

Annexe TIPE  
*Étude et optimisation d'un outil d'ingénierie du bâtiment*

*Thomas CREUSET*

*numéro de candidat : 11909*

## Contents

<b>1</b>	<b><u>code C : calculateur</u></b>	<b>2</b>
1.1	standard_lib.h . . . . .	2
1.2	module_matrice.h . . . . .	2
1.3	module_matrice.c . . . . .	3
1.4	module_matrice.c . . . . .	13
<b>2</b>	<b><u>code Ocaml : Affichage</u></b>	<b>25</b>
2.1	Affichage_FEM_3D.ml . . . . .	25

# 1 code C : calculateur

## 1.1 standard\_lib.h

```
1  /* -Importations- */
2
3  #include <stdio.h>
4  #include <stdbool.h>
5  #include <stdlib.h>
6  #include <time.h>
7  #include <math.h>
```

## 1.2 module\_matrice.h

```
1  /* ~Matrices~ */
2
3  /* -Importations- */
4
5  #include "standard_lib.h"
6
7
8  /* -Types et structures- */
9
10 typedef double valeur;
11
12 struct matrice_s {
13     int lignes;
14     int colonnes;
15     valeur** contenu;
16 };
17
18 typedef struct matrice_s matrice;;
19
20 /* -Déclarations fonctions (f) et procédures (p)- */
21
22 matrice* creer_matrice(int lignes, int colonnes);
23 // f - crée une matrice de dimension 'lignes'x'colonnes' initialisé à 0 (
    valeur par default du type 'valeur' à changer si ce dernier change)
24
25 void supprimer_matrice(matrice* matriceEntree);
26 // p - vide la mémoire utilisée par la matrice 'matriceEntree'
27
28 matrice* sous_matrice(matrice* matriceEntree, int ligneDepart, int
    colonneDepart, int nbreLignes, int nbreColonnes);
29 // f - créer la sous matrice comme spécifiée
30
31 matrice* add_matrice(matrice* matriceA, matrice* matriceB);
32 // f - additionne les matrices 'matriceA' et 'matriceB' de manière non
    destructive
33
34 matrice* soustrait_matrice(matrice* matriceA, matrice* matriceB);
35 // f - soustrait la matrice 'matriceB' à la matrice 'matriceB' de manière non
    destructive
36
37 matrice* mult_matrice(matrice* matriceA, matrice* matriceB);
38 // f - multiplie les matrices 'matriceA' et 'matriceB' de manière non
    destructive
39
40 matrice* transp_matrice(matrice* matriceEntree);
```

```

41 // f - transpose la matrice 'matriceEntree' de manière non destructive
42
43 matrice* dilatation_matrice(valeur scalaire, matrice* matriceEntree);
44 // f - dilate la matrice par une scalaire (de type 'valeur') 'scalaire'
45
46 valeur det_matrice(matrice* matriceEntree);
47 // f - calcule le déterminant de la matrice carrée 'matriceEntree'
48
49 void echange_ligne(matrice* matriceEntree, int ligne1, int ligne2);
50 // p - échange les lignes d'indice 'ligne1' et 'ligne2' de la matrice '
    matriceEntree' par effet de bord
51
52 void echange_colonne(matrice* matriceEntree, int colonnel, int colonne2);
53 // p - échange les colonnes d'indice 'colonnel' et 'colonne2' de la matrice '
    matriceEntree' par effet de bord
54
55 void combinaison_lignes(matrice* matriceEntree, int ligneDest, valeur scalaire
    , int ligneAjout);
56 // p - affecte à la ligne d'indice 'ligneDest' elle-même plus la ligne d'
    indice 'ligneAjout' multipliée par un scalaire 'scalaire' par effet de bord
57
58 void combinaison_colonnes(matrice* matriceEntree, int colonneDest, valeur
    scalaire, int colonneAjout);
59 // p - affecte à la colonne d'indice 'colonneDest' elle-même plus la colonne d'
    indice 'colonneAjout' multipliée par un scalaire 'scalaire' par effet de
    bord
60
61 void dilatation_ligne(matrice* matriceEntree, valeur scalaire, int ligne);
62 // p - affecte à la ligne d'indice 'ligne' elle-même multipliée par un
    scalaire 'scalaire' non-nul par effet de bord
63
64 void dilatation_colonne(matrice* matriceEntree, valeur scalaire, int colonne);
65 // p - affecte à la colonne d'indice 'colonne' elle-même multipliée par un
    scalaire 'scalaire' non-nul par effet de bord
66
67 matrice* inv_matrice(matrice* matriceEntree, bool verifie);
68 // f - calcule la matrice inverse de la matrice carrée inversible '
    matriceEntree' de manière non destructive
69
70 matrice* affichage_matrice(matrice* matriceEntree);
71 // p - affiche la matrice 'matriceEntree'

```

### 1.3 module\_matrice.c

```

1 /* -Fichier entête- */
2
3 #include "module_matrice.h"
4
5
6 /* -Fonctions- */
7
8 matrice* creer_matrice(int lignes, int colonnes)
9 {
10     if (lignes <= 0 || colonnes <= 0)
11     {
12         fprintf(stderr, "création_impossible:\n");
13         fprintf(stderr, "\t-> les tailles 'lignes' et 'colonnes' doivent être
            des entiers non nuls.\n");
14         fprintf(stderr, "\t\t'lignes': %d\n", lignes);

