Università degli studi di Padova

Dipartimento di ingegneria civile, edile e ambientale

Department of civil, enviromental and architectural engineering

Corso: "Meccanica computazionale"

Progetto finale

Studente: Mohamed Labrag
Matricola:1239041

Anno accademico 2019/2020

Contents

1 Introduzione

L'obbiettivo di questo progetto è quello di realizzare un programma agli elementi finiti per fare ciò si è optato per la programmazione per classi su matlab. L'idea generale è infatti quella di scrivere una serie di classi che permettano la risoluzione di strutture reticolari composte da elementi beam e truss, che, volendo, in futuro può anche essere integrato con altri elementi aumentando la complessità delle strutture analizzabili. Per quanto riguarda il pre e il post processamento viene utilizzato Gid.
Per poter risolvere queste strutture è necessaria una corretta applicazione del metodo degli elementi finiti. I quale si presenta come alternativa alla risoluzione analitica quando bisogna studiare le tensioni e le deformazioni di un oggetto che subisce uno sforzo, questo infatti è molto utile soprattutto per strutture complesse in quando ci fornisce un metodo facilmente automatizzabile e dunque risolvibile con un calcolatore.
Si procede dunque come segue: si discretizza l'oggetto tramite l'ausilio di funzioni forma N_i e cercando la soluzione su ogni nodo a_i tramite la quale si ottiene una approssimazione della soluzione esatta su tutto il corpo in questione.

$$\underline{u} \approx \underline{\hat{u}} = \underline{\underline{N}}\ \underline{a}$$

Inoltre da qui possiamo anche ottenere le deformazioni che ha subito il nostro sistema mediante un operatore differenziale:

$$\underline{\varepsilon} = \underline{\underline{\delta}}\ \underline{u} = \underline{\underline{\delta}}\ \underline{\underline{N}}\ \underline{a} = \underline{\underline{B}}\ \underline{a}$$

E una volta ottenute le deformazioni, nel caso di materiale elastico lineare e isotropo diventa immediato ottenere le tensioni mediante la matrice D.
Scrivendo poi il teorema dei lavori virtuali e sostituendo al suo interno le tensioni e le deformazioni otteniamo:

$$\int_\Omega \underline{\underline{B}}^T\ \underline{\underline{D}}\ \underline{\underline{B}} d\Omega\ \underline{a} = f^e$$

Dove f^e sono le forze applicate al nostro sistema, a è sempre la soluzione nodale e il risultato dell'integrale rappresenta la matrice di rigidezza K, che dipende dal materiale e dalle derivate delle funzioni forma che abbiamo utilizzato.

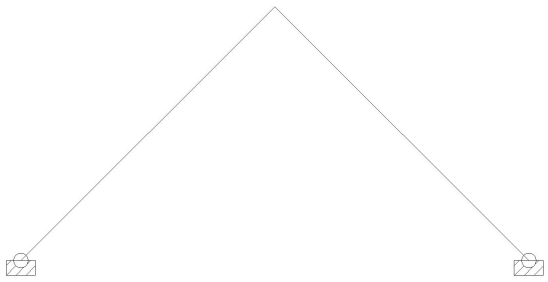
2 Assemblaggio della matrice K

Come è già stato anticipato in precedenza un aspetto essenziale prima di poter procedere alla risoluzione del problema è proprio quello di ottenere la matrice di rigidezza K, questa si ricava mediante l'assemblaggio di matrici locali che descrivono il comportamento dei singoli elementi.

Nel caso in questione saranno utilizzate le matrici di rigidezza degli elementi truss(che dunque lavorano solo in modo assiale) e beam di Eulero-Bernoulli (lavorano anche a flessione).

Si realizzano per l'appunto delle classi che si occupano di computare per ogni singolo elemento la sua matrice di rigidezza, a seguito in base ai nodi che delimitano l'elemento si sommano le componenti della matrice locale in quella globale, contribuendo in qualche modo alla rigidezza del nodo. L'esempio più semplice per comprendere ciò è quello di una molla su cui applico una forza, se il nodo in cui questa è applicata fosse attaccato a due molle, allora indubbiamente sarà più rigido e a parità di forza avremo spostamenti minori.

