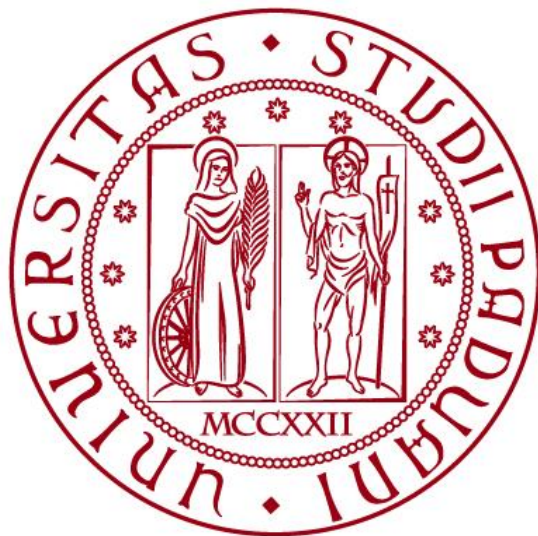


UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA CIVILE, EDILE E AMBIENTALE
Department Of Civil, Environmental and Architectural Engineering

Corso di Meccanica Computazionale



Codice numerico agli elementi finiti Straus Destroyer

Docenti: Prof. Ing. Carmelo Maiorana
Prof. Ing. Gianluca Mazzucco
Prof. Ing. Beatrice Pomaro

Studentesse: Guglietta Nikita
Righetto Angelica

Matricole: 1217811
1220412

ANNO ACCADEMICO 2019-2020

SOMMARIO

| | | |
|-----|--------------------------------|----|
| 1 | Introduzione..... | 3 |
| 2 | Codice numerico..... | 4 |
| 2.1 | Elementi truss | 4 |
| 2.2 | Classe dei carichi..... | 7 |
| 2.3 | Assemblaggio dei carichi | 10 |
| 2.4 | Programma Python | 14 |
| 3 | Test | 17 |
| 3.1 | Capriata 2D..... | 17 |
| 3.2 | Supporto fotovoltaico 3D | 20 |
| 3.3 | Carichi nodali misti | 24 |
| 3.4 | Elementi inclinati..... | 26 |
| 3.5 | Edificio multipiano | 30 |
| 3.6 | Passerella expo Milano..... | 32 |

1 INTRODUZIONE

Il codice agli elementi finiti ‘Straus Destroyer’ sviluppato durante il corso di meccanica computazionale è un codice numerico che sfrutta la programmazione ad oggetti in Matlab ed è in grado di calcolare gli spostamenti per alcune tipologie di strutture. All’interno del codice infatti, sono stati implementati solamente gli elementi monodimensionali *beam* e *truss*, e l’elemento bidimensionale *Quad8*. Il funzionamento del codice si basa sul calcolo della matrice di rigidezza del sistema K , assemblata a partire dalle matrici di rigidezza locali di ogni elemento, e sul calcolo del vettore delle forze esterne agenti F , con cui si ottiene il vettore degli spostamenti mediante il prodotto tra l’inverso della matrice di rigidezza K^{-1} e il vettore delle forze esterne F .

In questa relazione si tratterà nello specifico la costruzione delle classi TRUSS3D (contenente anche l’assemblaggio della matrice di rigidezza dell’elemento), LOAD, NODELOAD, POINTLOAD e DISTRLOAD, il calcolo delle forze nodali equivalenti nel caso di carichi puntuali e uniformemente distribuiti e l’assemblaggio dei carichi all’interno della classe SOLUTORE. Per creare il file di input contenente i dati necessari al codice, si è utilizzato GiD come preprocessore e postprocessore. Inoltre, è stato implementato uno script in Python in grado di modificare il file ‘.dat’ generato con GiD ed aggiungere eventuali carichi concentrati/distribuiti sugli elementi monodimensionali. Per la lettura dei dati del nuovo file di input è stata poi modificata anche la *function* READ_INPUT_FILE.

Una volta ultimata la scrittura del codice, si sono eseguiti una serie di test volti a verificare l’accuratezza dei risultati calcolati con ‘Straus Destroyer’, mettendoli a confronto con i risultati del programma agli elementi finiti commerciale Straus7.

2 CODICE NUMERICO

2.1 ELEMENTI TRUSS

L'elemento *truss* è un elemento monodimensionale caratterizzato da comportamento assiale dovuto al fatto che la rigidezza dell'elemento risulta diversa da zero solo lungo il suo asse. Per questo motivo l'elemento *truss* viene utilizzato principalmente per modellare strutture reticolari, per collegare due corpi o in fase di predimensionamento, altrimenti si preferisce utilizzare elementi *beam* con comportamento flessionale. L'elemento *truss* ha 2 o 3 gradi di libertà per nodo, a seconda che si tratti di un elemento nello spazio bidimensionale o tridimensionale, rispettivamente.

All'interno della classe TRUSS3D, che per la precisione è una classe figlia della classe ELEMENT, sono state assegnate due proprietà fisse, ovvero il numero di nodi dell'elemento (*totnodes*) e il numero di gradi di libertà per nodo (*dimdof*), in quanto queste ultime caratterizzano tutti gli elementi *truss* tridimensionali. Si sono poi assegnate anche tre proprietà che variano invece per ogni elemento: gli identificativi dei nodi di estremità (*nodes_id*), l'identificativo delle proprietà fisiche dell'elemento (*prop_id*) e le coordinate spaziali dei nodi (*X*).

Si distinguono inoltre due *functions* diverse, la prima crea l'elemento *truss* identificando i due nodi di estremità e assegnando le proprietà all'elemento: il modulo elastico del materiale di cui è costituito il *truss*, la lunghezza e l'area della sezione dell'asta; la seconda *function* invece si occupa dell'assemblaggio della matrice di rigidezza dell'elemento.

```
%E -> modulo elastico
%A -> area
%L -> lunghezza asta
%x1-> coordinate del primo nodo dell'elemento
%x2-> coordinate del secondo nodo dell'elemento
classdef TRUSS3D < handle
    %-----
    properties(Constant)
        DIMDOF = 3
        TOTNODES = 2
    end
    %-----
    properties
        nodes_id = [];
        prop_id;
        X = []; %lista di vettori posizione [(x1,y1,z1);(x2,y2,z2)]
    end
    %-----
    methods
        %-----
        function this = TRUSS3D(nodes_id, property_id)
            this.nodes_id = nodes_id;
            this.prop_id = property_id;
        end
        %-----
    end
end
```

Figura 1: Classe TRUSS3D, proprietà dell'elemento e function che crea l'elemento

```

function [K]=globalStiffness(this,prop,X)
    this.X = X;
    young = prop.mat.E;
    area = prop.sec.A;
    gl = GLOBAL();
    x1 = X(1,:);
    x2 = X(2,:);
    L = gl.distance(x1, x2);
    if L==0.0
        fprintf('WARNING in globalStiffnessMatrix: elemento\n');
        fprintf('                di lunghezza nulla\n');
        return
    end
    %computing direction cosins matrix for truss element in a
    %three-dimensional space

    Cx=(x2(1)-x1(1))/L;
    Cy=(x2(2)-x1(2))/L;
    Cz=(x2(3)-x1(3))/L;

    C=[ Cx^2    Cx*Cy  Cx*Cz  -Cx^2    -Cx*Cy  -Cx*Cz;
        Cx*Cy   Cy^2   Cy*Cz  -Cx*Cy   -Cy^2   -Cy*Cz;
        Cx*Cz   Cy*Cz   Cz^2  -Cx*Cz   -Cy*Cz  -Cz^2;
        -Cx^2   -Cx*Cy  -Cx*Cz  Cx^2    Cx*Cy   Cx*Cz;
        -Cx*Cy  -Cy^2   -Cy*Cz  Cx*Cy   Cy^2    Cy*Cz;
        -Cx*Cz  -Cy*Cz  -Cz^2  Cx*Cz   Cy*Cz   Cz^2];
    %global stiffness matrix
    K=area*young/L*C;
end

```

Figura 2: function per l'assemblaggio della matrice di rigidezza dell'elemento truss

La *function* per l'assemblaggio della matrice di rigidezza si basa sull'impiego di una matrice di rotazione T , attraverso la quale è possibile esprimere il vettore delle forze $\{f'\}$ e degli spostamenti $\{s'\}$ riferiti al sistema di riferimento locale in termini di sistema di riferimento globale

$$\{s'\} = [T] \{s\}, \quad \{f\} = [T]^T \{f'\}$$

In generale, le forze e gli spostamenti locali sono legati tra loro da una matrice di rigidezza locale $[k']$ tale per cui

$$\{f'\} = [k'] \{s'\}$$

mentre le forze e gli spostamenti globali sono legati dalla matrice di rigidezza globale

$$\{f\} = [k] \{s\}$$

A partire dalle relazioni precedenti, è facile riscrivere il vettore delle forze globali sostituendo $\{s'\}$ e $\{f'\}$ per ottenere

$$\{f\} = [T]^T [k'] [T] \{s\}$$

Dal confronto diretto tra le ultime due equazioni appare allora chiaro che la matrice di rigidezza rispetto al sistema di riferimento globale risulti

$$[k] = [T]^T [k'] [T]$$

e rappresenta la matrice di rigidezza per un elemento orientato in maniera arbitraria. La matrice di rotazione T è una matrice ortogonale che raggruppa i coseni direttori dell'elemento, ovvero le proiezioni del vettore unitario associato all'asse x (assunto come asse baricentrico dell'elemento) del sistema di riferimento locale sui vettori unitari associati agli assi del sistema di riferimento globale

$$[T] = \begin{bmatrix} C_x & C_y & C_z & 0 & 0 & 0 \\ 0 & 0 & 0 & C_x & C_y & C_z \end{bmatrix}$$

$$C_x = (x_2 - x_1)/L$$

$$C_y = (y_2 - y_1)/L$$

$$C_z = (z_2 - z_1)/L$$

Mentre la matrice di rigidezza espressa rispetto al sistema locale dell'elemento risulta

$$[k'] = \frac{AE}{L} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Infine, la matrice di rigidezza dell'elemento *truss* rispetto ad un sistema di riferimento globale arbitrario è

$$[k] = \frac{AE}{L} \begin{bmatrix} C_x^2 & C_x C_y & C_x C_z & C_x^2 & C_x C_y & C_x C_z \\ & C_y^2 & C_y C_z & C_x C_y & C_y^2 & C_y C_z \\ & & C_z^2 & C_x C_z & C_y C_z & C_z^2 \\ & & & C_x^2 & C_x C_y & C_x C_z \\ Sym & & & & C_y^2 & C_y C_z \\ & & & & & C_z^2 \end{bmatrix}$$

E corrisponde alla matrice C computata alla fine della *function* 'globalStiffness' che a partire dalle coordinate dei due nodi dell'elemento e dalla lunghezza di quest'ultimo, calcola i tre coseni direttori con cui costruire la matrice di rigidezza.

2.2 CLASSE DEI CARICHI

Il codice agli elementi finiti sviluppato sfrutta la programmazione a classi di Matlab, la quale permette di creare delle classi in grado di combinare dati (proprietà) e funzioni associate (metodi).

Le classi si differenziano inoltre in classi “madri” e classi “figlie”, per quanto riguarda l’implementazione dei carichi applicabili ad un elemento monodimensionale, all’interno del codice si è costruita una classe madre LOAD e tre classi figlie NODELOAD, POINTLOAD, DISTRLOAD che gestiscono le diverse tipologie di carico.

CLASSE LOAD

```
classdef LOAD < handle
%-----
properties
    type_load;    %tipo di carico:
                  %          1) carico nodale
                  %          2) carico puntuale
                  %          3) carico superficiale

    load = [];    %vettore con le componenti del carico lungo x,y,z
    node_id;      %numero identificativo del nodo
    el_id;        %identificativo dell'elemento su cui applica il carico
    a;            %distanza di applicazione del carico puntuale
    loading;      %classe del carico
end
%-----
```

Figura 3: Proprietà della classe LOAD

```
methods
%-----
function this=LOAD(type_load,load,varargin)
    this.type_load = type_load;
    this.load = load;
    if type_load == 1
        this.node_id = varargin{1};
        this.loading = NODELOAD(type_load,load,this.node_id);
    elseif type_load == 2
        this.el_id = varargin {1};
        this.a = varargin{2};
        this.loading = POINTLOAD(type_load,load,this.el_id,this.a);
    elseif type_load == 3
        this.el_id = varargin {1};
        this.loading = DISTRLOAD(type_load,load,this.el_id);
    elseif type_load == 4
        this.el_id = varargin {1};
        this.loading = FACELOAD(type_load,load,this.el_id);
    end
end
```

Figura 4: Metodi della classe LOAD

All’interno della classe c’è la funzione LOAD che crea l’oggetto assegnando alcuni parametri presi in input in base al tipo di carico, il quale viene opportunamente controllato attraverso una serie di istruzioni if-else. All’interno di queste istruzioni viene creato e salvato l’oggetto relativo allo specifico tipo di carico, trattato come una classe figlia di LOAD, assegnandolo alla variabile *loading*.

Oltre a questa funzione, nella classe sono presenti anche delle funzioni *get* che permettono di accedere ai valori delle proprietà dell’oggetto della classe.

CLASSE NODELOAD (carichi nodali)

```
classdef NODELOAD < handle
    %classe figlia per i carichi nodali (forze e momenti puntuali)
    properties
        node_id;
        type_load;
        load = [];
    end
    methods
        function this = NODELOAD(type_load,load,node_id)
            this.node_id = node_id;
            this.type_load = type_load;
            this.load = load;
        end
    end
end
```

Figura 5: Classe NODELOAD

All'interno della classe figlia NODELOAD si è implementata una *function* che assegna il vettore dei carichi al nodo sul quale è applicato il carico, per il carico nodale non sono infatti necessari altri dati di input.

CLASSE POINTLOAD (carichi concentrati)

```
classdef POINTLOAD < handle
    % classe figlia per i carichi puntuali (forze conc e momenti)
    %-----
    properties
        el_id;          %numero dell' elemento a cui è applicato il carico
        load = [];      %vettore contenete le componenti del carico lungo x,y,z
        a;               %distanza di applicazione del carico puntuale
        type_load;       %tipo di carico
    end
    %-----
    methods
        %-----
        function this = POINTLOAD(type_load,load,el_id,a)
            this.el_id = el_id;
            this.type_load = type_load;
            this.load = load;
            this.a = a;
        end
    end
    %-----
end
```

Figura 6: Classe POINTLOAD

All'interno della classe figlia POINTLOAD si è implementata una *function* che assegna l'identificativo dell'elemento caricato, il vettore dei carichi e anche il valore di a , ovvero la distanza tra il primo nodo dell'elemento e il punto di applicazione del carico.

CLASSE DISTRLOAD (carichi uniformemente distribuiti)

```
classdef DISTRLOAD < handle
    % classe figlia per un carico distribuito
    %-----
    properties
        el_id;          %numero dell' elemento a cui è applicato il carico
        load = [];      %vettore contenete le componenti della pressione lungo x,y,z
        type_load;      %tipo di carico (in questo caso 3)
    end
    %-----
    methods
        function this = DISTRLOAD(type_load,load,el_id)
            this.el_id = el_id;
            this.type_load = type_load;
            this.load = load;
        end
    end
    %-----
end
```

Figura 7: Classe DISTRLOAD

All'interno della classe figlia DISTRLOAD si è implementata una *function* che assegna l'identificativo dell'elemento caricato e il vettore dei carichi.

2.3 ASSEMBLAGGIO DEI CARICHI

Per quanto riguarda l'assemblaggio dei carichi si è implementata una *function* all'interno della classe SOLUTORE che crea il vettore colonna dei carichi F_{ext} di dimensione $this.dim$, che corrisponde a (numero di nodi della struttura)*(gradi di libertà massimi), e verrà utilizzato per il calcolo degli spostamenti nella formula: $u = K^{-1} \cdot F$.

```
function [F_ext]=assemblyF(this)
    totloads = length(this.loads);
    F_ext = zeros(this.dim,1);
    for l=1:totloads
        f_ext = this.loads(l).getload();
        %.....
        if this.loads(l).type_load == 1 %carico nodale
            node_id = this.loads(l).getnodes();
            for dof = 1:length(f_ext)
                F_ext((node_id-1)*this.gdlmax+dof) = F_ext((node_id-1)*this.gdlmax+dof) + f_ext(dof);
            end
        end
    end
```

Figura 8: Assemblaggio carichi, creazione F_{ext} e assemblaggio carichi nodali

Utilizzando un ciclo for, che scorre ogni carico applicato alla struttura, si costruisce il vettore f_{ext} contenente le componenti del carico applicato ad ogni singolo elemento. A seconda del tipo di carico ($type_load$), un costrutto if-else permette la differenziazione dell'assemblaggio.