```

```

15     fprintf(stderr, "\t\t'colonnes': _{%d}\n", colonnes);
16     exit(EXIT_FAILURE);
17 }
18
19     matrice* matriceSortie = malloc(sizeof(matrice));
20     matriceSortie->lignes = lignes;
21     matriceSortie->colonnes = colonnes;
22     matriceSortie->contenu = malloc(sizeof(valeur)*lignes);
23
24     for (int ligne = 0; ligne < lignes; ++ligne)
25     {
26         matriceSortie->contenu[ligne] = malloc(sizeof(valeur)*colonnes);
27
28         for (int colonne = 0; colonne < colonnes; ++colonne)
29         {
30             matriceSortie->contenu[ligne][colonne] = 0.0;
31         }
32     }
33
34     return matriceSortie;
35 }
36
37 void supprimer_matrice(matrice* matriceEntree)
38 {
39     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
40     {
41         free(matriceEntree->contenu[ligne]);
42     }
43
44     free(matriceEntree->contenu);
45     free(matriceEntree);
46 }
47
48 matrice* sous_matrice(matrice* matriceEntree, int ligneDepart, int
    colonneDepart, int nbreLignes, int nbreColonnes)
49 {
50     matrice* matriceSortie = creer_matrice(nbreLignes, nbreColonnes);
51
52     for (int i = 0; i < nbreLignes; ++i)
53     {
54         for (int j = 0; j < nbreColonnes; ++j)
55         {
56             matriceSortie->contenu[i][j] = matriceEntree->contenu[i+
                ligneDepart][j+colonneDepart];
57         }
58     }
59
60     return matriceSortie;
61 }
62
63 matrice* add_matrice(matrice* matriceA, matrice* matriceB)
64 {
65     if (matriceA->lignes != matriceB->lignes || matriceA->colonnes != matriceB
        ->colonnes)
66     {
67         fprintf(stderr, "addition_impossible:\n");
68         fprintf(stderr, "\t->_les_deux_matrices_doivent_être_de_taille_
            identique.\n");
69         fprintf(stderr, "\t\t'matriceA': _{%d}_lignes\n", matriceA->lignes);

```

```

70     fprintf(stderr, "\\t\\t'matriceA ':_{%d}_colonnes\\n", matriceA->colonnes)
71     ;
72     fprintf(stderr, "\\t\\t'matriceB ':_{%d}_lignes\\n", matriceB->lignes);
73     fprintf(stderr, "\\t\\t'matriceB ':_{%d}_colonnes\\n", matriceB->colonnes)
74     ;
75     exit(EXIT_FAILURE);
76 }
77
78 matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceA->
79     colonnes);
80
81 for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
82 {
83     for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
84     {
85         matriceSortie->contenu[ligne][colonne] = matriceA->contenu[ligne][
86             colonne] + matriceB->contenu[ligne][colonne];
87     }
88 }
89
90 return matriceSortie;
91 }
92
93 matrice* soustrait_matrice(matrice* matriceA, matrice* matriceB)
94 {
95     if (matriceA->lignes != matriceB->lignes || matriceA->colonnes != matriceB
96         ->colonnes)
97     {
98         fprintf(stderr, "soustraction_impossible:\\n");
99         fprintf(stderr, "\\t->_les_deux_matrices_doivent_être_de_taille_
100             identique.\\n");
101         fprintf(stderr, "\\t\\t'matriceA ':_{%d}_lignes\\n", matriceA->lignes);
102         fprintf(stderr, "\\t\\t'matriceA ':_{%d}_colonnes\\n", matriceA->colonnes)
103         ;
104         fprintf(stderr, "\\t\\t'matriceB ':_{%d}_lignes\\n", matriceB->lignes);
105         fprintf(stderr, "\\t\\t'matriceB ':_{%d}_colonnes\\n", matriceB->colonnes)
106         ;
107         exit(EXIT_FAILURE);
108     }
109
110 matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceA->
111     colonnes);
112
113 for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
114 {
115     for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
116     {
117         matriceSortie->contenu[ligne][colonne] = matriceA->contenu[ligne][
118             colonne] - matriceB->contenu[ligne][colonne];
119     }
120 }
121
122 return matriceSortie;
123 }
124
125 matrice* mult_matrice(matrice* matriceA, matrice* matriceB)
126 {
127     if (matriceA->colonnes != matriceB->lignes)
128     {

```

```

119     fprintf(stderr, "multiplication_impossible:\n");
120     fprintf(stderr, "\t->la matrice 'matriceA' doit avoir autant de
        colonnes que la matrice 'matriceB' a de lignes.\n");
121     fprintf(stderr, "\t\t'matriceA': %d colonnes\n", matriceA->colonnes)
        ;
122     fprintf(stderr, "\t\t'matriceB': %d lignes\n", matriceB->lignes);
123     exit(EXIT_FAILURE);
124 }
125
126 int tailleCommune = matriceA->colonnes;
127 matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceB->
        colonnes);
128
129 for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
130 {
131     for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
132     {
133         valeur somme = 0;
134
135         for (int k = 0; k < tailleCommune; ++k)
136         {
137             somme += matriceA->contenu[ligne][k] * matriceB->contenu[k][
                colonne];
138         }
139
140         matriceSortie->contenu[ligne][colonne] = somme;
141     }
142 }
143
144 return matriceSortie;
145 }
146
147 matrice* transp_matrice(matrice* matriceEntree)
148 {
149     matrice* matriceSortie = creer_matrice(matriceEntree->colonnes,
        matriceEntree->lignes);
150
151     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
152     {
153         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
154         {
155             matriceSortie->contenu[ligne][colonne] = matriceEntree->contenu[
                colonne][ligne];
156         }
157     }
158
159     return matriceSortie;
160 }
161
162 matrice* dilatation_matrice(valeur scalaire, matrice* matriceEntree)
163 {
164     matrice* matriceSortie = creer_matrice(matriceEntree->lignes,
        matriceEntree->colonnes);
165
166     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
167     {
168         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
169         {
170             matriceSortie->contenu[ligne][colonne] = scalaire * matriceEntree

