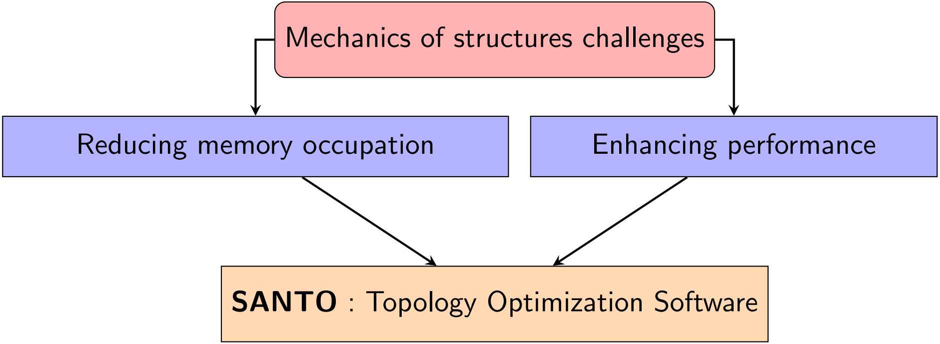
NURBS Surfaces for Topology Optimization Problems

2023/2024

EL GODE Amal NAIM Mouna
REDOUANE Mohamed Youssef TOULER FOKA Camille Willy
ZAHI Ziad

Encadrant : ZERROUQ Salah-eddine

Introduction



Introduction

- **Advantages:**
  - Modeling accuracy
  - Design flexibility
  - Material Optimization
  - Cost Reduction
- **Challenges:**
  - Computational complexity
  - Memory Management

Primary Challenges Targeted

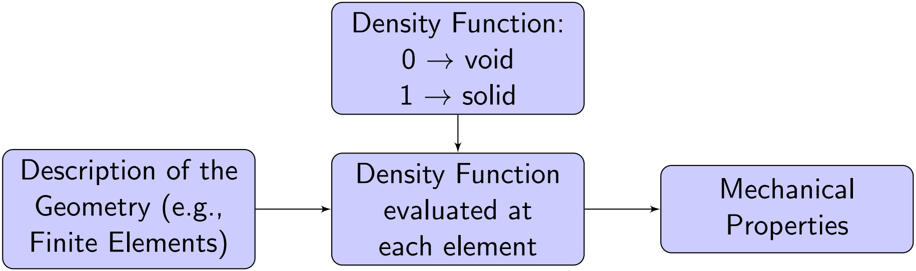
- The software is entirely written in **Python**
- **Python:**
  - Dynamic language
  - More flexible
  - Automatic memory management
- **C/C++:**
  - Low-level language
  - More concerned with memory management and system resources
  - Detailed and complex code
  - Typically faster

$\Rightarrow$ *Goal: convert computations to a lower level language which is* **C**

Topological Optimization (TO)

Mathematical method to find the optimum material distribution in a given volume subject to constraints

$\rightarrow$ **Solid Isotropic Material with Penalisation (SIMP)**



Non-Uniform Rational B-Splines (NURBS)

Mathematical method for representing geometries

- **Spline:** Piecewise Polynomial Function
- **B-spline:** Spline Function with Minimal Support
- **NURBS:** Piecewise Rational Fraction Representation

Advantages

- Ease and accuracy of shape evaluation
- Ability to approximate complex shapes
- Compatible with CAD software

Non-Uniform Rational B-Splines (NURBS)

The NURBS surface is defined as:

$$S(u,v) = \sum_{i=0}^{n_u} \sum_{j=0}^{n_v} R_{i,j}(u,v) P_{i,j}$$

where:

- $R_{i,j}(u,v)$: Piecewise Rational Basis Functions
- $P_{i,j} = \{x_{i,j}, y_{i,j}, z_{i,j}\}$: Control Points

$$R_{i,j}(u,v) = \frac{N_{i,p}(u) N_{j,q}(v) w_{i,j}}{\sum_{k=0}^{n_u} \sum_{l=0}^{n_v} N_{k,p}(u) N_{l,q}(v) w_{k,l}}$$

where:

- $N_{i,p}(u), N_{j,q}(v)$: Blending Functions
- p and q: NURBS Degrees along u and v directions
- $w_{i,j}$: Weights

Non-Uniform Rational B-Splines (NURBS)

