la méthode de Nelder-Mead.

5.2.3 Perspectives Futures

Plusieurs axes d'amélioration et d'extension peuvent être envisagés pour la suite des travaux :

- Optimisation plus poussée afin de minimiser la fonction coût et d'améliorer la précision de l'identification des paramètres.
- Intégration de données thermiques dans l'analyse pour une approche multi-physique combinant aspects mécaniques et thermiques.
- Extension de la méthodologie à d'autres types de matériaux et à différentes conditions d'essais mécaniques.

6 Références

References

[1] Roux, E. *Assemblage Mécanique : Stratégies d'optimisation des procédés et d'identification des comportements mécaniques des matériaux.* Thèse de doctorat, École nationale supérieure des mines de Paris, 2011.

[2] Pottier, T. *Identification paramétrique par recalage de modèles éléments finis couplée à des mesures de champs cinématiques et thermiques.* Thèse de doctorat, Université de Savoie, 2010.

[3] Robert, L., Velay, V., Decultot, N., & Ramde, S. *Identification of hardening parameters using finite element models and full-field measurements: some case studies.* Journal of Strain Analysis for Engineering Design, 47(1), 3–17, (2012). `https://doi.org/10.1177/0309324711430022`

[4] Martins, J. M. P., Andrade-Campos, A., & Thuillier, S. *Comparison of inverse identification strategies for constitutive mechanical models using full-field measurements*. International Journal of Mechanical Sciences, 145, 330–345, (2018).

7 Remerciements

Nous tenons à remercier nos encadrants, Pierre-oliver Bouchard et Guillome Corvec, pour leur soutien, leurs conseils et le temps agréable passé à travailler sur ce projet. Nous remercions également les techniciens et les membres du laboratoire pour leur aide précieuse dans la réalisation des essais expérimentaux.

8 Annexes

8.1 Code

Voici le code Python utilisé pour l'analyse des données :

```python
from abaqus import *
from abaqusConstants import *
from odbAccess import openOdb
import numpy as np


def get_abaqus_time(odb_name):
    # open ODB file
    odb = openOdb(path=odb_name)

    # get the step1 and its data
    last_step = list(odb.steps.keys())[-1] # setp1
    step1_frames = odb.steps[last_step].frames # data of step1

    abqtime = []

    for frame in step1_frames:
        abqtime.append(frame.frameValue)

    return abqtime

if __name__ == '__main__':
    time_abaqus = get_abaqus_time('E:/2023_2025/M2/Projet/Abaqus_simulation/traction.odb')
    output_path = "E:/2023_2025/M2/Projet/Abaqus_simulation/time_abaqus.csv"
    np.savetxt(output_path, time_abaqus, fmt="%.3f", delimiter=",", header="time", comments="")
    print(time_abaqus)
```

Listing 1: Code Python pour récupérer la grille de temps Abaqus

8.2 Code

Voici le code Python utilisé pour l'analyse des données :

```python
from abaqus import *
from abaqusConstants import *
from odbAccess import openOdb
#import matplotlib.pyplot as plt
import numpy as np


# open ODB file
odb_path = r'E:\\2023_2025\\M2\\Projet\\Abaqus_simulation\\traction.odb'
odb = openOdb(path=odb_path)
print(f"ODB opened: {odb}")

# get the step1 and its data
last_step = list(odb.steps.keys())[-1] # setp1
step1_frames = odb.steps[last_step].frames # data of step1

```

```
# difine initial lists to save displacement field
U2 = []
RF2 = []

assembly = odb.rootAssembly
region_name = 'SET-RP'  # replace set name
region = assembly.nodeSets[region_name]


for frame in step1_frames:
    # get reaction force RF and displacement U data
    force = 0.0
    disp = 0.0

    # get RF2
    RF2_field = frame.fieldOutputs['RF'] # get reaction force RF data
    RF2_values = RF2_field.getSubset(region=region).values # get
   reaction force RF values
    for val in RF2_values:
        force += val.data[1]  # index of RF: [0]=X, [1]=Y, [2]=Z

    # get U2
    U2_field = frame.fieldOutputs['U']  # get displacement U data
    U2_values = U2_field.getSubset(region=region).values # get
   displacement U values
    for val in U2_values:
        disp += val.data[1]  # index of U same like before

    # save present Frame data
    RF2.append(force)
    U2.append(disp)

# close ODB field..
odb.close()

FD = np.zeros((len(U2),2))
FD[:,0] = U2
FD[:,1] = RF2
# plt.figure()
# plt.plot(U2, RF2, label = 'FD')
# plt.title('force-displacemnt')
# plt.xlabel('displacemnt')
# plt.ylabel('force')
# plt.legend()
# plt.show()


output_path = r"E:\\2023_2025\\M2\\Projet\\Abaqus_simulation\\
   numeric_data\\force_displacement\\force_displacement.csv"

# save as a csv file
np.savetxt(output_path, FD, fmt="%.3f", delimiter=",",
           header="displacement, force", comments="")
# FX and FY represent the deformation field
print(f"Data saved to {output_path}")


# from abaqus import *
```

```
# from abaqusConstants import *
# from odbAccess import openOdb

# odb_path = r'E:\\2023_2025\\M2\\Projet\Abaqus_simulation\\traction.odb'
# odb = openOdb(path=odb_path)
# print(f"ODB opened: {odb}")
# assembly = odb.rootAssembly

# node_set_names = [node_set.name for node_set in assembly.nodeSets.values()]
# print("Available node set names:", node_set_names)

# odb.close()
```