```

```

        ->contenu[ligne][colonne];
171     }
172 }
173
174     return matriceSortie;
175 }
176
177 valeur det_matrice(matrice* matriceEntree)
178 {
179     if (matriceEntree->colonnes != matriceEntree->lignes)
180     {
181         fprintf(stderr, "calculé du déterminant impossible:\n");
182         fprintf(stderr, "\t-> la matrice 'matriceEntree' doit avoir autant de
            colonnes que de lignes.\n");
183         fprintf(stderr, "\t\t'matriceEntree':_ {%d}_lignes\n", matriceEntree->
            lignes);
184         fprintf(stderr, "\t\t'matriceEntree':_ {%d}_colonnes\n", matriceEntree
            ->colonnes);
185         exit(EXIT_FAILURE);
186     }
187
188     int tailleCommune = matriceEntree->colonnes;
189
190     // cas d'arrêt
191
192     if (tailleCommune == 1)
193     {
194         return matriceEntree->contenu[0][0];
195     }
196
197     // recherche meilleur ligne/colonne (celle possédant le moins de 0)
198
199     int idMeilleur = 0;
200     int nbreZeroMax = 0;
201     int nbreZero = 0;
202     bool estVertical = false;
203
204     for (int ligne = 0; ligne < tailleCommune; ++ligne)
205     {
206         nbreZero = 0;
207
208         for (int colonne = 0; colonne < tailleCommune; ++colonne)
209         {
210             if (matriceEntree->contenu[ligne][colonne] == 0.0)
211             {
212                 nbreZero += 1;
213             }
214         }
215
216         if (nbreZero > nbreZeroMax)
217         {
218             nbreZeroMax = nbreZero;
219             idMeilleur = ligne;
220         }
221     }
222
223     for (int colonne = 0; colonne < tailleCommune; ++colonne)
224     {
225         nbreZero = 0;

```

```

226
227     for (int ligne = 0; ligne < tailleCommune; ++ligne)
228     {
229         if (matriceEntree->contenu[ligne][colonne] == 0.0)
230         {
231             nbreZero += 1;
232         }
233     }
234
235     if (nbreZero > nbreZeroMax)
236     {
237         nbreZeroMax = nbreZero;
238         idMeilleur = colonne;
239         estVertical = true;
240     }
241 }
242
243 // calcule du déterminant (on se ramène à la transposé si le calcule le
    // plus intéressant est sur une colonne)
244
245 // cas simple
246
247 if (nbreZeroMax == tailleCommune)
248 {
249     return 0.0;
250 }
251
252 // cas général
253
254 valeur det = 0.0;
255 valeur signe = (idMeilleur % 2 == 0) ? 1.0 : -1.0;
256 int ligneTemp;
257 int colonneTemp;
258
259 if (estVertical)
260 {
261     matriceEntree = transp_matrice(matriceEntree);
262 }
263
264 matrice* matriceTemp = creer_matrice(tailleCommune-1, tailleCommune-1);
265
266 for (int colonneEnCours = 0; colonneEnCours < tailleCommune; ++
    colonneEnCours)
267 {
268     if (matriceEntree->contenu[idMeilleur][colonneEnCours] != 0.0)
269     {
270         ligneTemp = 0;
271
272         for (int ligne = 0; ligne < tailleCommune; ++ligne)
273         {
274             colonneTemp = 0;
275
276             if (ligne != idMeilleur)
277             {
278                 for(int colonne = 0; colonne < tailleCommune; ++colonne)
279                 {
280                     if (colonne != colonneEnCours)
281                     {
282                         matriceTemp->contenu[ligneTemp][colonneTemp] =

```



```

283             matriceEntree->contenu[ligne][colonne];
284             colonneTemp++;
285         }
286     }
287     ligneTemp++;
288 }
289 }
290
291     det += signe * matriceEntree->contenu[idMeilleur][colonneEnCours]
292         * det_matrice(matriceTemp);
293 }
294     signe *= -1.0;
295 }
296
297 if (estVertical)
298 {
299     supprimer_matrice(matriceEntree);
300 }
301 supprimer_matrice(matriceTemp);
302
303 return det;
304 }
305
306 void echange_ligne(matrice* matriceEntree, int ligne1, int ligne2)
307 {
308     if (ligne1 >= matriceEntree->lignes || ligne1 < 0 || ligne2 >=
309         matriceEntree->lignes || ligne2 < 0)
310     {
311         fprintf(stderr, "échange_des_lignes_impossible:\n");
312         fprintf(stderr, "\t-> les_lignes 'ligne1' et 'ligne2' doivent_exister
313             .\n");
314         fprintf(stderr, "\t\t'ligne1': %d\n", ligne1);
315         fprintf(stderr, "\t\t'ligne2': %d\n", ligne2);
316         exit(EXIT_FAILURE);
317     }
318
319     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
320     {
321         valeur stockageTemp = matriceEntree->contenu[
322             ligne1][colonne];
323         matriceEntree->contenu[ligne1][colonne] = matriceEntree->contenu[
324             ligne2][colonne];
325         matriceEntree->contenu[ligne2][colonne] = stockageTemp;
326     }
327 }
328
329 void echange_colonne(matrice* matriceEntree, int colonnel, int colonne2)
330 {
331     if (colonnel >= matriceEntree->colonnes || colonnel < 0 || colonne2 >=
332         matriceEntree->colonnes || colonne2 < 0)
333     {
334         fprintf(stderr, "échange_des_colonnes_impossible:\n");
335         fprintf(stderr, "\t-> les_colonnes 'colonnel' et 'colonne2' doivent_
336             exister.\n");
337         fprintf(stderr, "\t\t'colonnel': %d\n", colonnel);
338         fprintf(stderr, "\t\t'colonne2': %d\n", colonne2);
339         exit(EXIT_FAILURE);