Per quanto riguarda il carico nodale, identificato dal $type_load = 1$, si è implementato un ciclo for che scorre ogni componente di f_{ext} e la assegna alla corretta posizione all'interno di F_{ext} , grazie al numero identificativo del nodo a cui il carico corrisponde ($node_id$).

Nel caso di carichi applicati sull'elemento *beam*, che siano concentrati ($type_load = 2$) o uniformemente distribuiti ($type_load = 3$), è necessario calcolare le forze equivalenti nodali risultanti dall'applicazione del carico per inserirle all'interno di F_{ext} , poiché nel metodo agli elementi finiti le soluzioni vengono calcolate sui nodi e poi interpolate all'interno dell'elemento.

E' quindi necessario individuare i nodi di estremità di quest'ultimo, attraverso il numero identificativo dell'elemento.

```
%carico concentrato o distribuito sull'elemento beam
elseif this.loads(l).type_load == 2 || this.loads(l).type_load == 3
    el_id = this.loads(l).getelement_id();
    element = this.elements(el_id).getelement();
    X = element.X;
    nodes_id = element.nodes_id;
    node_id_1 = nodes_id(1);
    node_id_2 = nodes_id(2);
    gl = GLOBAL();
    x1 = X(1,:);
    x2 = X(2,:);
    L = gl.distance(x1, x2);
```

Figura 9: Assemblaggio carichi, individuazione elemento e relativi nodi a cui è associato il carico

Come prima cosa, ci si posiziona sul sistema di riferimento locale dell'elemento. Si ottiene, dunque, il vettore delle forze locali applicate all'elemento ruotando il vettore f_{ext} mediante la matrice di rotazione T. Tale matrice quadrata deve avere la dimensione della lunghezza del vettore che si vuol

ruotare, per questo motivo viene costruita con più sottomatrici lambda (calcolate all'interno della classe BEAM3D a partire dalle coordinate dei nodi associati all'elemento) e sottomatrici nulle.

```
%----- passare da forze f_ext globali a locali-----
lambda = element.lambdaForT(X);
T = [lambda zeros(3,3)
     zeros(3,3) lambda];
f_loc = f_ext * T^(-1);
```

Figura 10: Assemblaggio carichi, passaggio da sistema di riferimento globale a locale

Successivamente, per ogni carico, si crea un vettore colonna F_{loc} che conterrà 12 termini, 6 componenti di carico per ognuno dei due nodi dell'elemento.

A questo punto, differenziamo a seconda del tipo di carico che si sta trattando. Avremo infatti delle formule diverse per quanto riguarda il calcolo delle forze equivalenti in caso di carico concentrato o distribuito.

I coefficienti utilizzati nel calcolo delle forze e dei momenti equivalenti (in particolare *coeff1*, *coeff2*, *coeff3* e *coeff4* nel caso di carico concentrato) sono stati presi dal libro “A First Course in the Finite Element Method” di Daryl L. Logan.

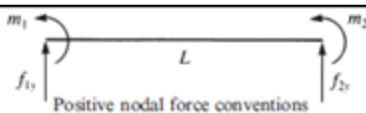
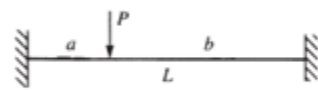
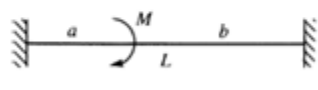
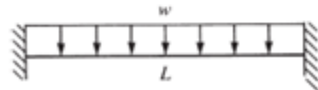
|  | | | | |
|---|------------------------|---|----------------------------------|------------------------|
| f_{1y} | m_1 | Loading case | f_{2y} | m_2 |
| $\frac{-Pb^2(L+2a)}{L^3}$ | $\frac{-Pab^2}{L^2}$ |  | $\frac{-Pa^2(L+2b)}{L^3}$ | $\frac{Pa^2b}{L^2}$ |
| $\frac{-M(a^2+b^2-4ab-L^2)}{L^3}$ | $\frac{Mb(2a-b)}{L^2}$ |  | $\frac{M(a^2+b^2-4ab-L^2)}{L^3}$ | $\frac{Ma(2b-a)}{L^2}$ |
| $\frac{-wL}{2}$ | $\frac{-wL^2}{12}$ |  | $\frac{-wL}{2}$ | $\frac{wL^2}{12}$ |

Figura 11: estratto da “A First Course in the Finite Element” Method, appendice D

Con una serie di istruzioni if-else si gestisce il fatto che forze e momenti applicati su assi diversi generano contributi di forza e momento diversi.

```

F_loc = zeros(12,1); %fx1,fy1,...fx2,fy2...mz2
node_1 = node_id_1;
node_2 = node_id_2;
if this.loads(1).type_load == 2 %pointload
%coefficienti di moltiplicazione per il calcolo dei carichi nodali equivalenti per
%il caso di doppio incastro
a = (this.loads(1).geta()) * L;
b = L - a;
coff1 = [(b^2*(L+2*a))/L^3, (a^2*(L+2*b))/L^3];
coff2 = [(a*b^2)/L^2, -(b*a^2)/L^2];
coff3 = [(a^2+b^2-(4*a*b)-L^2)/L^3, -(a^2+b^2-(4*a*b)-L^2)/L^3];
coff4 = [-(b*(2*a-b))/L^2, -(a*(2*b-a))/L^2];

for dof = 1:length(f_loc)
    f_p = f_loc(dof)*coff1; %forza equivalente dovuta a carico P concentrato
    m_p = f_loc(dof)*coff2; %momento equivalente dovuto a carico P concentrato
    f_M = f_loc(dof)*coff3; %forza equivalente dovuto a momento M concentrato
    m_M = f_loc(dof)*coff4; %momento equivalente dovuto a momento M concentrato
    if dof == 1 %fx -> fx
        F_loc(dof) = F_loc(dof) + f_p(1); %nodo1
        F_loc(dof+6) = F_loc(dof+6) + f_p(2); %nodo2
    elseif dof == 2 %fy -> fy,mz
        %nodo 1
        F_loc(dof) = F_loc(dof) + f_p(1);
        F_loc(dof+4) = F_loc(dof+4) + m_p(1);
        %nodo 2
        F_loc(dof+6) = F_loc(dof+6) + f_p(2);
        F_loc(dof+10) = F_loc(dof+10) + m_p(2);
    elseif dof == 3 %fz -> fz,my
        %nodo 1
        F_loc(dof) = F_loc(dof) + f_p(1);
        F_loc(dof+2) = F_loc(dof+2) - m_p(1);
        %nodo 2
        F_loc(dof+6) = F_loc(dof+6) + f_p(2);
        F_loc(dof+8) = F_loc(dof+8) - m_p(2);
    elseif dof == 5 %my -> fz,my
        %nodo 1
        F_loc(dof-2) = F_loc(dof-2) - f_M(1);
        F_loc(dof) = F_loc(dof) + m_M(1);
        %nodo 2
        F_loc(dof+4) = F_loc(dof+4) - f_M(2);
        F_loc(dof+6) = F_loc(dof+6) + m_M(2);
    elseif dof == 6 %Mz -> fy,mz
        %nodo 1
        F_loc(dof-4) = F_loc(dof-4) + f_M(1);
        F_loc(dof) = F_loc(dof) + m_M(1);
        %nodo 2
        F_loc(dof+2) = F_loc(dof+2) + f_M(2);
        F_loc(dof+6) = F_loc(dof+6) + m_M(2);
    end
end
end

```

Figura 12: assemblaggio dei carichi, creazione F_loc e calcolo forze equivalente nel caso di carico concentrato

```

elseif this.loads(1).type_load == 3 %distrload
    for dof = 1:length(f_loc)
        %forza equivalente dovuta a carico q distribuito
        f_q = f_loc(dof)*[L/2;L/2];
        %momento equivalente dovuto a carico q distribuito
        m_q = f_loc(dof)*[L^2/12;-L^2/12];
        if dof == 1 %qx -> fx
            F_loc(dof) = F_loc(dof) + f_q(1); %nodo1
            F_loc(dof+6) = F_loc(dof+6) + f_q(2); %nodo2
        elseif dof == 2 %qy -> fy, Mz
            %nodo 1
            F_loc(dof) = F_loc(dof) + f_q(1);
            F_loc(dof+4) = F_loc(dof+4) + m_q(1);
            %nodo 2
            F_loc(dof+6) = F_loc(dof+6) + f_q(2);
            F_loc(dof+10) = F_loc(dof+10) + m_q(2);
        elseif dof == 3 %qz -> fz, my
            %nodo 1
            F_loc(dof) = F_loc(dof) + f_q(1);
            F_loc(dof+2) = F_loc(dof+2) - m_q(1);
            %nodo 2
            F_loc(dof+6) = F_loc(dof+6) + f_q(2);
            F_loc(dof+8) = F_loc(dof+8) - m_q(2);
        end
    end
end
end
end

```

Figura 13: assemblaggio carichi, calcolo forze equivalenti nel caso di carico distribuito

Una volta calcolato il vettore F_loc , contenente le risultanti delle forze e dei momenti sui nodi associati all'elemento per un determinato carico, si ritorna al sistema di riferimento globale per poter assemblare tutti i carichi in F_ext .

Impieghiamo una matrice di rotazione T_1 di dimensione maggiore rispetto alla precedente, in quanto F_loc è di dimensione doppia rispetto a f_ext , ma costruita con lo stesso principio della precedente.

Come per i carichi nodali, anche per quelli applicati sul *beam*, vengono inseriti i carichi contenuti in F_gl nel posto corretto di F_ext facendo riferimento all'id del nodo.

```

%portare F_loc sul sistema di riferimento globale
T_1 = [lambda zeros(3,9)
        zeros(3) lambda zeros(3,6)
        zeros(3,6) lambda zeros(3)
        zeros(3,9) lambda];
F_gl = T_1' * F_loc;

%inserire valori F_gl in F_ext
for dof = 1:(length(F_gl)/2)
    %nodo1
    F_ext((node_1-1)*this.gdlmax+dof) = F_ext((node_1-1)*this.gdlmax+dof) + F_gl(dof);
    %nodo2
    F_ext((node_2-1)*this.gdlmax+dof) = F_ext((node_2-1)*this.gdlmax+dof) + F_gl(dof+6);
end

```

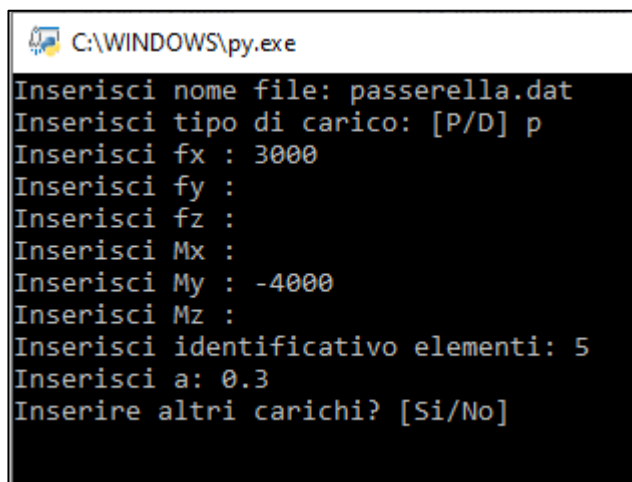
Figura 14: assemblaggio carichi, ritorno al sistema di riferimento globale e costruzione di F_ext

2.4 PROGRAMMA PYTHON

Dal momento che con il *problemtype* fornito per il preprocessore di GiD non è possibile inserire i dati relativi ai carichi puntuali e distribuiti su un elemento *beam*, si è pensato di creare un piccolo programma eseguibile in Python che, a partire dai dati sui carichi in input, aggiunge delle righe al file '.dat' creato con il preprocessore di GiD.

Il programma richiede in input: il nome del file, precedentemente creato con il preprocessore di GiD e al quale si vogliono aggiungere dei nuovi carichi; il tipo di carico che si vuole applicare (puntuale/distribuito) e in base alla risposta fornita sono richieste le informazioni attinenti:

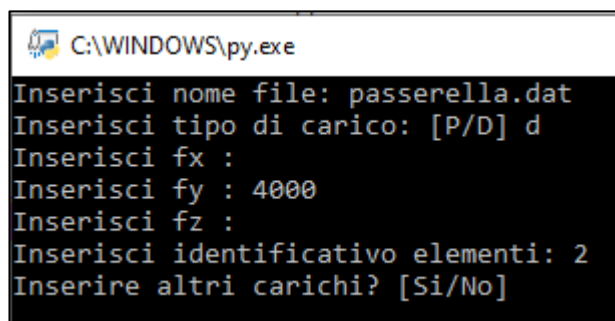
- Nel caso di carico puntuale, avendo implementato nel codice Matlab sia l'applicazione delle forze che quella dei momenti, vengono chiesti i valori della forza e del momento nelle tre direzioni (x,y,z); è necessario inserire anche *a* (distanza di applicazione del carico sull'elemento beam) e il numero identificativo dell'elemento a cui il carico è associato;



```
C:\WINDOWS\py.exe
Inserisci nome file: passerella.dat
Inserisci tipo di carico: [P/D] p
Inserisci fx : 3000
Inserisci fy :
Inserisci fz :
Inserisci Mx :
Inserisci My : -4000
Inserisci Mz :
Inserisci identificativo elementi: 5
Inserisci a: 0.3
Inserire altri carichi? [Si/No]
```

Figura 15: Finestra esecuzione del programma, esempio di applicazione di un carico puntuale

- Nel caso di carico distribuito, sono invece richieste solo le forze nelle tre direzioni (x,y,z) e il numero dell'elemento su cui è applicato il carico.



```
C:\WINDOWS\py.exe
Inserisci nome file: passerella.dat
Inserisci tipo di carico: [P/D] d
Inserisci fx :
Inserisci fy : 4000
Inserisci fz :
Inserisci identificativo elementi: 2
Inserire altri carichi? [Si/No]
```

Figura 16: Finestra d'esecuzione del programma, esempio di applicazione di un carico distribuito

Le richieste sono gestite da un ciclo *while* che viene interrotto nel momento in cui si risponde alla domanda "Inserire altri carichi?" in modo negativo.

```

def ADDLOAD():
    file = str(input("Inserisci nome file: "))

    to_request = ["fx" , "fy", "fz", "Mx", "My", "Mz"]

    stop = False
    loads = []
    while not stop:
        load = ""
        type_load = str(input("Inserisci tipo di carico: [P/D] ")).lower()
        for req in to_request:
            value = str(input("Inserisci " + req + " : "))
            if value == "":
                value = "0"
            value = "{0:.7e}".format(float(value))
            load = load + 5*" " + value
            if type_load == "d" and req == "fz" :
                load = load + 3*(5*" " + "{0:.7e}".format(0))
                break
        el_id = str(input("Inserisci identificativo elementi: "))
        a = ''
        n_load = str(3)    #carico distribuito d
        if type_load == "p":
            a = str(input("Inserisci a: " ))
            n_load = str(2)
        data = 4*' ' + n_load + load + 6*" " + el_id + 7*" " + a + "\n"
        loads.append(data)
        if str(input("Inserire altri carichi? [Si/No] ")).lower() == "no":
            stop = True

```

Figura 17: Python, prima parte di codice del programma

Alla fine di questa prima parte di codice vengono create all'interno della lista *loads* le nuove righe (formattate nel modo in cui il programma Matlab non avrà problemi a leggerle) da inserire nel file '.dat'.

La seconda parte del programma si occupa della lettura del file, di cui si è fornito il nome, al fine di memorizzarne il contenuto all'interno della lista *l*. Tale lista sarà costituita da due stringhe che conterranno rispettivamente tutto il testo prima e dopo della parola *loads* presente nel documento letto.

```

infile = open(file,"r",encoding="UTF-8")
lista = infile.readlines()
text_file = ''
for i in lista:
    text_file = text_file + i
infile.close()
l = text_file.split("loads")

```

Figura 18: Python, seconda parte di codice del programma

La terza e ultima parte del codice si occupa della creazione di un nuovo file che contenga tutte le informazioni prese dal documento letto nella seconda parte del programma illustrata e le nuove righe create nella prima parte del programma.

```
outfile = open("new_" + file, "w", encoding="UTF-8")
l[0] = l[0] + "loads\n"
outfile.write(l[0])
l[1] = l[1].rstrip('\n')
l_righe = l[1].split('\n')
for i in range(0, len(l_righe)-2):
    outfile.write(l_righe[i] + '\n')
for new_r in loads:
    outfile.write(new_r)
outfile.write("end")
outfile.close()
```

ADDLOAD()

Figura 19: Python, terza parte di codice del programma

3 TEST

Con lo scopo di verificare l'accuratezza dei risultati derivanti dal codice sviluppato, si sono eseguiti una serie di test di confronto mediante il programma commerciale Straus7. Si sono eseguiti più test per valutare tutti gli aspetti del codice singolarmente, una volta confermata l'accuratezza dei risultati si è scelto di modellare una struttura più complessa che comprendesse tutte le funzionalità precedentemente testate.