Listing 2: Extraction force-déplacement numérique

```
import os
from abaqus import *
from abaqusConstants import *

N_folder = 2
#  A_fit, B_fit, n_fit are optimized
A_fit = 500
B_fit = 300
n_fit = 0.3

# set CAE file path
cae_file_path = 'E:/2023_2025/M2/Projet/Abaqus_simulation/simulation_traction.cae'
# open CAE file
openMdb(pathName = cae_file_path)

#  Modifier les parametres du material
# choose the modele
model_name = "Model-1"
material_name = "Alu-7075"

# open Modele and Materiele
model = mdb.models[model_name]
material = model.materials[material_name]

# update Johnson-Cook parameters
material.Plastic(hardening=JOHNSON_COOK, table=((A_fit, B_fit, n_fit),))

folder_path = []
for i in range(1, N_folder+1):
    folder_name = f'Job_opt{i}'
    folder_path = f'E:/2023_2025/M2/Projet/Abaqus_simulation/Jobs_opt/{folder_name}'

    # creat a folder if it doesn't exist
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
        print(f"folder '{folder_path}' created")
    else:
        print(f"folder '{folder_path}' exists before")

    # creer ou mise a jour le job
```

```
    job_name = f"Optimized_Job{i}"

    # si job optimise existe, remplace le
    if job_name in mdb.jobs.keys():
        del mdb.jobs[job_name]

    # set Job path
    os.chdir(folder_path)

    # creer nouveau job
    mdb.Job(name=job_name, model=model_name)

    # submit job
    mdb.jobs[job_name].writeInput()
    mdb.jobs[job_name].submit(consistencyChecking=OFF)

    # wait for the job complet
    mdb.jobs[job_name].waitForCompletion()

    cae_name = f"simulation_traction{i}"
    # savgarder du modele
    mdb.saveAs(os.path.join(folder_path, cae_name))
```

Listing 3: Code Python pour l'automatisation des simulations Abaqus

```
from abaqus import *
from abaqusConstants import *
from odbAccess import openOdb
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.optimize import differential_evolution
import numpy as np
import os


root_path = r"E:\\2023_2025\\M2\\Projet"
# CAE file path
cae_file_path = os.path.join(root_path, "Abaqus_simulation","
    simulation_traction.cae")
# open CAE file
openMdb(pathName = cae_file_path)


def abaqus_FD(JC_params):
    # ODB file path
    odb_path = os.path.join(root_path, "Abaqus_simulation", "traction.
    odb")

    # CAE file path
    cae_file_path = os.path.join(root_path, "Abaqus_simulation","
    simulation_traction.cae")
    # open CAE file
    openMdb(pathName = cae_file_path)

    # get elastic and Johnson-Cook paramamters

    #E, v, A, B, n = JC_params
    A, B, n = JC_params

```

```
    # updata paramaters
    model_name = "Model-1"
    material_name = "Alu-7075"

    # open Modele and Materiele
    model = mdb.models[model_name]
    material = model.materials[material_name]
    #material.Elastic(table=((E, v),))
    material.Plastic(hardening=JOHNSON_COOK, table=((A, B, n),))

    # save ODB file and submit the job
    os.chdir(r"E:\\2023_2025\\M2\\Projet\\Abaqus_simulation")
    job_name = 'traction'
    mdb.Job(name=job_name, model=model_name)
    mdb.jobs[job_name].submit(consistencyChecking=OFF)
    mdb.jobs[job_name].waitForCompletion()

    cae_name = "simulation_traction"
    # savgarder du modele
    mdb.saveAs(os.path.join(root_path, "Abaqus_simulation", cae_name))

    # open ODB file
    odb = openOdb(path=odb_path)
    print(f"ODB_traction_yy opened: {odb}")

    # get the step1 and its data
    last_step = list(odb.steps.keys())[-1] # setp1
    step1_frames = odb.steps[last_step].frames # data of step1

    # difine initial lists to save displacement field
    U2 = []
    RF2 = []

    assembly = odb.rootAssembly
    region_name = 'SET-RP'  # replace set name
    region = assembly.nodeSets[region_name]


    for frame in step1_frames:
        # get reaction force RF and displacement U data
        force = 0.0
        disp = 0.0

        # get RF2
        RF2_field = frame.fieldOutputs['RF'] # get reaction force RF
  data
        RF2_values = RF2_field.getSubset(region=region).values # get
  reaction force RF values
        for val in RF2_values:
            force += val.data[1]  # index of RF: [0]=X, [1]=Y, [2]=Z

        # get U2
        U2_field = frame.fieldOutputs['U']  # get displacement U data
        U2_values = U2_field.getSubset(region=region).values # get
  displacement U values
        for val in U2_values:
            disp += val.data[1]  # index of U same like before

```

```
        # save present Frame data
        RF2.append(force)
        U2.append(disp)

    # close ODB field..
    odb.close()

    # FD = np.zeros((len(U2),2))
    # FD[:,0] = U2
    # FD[:,1] = RF2
    return np.array(RF2), np.array(U2)#, FD


#
    ##############################################################################


#
    ##############################################################################


# load experimental data
exp_path = os.path.join(root_path, "FD_exp.csv")
data_exp = np.genfromtxt(exp_path, delimiter=",", skip_header=1)

# define numeric and experimental force and displacement

force_exp, displacement_exp = data_exp[:, 1], data_exp[:, 0]

# define interpolate domain
N = len(displacement_exp)*1 # number of usual points
common_displacement = np.linspace(min(displacement_exp), max(
    displacement_exp), N)


# difine cost function
def cost_fct(JC_params):

    try:
        force_num, displacement_num = abaqus_FD(JC_params)
    except Exception as e:
        print(f"Error during abaqus_FD: {e}")
        return float('inf')  # return infinit value

    # check NaN value
    if np.any(np.isnan(force_num)) or np.any(np.isnan(displacement_num))
    :
        print("NaN encountered in numerical results")
        return float('inf')

    #force_num, displacement_num = abaqus_FD(JC_params)

    exp_force_interp = np.interp(common_displacement, displacement_exp,
    force_exp)
    num_force_interp = np.interp(common_displacement, displacement_num,
    force_num)