```

```

334     }
335
336     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
337     {
338         valeur stockageTemp                = matriceEntree->contenu[ligne
339             ][ colonne1 ];
340         matriceEntree->contenu[ligne][ colonne1 ] = matriceEntree->contenu[ligne
341             ][ colonne2 ];
342         matriceEntree->contenu[ligne][ colonne2 ] = stockageTemp;
343     }
344 }
345
346 void combinaison_lignes(matrice* matriceEntree, int ligneDest, valeur scalaire
347     , int ligneAjout)
348 {
349     if (ligneDest >= matriceEntree->lignes || ligneDest < 0 || ligneAjout >=
350         matriceEntree->lignes || ligneAjout < 0)
351     {
352         fprintf(stderr, "combinaison_des_lignes_impossible:\n");
353         fprintf(stderr, "\t-> les_lignes 'ligneDest' et 'ligneAjout' doivent_
354             exister.\n");
355         fprintf(stderr, "\t\t 'ligneDest':_{%d}\n", ligneDest);
356         fprintf(stderr, "\t\t 'ligneAjout':_{%d}\n", ligneAjout);
357         exit(EXIT_FAILURE);
358     }
359
360     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
361     {
362         matriceEntree->contenu[ligneDest][colonne] = matriceEntree->contenu[
363             ligneDest][colonne] + scalaire * matriceEntree->contenu[ligneAjout
364             ][colonne];
365     }
366 }
367
368 void combinaison_colonne(matrice* matriceEntree, int colonneDest, valeur
369     scalaire, int colonneAjout)
370 {
371     if (colonneDest >= matriceEntree->colonnes || colonneDest < 0 ||
372         colonneAjout >= matriceEntree->colonnes || colonneAjout < 0)
373     {
374         fprintf(stderr, "combinaison_des_colonnes_impossible:\n");
375         fprintf(stderr, "\t-> les_colonnes 'colonneDest' et 'colonneAjout'
376             doivent_exister.\n");
377         fprintf(stderr, "\t\t 'colonneDest':_{%d}\n", colonneDest);
378         fprintf(stderr, "\t\t 'colonneAjout':_{%d}\n", colonneAjout);
379         exit(EXIT_FAILURE);
380     }
381
382     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
383     {
384         matriceEntree->contenu[ligne][colonneDest] = matriceEntree->contenu[
385             ligne][colonneDest] + scalaire * matriceEntree->contenu[ligne][
386             colonneAjout];
387     }
388 }
389
390 void dilatation_ligne(matrice* matriceEntree, valeur scalaire, int ligne)
391 {
392     if (ligne >= matriceEntree->lignes || ligne < 0)

```

```

381     {
382         fprintf(stderr, "dilatation_de_la_ligne_impossible:\n");
383         fprintf(stderr, "\t->_la_ligne_'ligne'_doit_exister.\n");
384         fprintf(stderr, "\t\t'ligne':_{%d}\n", ligne);
385         exit(EXIT_FAILURE);
386     }
387
388     if (scalaire == 0)
389     {
390         fprintf(stderr, "dilatation_de_la_ligne_impossible:\n");
391         fprintf(stderr, "\t->_le_scalaire_'scalaire'_doit_être_non_nul.\n");
392         fprintf(stderr, "\t\t'scalaire':_{%f}\n", scalaire); // à modifier si
393             valeur change de type
394         exit(EXIT_FAILURE);
395     }
396
397     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
398     {
399         matriceEntree->contenu[ligne][colonne] = scalaire * matriceEntree->
400             contenu[ligne][colonne];
401     }
402
403 void dilatation_colonne(matrice* matriceEntree, valeur scalaire, int colonne)
404 {
405     if (colonne >= matriceEntree->colonnes || colonne < 0)
406     {
407         fprintf(stderr, "dilatation_de_la_colonne_impossible:\n");
408         fprintf(stderr, "\t->_la_colonne_'colonne'_doit_exister.\n");
409         fprintf(stderr, "\t\t'colonne':_{%d}\n", colonne);
410         exit(EXIT_FAILURE);
411     }
412
413     if (scalaire == 0)
414     {
415         fprintf(stderr, "dilatation_de_la_colonne_impossible:\n");
416         fprintf(stderr, "\t->_le_scalaire_'scalaire'_doit_être_non_nul.\n");
417         fprintf(stderr, "\t\t'scalaire':_{%f}\n", scalaire); // à modifier si
418             valeur change de type
419         exit(EXIT_FAILURE);
420     }
421
422     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
423     {
424         matriceEntree->contenu[ligne][colonne] = scalaire * matriceEntree->
425             contenu[ligne][colonne];
426     }
427 }
428
429 matrice* inv_matrice(matrice* matriceEntree, bool verifie)
430 {
431     if (matriceEntree->colonnes != matriceEntree->lignes)
432     {
433         fprintf(stderr, "calcul_de_l'inverse_impossible:\n");
434         fprintf(stderr, "\t->_la_matrice_'matriceEntree'_doit_avoir_autant_de_
435             colonnes_que_de_lignes.\n");
436         fprintf(stderr, "\t\t'matriceEntree':_{%d}_lignes\n", matriceEntree->
437             lignes);
438         fprintf(stderr, "\t\t'matriceEntree':_{%d}_colonnes\n", matriceEntree->