3.1 CAPRIATA 2D

Nel primo test si è scelto di collaudare gli elementi *truss*, sia diritti che inclinati, soggetti a forze nodali di diversa entità e direzione di applicazione. Si è pensato dunque di modellare una semplice capriata all'inglese bidimensionale in Straus7 (Fig. 21).

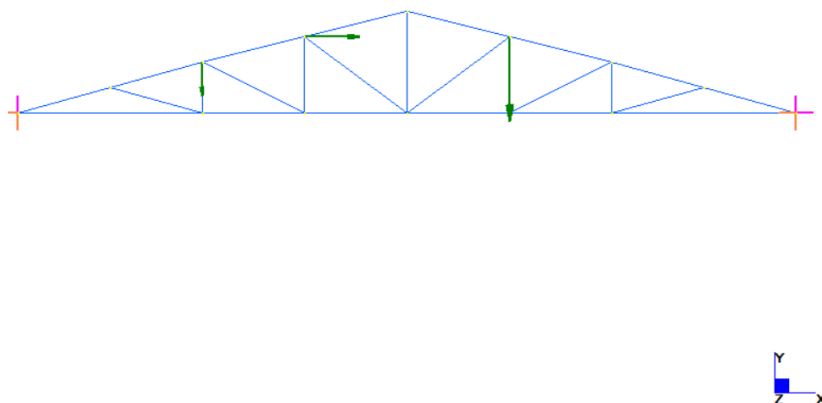


Figura 21: Straus7, capriata comprensiva di carichi e vincoli

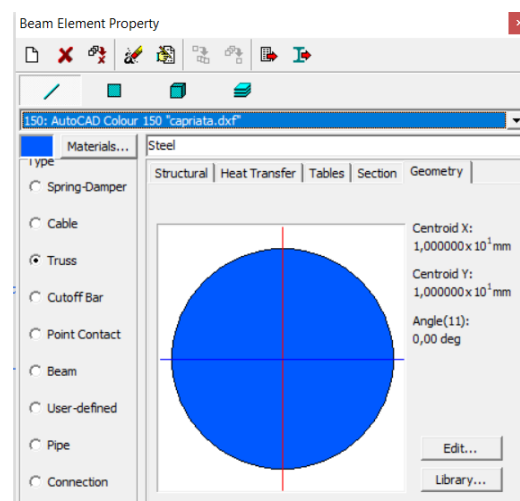


Figura 20: Straus7, proprietà dell'elemento truss

Per quanto riguarda il preprocessamento, si è utilizzato il software GiD per definire la mesh, comprensiva di vincoli e carichi (Fig. 22), e creare così il file con estensione '.dat' dal quale partire per calcolare le soluzioni all'interno del codice. GiD è stato poi impiegato anche in fase di postprocessamento per la visualizzazione dei risultati nella mesh.

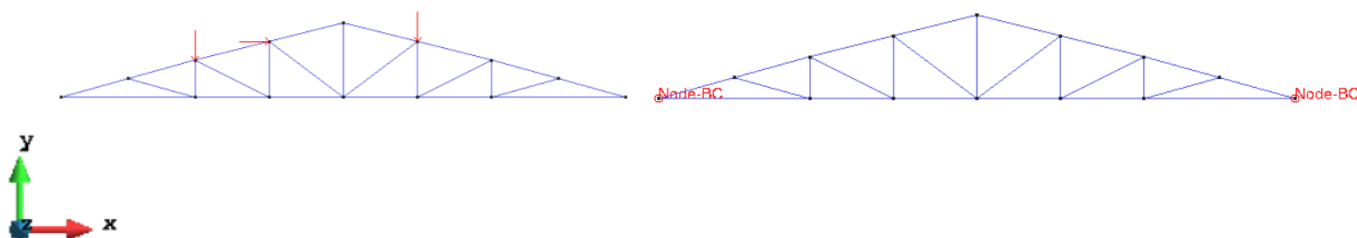


Figura 22: GiD, capriata comprensiva di carichi (sinistra) e vincoli (destra)

Dal confronto con i risultati in termini di spostamento ottenuti con Straus7, si può concludere che il codice implementato funziona correttamente sia per i risultati in direzione X che per i risultati in direzione Y, con un errore dell'ordine di 10^{-2} .

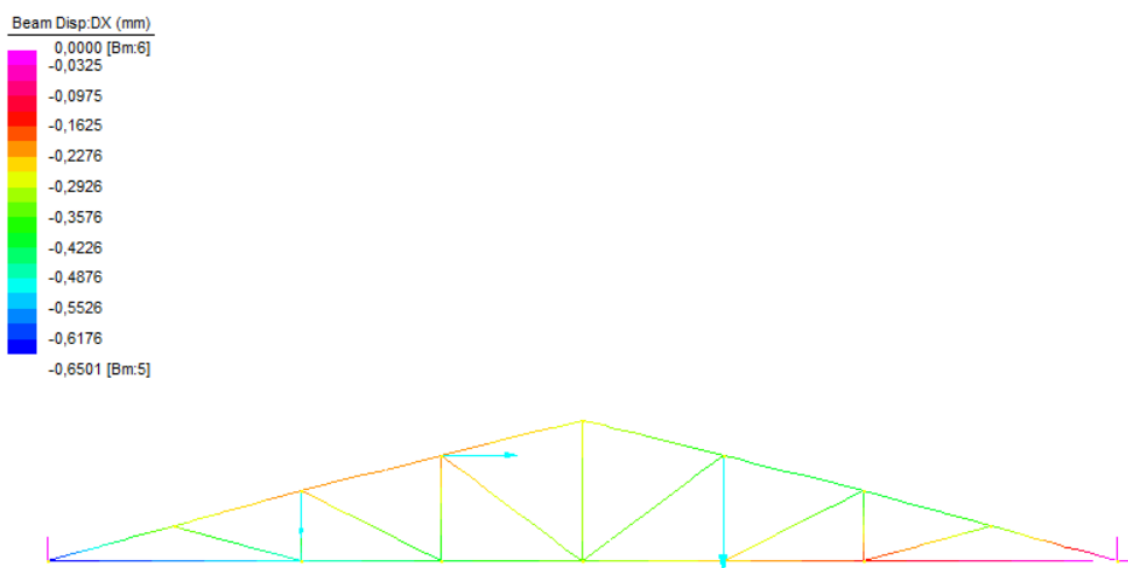


Figura 24: Straus7, contour dei risultati relativamente allo spostamento lungo x

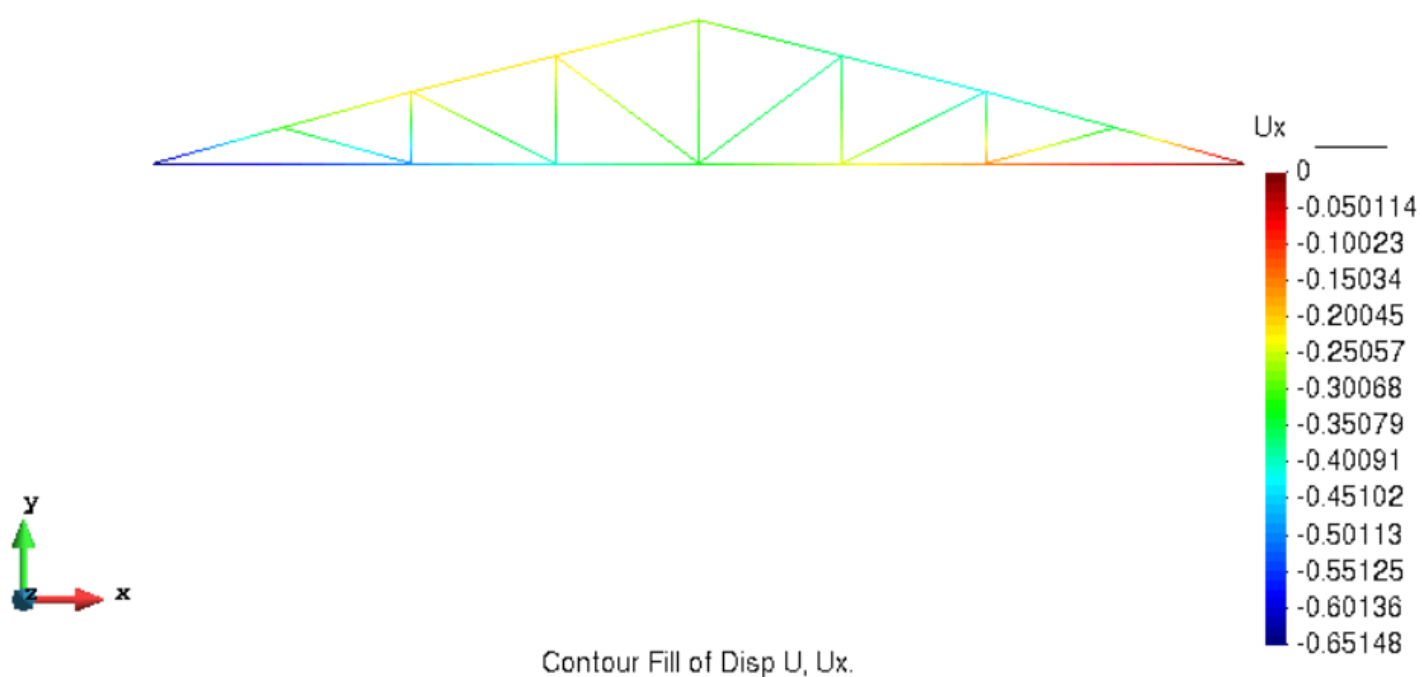


Figura 23: GiD, contour dei risultati relativamente allo spostamento lungo x

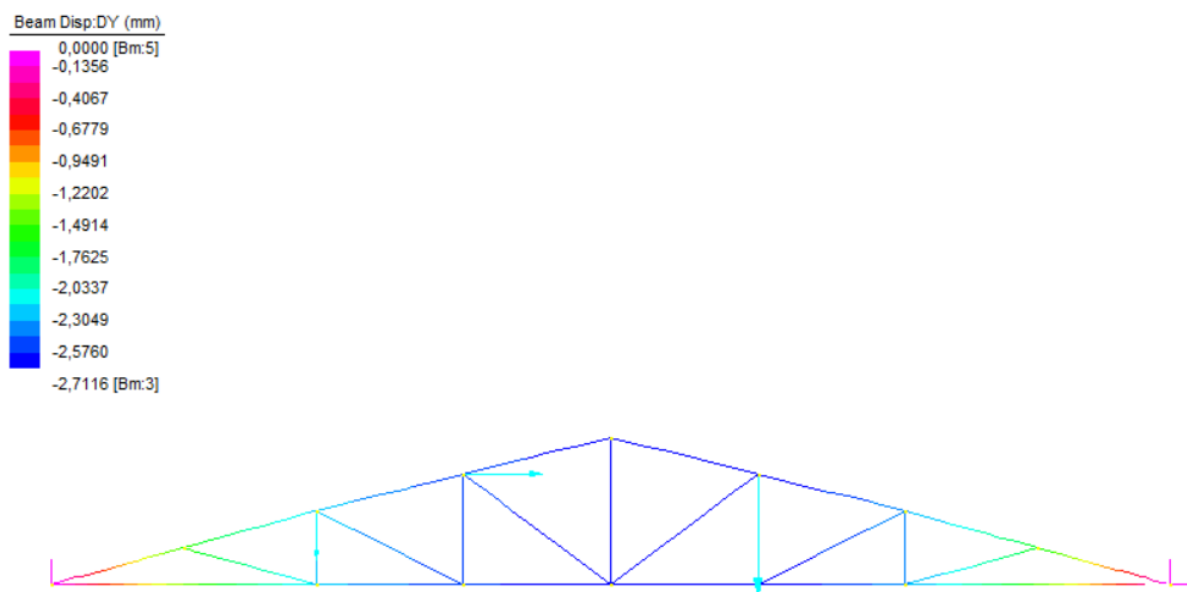


Figura 25: Straus7, contour dei risultati relativamente allo spostamento lungo y

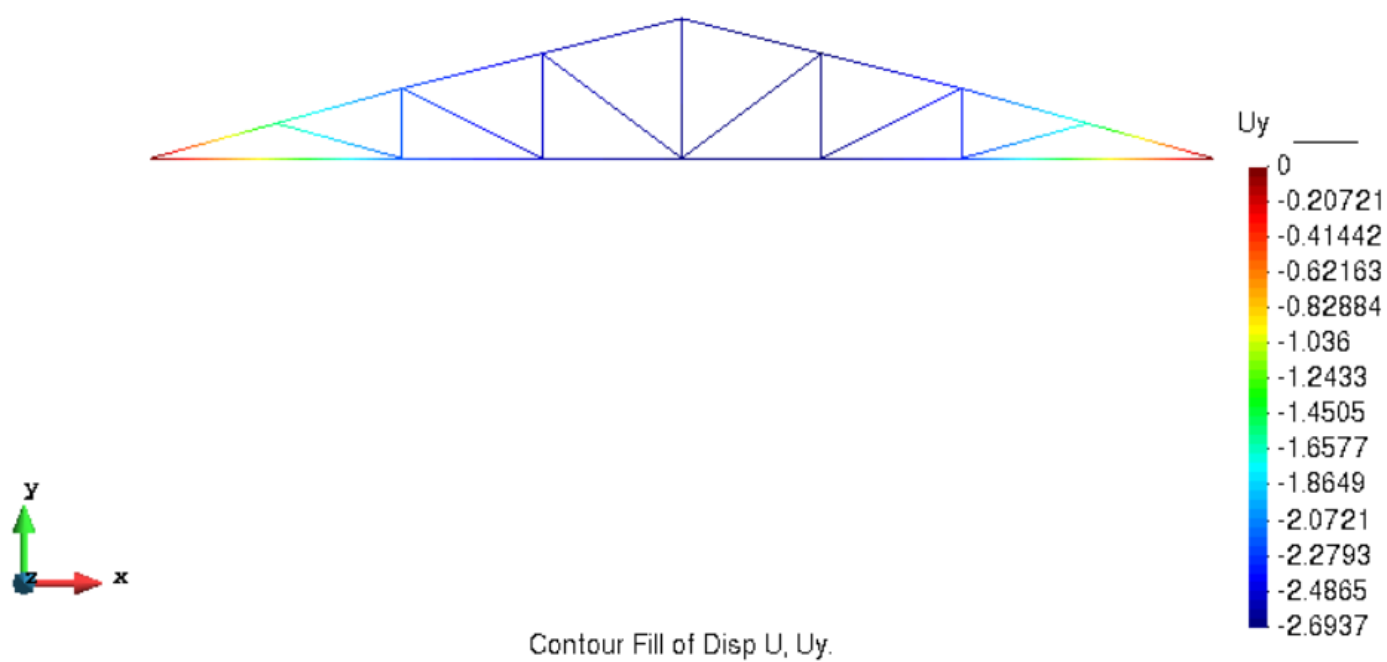


Figura 26: GiD, contour dei risultati relativamente allo spostamento lungo y

3.2 SUPPORTO FOTOVOLTAICO 3D

Il secondo test è stato pensato per collaudare nuovamente i *truss*, ma all'interno di una struttura 3D, in particolare la struttura di supporto per l'impianto fotovoltaico in Figura 27.



Figura 27: supporto impianto fotovoltaico

Il modello è stato riprodotto sul programma agli elementi finiti Straus7 e su GiD.