```

```
    # normalized error
    denominator = np.sum(exp_force_interp ** 2)
    if denominator == 0:
        raise ValueError("Experimental force data sum is zero, normalization is invalid.")
    err = np.sum((exp_force_interp - num_force_interp) ** 2) / denominator
    return err


JCparameters = []

JCparams = [450, 270, 0.3]
#JCparams = [71000, 0.3, 450, 270, 0.3]

# optimize iterations
nb_iter = 1
for i in range(1, nb_iter+1):
    print(f" A = {JCparams[0]}, B = {JCparams[1]}, n = {JCparams[2]}")
    #print(f"E = {JCparams[0]}, v = {JCparams[1]},  A = {JCparams[2]}, B = {JCparams[3]}, n = {JCparams[4]}")

    # optimization

    #bound = [(50000,71000), (0.15, 0.39), (300, 650), (200, 600), (0.1, 0.7)]
    bound = [(300, 650), (200, 600), (0.1, 0.7)]
    result = minimize(cost_fct, JCparams, method='Nelder-Mead', bounds=bound, options={'maxiter': 1})#, 'disp': True})
    #result = differential_evolution(cost_fct,  bounds=bound, maxiter=1)
    print("optimized Johnson Cook model:", result)

    # updata parameters
    JCparams = result.x

    JCparameters.append(list(JCparams) + [result.fun])

print(f"Johnson Cook parameters and errors{JCparameters}")

params_path = os.path.join(root_path, "Abaqus_simulation", "numeric_data", "JC_parameters.csv")

# save as a csv file
np.savetxt(params_path, JCparameters, fmt="%.3f", delimiter=",",
           header="Young's Module, Poisson coeff, A,B,n, error", comments="")
# FX and FY represent the deformation field
print(f"Data saved to {params_path}")

#
    ##########################################################################################


#
    ##########################################################################################


# force_num, displacement_num = abaqus_FD(JCparams)
```

```
# num_force_interp = np.interp(common_displacement, displacement_num, force_num)

# plt.figure(figsize=(10, 6))
# plt.plot(displacement_exp, force_exp, label="Experimental", color="blue", linestyle='--')
# plt.plot(num_force_interp, force_num, label="Numerical", color="red")
# plt.xlabel("Displacement")
# plt.ylabel("Force")
# plt.legend()
# plt.title(f"Comparison of Experimental and Numerical Results (Iter {i})")
# plt.grid()
# plt.savefig(os.path.join(root_path, "Abaqus_simulation", "plots", f"comparison_iter_{i}.png"))
# plt.close()
```

Listing 4: Optimisation force-déplacement

```
from abaqus import *
from abaqusConstants import *
from odbAccess import openOdb
import job
import os
import shutil
import numpy as np

# extraire les champs numeric pour tout les neuds au cours du temps

# open ODB file
odb_path = 'E:/2023_2025/M2/Projet/Abaqus_simulation/traction.odb'
odb = openOdb(path=odb_path)
print(f"ODB opened: {odb}")

# get the step1 and the initial time
last_step = odb.steps.keys()[-1] # setp1
first_frame = odb.steps[last_step].frames[0] # initial time

num_frames = len(odb.steps[last_step].frames)
print(f"Total frames: {num_frames}")


for i in range(num_frames):

    frame = odb.steps[last_step].frames[i]
    # get force and displacement data
    displacement_field = frame.fieldOutputs['U']

    # # get instance
    instance_name = list(odb.rootAssembly.instances.keys())[0]
    instance = odb.rootAssembly.instances[instance_name]

    # difine initial lists to save displacement field
    node_labels = []
    x_coords = []
    y_coords = []
    xdisplacements = []
    ydisplacements = []

```

```
41
42      # append datas
43      for value in displacement_field.values:
44
45          node_label = value.nodeLabel  # node number
46          node_labels.append(node_label)
47          node = instance.nodes[node_label - 1] # get instance nodes
48
49          x_coords.append(node.coordinates[0])  # X coord
50          y_coords.append(node.coordinates[1])  # Y coord
51          xdisplacements.append(value.data[0])  # X displacement
52          ydisplacements.append(value.data[1])
53
54      # Create a numpy array to store the displacement field
55      disp_field = np.zeros((len(node_labels), 7))
56      disp_field[:,0] = node_labels
57      disp_field[:,1] = x_coords
58      disp_field[:,2] = y_coords
59      disp_field[:,3] = xdisplacements
60      disp_field[:,4] = ydisplacements
61      # defomed coord.
62      disp_field[:,5] = disp_field[:,1] + disp_field[:,3]
63      disp_field[:,6] = disp_field[:,2] + disp_field[:,4]
64
65
66
67      # if the file already exists, automatically back it up
68      output_path = f"E:/2023_2025/M2/Projet/Abaqus_simulation/
    numeric_data/champs_deformation/deformation_data_{i}.csv"
69
70      if not os.path.exists(output_path):
71          base_dir = "E:/2023_2025/M2/Projet/Abaqus_simulation/
    numeric_data/champs_deformation"
72          os.makedirs(base_dir, exist_ok=True)
73
74      # save as a csv file
75      np.savetxt(output_path, disp_field, fmt="%.3f", delimiter=",",
76              header="NodeLabel,X,Y,UX,UY, defmX, defmY", comments="")
77      # FX and FY represent the deformation field
78      print(f"Data saved to {output_path}")
79
80  # # close ODB field
81  odb.close()
```

Listing 5: Extraction des champs numériques

```
1   import numpy as np
2   import os
3   import matplotlib.pyplot as plt
4
5   # initial orgin and ratio number
6   ORIGIN_X = 354
7   ORIGIN_Y = 1522
8   PIXEL_TO_MM = 0.0694078947368421
9
10  def load_and_process_data(i):
11      # file path
12      file_path = f"E:/2023_2025/M2/Projet/projet_data/Essai_traction/
    data_traction/alu_7075/image-00000{i:03}_0.csv"
```

```
13
14      # check the existence of files
15      if not os.path.exists(file_path):
16          print(f"File not found: {file_path}")
17          return None
18
19      # read 8 and 9 colonnes
20      data = np.loadtxt(file_path, delimiter=",", skiprows=1, usecols=(7,
        8))
21
22      # change coordinate system
23      data[:, 0] -= ORIGIN_X  # X tanslation
24      data[:, 1] -= ORIGIN_Y  # Y translation
25
26      # pixels to mm
27      data *= PIXEL_TO_MM
28
29      coord_label = np.linspace(1, len(data[:,0]), len(data[:,0]))
30      def_data = np.zeros((len(data[:,0]), 3))
31      def_data[:,0] = coord_label
32      def_data[:,1] = data[:,0]
33      def_data[:,2] = data[:,1]
34
35      return data
36
37  def save_data(data, i):
38      # save new data
39      output_path = f"E:/2023_2025/M2/Projet/Abaqus_simulation/
        experimental_data/exp_deformation_{i:03}.csv"
40      np.savetxt(output_path, data, fmt="%.3f", delimiter=",", header="X,
        Y", comments="")
41      print(f"Data saved to {output_path}")
42
43  # main program
44  if __name__ == "__main__":
45      data_exp = []  # initialize the list
46      for i in range(0, 289):
47          data = load_and_process_data(i)
48          if data is not None:
49              data_exp.append(data)  # save data in the list
50              save_data(data, i)
51
52
53  data_0 = load_and_process_data(0)
54  data_288 = load_and_process_data(288)
55  plt.figure()
56  plt.scatter(data_0[:,0], data_0[:,1], color = 'r')
57  plt.scatter(data_288[:,0], data_288[:,1], color = 'b')
58  plt.xlabel("X")
59  plt.ylabel("Y")
60  plt.legend()
61  plt.show()
```