```

```

    ->colonnes);
434     exit(EXIT_FAILURE);
435 }
436
437 if (verifie)
438 {
439     valeur_det = det_matrice(matriceEntree);
440     printf("%f\n", det);
441
442     if (det == 0)
443     {
444         fprintf(stderr, "calcul_de_l'inverse_impossible:\n");
445         fprintf(stderr, "\t->_la_matrice_'matriceEntree'_est_de_dé
            terminant_nul.\n");
446         fprintf(stderr, "\t\tdéterminant:_{%f}\n", det); // à modifier si
            valeur change de type
447         exit(EXIT_FAILURE);
448     }
449 }
450
451 // mise en place
452
453 int tailleCommune = matriceEntree->colonnes;
454 matrice* matriceTemp = creer_matrice(tailleCommune, 2*tailleCommune);
455
456 for (int ligne = 0; ligne < tailleCommune; ++ligne)
457 {
458     for (int colonne = 0; colonne < tailleCommune; ++colonne)
459     {
460         matriceTemp->contenu[ligne][colonne] = matriceEntree->contenu[
            ligne][colonne];
461
462         if (ligne == colonne)
463         {
464             matriceTemp->contenu[ligne][colonne+tailleCommune] = 1;
465         }
466         else
467         {
468             matriceTemp->contenu[ligne][colonne+tailleCommune] = 0;
469         }
470     }
471 }
472
473 // algorithme de Gauss-Jordan
474
475 int lignePivot = -1;
476
477 for (int colonne = 0; colonne < tailleCommune; ++colonne)
478 {
479     int ligneMax = lignePivot+1;
480     int maximum = matriceTemp->contenu[lignePivot+1][colonne];
481
482     for (int ligne = lignePivot+2; ligne < tailleCommune; ++ligne)
483     {
484         if (matriceTemp->contenu[ligne][colonne] > maximum)
485         {
486             maximum = matriceTemp->contenu[ligne][colonne];
487             ligneMax = ligne;
488         }
489     }
490 }

```

```

489     }
490 }
491
492 if (matriceTemp->contenu[ligneMax][colonne] != 0)
493 {
494     lignePivot += 1;
495     dilatation_ligne(matriceTemp, 1/matriceTemp->contenu[ligneMax][
        colonne], ligneMax);
496
497     if (ligneMax != lignePivot)
498     {
499         echange_ligne(matriceTemp, lignePivot, ligneMax);
500     }
501
502     for (int ligne = 0; ligne < tailleCommune; ++ligne)
503     {
504         if (ligne != lignePivot)
505         {
506             combinaison_lignes(matriceTemp, ligne, (-1)*matriceTemp->
                contenu[ligne][colonne], lignePivot);
507         }
508     }
509 }
510
511 // recopie de la matrice inverse
512
513 matrice* matriceSortie = creer_matrice(tailleCommune, tailleCommune);
514 for (int ligne = 0; ligne < tailleCommune; ++ligne)
515 {
516     for (int colonne = 0; colonne < tailleCommune; ++colonne)
517     {
518         matriceSortie->contenu[ligne][colonne] = matriceTemp->contenu[
            ligne][colonne+tailleCommune];
519     }
520 }
521
522 supprimer_matrice(matriceTemp);
523 return matriceSortie;
524 }
525
526 matrice* affichage_matrice(matrice* matriceEntree)
527 {
528     printf("Affichage:\n");
529
530     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
531     {
532         printf("|");
533         for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
534         {
535             printf("_{%f}_", matriceEntree->contenu[ligne][colonne]);
536         }
537         printf("\n");
538     }
539 }
540

```

#### 1.4 module\_matrice.c

```

1  /* -Fichier entête- */

```

```

2
3 #include "module_matrice.h"
4
5
6 /* -Fonctions- */
7
8 matrice* creer_matrice(int lignes, int colonnes)
9 {
10     if (lignes <= 0 || colonnes <= 0)
11     {
12         fprintf(stderr, "création impossible:\n");
13         fprintf(stderr, "\t-> les tailles 'lignes' et 'colonnes' doivent être des entiers non nuls.\n");
14         fprintf(stderr, "\t\t'lignes': %d\n", lignes);
15         fprintf(stderr, "\t\t'colonnes': %d\n", colonnes);
16         exit(EXIT_FAILURE);
17     }
18
19     matrice* matriceSortie = malloc(sizeof(matrice));
20     matriceSortie->lignes = lignes;
21     matriceSortie->colonnes = colonnes;
22     matriceSortie->contenu = malloc(sizeof(valeur)*lignes);
23
24     for (int ligne = 0; ligne < lignes; ++ligne)
25     {
26         matriceSortie->contenu[ligne] = malloc(sizeof(valeur)*colonnes);
27
28         for (int colonne = 0; colonne < colonnes; ++colonne)
29         {
30             matriceSortie->contenu[ligne][colonne] = 0.0;
31         }
32     }
33
34     return matriceSortie;
35 }
36
37 void supprimer_matrice(matrice* matriceEntree)
38 {
39     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
40     {
41         free(matriceEntree->contenu[ligne]);
42     }
43
44     free(matriceEntree->contenu);
45     free(matriceEntree);
46 }
47
48 matrice* sous_matrice(matrice* matriceEntree, int ligneDepart, int
    colonneDepart, int nbreLignes, int nbreColonnes)
49 {
50     matrice* matriceSortie = creer_matrice(nbreLignes, nbreColonnes);
51
52     for (int i = 0; i < nbreLignes; ++i)
53     {
54         for (int j = 0; j < nbreColonnes; ++j)
55         {
56             matriceSortie->contenu[i][j] = matriceEntree->contenu[i+
                ligneDepart][j+colonneDepart];
57         }