The blending functions are recursively defined by means of the Bernstein's polynomials:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } U_i \leq u < U_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(u) = \frac{u - U_i}{U_{i+p} - U_i} N_{i,p-1}(u) + \frac{U_{i+p+1} - u}{U_{i+p+1} - U_{i+1}} N_{i+1,p-1}(u)$$

$\boldsymbol{U}$, $\boldsymbol{V}$: knot vectors

Local Support Property

$R_{i,j} = 0$ if (u, v) outside $[U_i, U_{i+p+1}[\times[V_j, v_{j+q+1}[$

Non-Uniform Rational B-Splines (NURBS)

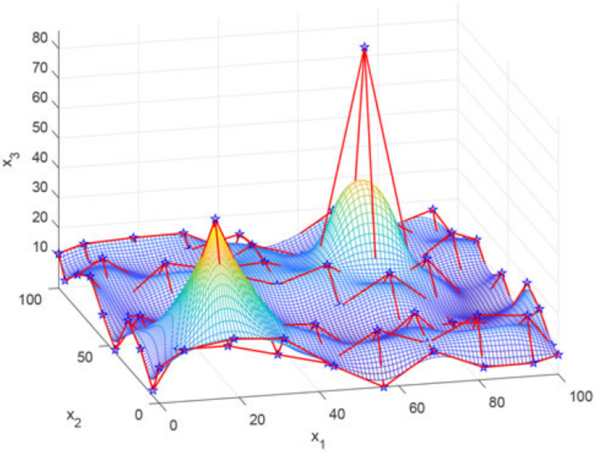


Figure 1: Example of NURBS Surface. Source: "NURBS hyper-surfaces for 3D topology optimization problems", *Giulio Costa, Marco Montemurro and Jérôme Pailhès*

NURBS-Based SIMP Method

Compliance ($\neq$ Stiffness)

How much a material deforms under an applied load

SIMP Method for Minimum Compliance Problem

Given a volume constraint, what is the optimal topology that minimizes compliance?

SIMP Method for Minimum Compliance Problem

Optimization Problem

$$
\begin{aligned}
\min_{\rho_e} \quad & c(\rho_e) \\
\text{subject to:} \quad & [K]\{U_{FEM}\} = \{F\} \\
& V(\rho_e) = fV_{\mathrm{tot}} \\
& \rho_{\min} \leq \rho_e \leq 1, \quad e = 1, \ldots, N_e
\end{aligned}
$$

NURBS-Based SIMP Method

$$\rho(u,v) = \sum_{i=0}^{n_u} \sum_{j=0}^{n_v} R_{i,j}(u,v)\bar{\rho}_{i,j}$$

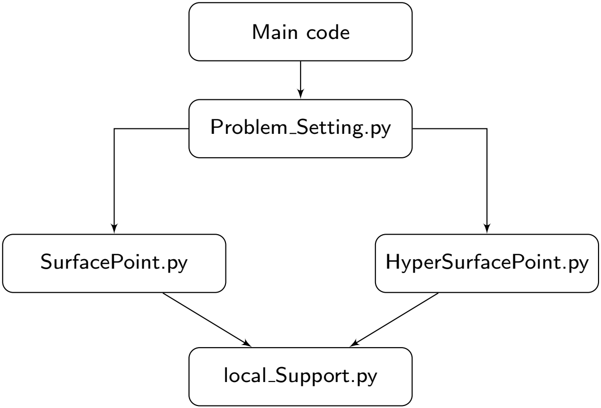
where $\bar{\rho}_{i,j}$ is the value of the fictitious density at each control point

- $\bar{\rho}_{i,j}$ and $w_{i,j}$ are collected in $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$

$$\begin{aligned} \min_{\boldsymbol{\xi},\boldsymbol{\eta}} \quad & c(\rho(\boldsymbol{\xi},\boldsymbol{\eta})) \\ \text{subject to:} \quad & [K]\{U_{FEM}\} = \{F\} \\ & V(\rho_e) = fV_{\text{tot}} \\ & \{g(\boldsymbol{\xi},\boldsymbol{\eta})\} \leq \{0\} \end{aligned}$$

where $\{g(\boldsymbol{\xi},\boldsymbol{\eta})\}$ is the vector collecting the constraints of different nature