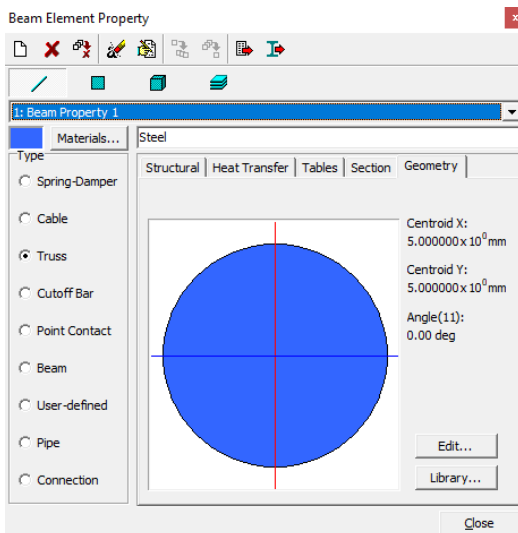


Figura 28: caratteristiche sezione della struttura

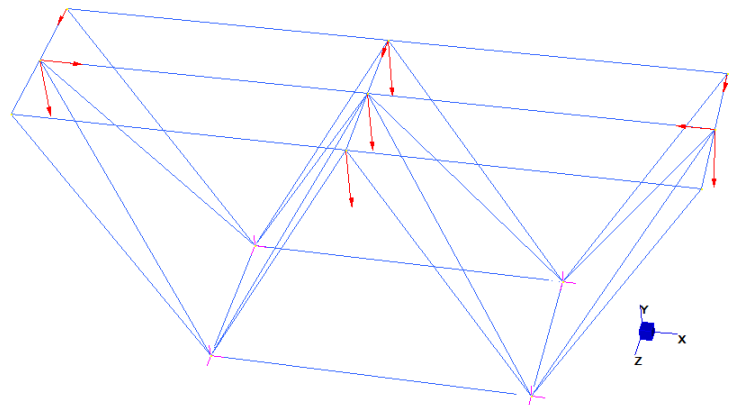


Figura 29: modello Straus7 vincolato e caricato con carichi nodali.

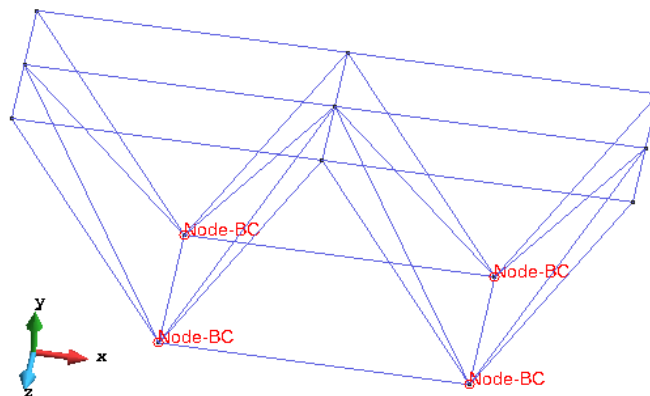


Figura 30: modello GiD con indicazioni sui nodi incastrati.

Dal confronto tra i risultati in termini di spostamento ottenuti da Straus7 e dal programma agli elementi finiti implementato (che visualizziamo sul postprocessore di GiD), è possibile osservare che i risultati sono pressoché corrispondenti nelle tre direzioni x,y e z. E' quindi possibile confermare il funzionamento degli elementi truss, dei carichi nodali e dell'assemblaggio dei carichi.

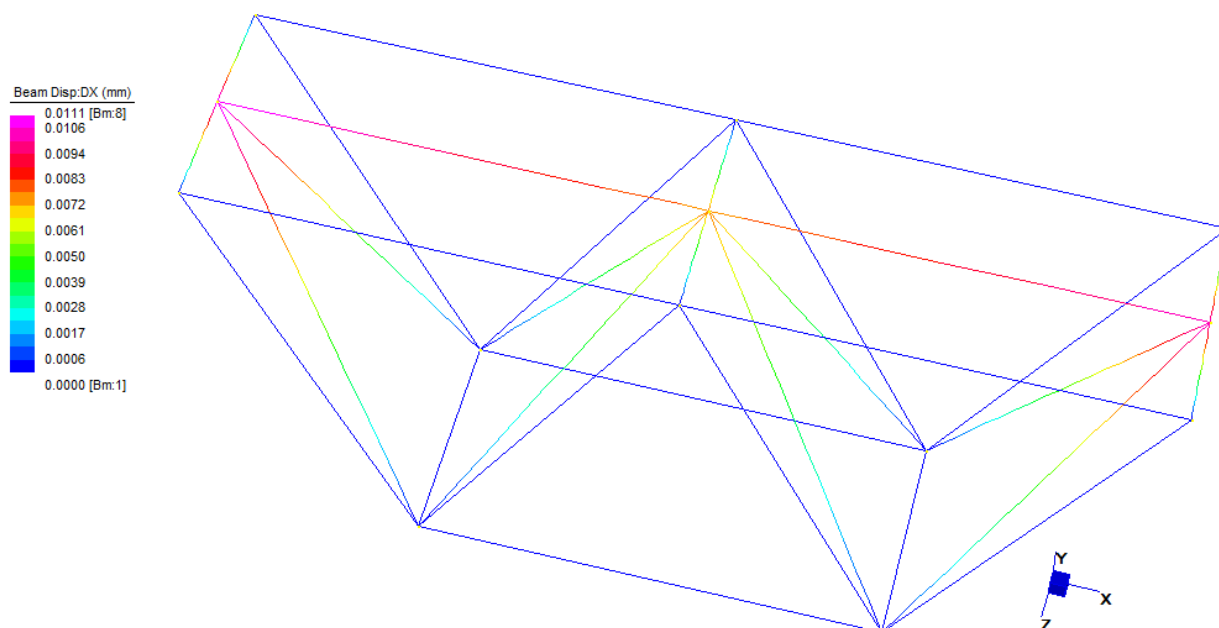


Figura 31: Straus7, contour relativo agli spostamenti lungo x

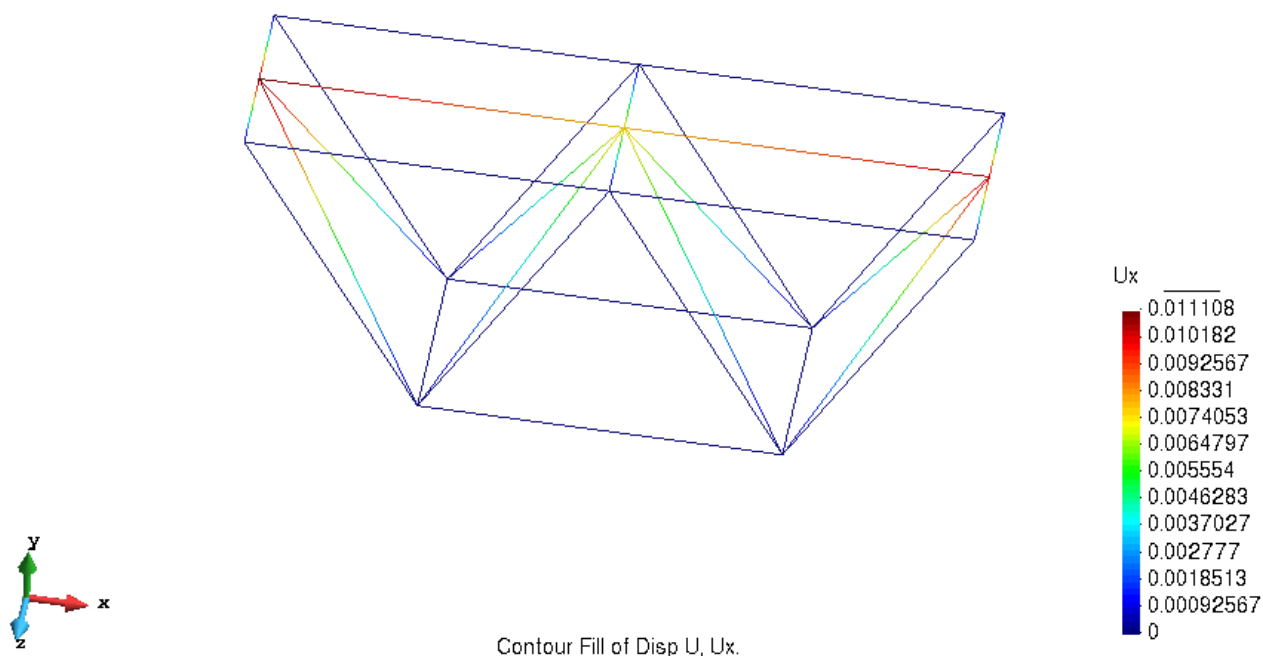


Figura 32: GiD, contour relativo agli spostamenti lungo x

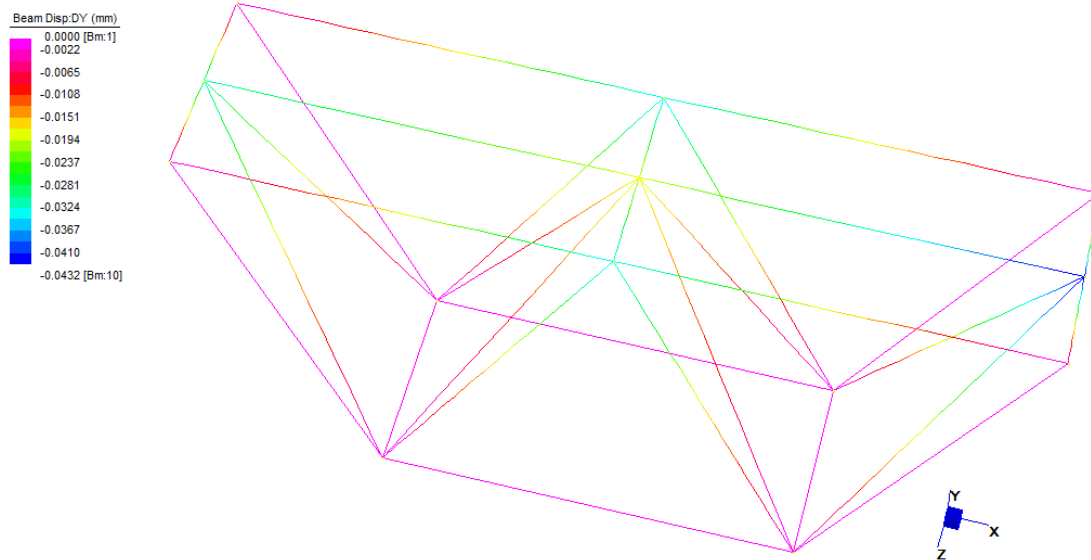


Figura 33: Straus7, contour relativo agli spostamenti lungo y

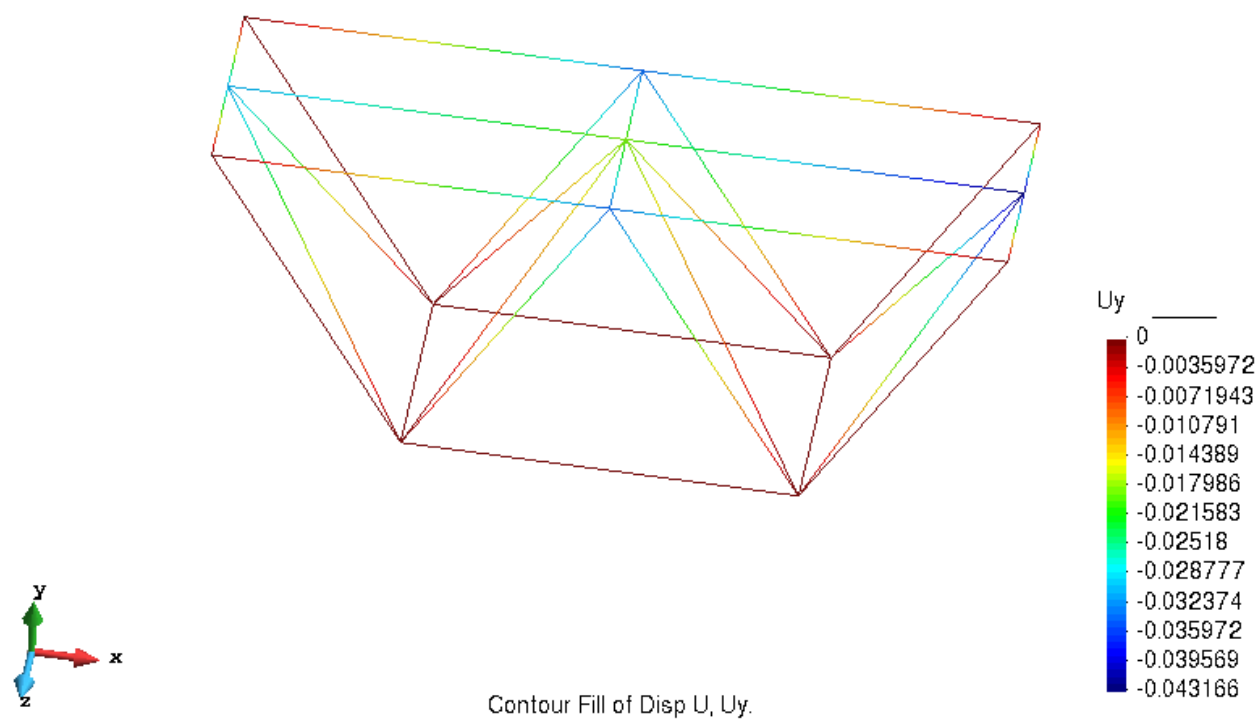


Figura 34: GiD, contour relativo agli spostamenti lungo y

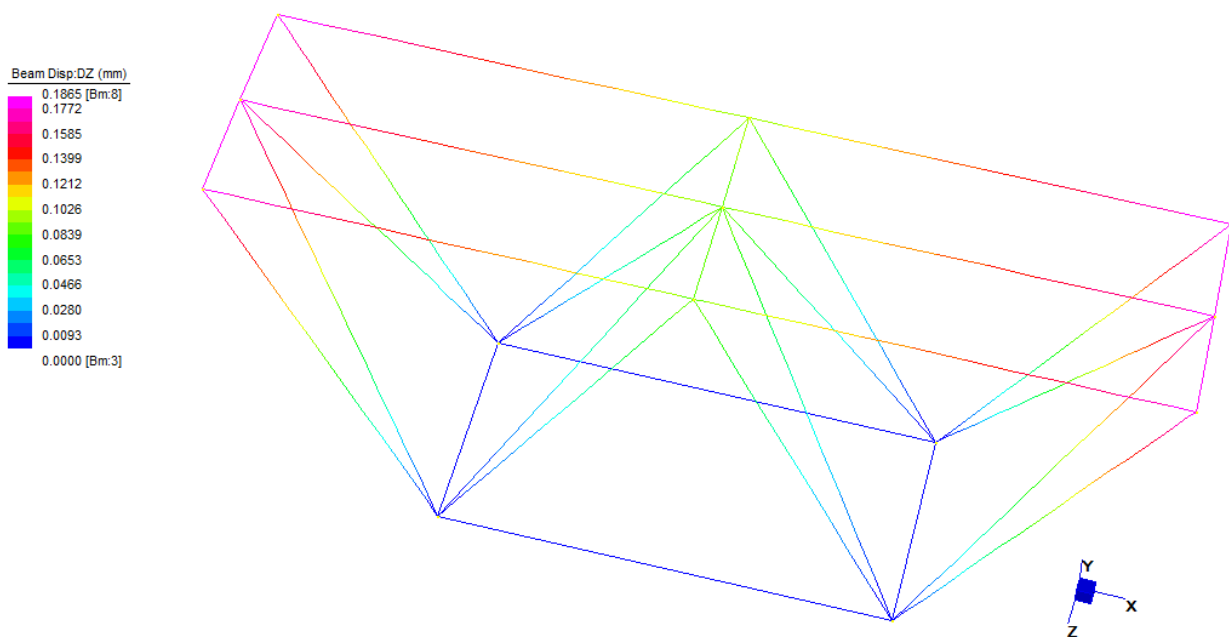


Figura 35: Straus7, contour relativo agli spostamenti lungo z

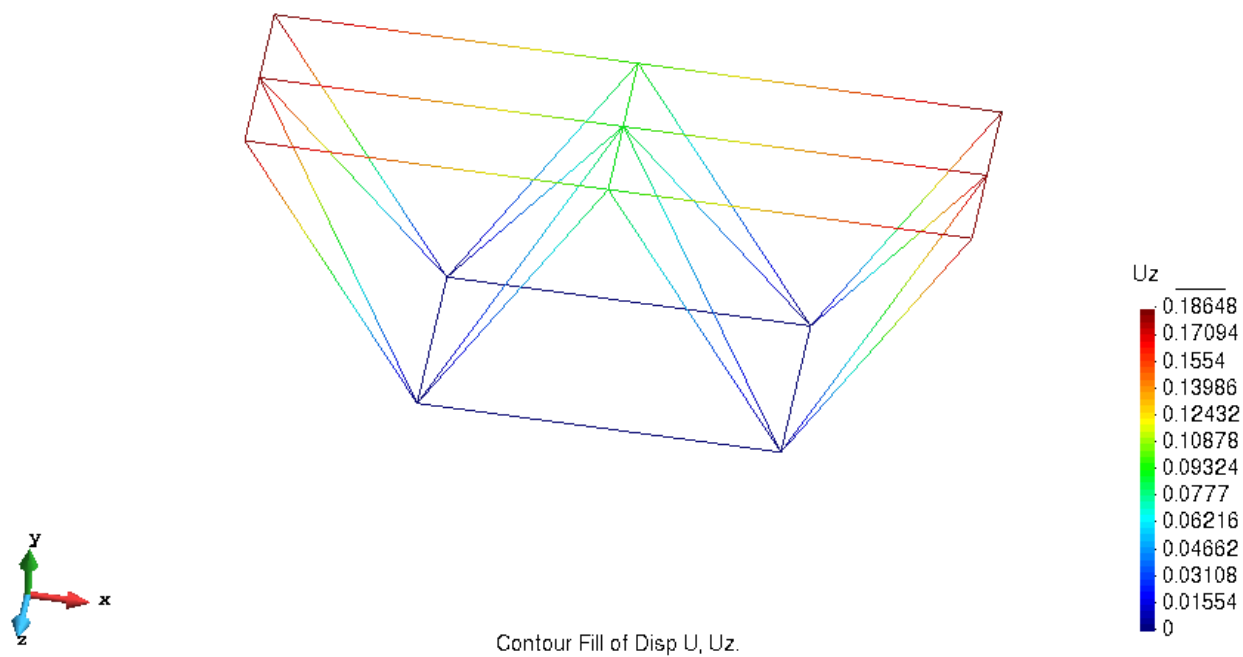


Figura 36: GiD, contour relativo agli spostamenti lungo z

3.3 CARICHI NODALI MISTI

Con lo scopo di testare anche i momenti nodali, oltre alle forze già utilizzate nei test precedenti, si è realizzato un semplice telaio 3D composto però da elementi *beam*, invece che elementi *truss*, in modo da tener conto anche del comportamento flessionale della struttura.