Descriviamo il procedimento di assemblaggio mediante il seguente esempio, supponiamo di considerare un sistema composto da 3 nodi e due aste all'interno di uno spazio bidimensionale:



Possiamo pensare dunque che la matrice K sarà divisa in $n \times n$ blocchi dove

n corrisponde al numero di nodi presenti nel mio sistema, e dove ogni blocco ha dimensione $m \times m$ in cui m è il numero di gradi di libertà per nodo, nel caso bidimensionale dunque sono quattro se consideriamo elementi beam.
L'assemblaggio consiste dunque nel prendere le matrici di rigidezza locali dei due elementi(le quali sono composte da quattro blocchi, in quanto poggiano su due nodi) scomporle per poi andare a sommarli alle entrate corrispondenti nella matrice di rigidezza globale .

$$K = \begin{pmatrix} \underline{\underline{K}}^1_{11} & \underline{\underline{K}}^1_{12} & 0 \\ \underline{\underline{K}}^1_{21} & \underline{\underline{K}}^1_{22} + \underline{\underline{K}}^2_{11} & \underline{\underline{K}}^2_{12} \\ 0 & \underline{\underline{K}}^2_{21} & \underline{\underline{K}}^2_{22} \end{pmatrix}$$

Dunque è sufficiente ricavare le singole matrici locali di ogni elemento, e considerando l'indice globale dei loro nodi.

Codice

Per l'algoritmo ovviamente è necessario in un primo momento definire la classe "SOLUTORE" , indicando le proprietà di questa, dopodiché realizziamo il costruttore, e solo a seguito possiamo cominciare con la realizzazione delle function che ci servono.

Per l'assemblaggio partiamo innanzitutto inizializzando la nostra matrice globale:

```
K=sparse(this.dim,this.dim, 0);        %inizializzazione della
                                        matrice di rigidezza
dimel=length(this.elements);           %dimensione del numero di
                                          elementi
```

In seguito introduciamo un controllo che ci assicuri che la nostra classe abbia effettivamente ricevuto i dati dei nodi e degli elementi:

```
if dimel==0 || this.dim==0
     fprintf('WARNING sul solutore non ci sono elementi o nodi! \n');
end
```

Dopodiché iniziamo un ciclo for per tutti i nostri elementi, e per ciascuno di questi otteniamo il numero dei nodi totali definiti nell'elemento:

```
n_nod_el=this.elements(e).el.TOTNODES;
```

Una matrice X che contenga le coordinate dei nodi dell'elemento:

```
for i=1:n_nod_el
     X(i,:)=this.nodes(this.elements(e).el.nodes_id(i) ).x;
end
```

E la matrice di rigidezza locale del suddetto elemento di cui otteniamo la dimensione e i gradi di libertà ammessi per nodo.

```
elK=this.elements(e).stiffness(this.property(this.elements(e).el.prop_id),X);

local_dim=length(elK);
gdl=local_dim/n_nod_el;
```

A seguito si fa partire il ciclo di assemblaggio, il quale non è altro che due cicli "for", ciascuno fatto sul numero di nodi definiti dall'elemento, e in questi si cercano le corrispettive coordinate nella matrice globale dei valori della matrice locale appena calcolata:

```
for local_i=1:n_nod_el
    global_i=this.elements(e).el.nodes_id(local_i);
    for local_j=1:n_nod_el
       global_j=this.elements(e).el.nodes_id(local_j);
       global_coord_i=[(global_i-1)*this.gdlmax+1:
                       (global_i-1)*this.gdlmax+gdl];

       global_coord_j=[(global_j-1)*this.gdlmax+1:
                      (global_j-1)*this.gdlmax+gdl];
```

```
        local_coord_i=[(local_i-1)*gdl+1: local_i*gdl];
        local_coord_j=[(local_j-1)*gdl+1: local_j*gdl];


        K(global_coord_i,global_coord_j)=K(global_coord_i,global_coord_j)
                                        +elK(local_coord_i,local_coord_j);

   end
end
```

Una volta giunti a questo punto abbiamo completato l'assemblaggio della matrice di rigidezza.

Condizioni al contorno

Un aspetto importante all'interno della risoluzione di un problema agli elementi finiti è sicuramente quello delle condizioni al contorno, senza le quali sarebbe impossibile la risoluzione. Queste non sono altro che l'insieme di gradi di libertà, traslazionali e/o rotazionali, che possono essere bloccati nei miei nodi.

Nella funzione di assemblaggio delle condizioni al contorno otteniamo come output tre vettori :

- Un primo vettore che contiene al suo interno gli indici posizione degli spostamenti incogniti all'interno del mio sistema, e dunque dei gradi di libertà liberi.
- Un secondo vettore contenente gli indici di posizione dei gradi di libertà vincolati.
- Un terzo vettore della stessa dimensione del secondo contenete i valori dei gradi di libertà imposti.