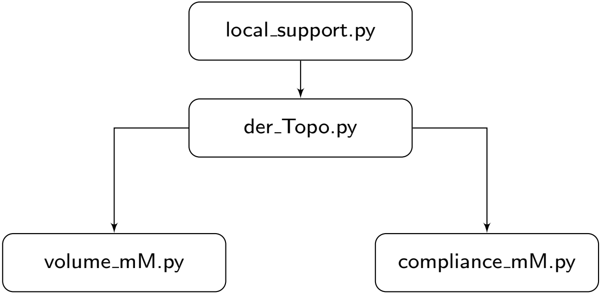
Python Code Architecture



`Output:`

- Local_support
- BF_supprort
- IND_mask_active

Python Code Architecture



`Output:`

- Volume gradient
- Compliance gradient

C Code Architecture

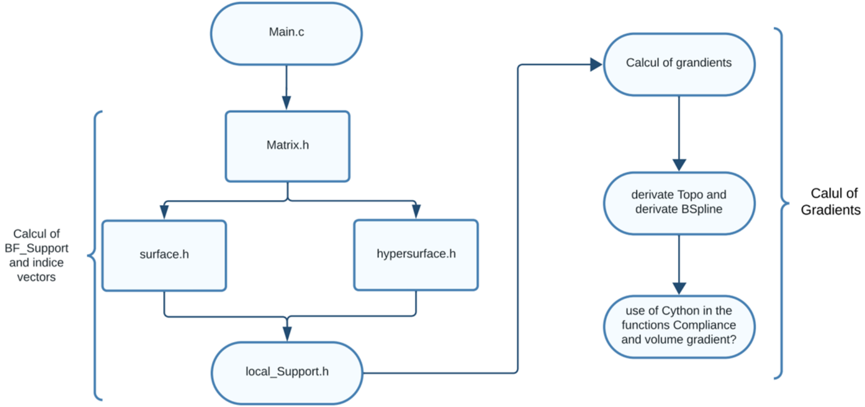


Figure 2: Architecture of C version

Data Structures

Two problems faced in **C** implementation:

- No predifined structures for vectors , 2D/3D matrices or list vectors
- Big usage of memory space for large data structures (`BF_support` = 5GB matrix)

> $\Rightarrow$ *Solution:* creation of new data structures suitable for our problem

Data Structures

```
typedef struct {
    double *data;
    int rows;
    int cols;
    int depth;
} Matrix;
```

Methods:

- Loading matrices
- Initializing matrices
- Matrix-vector product
- Hadamard product
- Extracting values/columns/ rows ..

Data Structures

```
typedef struct {
    double *data;
    int length;
} Vector;

typedef struct {
    Vector *vectors;
    int size;
} ListOfVectors;
```

Methods:

- Loading vectors
- Initializing vectors
- Adding vectors to lists
- Freeing list of vectors
- Eliminate duplicated values with hash function

Data structures

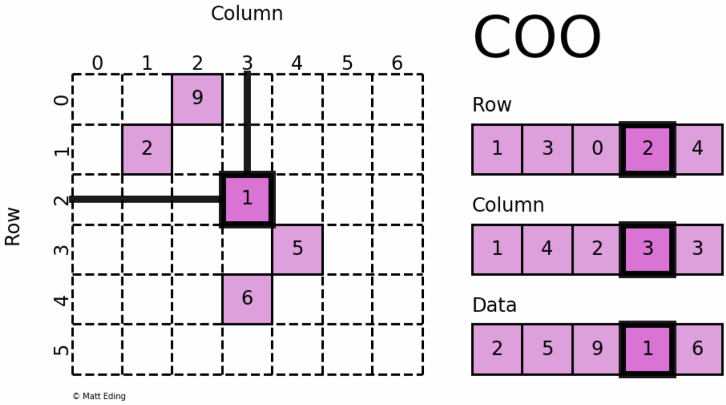


Figure 3: Matrix in COO format

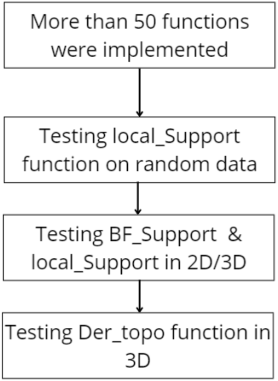
Data Structures

```
typedef struct {
    double *values;
    int *rowsIndices;
    int *colsIndices;
    int *depthsIndices;
    int nonZeroCount;
    int rows;
    int cols;
    int depth;
} COOMatrix;
```