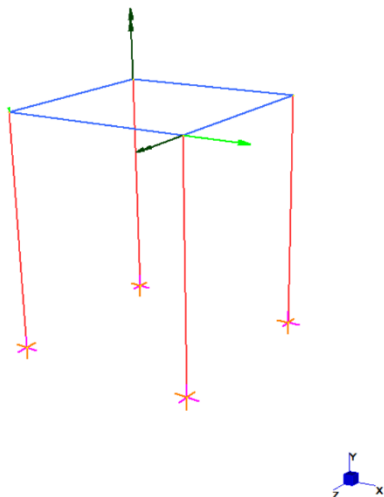


Figura 38: Straus7, telaio 3D

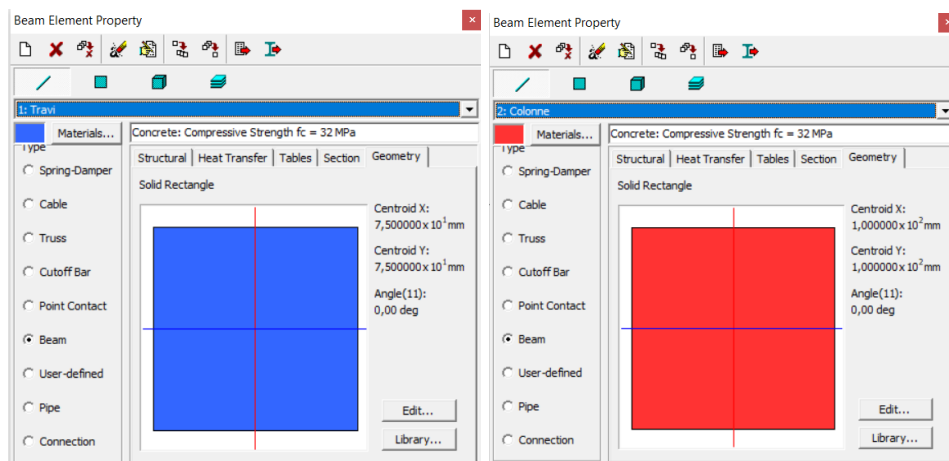
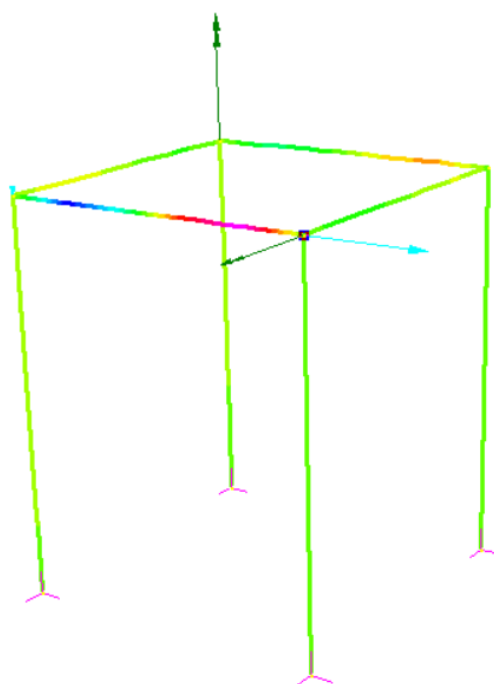
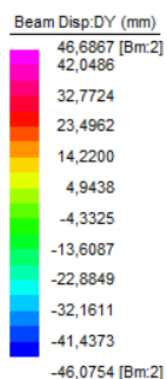


Figura 37: Straus7, proprietà degli elementi beam

La struttura è incastrata alla base e soggetta a carichi nodali di diversa entità, agenti in più punti della struttura come si vede in figura 38.



test_box_F-MNOD Nodes:1: Load Case 1

| Node | 3 | DX (mm) | 678,7552 |
|--------------|---|--------------|-----------|
| Quantity | | DY (mm) | -1,6843 |
| Displacement | | DZ (mm) | -97,1317 |
| | | RX (deg) | -1,9205 |
| | | RY (deg) | 8,1631 |
| | | RZ (deg) | -14,2931 |
| | | D(XY) (mm) | 678,7573 |
| | | D(YZ) (mm) | 97,1463 |
| | | D(ZX) (mm) | 685,6699 |
| | | D(XYZ) (mm) | 685,6720 |
| | | R(XY) (deg) | 8,3860 |
| | | R(YZ) (deg) | 16,4599 |
| | | R(ZX) (deg) | 14,4216 |
| | | R(XYZ) (deg) | 16,5716 |
| | | X | 2000,0000 |
| | | Y | 2500,0000 |
| | | Z | 0,0000 |
| | | X + DX | 2678,7552 |
| | | Y + DY | 2498,3157 |
| | | Z + DZ | -97,1317 |

Figura 39: Straus7, spostamenti e rotazioni relative al nodo 3 (ovvero il nodo 7 in GiD)

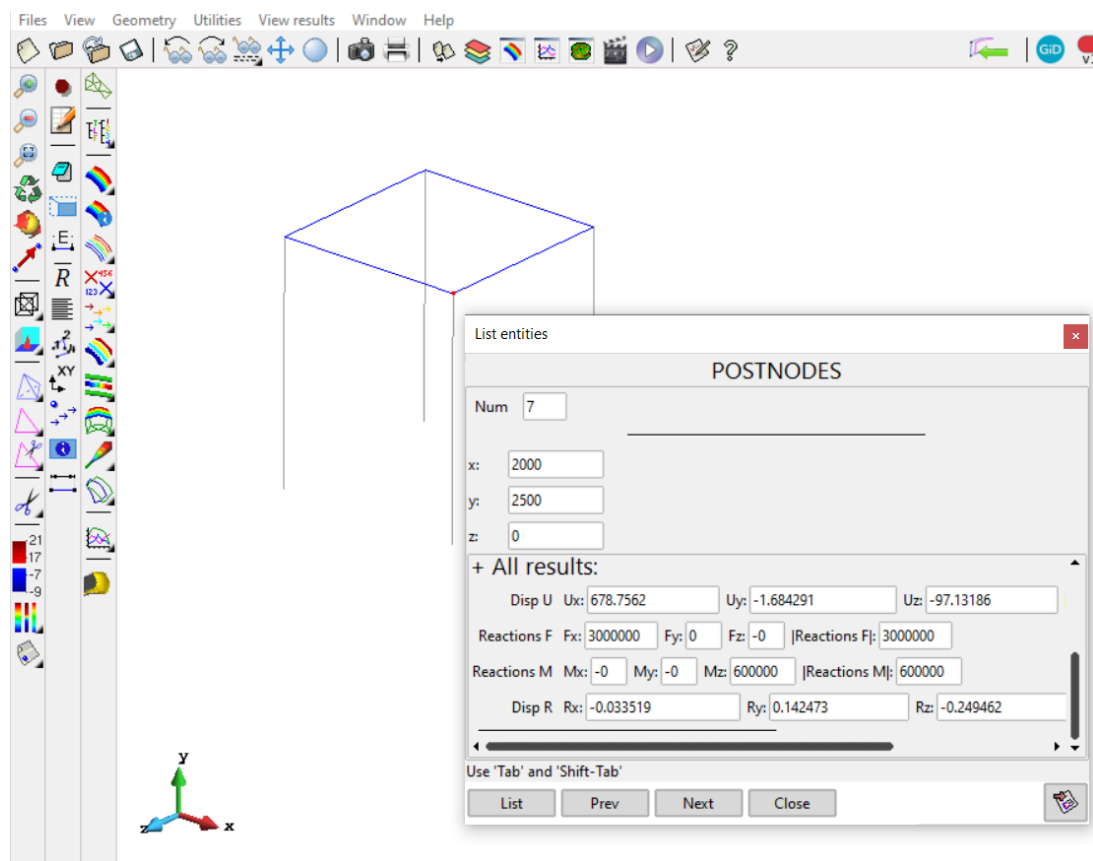


Figura 40: GiD, spostamenti e rotazioni relative al nodo 7

Dall'analisi dei risultati derivanti da Straus7 e da GiD si può concludere che anche il momento nodale è implementato correttamente, ma soprattutto che l'assemblaggio dei carichi avviene esattamente anche nel caso del nodo 7 soggetto sia ad una forza che ad un momento nodali. Gli spostamenti calcolati sui nodi risultano congruenti, con un errore dell'ordine di 10^{-2} , mentre la soluzione all'interno dell'elemento *beam* non è precisa in quanto all'interno di GiD le soluzioni nodali sono interpolate linearmente, mentre Straus7 calcola la soluzione esatta in ogni punto dell'elemento. Per quanto riguarda le rotazioni, anch'esse sono esatte solo che in GiD sono espresse in radianti.

| | STR | GID |
|----|----------|----------|
| RX | -1,9205 | -0,03352 |
| RY | 8,1631 | 0,142473 |
| RZ | -14,2931 | -0,24946 |

Le medesime considerazioni valgono per le soluzioni in tutti i nodi della struttura, che sono state anch'esse confrontate con gli spostamenti calcolati in Straus7.

3.4 ELEMENTI INCLINATI

Con lo scopo di testare le matrici di rotazione nell'assemblaggio di carichi, anche nel caso di un elemento *beam* inclinato e sottoposto a carico concentrato, si è pensato di fare un test sul box in figura 42, al quale sono stati applicati sia forze che momenti concentrati.

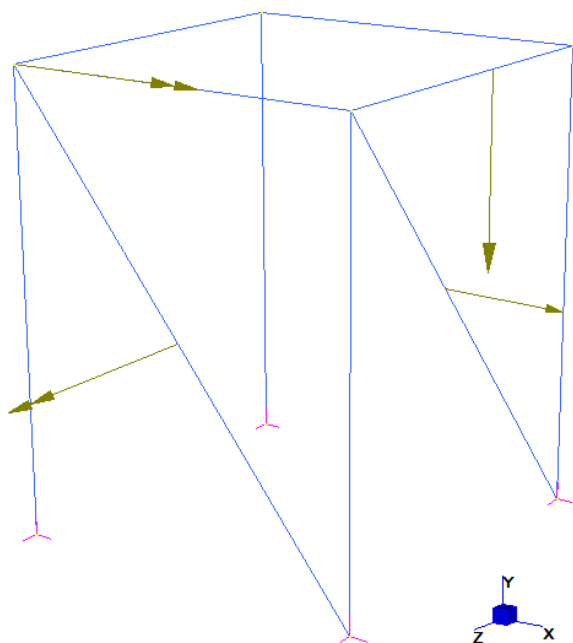


Figura 42: Straus7, box caricato e vincolato

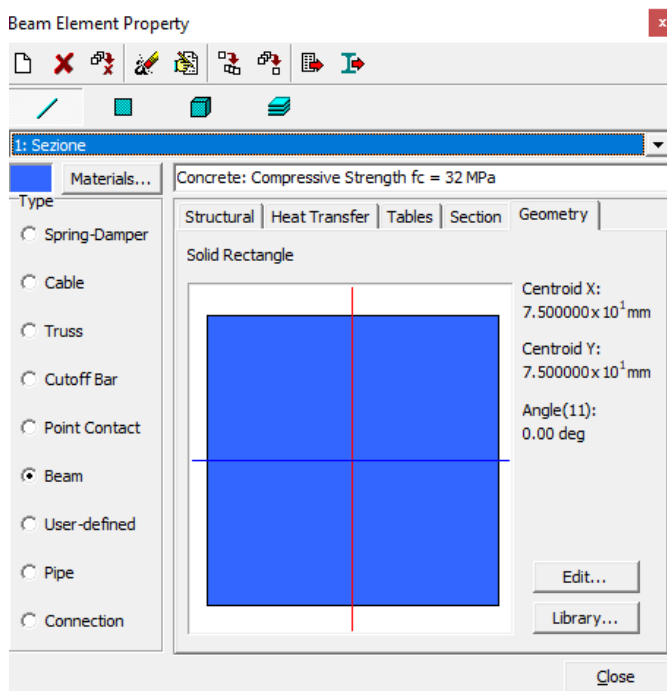


Figura 41: Straus7, caratteristiche sezione box

Come già detto, con il preprocessore di GiD non è possibile aggiungere questo tipo di carico, quindi dopo aver creato con questo il modello della struttura e aver inserito i vincoli si passa ad aggiungere delle righe al file '.dat' con lo script Python creato. A titolo di esempio si riporta di seguito la finestra d'esecuzione del programma e la parte del file '.dat' relativa ai carichi creata.

The image shows a Windows desktop with two open windows. The background window is a Python script execution window titled 'C:\WINDOWS\py.exe'. It displays a series of prompts for user input, which have been answered. The prompts include file name, load type, force components (fx, fy, fz), moment components (Mx, My, Mz), number of elements, and a parameter 'a'. The user has entered 'test_box2.dat' as the file name, 'p' for point load, and various numerical values for the other parameters. The foreground window is a Notepad application titled 'new_test_box2.dat - Blocco note di Windows'. It contains the output of the Python program, which is a list of data points for each element. The data is formatted as follows:

```
end
loads
  2  2.5000000e+07  0.0000000e+00  0.0000000e+00
0.0000000e+00  0.0000000e+00  0.0000000e+00  10  0.4
  2  0.0000000e+00  0.0000000e+00  0.0000000e+00
2.0000000e+06  0.0000000e+00  0.0000000e+00  2  0
  2  0.0000000e+00  0.0000000e+00  0.0000000e+00
0.0000000e+00  0.0000000e+00  4.0000000e+04  9  0.5
  2  0.0000000e+00 -5.0000000e+07  0.0000000e+00
0.0000000e+00  0.0000000e+00  0.0000000e+00  8  0.6
end
```

Figura 43: finestra d'esecuzione del programma Python per la creazione delle stringhe relative all'applicazione dei carichi sul file '.dat'

Si riportano di seguito i risultati per quanto riguarda gli spostamenti e le rotazioni in x,y e z in due punti della struttura, al fine di poter concludere che non ci sono differenze tra i risultati ottenuti con Straus7 e il codice implementato, e quindi l'assemblaggio delle forze risulta esatto.