Per fare ciò il procedimento è stato il seguente:
Come primo passaggio è stata trovata la dimensione degli ultimi due vettori, per poter preallocare lo spazio nella memoria per questi:

```
long=0;
for i=1:length(this.constraint)
   long=long+length(this.constraint(i).u_vinc);
end
ind_u_vinc=zeros(long,1);
u_vinc=zeros(long,1);
```

A seguito andiamo a fare un ciclo sui vincoli che abbiamo e andiamo a inserire all'interno dei due vettori le loro rispettive entrate:

```
for i=1:length(this.constraint)
   cont_end=cont_start+length(this.constraint(i).u_vinc);
   ind_u_vinc(cont_start+1:cont_end)=this.constraint(i).ind_u_vinc;
   u_vinc(cont_start+1:cont_end)=this.constraint(i).u_vinc;
   cont_start=cont_end;
end
```

Dopodiché procediamo con la costruzione del primo vettore, che come è già stato annunciato, contiene le posizioni dei gradi di libertà svincolati nel sistema analizzato:

```
ind_u_inc=zeros(this.dim-length(ind_u_vinc),1);
temp=zeros(this.dim,1);
temp(ind_u_vinc)=1;
cont=0;
for i=1:this.dim
   if temp(i)==0
      cont=cont+1;
      ind_u_inc(cont)=i;
   end
end
```

3 Solutori

Per quanto riguarda i solutori e dunque l'applicazione delle condizioni al contorno sono stati sviluppati due diversi metodi uno che sfrutta il penalty, e uno che lavora senza quest'ultimo.

3.1 Solutore senza penalty

All'interno di questo solutore vengono sfruttati tutti e tre i vettori calcolati durante l'assemblaggio delle condizioni al contorno, andando a risolvere solo le entrate incognite del nostro sistema, come se eliminassimo, le righe e le colonne dei gradi di libertà vincolati, avendone già lo spostamento.
Per cominciare assembliamo la matrice di rigidezza il vettore delle forze e le condizioni al contorno:

```
K=assemblyK(this);
F=assemblyF(this);
[ind_u_inc, ind_u_vinc, u_vinc]=assemblyU(this);
```

A seguito andiamo ricavare i vettori che ci servono per la nostra soluzione:

```
u_temp=zeros(this.dim,1);
u_temp(ind_u_vinc)=u_vinc;
F_temp=F-K*u_temp;
```

Avendo già ottenuto gli indici di posizione dei gradi di libertà incogniti a questo punto è sufficiente:

```
u_inc=K(ind_u_inc,ind_u_inc)\F_temp(ind_u_inc);
```

A questo punto per ottenere i vettori u e f è sufficiente procedere come segue:

```
u=u_temp;
u(ind_u_inc)=u_inc;
f=K*u;
```

3.2 Solutore con penalty

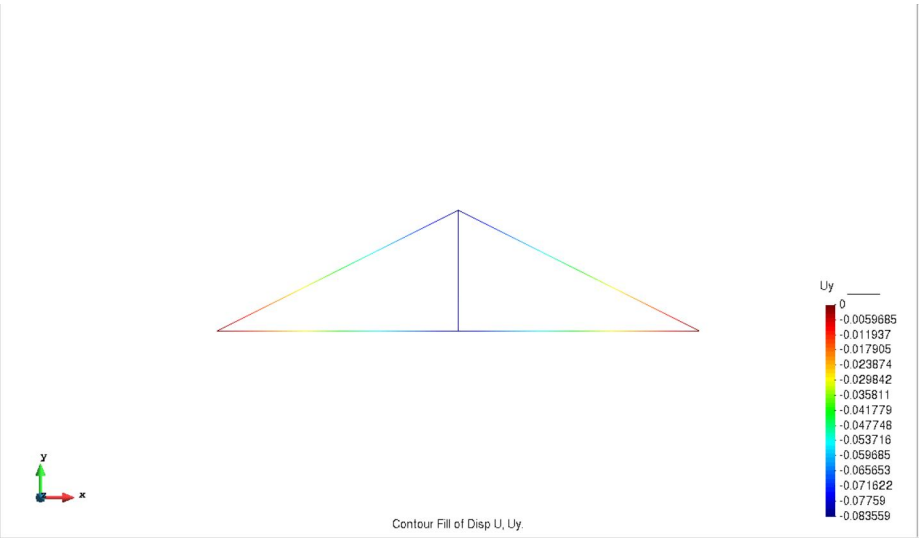
Un alternativa al solutore precedentemente descritto è il seguente; l'idea alla base consiste nel evitare di togliere le righe e le colonne corrispondenti ai gradi di libertà vincolati, ma di modificare la matrice di rigidezza, andando