Listing 6: Convertion des données expérimentales

```
1  from abaqus import *
2  from abaqusConstants import *
3  from odbAccess import openOdb
4  import numpy as np
```

```
5
6
7  def get_abaqus_time(odb_name):
8      # open ODB file
9      odb = openOdb(path=odb_name)
10
11     # get the step1 and its data
12     last_step = list(odb.steps.keys())[-1] # setp1
13     step1_frames = odb.steps[last_step].frames # data of step1
14
15     abqtime = []
16
17     for frame in step1_frames:
18         abqtime.append(frame.frameValue)
19
20     return abqtime
21
22 if __name__ == '__main__':
23     time_abaqus = get_abaqus_time('E:/2023_2025/M2/Projet/
       Abaqus_simulation/traction.odb')
24     output_path = "E:/2023_2025/M2/Projet/Abaqus_simulation/time_abaqus.
       csv"
25     np.savetxt(output_path, time_abaqus, fmt="%.3f", delimiter=",",
       header="time", comments="")
26     print(time_abaqus)
```

Listing 7: Code Python pour récupérer la grille de temps Abaqus

```
1  import numpy as np
2  from scipy.interpolate import interp1d
3  import os
4
5
6  # files path definition
7  input_folder = "E:/2023_2025/M2/Projet/Abaqus_simulation/
       experimental_data"
8  output_folder = "E:/2023_2025/M2/Projet/Abaqus_simulation/
       interpolated_exp_data"
9  os.makedirs(output_folder, exist_ok=True)
10
11 def load_exp_data(i):
12     data_path = os.path.join(input_folder, f"exp_deformation_{i:03}.csv"
       )
13
14     if not os.path.exists(data_path):
15         print(f"File not found: {data_path}")
16         return None
17
18     data = np.loadtxt(data_path, delimiter=",", skiprows=1)
19
20     return data
21
22
23 def spatial_interp(i, N):
24
25     data = load_exp_data(i)
26     if data is None:
27         print(f"None file i={i}")
28         return None
```

```
    x = data[:, 1]
    y = data[:, 2]
    M = len(x)

    dx = np.diff(x)
    dy = np.diff(y)
    ds = np.sqrt(dx**2 + dy**2)
    t = np.zeros(M)
    t[1:] = np.cumsum(ds)
    t /= t[-1]      # normalize t

    t_new = np.linspace(0, 1, N)

    # interpolate for x and y
    f_x = interp1d(t, x, kind='cubic', fill_value="extrapolate")
    f_y = interp1d(t, y, kind='cubic', fill_value="extrapolate")
    x_new = f_x(t_new)
    y_new = f_y(t_new)

    return np.column_stack((x_new, y_new))


# Load original time for each file
time_original = np.loadtxt("E:/2023_2025/M2/Projet/Abaqus_simulation/
    time_abaqus.csv", skiprows=1)
# Uniform time nodes
N = 290  # Number of time nodes
time_uniform = np.linspace(time_original[0], time_original[-1], N)


# Load and organize data by time step
data_dict = {}
N_points = 8000

for i, t in enumerate(time_original):
    data = spatial_interp(i, N_points)
    if data is not None:  # Filter invalid data
        data_dict[t] = data

# Check for valid data
if not data_dict:
    raise ValueError("Error: data_dict is empty. Check experimental data
    files")

# Calculate maximum points (from valid data)
max_points = max(len(data) for data in data_dict.values())

# Initialize storage for interpolated data
interpolated_data = []

# Perform interpolation for each point across all time steps
for point_index in range(max_points):  # Using the correct range
    x_coords = []
    y_coords = []
    times_with_data = []

    # Collect x, y, and corresponding time for the current point
    for t, data in data_dict.items():
```

```
        if point_index < len(data):  # Only consider points that exist at this time step
            x_coords.append(data[point_index, 0])
            y_coords.append(data[point_index, 1])
            times_with_data.append(t)

    # If there are enough data points to interpolate, perform the interpolation
    if len(times_with_data) > 1:
        x_interp = interp1d(times_with_data, x_coords, kind='cubic', fill_value="extrapolate")(time_uniform)
        y_interp = interp1d(times_with_data, y_coords, kind='cubic', fill_value="extrapolate")(time_uniform)
    else:
        # If not enough data points, fill with NaN
        x_interp = np.full_like(time_uniform, np.nan)
        y_interp = np.full_like(time_uniform, np.nan)

    interpolated_data.append(np.column_stack((x_interp, y_interp)))

# Save results as CSV files
for i, t in enumerate(time_uniform):
    output_data = np.zeros((len(interpolated_data), 3))
    for j in range(len(interpolated_data)):
        output_data[j, 0] = j + 1  # Point number
        output_data[j, 1] = interpolated_data[j][i, 0]  # Interpolated x
        output_data[j, 2] = interpolated_data[j][i, 1]  # Interpolated y


    output_path = os.path.join(output_folder, f"image_{i:03}.csv")


    np.savetxt(output_path, output_data, delimiter=",", fmt="%.5f", header="num,x,y", comments="")
    print("Interpolation completed! Interpolated data saved to:", output_path)
```