Functions:

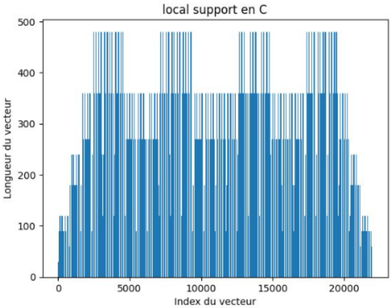
- Loading matrices in COO format
- Converting and storing dense Matrices in COO format
- Hadamard Matrix-vector product for COO matrices
- Subtract COOMatrices (also check if they are sorted)
- Select Row and columns

Validation 2D



- der_topo is validated only in 3D implementation

2D Validation of Local_Support Function



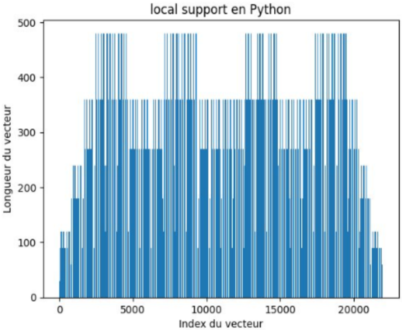


Figure 4: Local_support matrix en C and Python

2D Validation of Functions

Matrix	Local_Support	BF_Support	IND_Mask_Active
Method	is_equal	AD	is_equal
Tolerence	0	10^{-10}	0

Table 1: Comparaison between matrices in python and C version

3D Validation of Functions

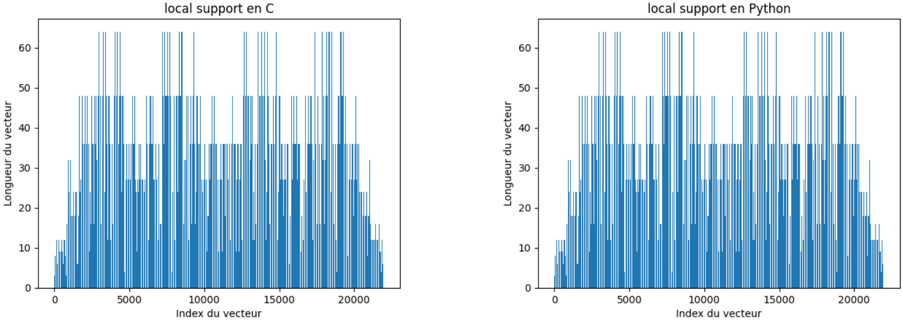


Figure 5: Local_support matrix en C and Python

3D Validation of Function

Matrix	Local_Support	BF_Support	IND_Mask_Active	der_W	der_CP	BF_Mask
Method	is_equal	AD	is_equal	AD	AD	is_equal
Tolerence	0	10^{-10}	0	10^{-7}	10^{-9}	0

Table 2: Comparaison between matrices in python and C version

Parallelism

Parallelism

Loops exhibiting significant temporal complexity were optimized through parallelization with OpenMP

Performance Comparaison

For 3D implementation:

Execution time to calculate Local_Support function

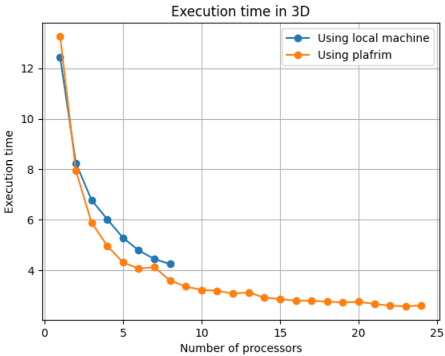
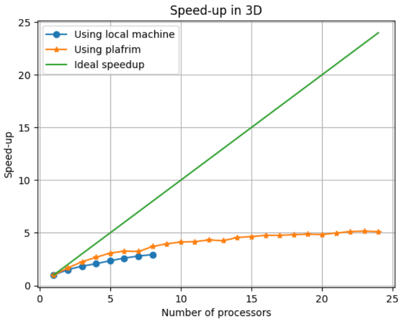