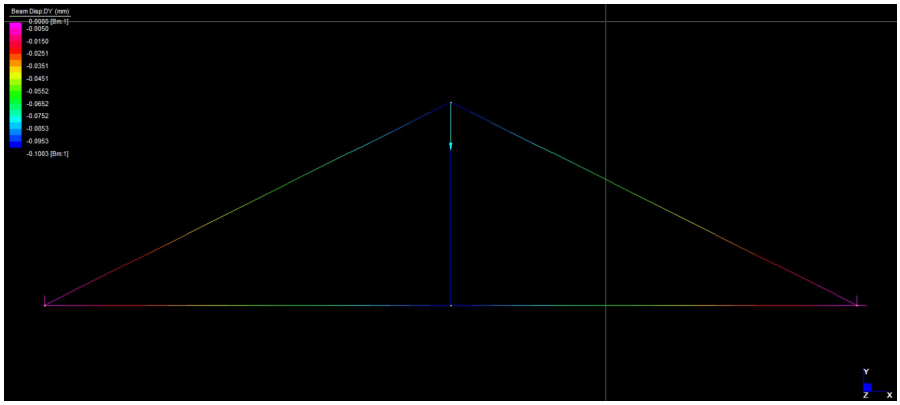
a sostituire nelle entrate diagonali che gli corrispondono un valore R detto penalty, il quale è un numero molto alto che ci permette di "manipolare" il risultato. Inoltre si sostituisce anche nel vettore delle forze in queste posizioni il penalty moltiplicato per lo spostamento imposto.

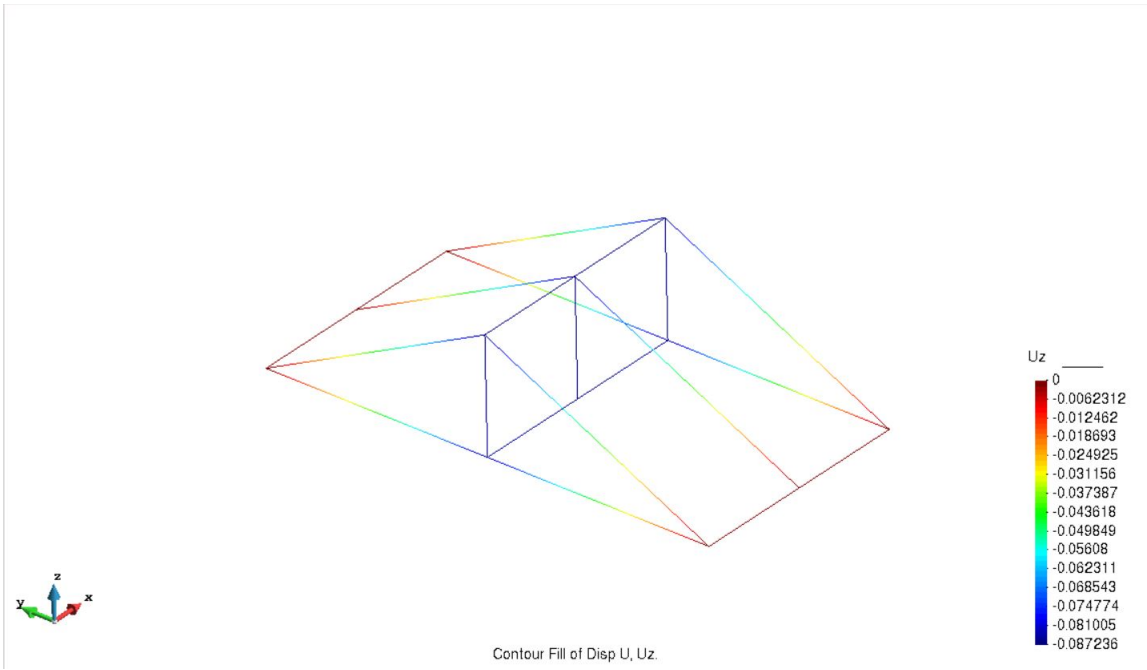
```
K=assemblyK(this);
F=assemblyF(this);
[~, ind_u_vinc, u_vinc]=assemblyU(this);

k=K; %k è la matrice K ma che verrà modificata con il penalty
f=F; %f è il vettore F ma che verrà modificato con il penalty
for i=1:length(ind_u_vinc)
   k(ind_u_vinc(i),ind_u_vinc(i))=this.R;
   f(ind_u_vinc(i))=this.R*u_vinc(i);
end

u=k\f;
F=K*u;
```