```

```

58     }
59
60     return matriceSortie;
61 }
62
63 matrice* add_matrice(matrice* matriceA, matrice* matriceB)
64 {
65     if (matriceA->lignes != matriceB->lignes || matriceA->colonnes != matriceB
        ->colonnes)
66     {
67         fprintf(stderr, "addition_impossible:\n");
68         fprintf(stderr, "\t->_les_deux_matrices_doivent_être_de_taille_
            identique.\n");
69         fprintf(stderr, "\t\t'matriceA':_{%d}_lignes\n", matriceA->lignes);
70         fprintf(stderr, "\t\t'matriceA':_{%d}_colonnes\n", matriceA->colonnes)
            ;
71         fprintf(stderr, "\t\t'matriceB':_{%d}_lignes\n", matriceB->lignes);
72         fprintf(stderr, "\t\t'matriceB':_{%d}_colonnes\n", matriceB->colonnes)
            ;
73         exit(EXIT_FAILURE);
74     }
75
76     matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceA->
        colonnes);
77
78     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
79     {
80         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
81         {
82             matriceSortie->contenu[ligne][colonne] = matriceA->contenu[ligne][
                colonne] + matriceB->contenu[ligne][colonne];
83         }
84     }
85
86     return matriceSortie;
87 }
88
89 matrice* soustrait_matrice(matrice* matriceA, matrice* matriceB)
90 {
91     if (matriceA->lignes != matriceB->lignes || matriceA->colonnes != matriceB
        ->colonnes)
92     {
93         fprintf(stderr, "soustraction_impossible:\n");
94         fprintf(stderr, "\t->_les_deux_matrices_doivent_être_de_taille_
            identique.\n");
95         fprintf(stderr, "\t\t'matriceA':_{%d}_lignes\n", matriceA->lignes);
96         fprintf(stderr, "\t\t'matriceA':_{%d}_colonnes\n", matriceA->colonnes)
            ;
97         fprintf(stderr, "\t\t'matriceB':_{%d}_lignes\n", matriceB->lignes);
98         fprintf(stderr, "\t\t'matriceB':_{%d}_colonnes\n", matriceB->colonnes)
            ;
99         exit(EXIT_FAILURE);
100    }
101
102    matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceA->
        colonnes);
103
104    for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
105    {

```

```

106         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
107         {
108             matriceSortie->contenu[ligne][colonne] = matriceA->contenu[ligne][
                colonne] - matriceB->contenu[ligne][colonne];
109         }
110     }
111
112     return matriceSortie;
113 }
114
115 matrice* mult_matrice(matrice* matriceA, matrice* matriceB)
116 {
117     if (matriceA->colonnes != matriceB->lignes)
118     {
119         fprintf(stderr, "multiplication_impossible:\n");
120         fprintf(stderr, "\t->_la_matrice_'matriceA'_doit_avoir_autant_de_
            colonnes_que_la_matrice_'matriceB'_a_de_lignes.\n");
121         fprintf(stderr, "\t\t'matriceA':_{%d}_colonnes\n", matriceA->colonnes)
            ;
122         fprintf(stderr, "\t\t'matriceB':_{%d}_lignes\n", matriceB->lignes);
123         exit(EXIT_FAILURE);
124     }
125
126     int tailleCommune = matriceA->colonnes;
127     matrice* matriceSortie = creer_matrice(matriceA->lignes, matriceB->
        colonnes);
128
129     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
130     {
131         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
132         {
133             valeur somme = 0;
134
135             for (int k = 0; k < tailleCommune; ++k)
136             {
137                 somme += matriceA->contenu[ligne][k] * matriceB->contenu[k][
                    colonne];
138             }
139
140             matriceSortie->contenu[ligne][colonne] = somme;
141         }
142     }
143
144     return matriceSortie;
145 }
146
147 matrice* transp_matrice(matrice* matriceEntree)
148 {
149     matrice* matriceSortie = creer_matrice(matriceEntree->colonnes,
        matriceEntree->lignes);
150
151     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
152     {
153         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
154         {
155             matriceSortie->contenu[ligne][colonne] = matriceEntree->contenu[
                colonne][ligne];
156         }
157     }

```



```

158
159     return matriceSortie;
160 }
161
162 matrice* dilatation_matrice(valeur scalaire , matrice* matriceEntree)
163 {
164     matrice* matriceSortie = creer_matrice(matriceEntree->lignes ,
165         matriceEntree->colonnes);
166
167     for (int ligne = 0; ligne < matriceSortie->lignes; ++ligne)
168     {
169         for (int colonne = 0; colonne < matriceSortie->colonnes; ++colonne)
170         {
171             matriceSortie->contenu[ligne][colonne] = scalaire * matriceEntree
172                 ->contenu[ligne][colonne];
173         }
174     }
175
176     return matriceSortie;
177 }
178
179 valeur det_matrice(matrice* matriceEntree)
180 {
181     if (matriceEntree->colonnes != matriceEntree->lignes)
182     {
183         fprintf(stderr , "calculé_du_déterminant_impossible:\n");
184         fprintf(stderr , "\t->_la_matrice_'matriceEntree'_doit_avoir_autant_de_
185             colonnes_que_de_lignes.\n");
186         fprintf(stderr , "\t\t'matriceEntree':_{{%d}}_lignes\n", matriceEntree->
187             lignes);
188         fprintf(stderr , "\t\t'matriceEntree':_{{%d}}_colonnes\n", matriceEntree
189             ->colonnes);
190         exit(EXIT_FAILURE);
191     }
192
193     int tailleCommune = matriceEntree->colonnes;
194
195     // cas d'arrêt
196
197     if (tailleCommune == 1)
198     {
199         return matriceEntree->contenu[0][0];
200     }
201
202     // recherche meilleur ligne/colonne (celle possédant le moins de 0)
203
204     int idMeilleur = 0;
205     int nbreZeroMax = 0;
206     int nbreZero = 0;
207     bool estVertical = false;
208
209     for (int ligne = 0; ligne < tailleCommune; ++ligne)
210     {
211         nbreZero = 0;
212
213         for (int colonne = 0; colonne < tailleCommune; ++colonne)
214         {
215             if (matriceEntree->contenu[ligne][colonne] == 0.0)
216             {