Figure 6: Execution time over number of processors

Calculation of Speed-up and Efficiency

- Speed-up: $S(p) = \frac{T(1)}{T(p)}$
- Efficiency: $E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$



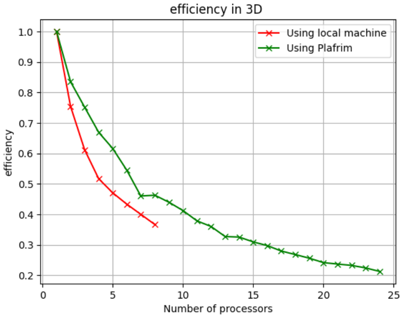


Figure 7: Calculation of Speed-up & Efficiency for 3D implementation of local_support function

Calculation of Speedup & Efficiency for der_Nurbs

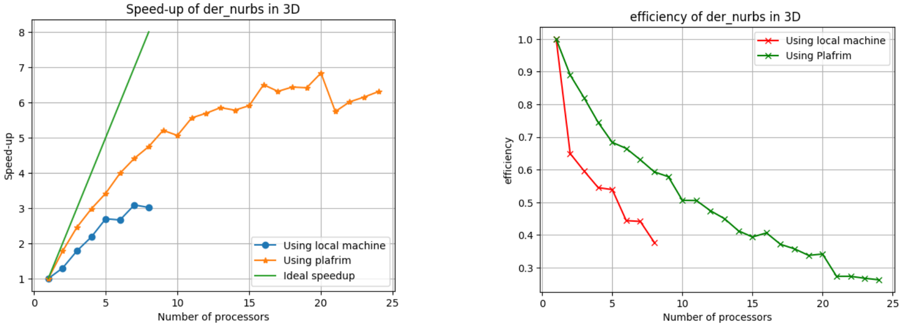


Figure 8: Calculation of Speed-up & Efficiency for 3D implementation of der_Nurbs

For 2D implementation:

Execution time to calculate Local_Support function in 2d

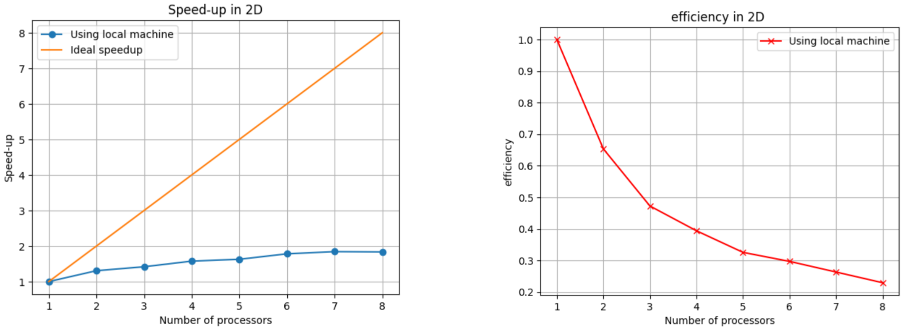


Figure 9: Calculation of Speed-up & Efficiency for 2D implementation of der_Nurbs

Performance Summary

function	Local_Support_fun 2D	Local_Support_fun 3D	der_nurbs 3D
Sequential Code in Python	20.01 s	23.62 s	64.92 s
Sequential Code in C	8.08 s	11,82 s	10.80 s
Parallel Code in C with OpenMP	6.13 s	2.56 s	1.20 s

Table 3: Time comparaison between functions in Python and C version

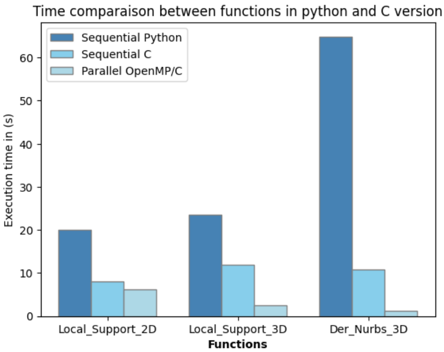


Figure 10: Performance summary

Conclusion

Parallelism was successfully implemented using OpenMP, enhancing the performance of the C version

Sparse COO matrices may not be as efficient and could be replaced with other types of sparse matrices

Cython library can be used to implement C in Python version