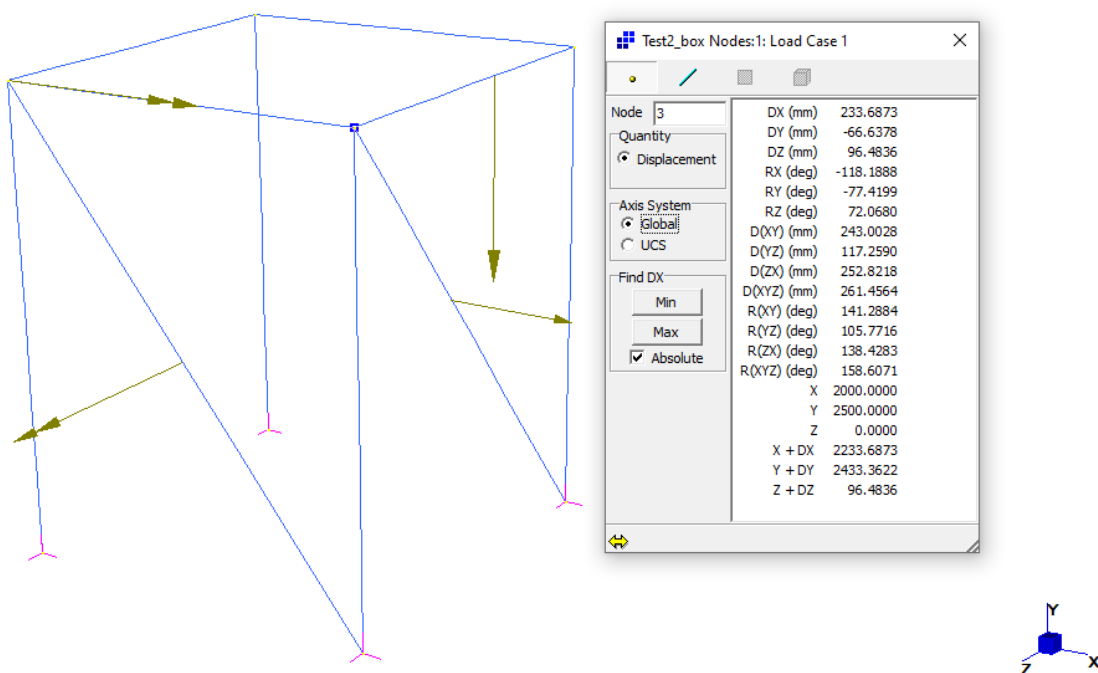


Figura 44: Straus7, spostamenti e rotazioni relative al primo nodo confrontato

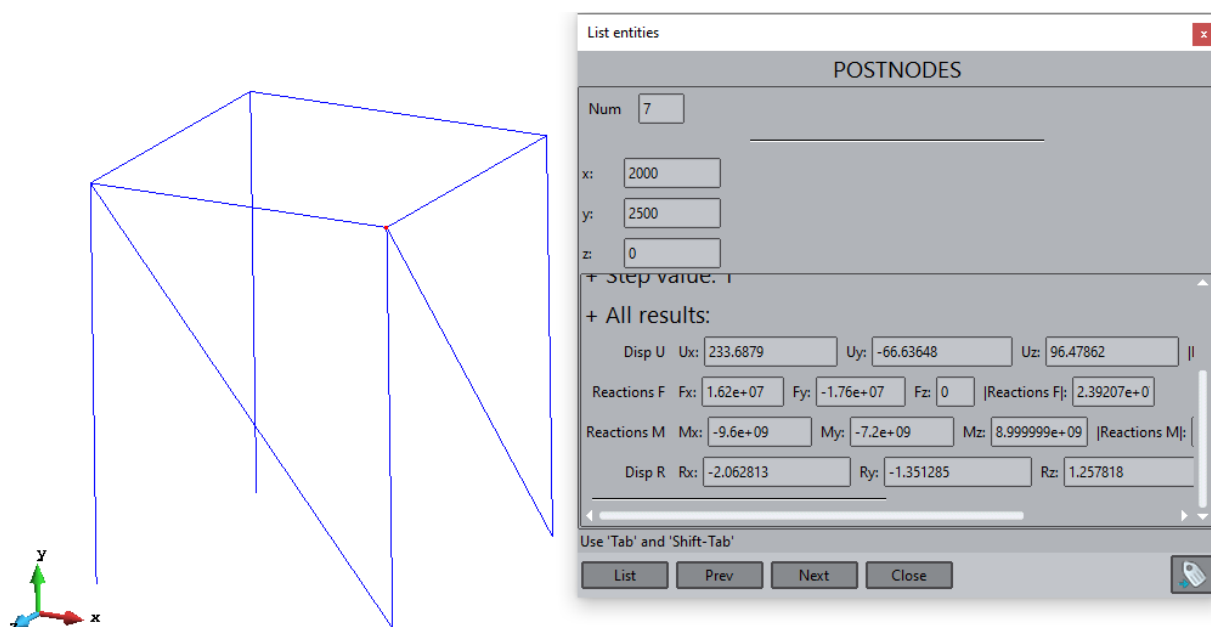


Figura 45: GiD, spostamenti e rotazioni relative al primo nodo confrontato

Al fine di un confronto immediato tra le rotazioni su Straus7 e GiD si è riportato in tabella la conversione delle rotazioni su GiD da radianti a gradi.

| | GiD (rad) | deg |
|----|-----------|-----------|
| RX | -2.06E+00 | -118.1905 |
| RY | -1.35E+00 | -77.4229 |
| RZ | 1.257818 | 72.06766 |

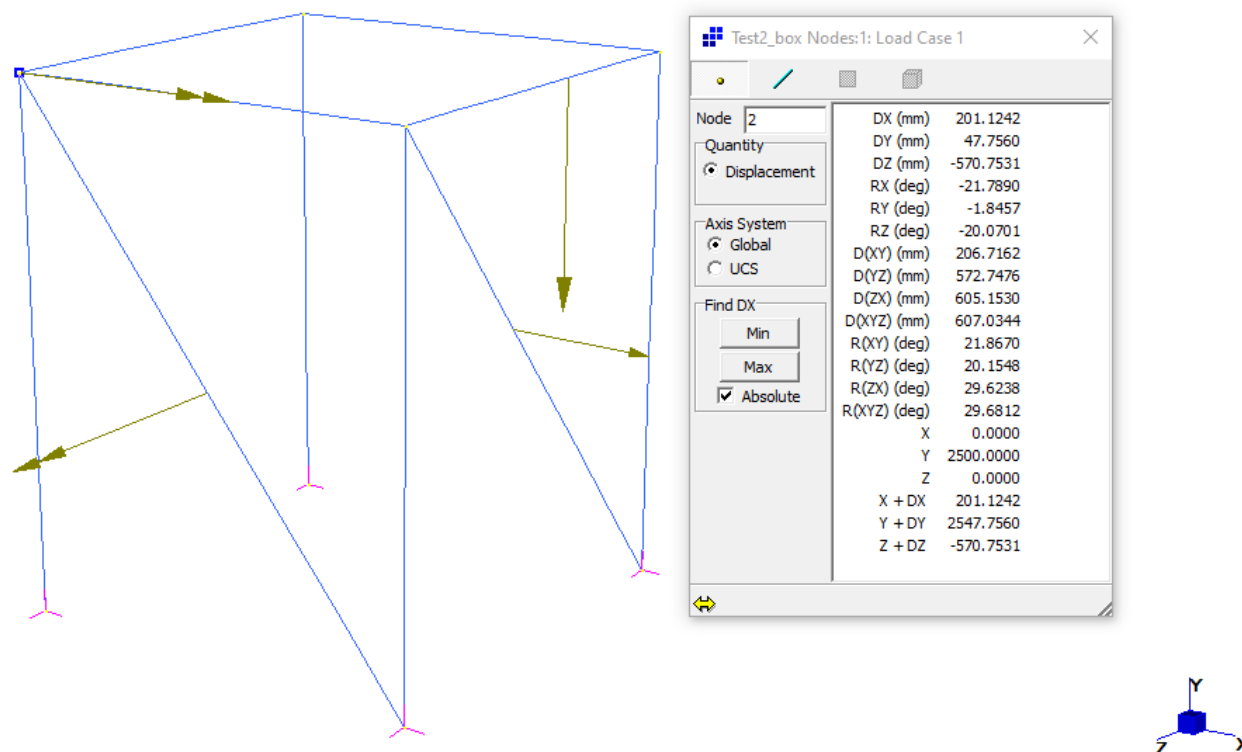


Figura 46: Straus7, spostamenti e rotazioni relative al secondo nodo confrontato

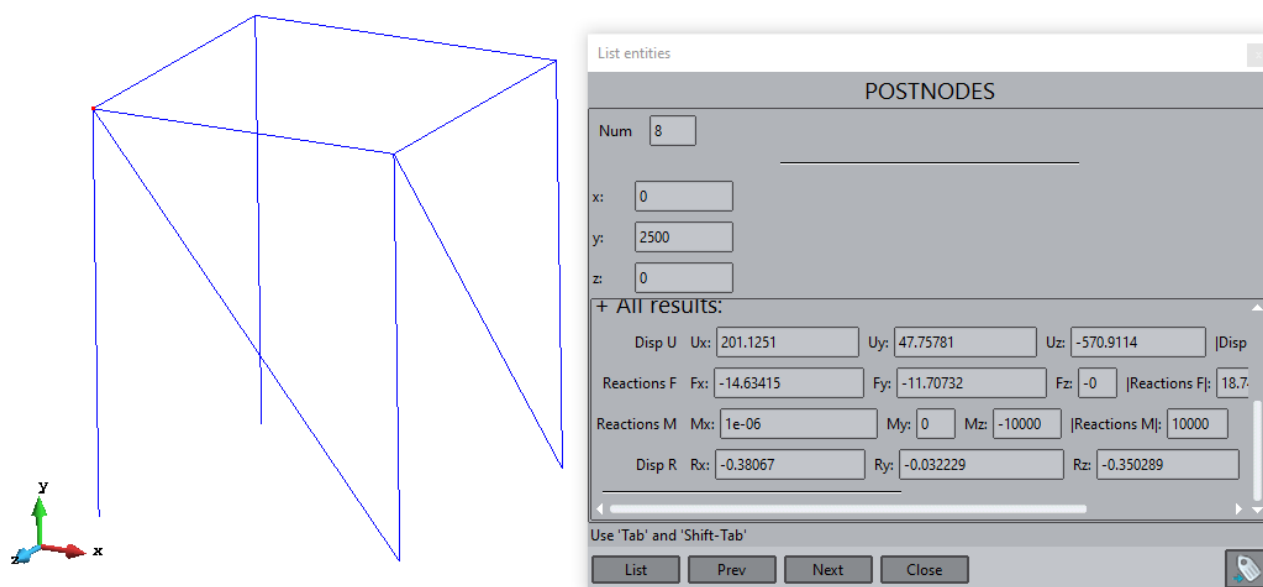


Figura 47: GID, spostamenti e rotazioni relative al secondo nodo confrontato

| | GID(rad) | deg |
|----|-----------|----------|
| RX | -3.81E-01 | -21.8108 |
| RY | -3.22E-02 | -1.84659 |
| RZ | -0.35029 | -20.0701 |

3.5 EDIFICIO MULTIPIANO

Una volta verificato il codice per quanto riguarda i carichi concentrati e gli elementi *beam* (anche quelli inclinati), si è implementato il carico uniformemente distribuito. Per testare le diverse tipologie di carico si è costruito un modello di dimensioni maggiori rispetto ai precedenti, ovvero un edificio multipiano in calcestruzzo.

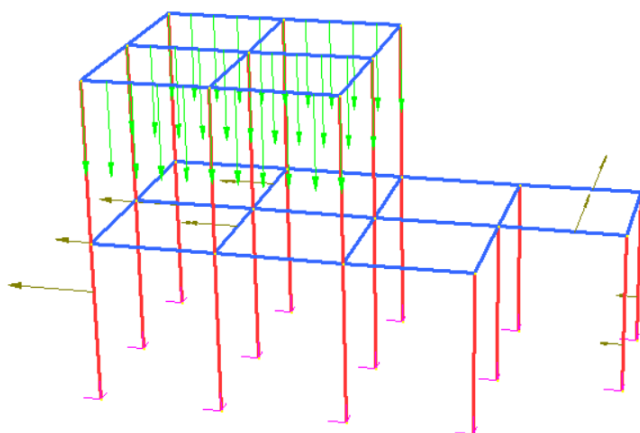


Figura 49: Straus7, modello comprensivo di carichi e vincoli

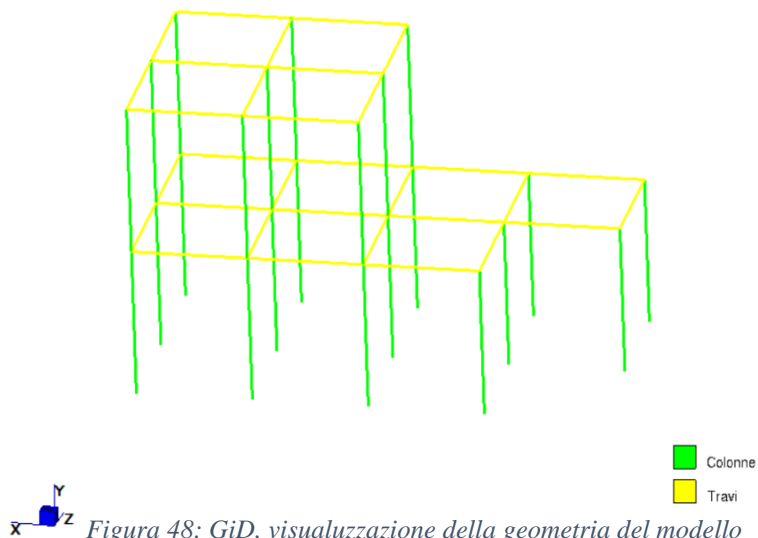


Figura 48: GiD, visualizzazione della geometria del modello

Dal confronto tra i risultati di Straus7 e quelli ottenuti in GiD, in corrispondenza del nodo 14, è evidente che sia il carico distribuito che l'assemblaggio dei carichi sono stati implementati correttamente.

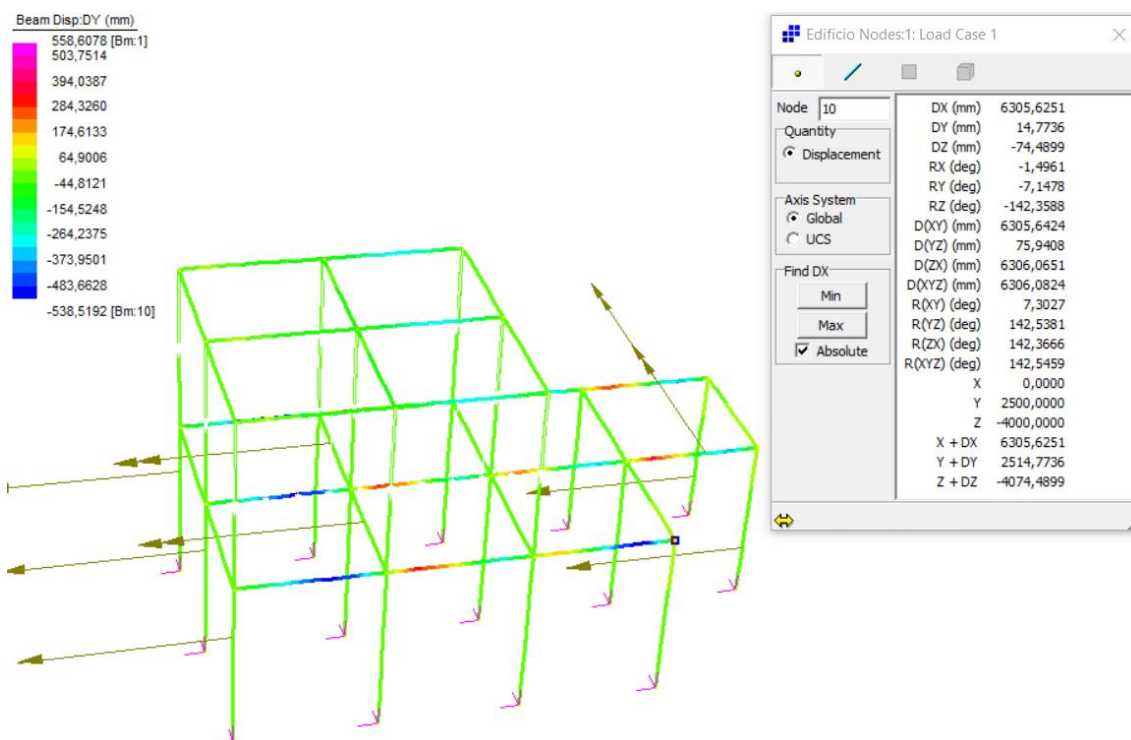


Figura 50: Straus7, spostamenti e rotazioni nel nodo 10 (ovvero il nodo 14 in GiD)