4 Test

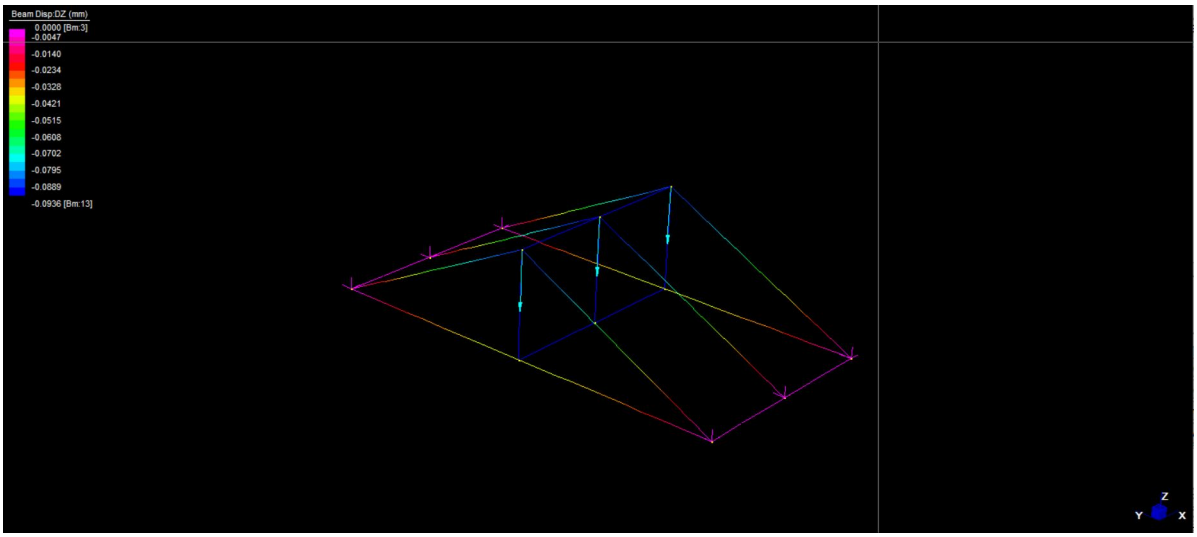
Alla fine dell'implementazione del programma sono stati fatti alcuni test per confrontare i risultati di questo algoritmo con quelli di un software commerciale quale "Straus7" e si può osservare che le soluzioni sono generalmente prossime.