Listing 8: Interpolation spatiale et temporelle

```
import numpy as np
import matplotlib.pyplot as plt
import os
from scipy.interpolate import griddata

# load temporel interpolated reshaped exprimental data
def load_exp_data(i):
    file_path = f"E:/2023_2025/M2/Projet/Abaqus_simulation/interpolated_exp_data/image_{i:03}.csv"
        # check the existence of files
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
        return None

    data = np.loadtxt(file_path, delimiter=",", skiprows=1)

    return data

def load_num_data(i):
    file_path =  f"E:/2023_2025/M2/Projet/Abaqus_simulation/numeric_data
```

```
    /champs_deformation/deformation_data_{i}.csv"
        # check the existence of files
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
        return None

    data = np.loadtxt(file_path, delimiter=",", skiprows=1)

    return data

if __name__ == "__main__":    # exp to num
    # load initial geometric data
    #data_num = load_num_data(0)
    N = 285
    x_num, y_num = load_num_data(N)[:,5], load_num_data(N)[:,6]
    x_exp, y_exp = load_exp_data(N)[:,1], load_exp_data(N)[:,2]

    #print("x exp", np.shape(x_exp), "y exp", np.shape(y_exp))

    # interpolate exp data to num data
    # itp_y_exp = griddata((x_exp, y_exp), y_exp, (x_num, y_num), method='cubic')
    # itp_x_exp = griddata((y_exp, x_exp), x_exp, (y_num, x_num), method='cubic')

    # interpolate num data to exp data
    itp_y_num = griddata((x_num, y_num), y_num, (x_exp, y_exp), method='cubic')
    itp_x_num = griddata((y_num, x_num), x_num, (y_exp, x_exp), method='cubic')

    # plot the initial superposition
    plt.figure()
    #plt.scatter(data_num[:,5], data_num[:,6], color = 'g')
    #plt.scatter(x_num, y_num, color = 'red', alpha=1, label = 'num_data')

    # interpolated plot
    #plt.scatter(itp_x_exp, itp_y_exp, color = "black", alpha=0.3, label="interpolated_exp_data")
    plt.scatter(itp_x_num, itp_y_num, color = "black", alpha=1, label="interpolated_num_data")

    plt.scatter(x_exp, y_exp, color = "red", alpha=0.05, label = 'exp_data')

    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title(f"Superposition at t= {N*144.04/290}s")
    plt.legend()
    plt.show()

# if __name__ == "__main__":    # pour tout le long du temps

#     for i in range(0, 289):
#         x_num, y_num = load_num_data(i)[:,5], load_num_data(i)[:,6]
#         x_exp, y_exp = load_exp_data(i)[:,1], load_exp_data(i)[:,2]

```

```
#         # interpolate exp data to num data
#         itp_y_exp = griddata((x_exp, y_exp), y_exp, (x_num, y_num), method='cubic')
#         itp_x_exp = griddata((y_exp, x_exp), x_exp, (y_num, x_num), method='cubic')

#         # interpolate num data to exp data
#         itp_y_num = griddata((x_num, y_num), y_num, (x_exp, y_exp), method='cubic')
#         itp_x_num = griddata((y_num, x_num), x_num, (y_exp, x_exp), method='cubic')

#         plt.figure()
#         plt.scatter(x_exp, y_exp, color = "black", alpha=0.9)
#         # interpolated plot
#         plt.scatter(itp_x_num, itp_y_num, color = "b", alpha=0.1)
#         plt.xlabel("X")
#         plt.ylabel("Y")
#         plt.title(f"Superposition at t= {i*144.04/290}s")
#         plt.legend()
#         plt.show()
```

Listing 9: Superposition

```
import numpy as np
import matplotlib.pyplot as plt
import os
from scipy.interpolate import griddata


# load temporel interpolated reshaped exprimental data
def load_exp_data(i):
    file_path = f"E:/2023_2025/M2/Projet/Abaqus_simulation/interpolated_exp_data/image_{i:03}.csv"
        # check the existence of files
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
        return None

    data = np.loadtxt(file_path, delimiter=",", skiprows=1)

    return data

def load_num_data(i):
    file_path =  f"E:/2023_2025/M2/Projet/Abaqus_simulation/numeric_data/champs_deformation/deformation_data_{i}.csv"
        # check the existence of files
    if not os.path.exists(file_path):
        print(f"File not found: {file_path}")
        return None

    data = np.loadtxt(file_path, delimiter=",", skiprows=1)
    index_sorted = np.argsort(data[:, 6])
    data = data[index_sorted]

    index = np.arange(len(data))
    rdata = np.column_stack((index, data[:,5], data[:,6]))
    return rdata