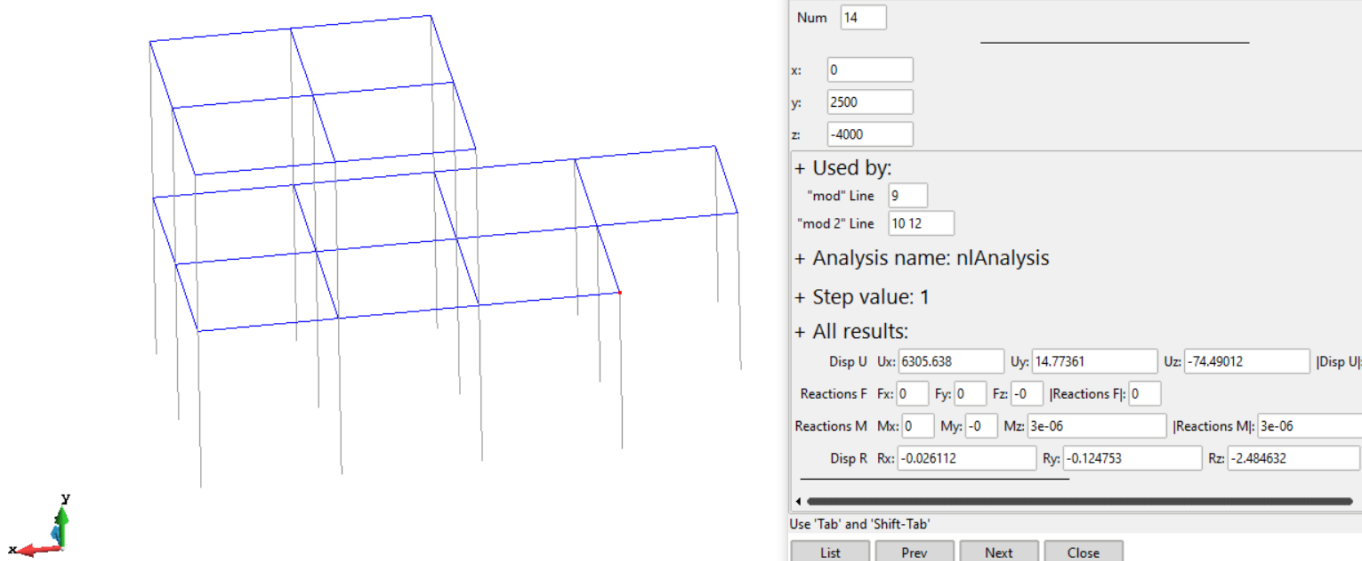


Figura 51: GiD, spostamenti e rotazioni nel nodo 14

| | STR | GID |
|----|----------|----------|
| RX | -1,4961 | -0,02611 |
| RY | -7,1478 | -0,12475 |
| RZ | -142,359 | -2,48463 |

Gli spostamenti e le rotazioni sono stati controllati in tutti i nodi, ovviamente per quanto riguarda la soluzione all'interno dell'elemento non si riescono ad ottenere i risultati esatti per via dell'interpolazione lineare all'interno di GiD.

3.6 PASSERELLA EXPO MILANO

Come ultimo test si è scelta una struttura più complessa, modellata con elementi *beam* ed elementi *truss* e sottoposta ai tre tipi di carichi implementati: forze e momenti nodali, forze e momenti puntuali e forze uniformemente distribuite.

La struttura scelta è la passerella ciclopedonale costruita per congiungere l'area Expo 2015 con la zona di Cascina Merlata. È una passerella pedonale che permette il superamento dell'autostrada A4 Torino-Milano, i binari della linea ferroviaria Milano-Domodossola e della linea ferroviaria ad Alta Velocità Torino-Milano. Ne è stata modellata la parte rappresentata nelle figure 52 e 53.



Figura 52: Passerella EXPO - Merlata



Figura 53: Passerella EXPO-Merlata, profilo

Di seguito sono riportati i modelli costruiti su Straus7 e GiD, con riferimento ai materiali, alle sezioni adottate e ai carichi che si è scelto di applicare.

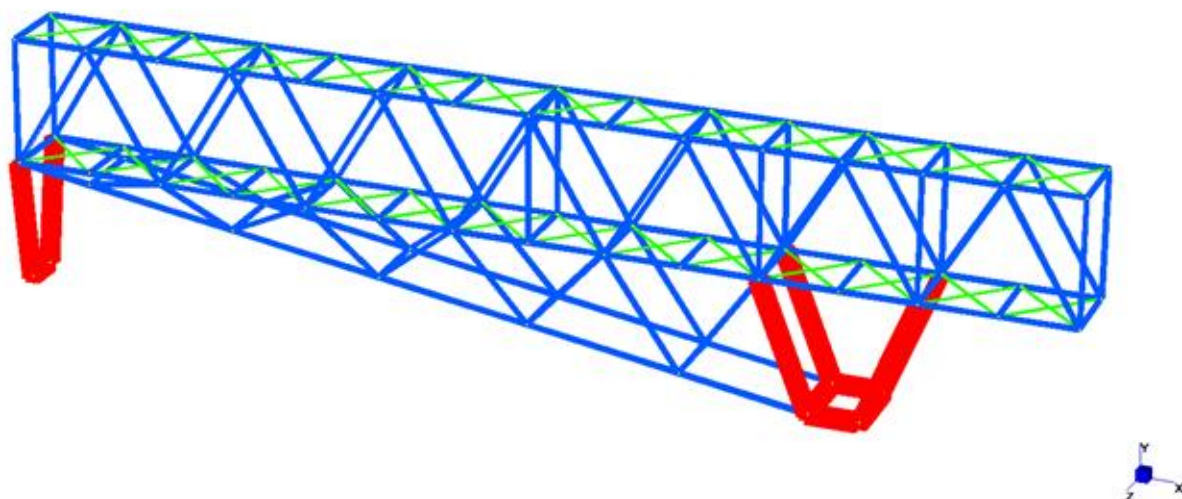


Figura 54: Straus7, visualizzazioni elementi in solid

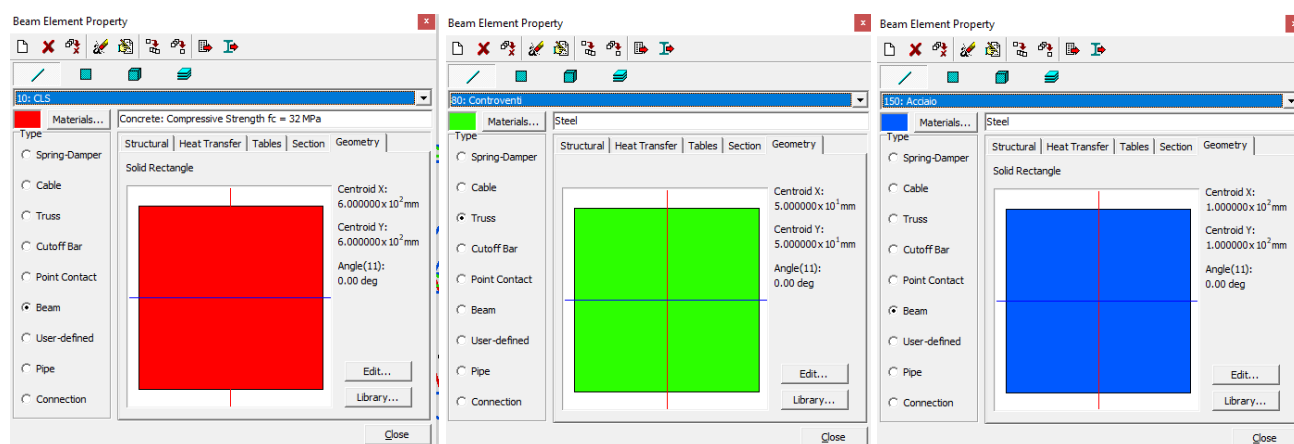


Figura 55: Straus7, sezioni utilizzate

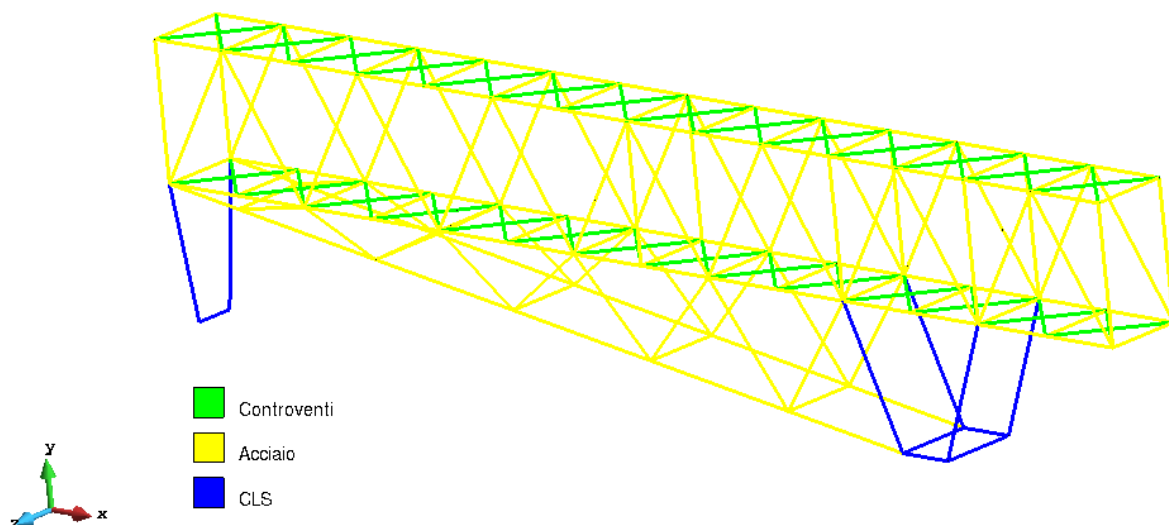


Figura 56: GiD, struttura con riferimento ai materiali utilizzati (corrispondenti per nome e caratteristiche con quelli indicati su Straus7)

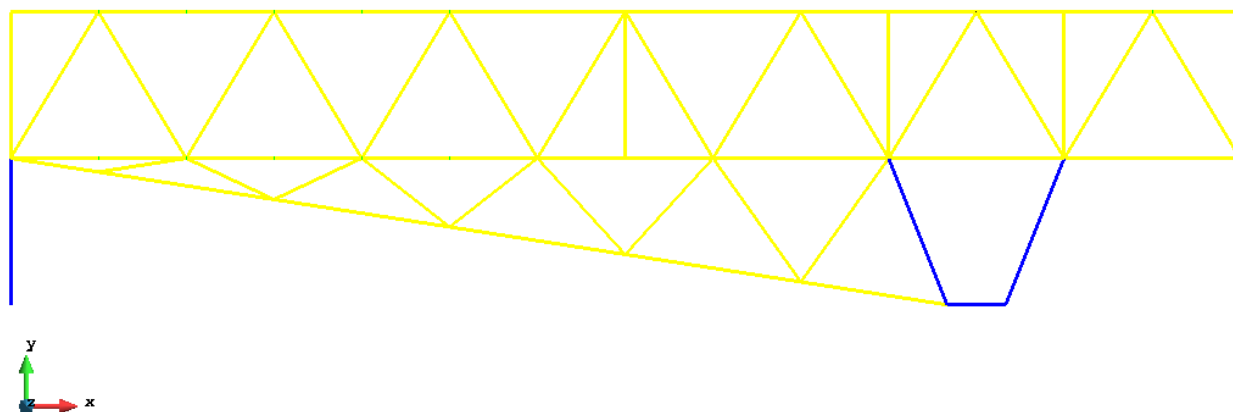


Figura 57: GiD, visualizzazione profilo della struttura

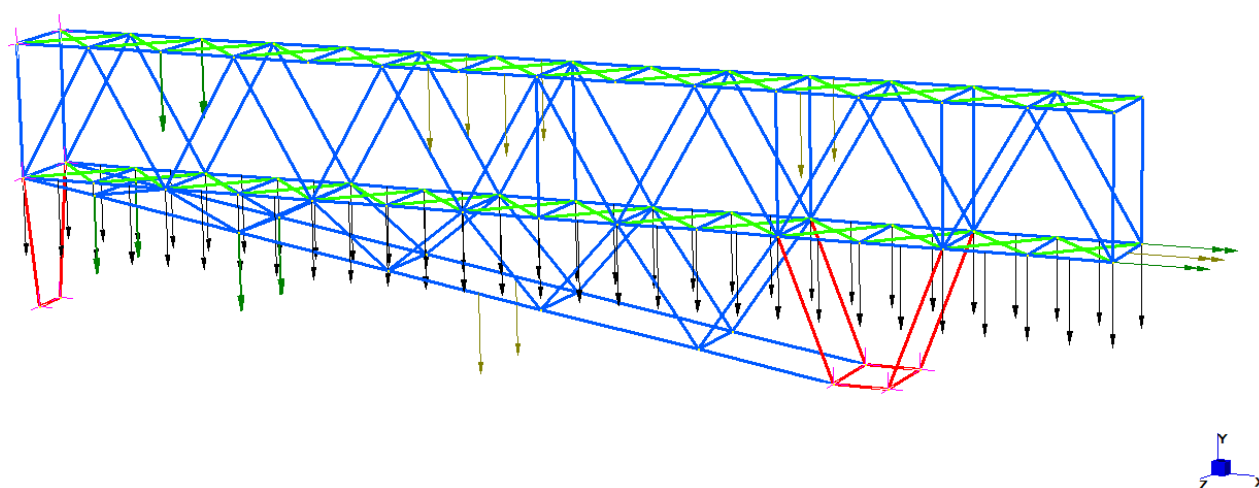


Figura 58: Straus7, visualizzazione dei vincoli imposti e dei carichi applicati alla struttura

Mettendo a confronto i risultati ottenuti con Straus7 e quelli ottenuti con il programma implementato (e visualizzati su GiD) in tre punti è possibile notare, ancora una volta, la buona corrispondenza tra i due, in quanto a spostamenti e rotazioni sui tre assi.

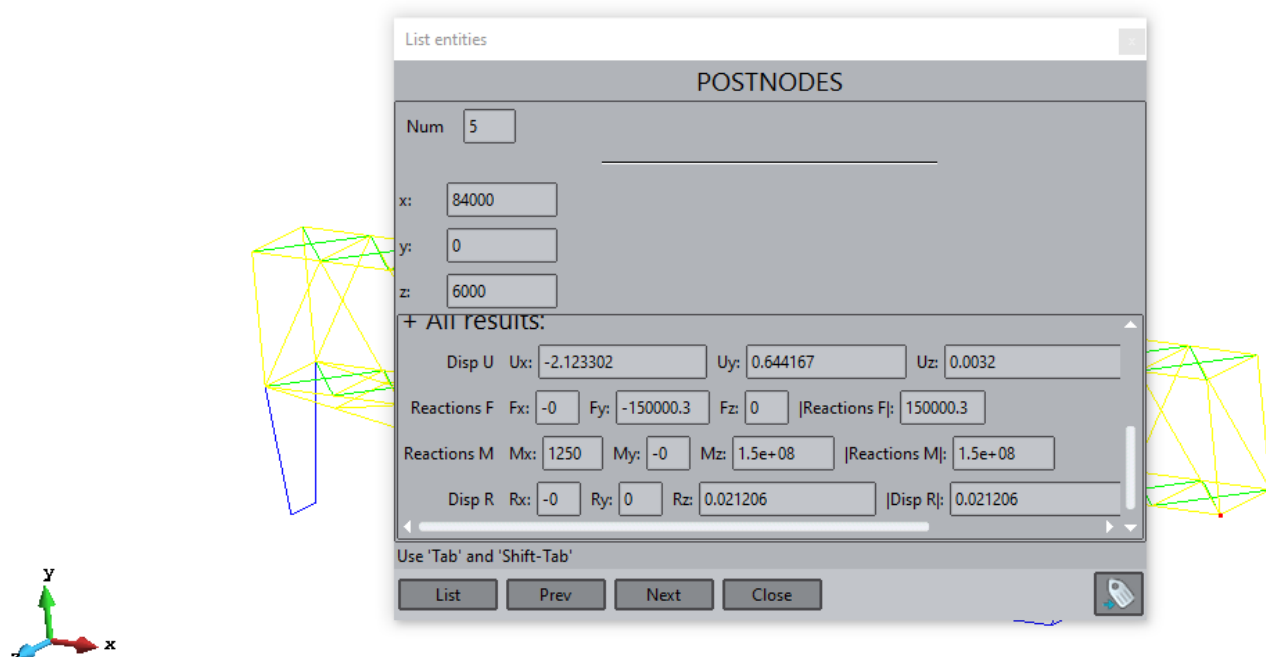


Figura 59: GiD, spostamenti e rotazioni del primo punto confrontato

| | GiD(rad) | deg |
|----|----------|----------|
| RX | 0.00E+00 | 0.0000 |
| RY | 0.00E+00 | 0 |
| RZ | 0.021206 | 1.215014 |