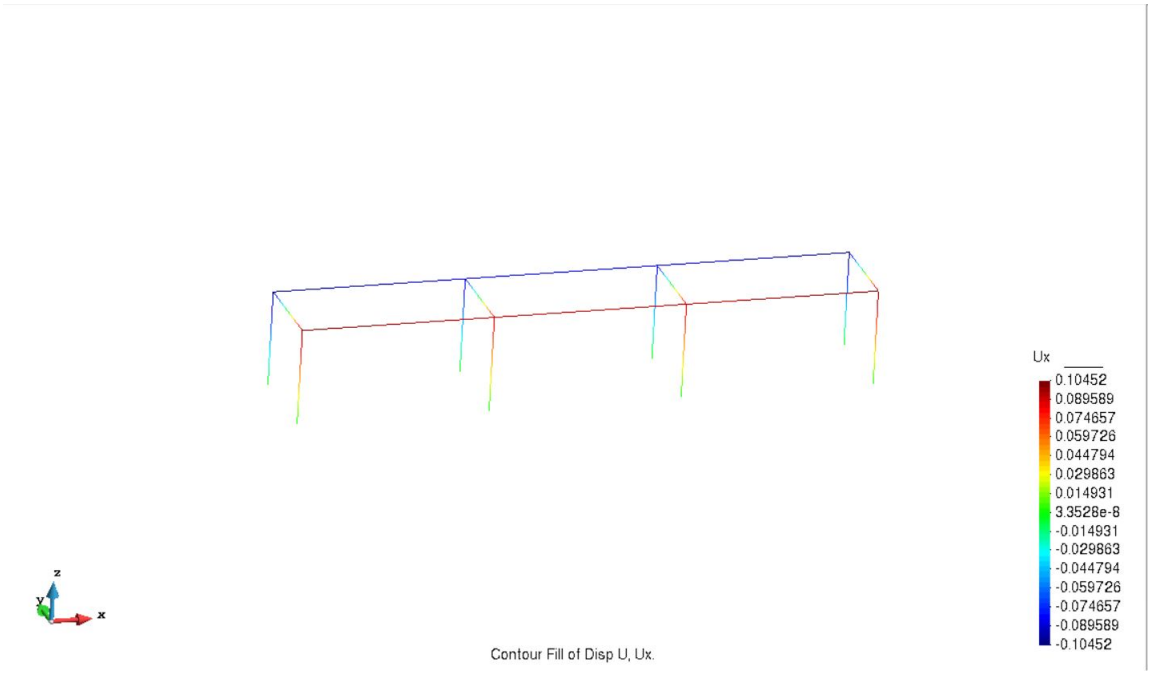




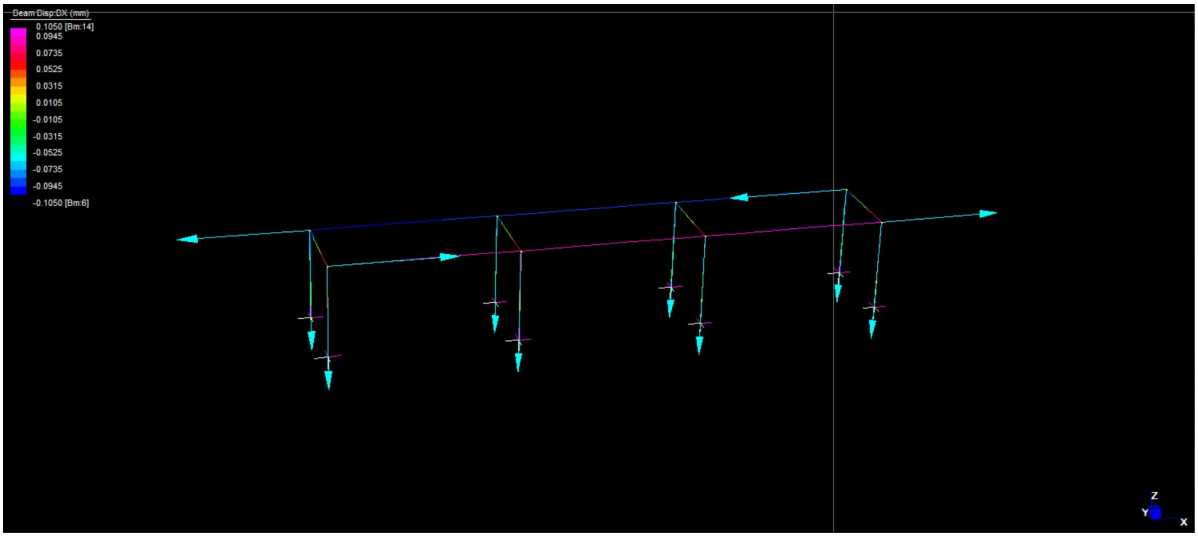


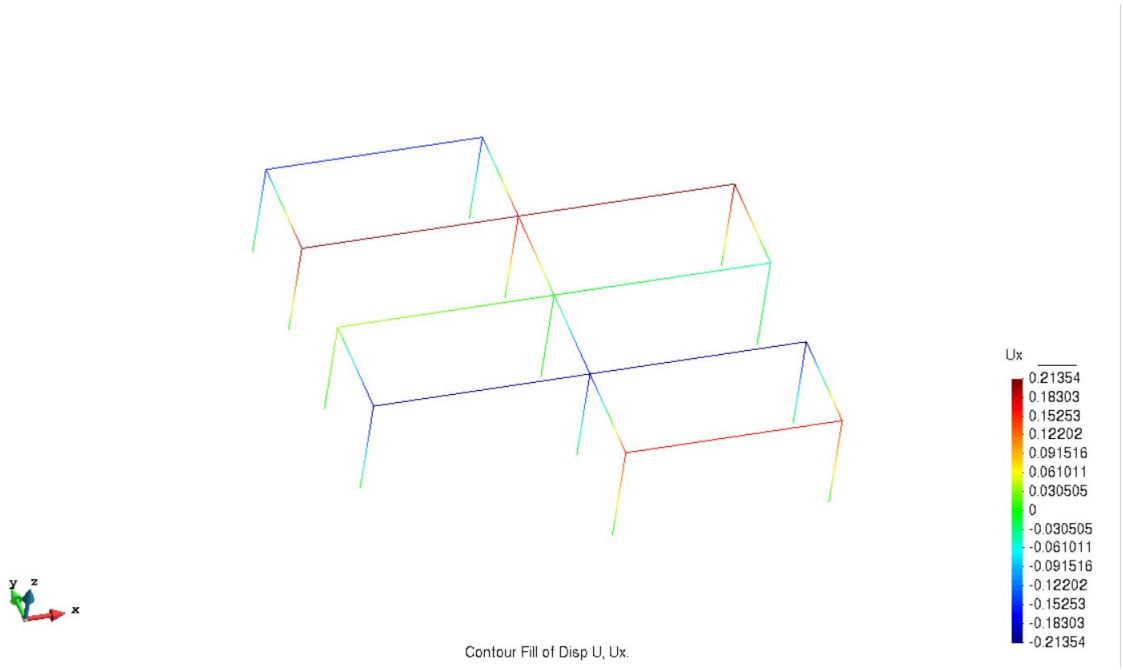
Contour Fill of Disp U, Uz.