```

```
# load initial geometric data
data_num_ini = load_num_data(0)
data_exp_ini = load_exp_data(0)

# center point of geometry (float)
y_cent_num = (max(data_num_ini[:,2]) + min(data_num_ini[:,2])) / 2
y_cent_exp = (max(data_exp_ini[:,2]) + min(data_exp_ini[:,2])) / 2

# difine y limits
y_max_exp,  y_min_exp = y_cent_exp + 15,  y_cent_exp -15
y_max_num,  y_min_num = y_cent_num + 15,  y_cent_num -15

# define geometric domain for observation
y_exp_filter = (data_exp_ini[:,2] >= y_min_exp) & (data_exp_ini[:,2] <=
    y_max_exp)
filtered_data_exp = data_exp_ini[y_exp_filter]
x_exp = filtered_data_exp[:,1]
y_exp = filtered_data_exp[:,2] # updata y data

y_num_filter = (data_num_ini[:,2] >= y_min_num) & (data_num_ini[:,2] <=
    y_max_num)
filtered_data_num = data_num_ini[y_num_filter]
x_num = filtered_data_num[:,1]
y_num = filtered_data_num[:,2] # updata y data

# get the coordinate index
############################################################
index_num = data_num_ini[:,0]
index_exp = data_exp_ini[:,0] - 1

filtered_index_num = index_num[y_num_filter] # Filtered indices
filtered_index_exp = index_exp[y_exp_filter] # Filtered indices
############################################################
i_min_num, i_max_num = int(min(filtered_index_num)), int(max(
    filtered_index_num))     #  minimum index and maximum index for
    useful numeric zone
i_min_exp, i_max_exp = int(min(filtered_index_exp)), int(max(
    filtered_index_exp))     #  minimum index and maximum index for
    useful experimental zone
############################################################


def superp(i): # num to exp
    # just use the useful zone xy coord.
    data_num = load_num_data(i)
    data_exp = load_exp_data(i)

    # Use boolean masks to select data
    num_mask = (index_num >= i_min_num) & (index_num <= i_max_num)
    exp_mask = (index_exp >= i_min_exp) & (index_exp <= i_max_exp)

    x_num = data_num[num_mask, 1]
    y_num = data_num[num_mask, 2]
    x_exp = data_exp[exp_mask, 1]
    y_exp = data_exp[exp_mask, 2]

    points_num = np.column_stack((x_num, y_num))
    points_exp = np.column_stack((x_exp, y_exp))
```

```
86
87      #interpolate num data to exp data
88      itp_x = griddata(points_num, x_num, points_exp, method='cubic')
89      itp_y = griddata(points_num, y_num, points_exp, method='cubic')
90
91      itp_data = np.column_stack((itp_x, itp_y))
92      data_exp = np.column_stack((x_exp, y_exp))
93
94      return itp_data, data_exp
95
96  if __name__ == '__main__':  # num to exp
97      i = 289
98      itp_data, exp_data = superp(i)
99      itp_x_num, itp_y_num = itp_data[:, 0], itp_data[:, 1]
100     x_exp, y_exp = exp_data[:, 0], exp_data[:, 1]
101
102     data_num = load_num_data(i)
103     num_mask = (index_num >= i_min_num) & (index_num <= i_max_num)
104     x_num = data_num[num_mask, 1]
105     y_num = data_num[num_mask, 2]
106
107     # plot the i superposition
108     plt.figure()
109     plt.scatter(x_num, y_num, color = 'black', label='num field')
110     plt.scatter(itp_x_num, itp_y_num, color = 'r', label= 'interpolated
        num field')
111     plt.scatter(x_exp, y_exp, color = "b", alpha=0.35, label='exp field'
        )
112     plt.xlabel("X")
113     plt.ylabel("Y")
114     plt.title(f"Superposition at t= {i*144.04/290}s")
115     plt.legend()
116     plt.show()
```

Listing 10: Superposition en zone utile

```
1   from abaqus import *
2   from abaqusConstants import *
3   from odbAccess import openOdb
4   import job
5
6   import numpy as np
7   #import matplotlib.pyplot as plt
8   import os
9   import gc
10  from scipy.interpolate import griddata
11  from scipy.optimize import minimize
12
13
14  # load temporel interpolated reshaped exprimental data
15  def load_exp_data(i):
16      file_path = f"E:/2023_2025/M2/Projet/Abaqus_simulation/
        interpolated_exp_data/image_{i:03}.csv"
17          # check the existence of files
18      if not os.path.exists(file_path):
19          print(f"File not found: {file_path}")
20          return None
21
22      data = np.loadtxt(file_path, delimiter=",", skiprows=1)
```

```
23
24      return data
25
26  # load abaqus deformation data
27  def load_num_data(i):
28      odb_path = 'E:/2023_2025/M2/Projet/Abaqus_simulation/traction.odb'
29      odb = openOdb(path=odb_path)
30      print(f"ODB opened: {odb}")
31
32      # get the step1 and the initial time
33      last_step = odb.steps.keys()[-1] # setp1
34      frames = odb.steps[last_step].frames[i] # for each time step
35
36      # get force and displacement data
37      displacement_field = frames.fieldOutputs['U']
38
39      # get instance
40      instance_name = list(odb.rootAssembly.instances.keys())[0]
41      instance = odb.rootAssembly.instances[instance_name]
42
43      # difine initial lists to save displacement field
44      node_labels = []
45      x_coords = []
46      y_coords = []
47      xdisplacements = []
48      ydisplacements = []
49
50      # append datas
51      for value in displacement_field.values:
52
53          node_label = value.nodeLabel  # node number
54          node_labels.append(node_label)
55          node = instance.nodes[node_label - 1] # get instance nodes
56
57          x_coords.append(node.coordinates[0])  # X coord
58          y_coords.append(node.coordinates[1])  # Y coord
59          xdisplacements.append(value.data[0])  # X displacement
60          ydisplacements.append(value.data[1])  # Y displacement
61
62      # close ODB field
63      odb.close()
64
65      # Create a numpy array to store the displacement field
66      disp_field = np.zeros((len(node_labels), 7))
67      disp_field[:,0] = node_labels
68      disp_field[:,1] = x_coords
69      disp_field[:,2] = y_coords
70      disp_field[:,3] = xdisplacements
71      disp_field[:,4] = ydisplacements
72      # displacement field
73      disp_field[:,5] = disp_field[:,1] + disp_field[:,3]
74      disp_field[:,6] = disp_field[:,2] + disp_field[:,4]
75
76      index_sorted = np.argsort(disp_field[:, 6])
77      filtered_data = disp_field[index_sorted]
78      index = np.arange(len(filtered_data))
79      data = np.column_stack((index, filtered_data[:,5], filtered_data
    [:,6]))
```

```
80
81      return data
82
83  # load initial geometric data
84  data_num_ini = load_num_data(0)
85  data_exp_ini = load_exp_data(0)
86
87  # center point of geometry (float)
88  y_cent_num = (max(data_num_ini[:,2]) + min(data_num_ini[:,2])) / 2
89  y_cent_exp = (max(data_exp_ini[:,2]) + min(data_exp_ini[:,2])) / 2
90
91  # difine y limits
92  y_max_exp,  y_min_exp = y_cent_exp + 15,  y_cent_exp -15
93  y_max_num,  y_min_num = y_cent_num + 15,  y_cent_num -15
94
95  # define geometric domain for observation
96  y_exp_filter = (data_exp_ini[:,2] >= y_min_exp) & (data_exp_ini[:,2] <=
        y_max_exp)
97  filtered_data_exp = data_exp_ini[y_exp_filter]
98  x_exp = filtered_data_exp[:,1]
99  y_exp = filtered_data_exp[:,2] # updata y data
100
101 y_num_filter = (data_num_ini[:,2] >= y_min_num) & (data_num_ini[:,2] <=
        y_max_num)
102 filtered_data_num = data_num_ini[y_num_filter]
103 x_num = filtered_data_num[:,1]
104 y_num = filtered_data_num[:,2] # updata y data
105
106 # get the coordinate index
107 ############################################################
108 index_num = data_num_ini[:,0]
109 index_exp = data_exp_ini[:,0] - 1
110
111 filtered_index_num = index_num[y_num_filter] # Filtered indices
112 filtered_index_exp = index_exp[y_exp_filter] # Filtered indices
113 ############################################################
114 i_min_num, i_max_num = int(min(filtered_index_num)), int(max(
        filtered_index_num))     #  minimum index and maximum index for
        useful numeric zone
115 i_min_exp, i_max_exp = int(min(filtered_index_exp)), int(max(
        filtered_index_exp))     #  minimum index and maximum index for
        useful experimental zone
116 ############################################################
117
118 # interpolate num data to exp data
119 itp_y_num_ini = griddata((x_num, y_num), y_num, (x_exp, y_exp), method='
        cubic')
120 itp_x_num_ini = griddata((y_num, x_num), x_num, (y_exp, x_exp), method='
        cubic')
121
122 # img_path = r"E:\\2023_2025\\M2\\Projet\\Abaqus_simulation\\images\\
        superposition_initial.png"
123 # # plot the initial superposition
124 # plt.figure()
125 # plt.scatter(itp_x_num_ini, itp_y_num_ini, color = 'r')
126 # plt.scatter(x_exp, y_exp, color = "y")
127 # plt.xlabel("X")
128 # plt.ylabel("Y")
```

```
# plt.title("Superposition at t=0")
# plt.legend()
# plt.savefig(img_path)
# plt.close()
# print(f"plot image is saved at: {img_path}")

root_path = r"E:\\2023_2025\\M2\\Projet"
def abaqus_sim(i, JC_params):

    # CAE file path
    cae_file_path = os.path.join(root_path, "Abaqus_simulation","
    simulation_traction.cae")
    # open CAE file
    openMdb(pathName = cae_file_path)

    # get elastic and Johnson-Cook paramamters

    #E, v, A, B, n = JC_params
    A, B, n = JC_params

    # updata paramaters
    model_name = "Model-1"
    material_name = "Alu-7075"

    # open Modele and Materiele
    model = mdb.models[model_name]
    material = model.materials[material_name]
    #material.Elastic(table=((E, v),))
    material.Plastic(hardening=JOHNSON_COOK, table=((A, B, n),))

    # save ODB file and submit the job
    os.chdir(r"E:\\2023_2025\\M2\\Projet\\Abaqus_simulation")
    job_name = 'traction'
    mdb.Job(name=job_name, model=model_name)
    mdb.jobs[job_name].submit(consistencyChecking=OFF)
    mdb.jobs[job_name].waitForCompletion()

    cae_name = "simulation_traction"
    # savgarder du modele
    mdb.saveAs(os.path.join(root_path, "Abaqus_simulation", cae_name))

    data = load_exp_data(i)

    return data

def superp(i, JC_params):
    # just use the useful zone xy coord.
    data_num = load_num_data(i)
    data_exp = abaqus_sim(i,JC_params)

    # filteration of data
    num_mask = (index_num >= i_min_num) & (index_num <= i_max_num)
    exp_mask = (index_exp >= i_min_exp) & (index_exp <= i_max_exp)
    x_num = data_num[num_mask, 1]
    y_num = data_num[num_mask, 2]
    x_exp = data_exp[exp_mask, 1]
    y_exp = data_exp[exp_mask, 2]