```

```

212         nbreZero += 1;
213     }
214 }
215
216     if (nbreZero > nbreZeroMax)
217     {
218         nbreZeroMax = nbreZero;
219         idMeilleur = ligne;
220     }
221 }
222
223 for (int colonne = 0; colonne < tailleCommune; ++colonne)
224 {
225     nbreZero = 0;
226
227     for (int ligne = 0; ligne < tailleCommune; ++ligne)
228     {
229         if (matriceEntree->contenu[ligne][colonne] == 0.0)
230         {
231             nbreZero += 1;
232         }
233     }
234
235     if (nbreZero > nbreZeroMax)
236     {
237         nbreZeroMax = nbreZero;
238         idMeilleur = colonne;
239         estVertical = true;
240     }
241 }
242
243 // calcule du déterminant (on se ramène à la transposé si le calcule le
    plus intéressant est sur une colonne)
244
245 // cas simple
246
247 if (nbreZeroMax == tailleCommune)
248 {
249     return 0.0;
250 }
251
252 // cas général
253
254 valeur det = 0.0;
255 valeur signe = (idMeilleur % 2 == 0) ? 1.0 : -1.0;
256 int ligneTemp;
257 int colonneTemp;
258
259 if (estVertical)
260 {
261     matriceEntree = transp_matrice(matriceEntree);
262 }
263
264 matrice* matriceTemp = creer_matrice(tailleCommune-1, tailleCommune-1);
265
266 for (int colonneEnCours = 0; colonneEnCours < tailleCommune; ++
    colonneEnCours)
267 {
268     if (matriceEntree->contenu[idMeilleur][colonneEnCours] != 0.0)

```

```

269     {
270         ligneTemp = 0;
271
272         for (int ligne = 0; ligne < tailleCommune; ++ligne)
273         {
274             colonneTemp = 0;
275
276             if (ligne != idMeilleur)
277             {
278                 for (int colonne = 0; colonne < tailleCommune; ++colonne)
279                 {
280                     if (colonne != colonneEnCours)
281                     {
282                         matriceTemp->contenu[ligneTemp][colonneTemp] =
283                             matriceEntree->contenu[ligne][colonne];
284                         colonneTemp++;
285                     }
286                 }
287                 ligneTemp++;
288             }
289         }
290
291         det += signe * matriceEntree->contenu[idMeilleur][colonneEnCours]
292             * det_matrice(matriceTemp);
293     }
294     signe *= -1.0;
295 }
296
297 if (estVertical)
298 {
299     supprimer_matrice(matriceEntree);
300 }
301 supprimer_matrice(matriceTemp);
302
303 return det;
304 }
305
306 void echange_ligne(matrice* matriceEntree, int ligne1, int ligne2)
307 {
308     if (ligne1 >= matriceEntree->lignes || ligne1 < 0 || ligne2 >=
309         matriceEntree->lignes || ligne2 < 0)
310     {
311         fprintf(stderr, "échange_des_lignes_impossible:\n");
312         fprintf(stderr, "\t-> les_lignes 'ligne1' et 'ligne2' doivent_exister
313             .\n");
314         fprintf(stderr, "\t\t'ligne1': _{%d}\n", ligne1);
315         fprintf(stderr, "\t\t'ligne2': _{%d}\n", ligne2);
316         exit(EXIT_FAILURE);
317     }
318
319     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
320     {
321         valeur stockageTemp = matriceEntree->contenu[
322             ligne1][colonne];
323         matriceEntree->contenu[ligne1][colonne] = matriceEntree->contenu[
324             ligne2][colonne];
325         matriceEntree->contenu[ligne2][colonne] = stockageTemp;

```

```

322     }
323 }
324
325 void echange_colonne(matrice* matriceEntree, int colonne1, int colonne2)
326 {
327     if (colonne1 >= matriceEntree->colonnes || colonne1 < 0 || colonne2 >=
        matriceEntree->colonnes || colonne2 < 0)
328     {
329         fprintf(stderr, "échange_des_colonnes_impossible:\n");
330         fprintf(stderr, "\t-> les_colonnes 'colonne1' et 'colonne2' doivent_
            exister.\n");
331         fprintf(stderr, "\t\t'colonne1': _{%d}\n", colonne1);
332         fprintf(stderr, "\t\t'colonne2': _{%d}\n", colonne2);
333         exit(EXIT_FAILURE);
334     }
335
336     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
337     {
338         valeur stockageTemp = matriceEntree->contenu[ligne
            ][colonne1];
339         matriceEntree->contenu[ligne][colonne1] = matriceEntree->contenu[ligne
            ][colonne2];
340         matriceEntree->contenu[ligne][colonne2] = stockageTemp;
341     }
342 }
343
344 void combinaison_lignes(matrice* matriceEntree, int ligneDest, valeur scalaire
    , int ligneAjout)
345 {
346     if (ligneDest >= matriceEntree->lignes || ligneDest < 0 || ligneAjout >=
        matriceEntree->lignes || ligneAjout < 0)
347     {
348         fprintf(stderr, "combinaison_des_lignes_impossible:\n");
349         fprintf(stderr, "\t-> les_lignes 'ligneDest' et 'ligneAjout' doivent_
            exister.\n");
350         fprintf(stderr, "\t\t'ligneDest': _{%d}\n", ligneDest);
351         fprintf(stderr, "\t\t'ligneAjout': _{%d}\n", ligneAjout);
352         exit(EXIT_FAILURE);
353     }
354
355     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
356     {
357         matriceEntree->contenu[ligneDest][colonne] = matriceEntree->contenu[
            ligneDest][colonne] + scalaire * matriceEntree->contenu[ligneAjout
            ][colonne];
358     }
359 }
360
361 void combinaison_colonne(matrice* matriceEntree, int colonneDest, valeur
    scalaire, int colonneAjout)
362 {
363     if (colonneDest >= matriceEntree->colonnes || colonneDest < 0 ||
        colonneAjout >= matriceEntree->colonnes || colonneAjout < 0)
364     {
365         fprintf(stderr, "combinaison_des_colonnes_impossible:\n");
366         fprintf(stderr, "\t-> les_colonnes 'colonneDest' et 'colonneAjout' _
            doivent_exister.\n");
367         fprintf(stderr, "\t\t'colonneDest': _{%d}\n", colonneDest);
368         fprintf(stderr, "\t\t'colonneAjout': _{%d}\n", colonneAjout);