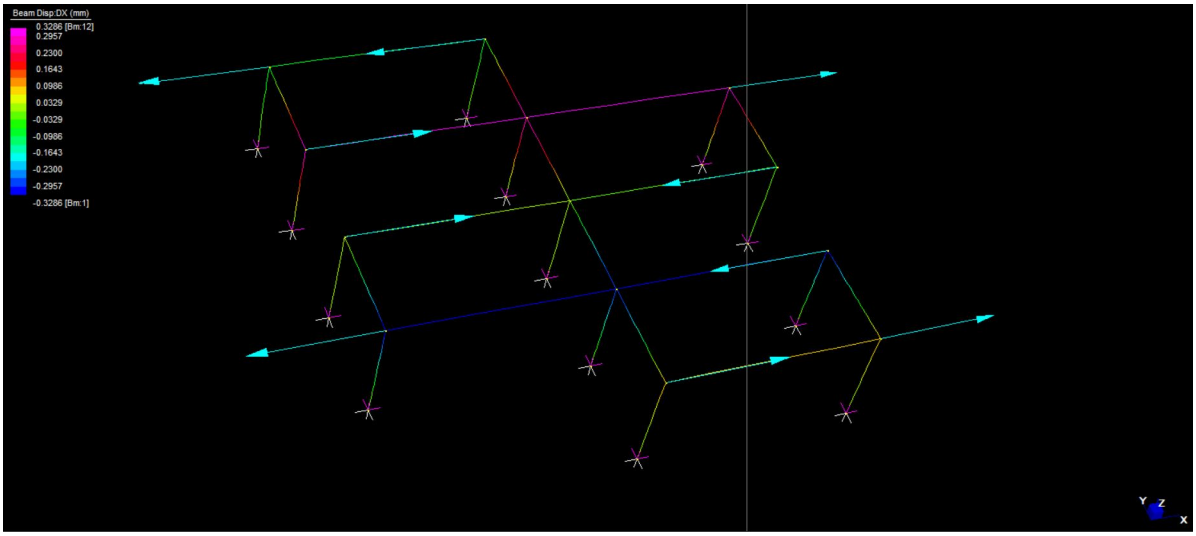


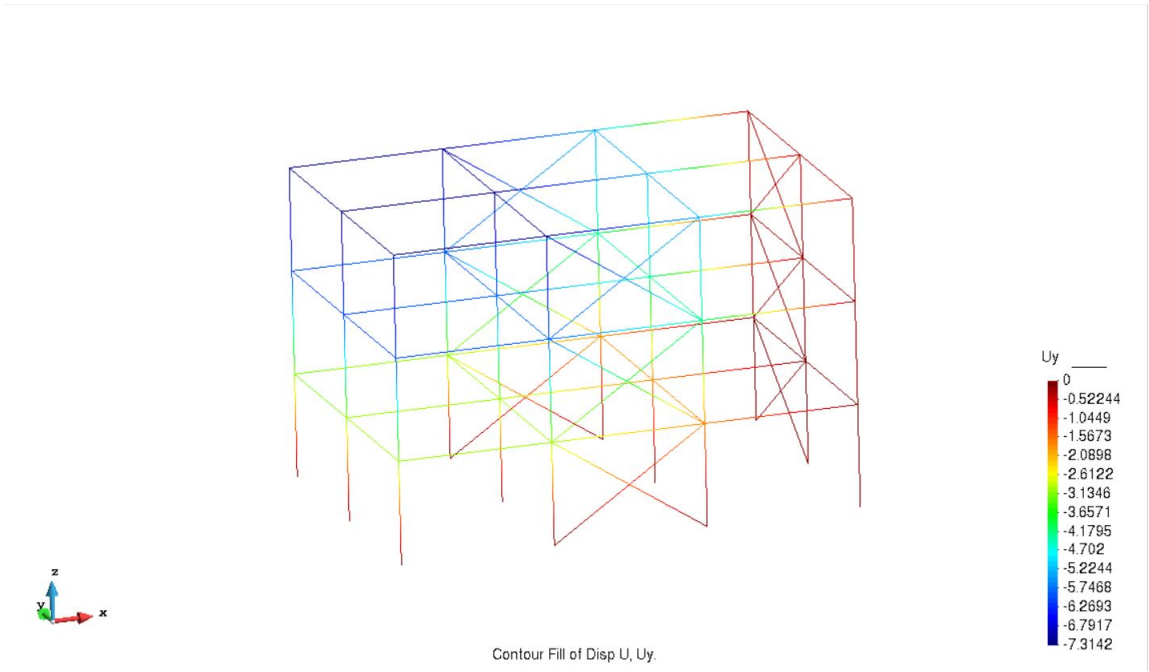
Contour Fill of Disp U, Ux.