```

```
    points_num = np.column_stack((x_num, y_num))
    points_exp = np.column_stack((x_exp, y_exp))

    data_exp = np.column_stack((x_exp, y_exp))
    data_num = np.column_stack((x_num, y_num))

    #interpolate num data to exp data
    # itp_x = griddata(points_num, x_num, points_exp, method='cubic')
    # itp_y = griddata(points_num, y_num, points_exp, method='cubic')

    #interpolate exp data to num data
    itp_x = griddata(points_exp, x_exp, points_num, method='linear')
    itp_y = griddata(points_exp, y_exp, points_num, method='linear')

    # nitp_data = np.column_stack((itp_x, itp_y))
    eitp_data = np.column_stack((itp_x, itp_y))

    # return nitp_data, data_exp
    return eitp_data, data_num

def cost_fct(JC_params, i):

    itp_data, data_exp = superp(i, JC_params)

    denominator = np.sum(data_exp ** 2)
    if denominator == 0:
        raise ValueError("Experimental force data sum is zero, normalization is invalid.")
    error = np.sqrt(np.sum((data_exp - itp_data)** 2)) / np.sqrt(denominator)

    return error


if __name__ == "__main__":

    JCparam_err = []
    JC_params = [450, 270, 0.3]  # initialize the list

    for i in range(122, 299):  # time of platic deformation
        try:
            result = minimize(cost_fct, JC_params, method='Nelder-Mead', options={'disp': False, 'maxiter': 50}, args=(i,))
            print("optimized Johnson Cook parameters:", result.x)

            if result.success:

                # updata parameters
                JCparams = result.x
                JCparam_err.append(list(JCparams) + [result.fun])
                print(f"Johnson Cook parameters and errors{JCparam_err}")

            # Explicit memory cleanup
            del result
            gc.collect()

        except Exception as e:
```

```
240             print(f"Optimization failed at step {i}: {str(e)}")
241             continue
```

Listing 11: Optimisation supérposition

8.3 Graphiques

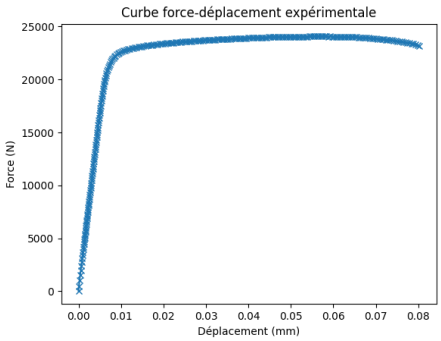
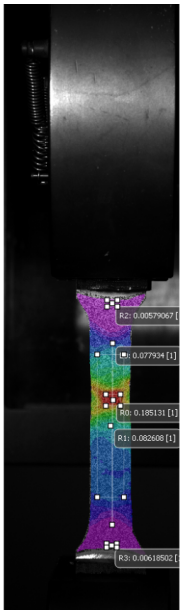
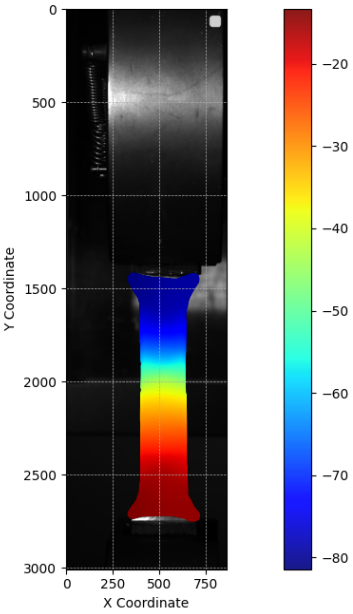


Figure 1: Courbe force-déplacement issue des essais expérimentaux.