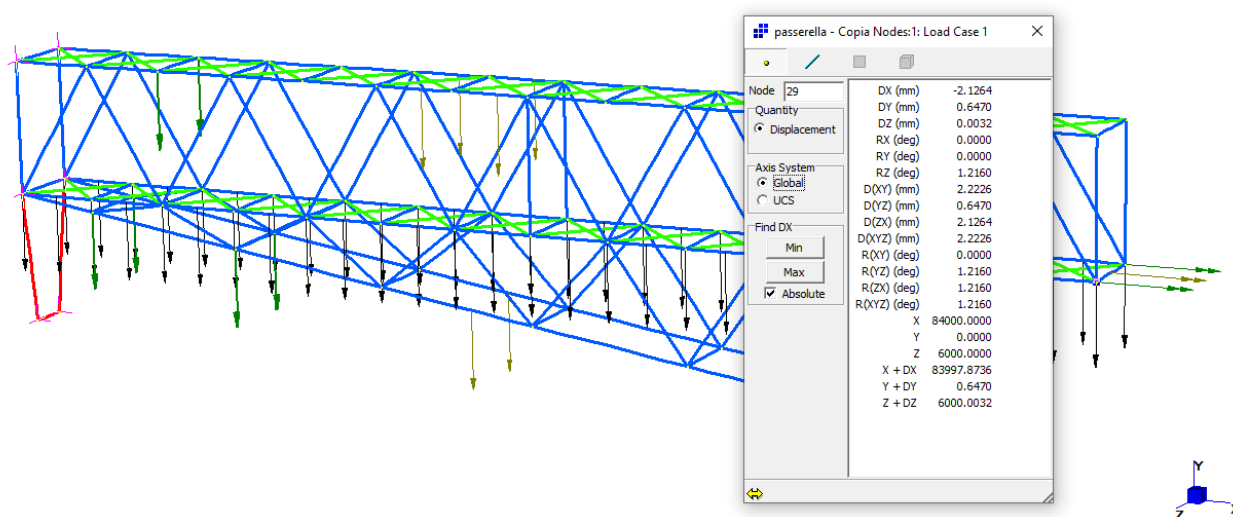


Figura 60: Straus7, spostamenti e rotazioni del primo punto confrontato

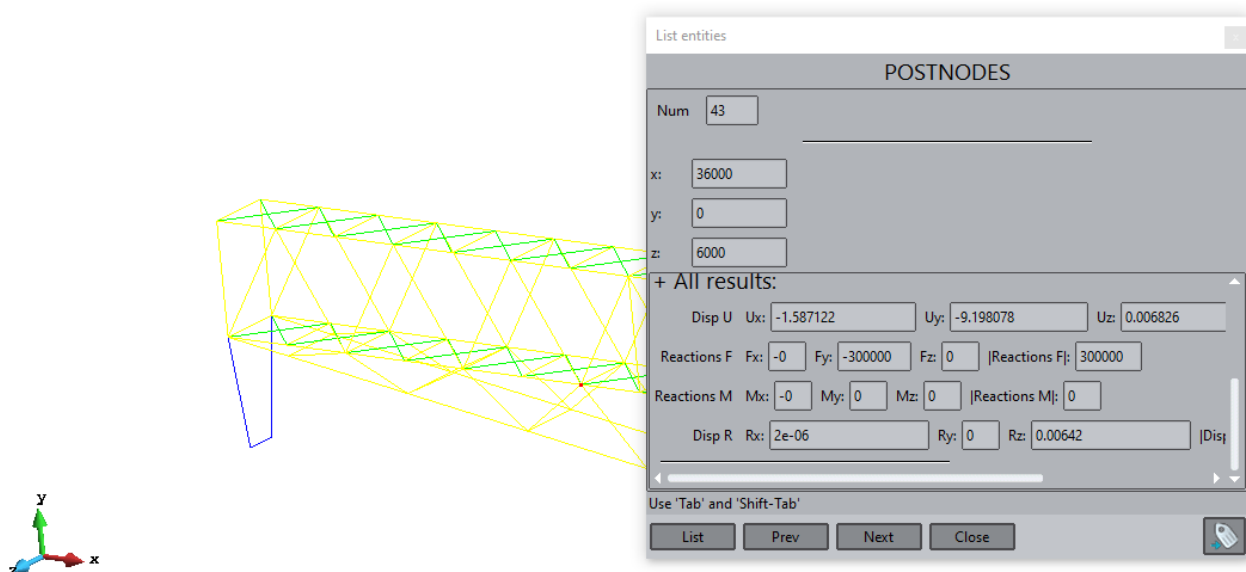


Figura 61: GiD, spostamenti e rotazioni del secondo punto confrontato

| | GiD(rad) | deg |
|----|----------|----------|
| RX | 2.00E-06 | 0.0001 |
| RY | 0.00E+00 | 0 |
| RZ | 0.00642 | 0.367839 |

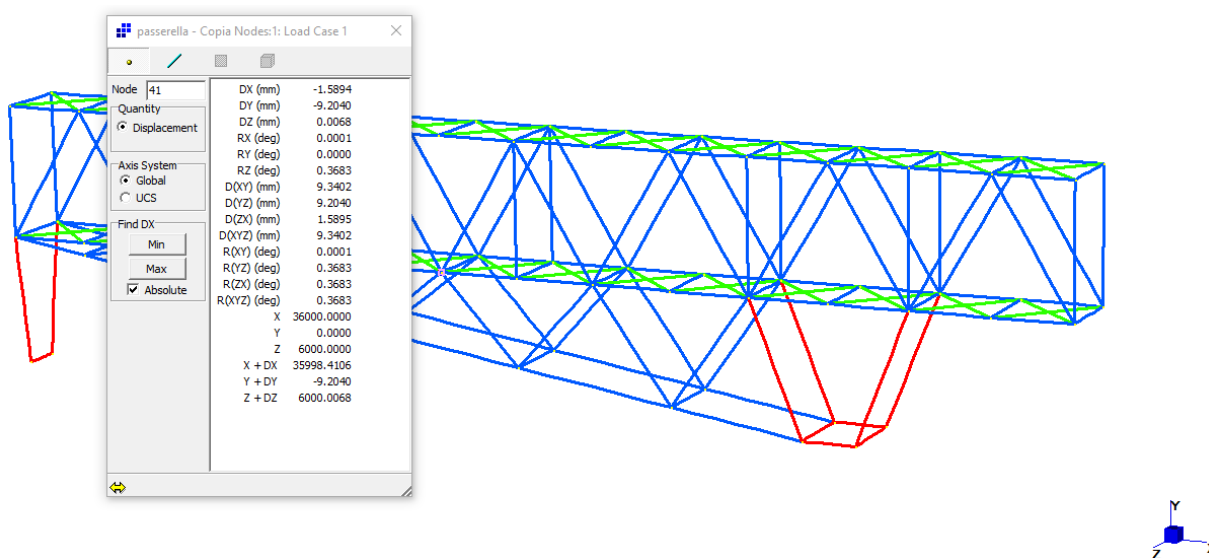


Figura 62: Straus7, spostamenti e rotazioni del secondo punto confrontato

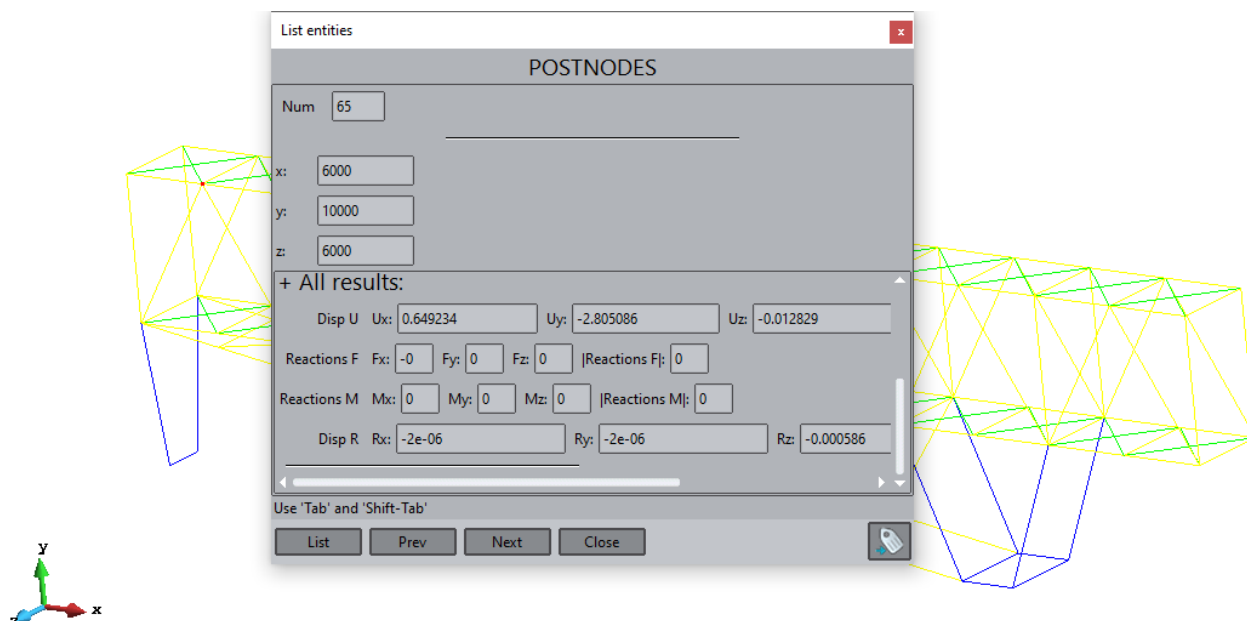


Figura 63: GiD, spostamenti e rotazioni del terzo punto confrontato

| | GiD(rad) | deg |
|----|-----------|----------|
| RX | -2.00E-06 | -0.0001 |
| RY | -2.00E-06 | -0.00011 |
| RZ | -0.00059 | -0.03358 |

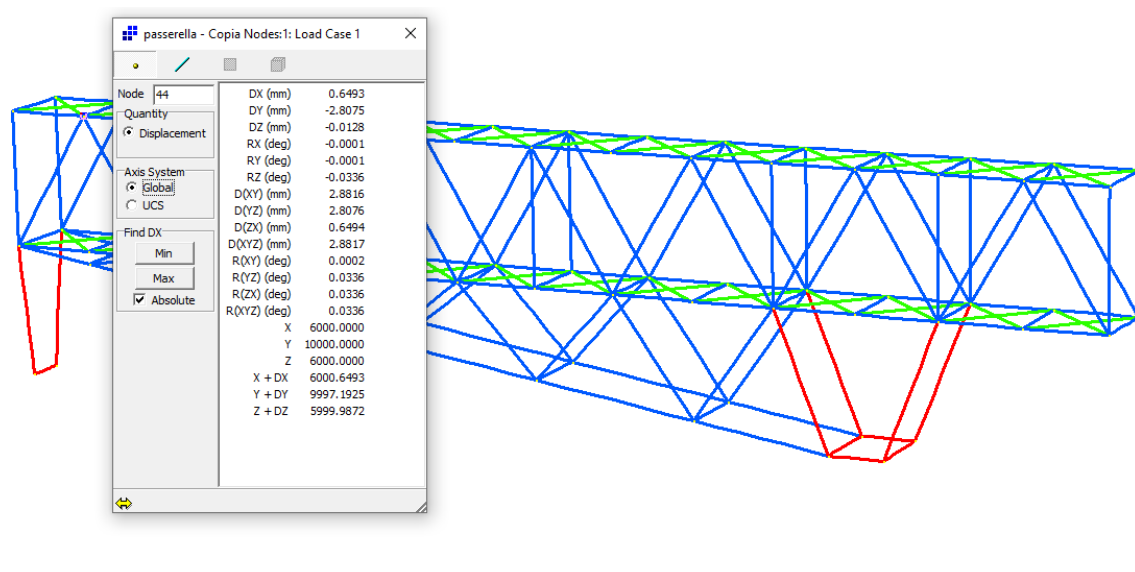


Figura 64: Straus7, spostamenti e rotazioni del terzo punto confrontato

La buona corrispondenza dei risultati, in maniera più generale, si può osservare anche dal contour degli spostamenti relativo all'asse y, la direzione lungo la quale la struttura è maggiormente sollecitata.

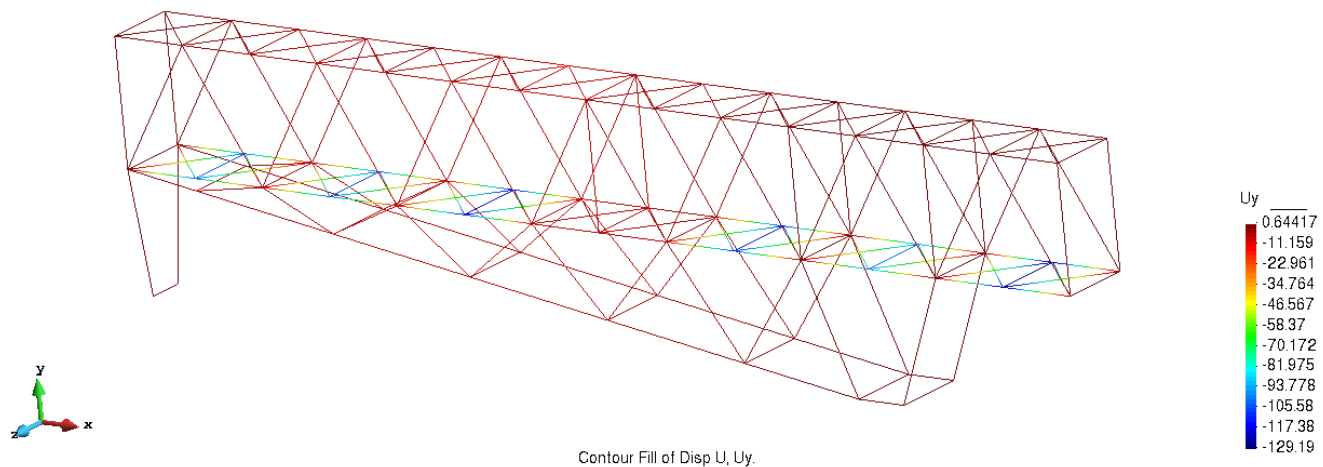


Figura 65: GiD, contour dei risultati relativi agli spostamenti lungo l'asse y

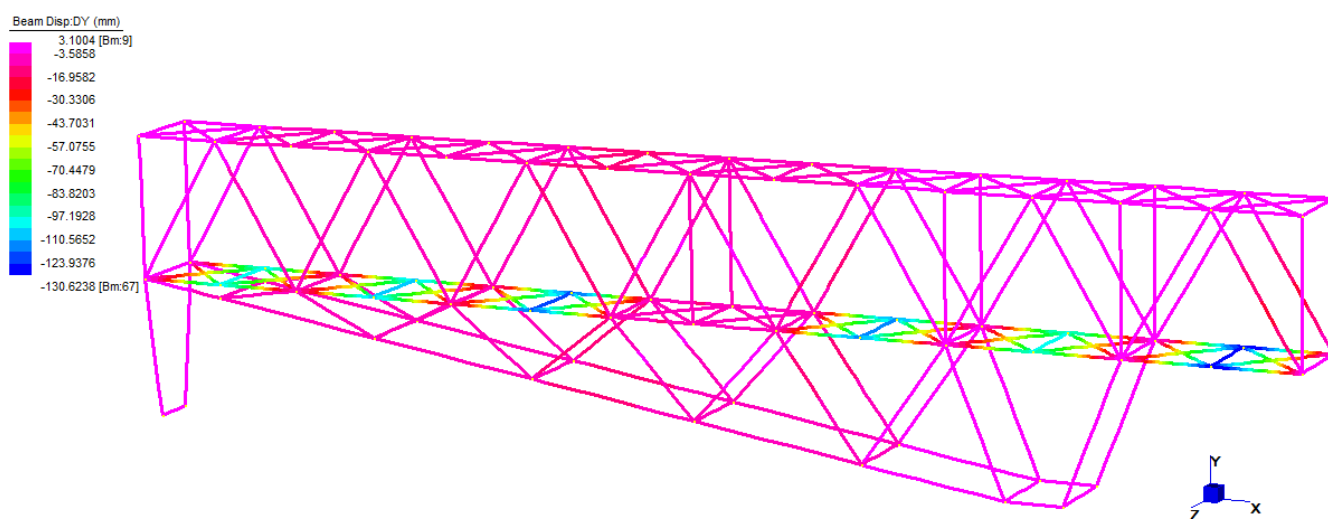


Figura 66: Straus7, contour dei risultati relativi agli spostamenti lungo l'asse y