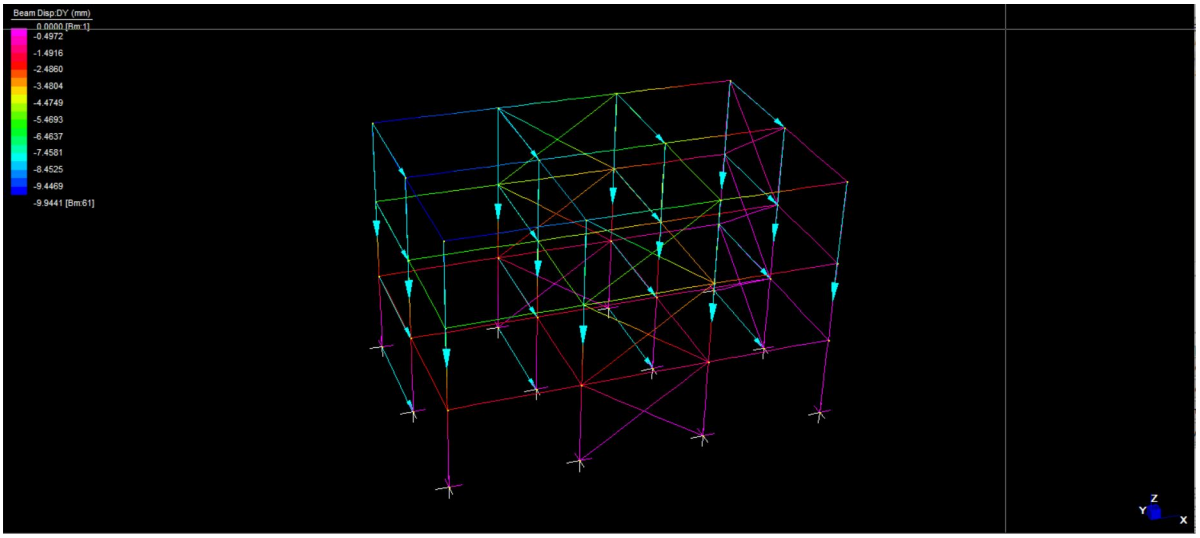


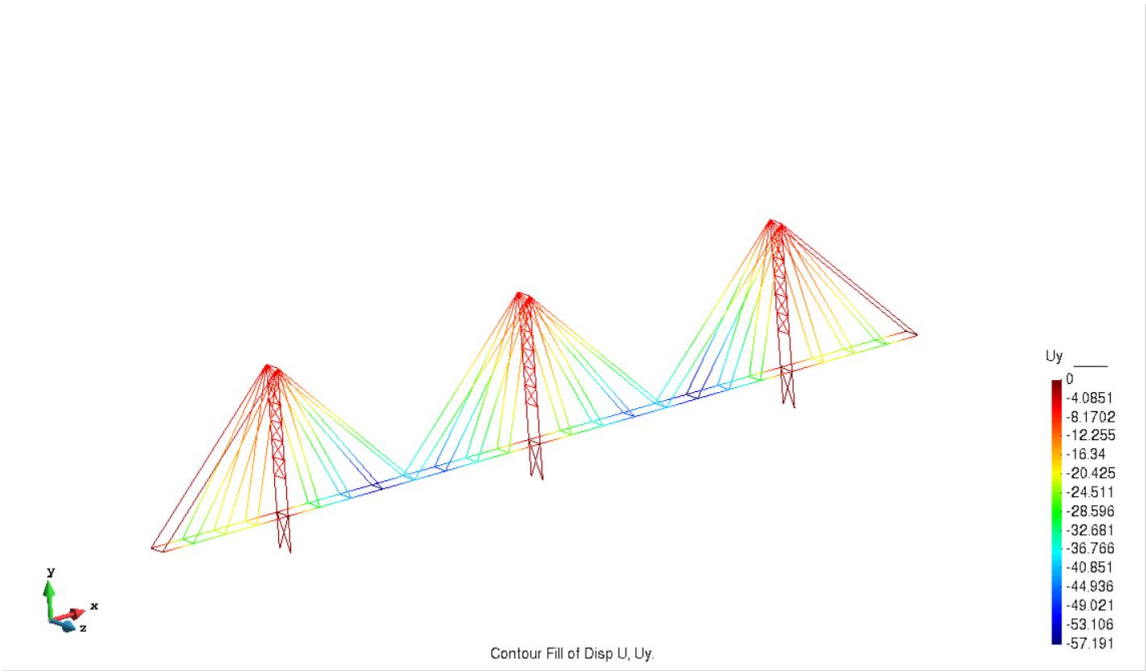
Contour Fill of Disp U, Ux.





Contour Fill of Disp U, Uy.





Contour Fill of Disp U, Uy.