(a) Image brute



(b) Image avec ROI et champ de déplacement



(c) Analyse DIC

Figure 2: Analyse DIC

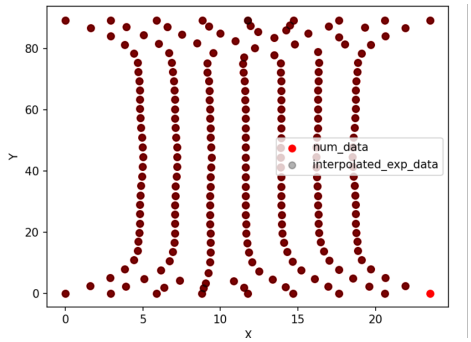


Figure 3: Champ de déplacement obtenu par FEM.

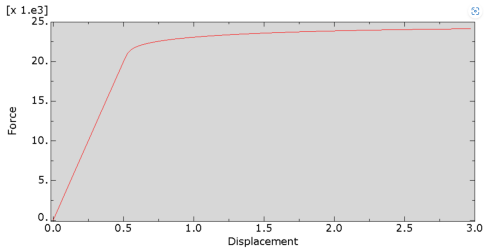


Figure 4: Courbe force-déplacement obtenue par FEM.

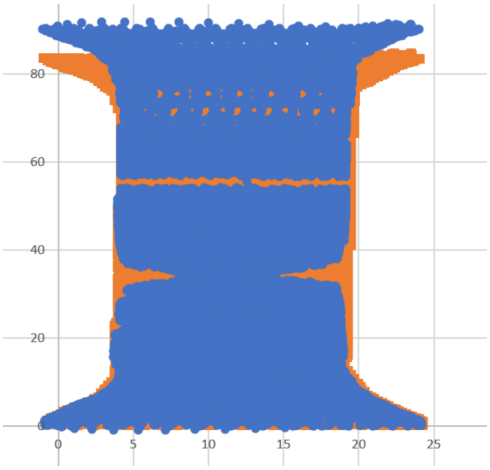


Figure 5: Champ de déplacement obtenu par DIC.

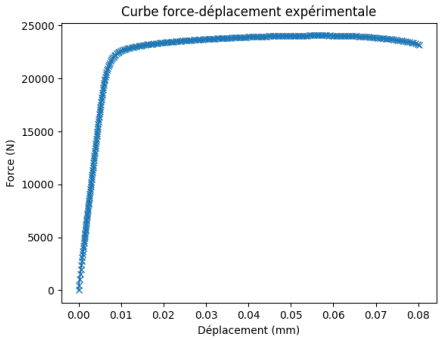


Figure 6: Courbe force-déplacement obtenue par DIC.

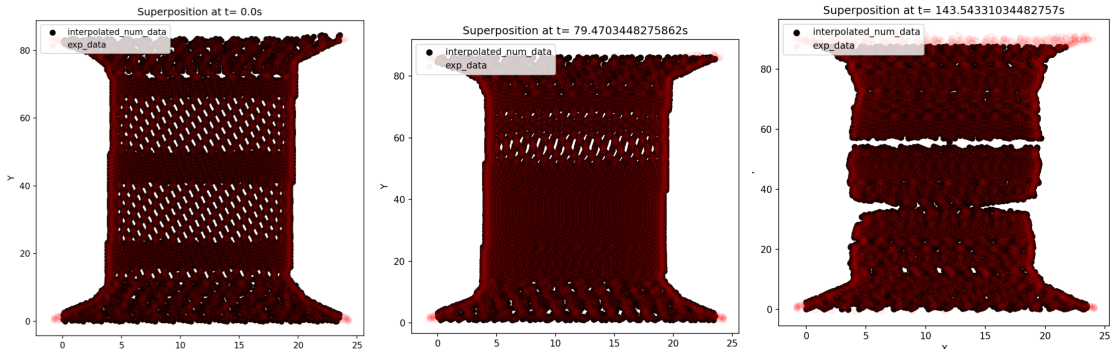


Figure 7: Superposition des champs DIC et FEM.

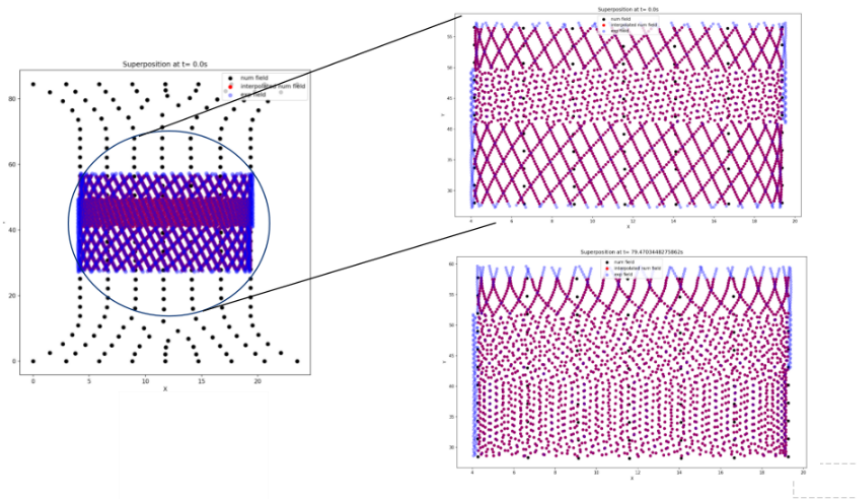


Figure 8: Superposition des champs dans la zone utile.