```

```

369     exit(EXIT_FAILURE);
370 }
371
372 for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
373 {
374     matriceEntree->contenu[ligne][colonneDest] = matriceEntree->contenu[
        ligne][colonneDest] + scalaire * matriceEntree->contenu[ligne][
        colonneAjout];
375 }
376 }
377
378 void dilatation_ligne(matrice* matriceEntree, valeur scalaire, int ligne)
379 {
380     if (ligne >= matriceEntree->lignes || ligne < 0)
381     {
382         fprintf(stderr, "dilatation_de_la_ligne_impossible:\n");
383         fprintf(stderr, "\t->_la_ligne_'ligne'_doit_exister.\n");
384         fprintf(stderr, "\t\t'ligne':_{"d}\n", ligne);
385         exit(EXIT_FAILURE);
386     }
387
388     if (scalaire == 0)
389     {
390         fprintf(stderr, "dilatation_de_la_ligne_impossible:\n");
391         fprintf(stderr, "\t->_le_scalaire_'scalaire'_doit_être_non_nul.\n");
392         fprintf(stderr, "\t\t'scalaire':_{"f}\n", scalaire); // à modifier si
            valeur change de type
393         exit(EXIT_FAILURE);
394     }
395
396     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
397     {
398         matriceEntree->contenu[ligne][colonne] = scalaire * matriceEntree->
            contenu[ligne][colonne];
399     }
400 }
401
402 void dilatation_colonne(matrice* matriceEntree, valeur scalaire, int colonne)
403 {
404     if (colonne >= matriceEntree->colonnes || colonne < 0)
405     {
406         fprintf(stderr, "dilatation_de_la_colonne_impossible:\n");
407         fprintf(stderr, "\t->_la_colonne_'colonne'_doit_exister.\n");
408         fprintf(stderr, "\t\t'colonne':_{"d}\n", colonne);
409         exit(EXIT_FAILURE);
410     }
411
412     if (scalaire == 0)
413     {
414         fprintf(stderr, "dilatation_de_la_colonne_impossible:\n");
415         fprintf(stderr, "\t->_le_scalaire_'scalaire'_doit_être_non_nul.\n");
416         fprintf(stderr, "\t\t'scalaire':_{"f}\n", scalaire); // à modifier si
            valeur change de type
417         exit(EXIT_FAILURE);
418     }
419
420     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
421     {
422         matriceEntree->contenu[ligne][colonne] = scalaire * matriceEntree->

```

```

        contenu[ligne][colonne];
423     }
424 }
425
426 matrice* inv_matrice(matrice* matriceEntree, bool verifie)
427 {
428     if (matriceEntree->colonnes != matriceEntree->lignes)
429     {
430         fprintf(stderr, "calcul_de_l'inverse_impossible:\n");
431         fprintf(stderr, "\t->_la_matrice_'matriceEntree'_doit_avoir_autant_de_
            colonnes_que_de_lignes.\n");
432         fprintf(stderr, "\t\t'matriceEntree':_{%d}_lignes\n", matriceEntree->
            lignes);
433         fprintf(stderr, "\t\t'matriceEntree':_{%d}_colonnes\n", matriceEntree
            ->colonnes);
434         exit(EXIT_FAILURE);
435     }
436
437     if (verifie)
438     {
439         valeur_det = det_matrice(matriceEntree);
440         printf("%f\n", det);
441
442         if (det == 0)
443         {
444             fprintf(stderr, "calcul_de_l'inverse_impossible:\n");
445             fprintf(stderr, "\t->_la_matrice_'matriceEntree'_est_de_dé
                terminant_nul.\n");
446             fprintf(stderr, "\t\tdéterminant:_{%f}\n", det); // à modifier si
                valeur change de type
447             exit(EXIT_FAILURE);
448         }
449     }
450 }
451
452 // mise en place
453
454 int tailleCommune = matriceEntree->colonnes;
455 matrice* matriceTemp = creer_matrice(tailleCommune, 2*tailleCommune);
456
457 for (int ligne = 0; ligne < tailleCommune; ++ligne)
458 {
459     for (int colonne = 0; colonne < tailleCommune; ++colonne)
460     {
461         matriceTemp->contenu[ligne][colonne] = matriceEntree->contenu[
            ligne][colonne];
462
463         if (ligne == colonne)
464         {
465             matriceTemp->contenu[ligne][colonne+tailleCommune] = 1;
466         }
467         else
468         {
469             matriceTemp->contenu[ligne][colonne+tailleCommune] = 0;
470         }
471     }
472 }
473
474 // algorithme de Gauss-Jordan

```

```

475
476     int lignePivot    = -1;
477
478     for (int colonne = 0; colonne < tailleCommune; ++colonne)
479     {
480         int ligneMax = lignePivot+1;
481         int maximum  = matriceTemp->contenu[lignePivot+1][colonne];
482
483         for (int ligne = lignePivot+2; ligne < tailleCommune; ++ligne)
484         {
485             if (matriceTemp->contenu[ligne][colonne] > maximum)
486             {
487                 maximum = matriceTemp->contenu[ligne][colonne];
488                 ligneMax = ligne;
489             }
490         }
491
492         if (matriceTemp->contenu[ligneMax][colonne] != 0)
493         {
494             lignePivot += 1;
495             dilatation_ligne(matriceTemp, 1/matriceTemp->contenu[ligneMax][
496                 colonne], ligneMax);
497
498             if (ligneMax != lignePivot)
499             {
500                 echange_ligne(matriceTemp, lignePivot, ligneMax);
501             }
502
503             for (int ligne = 0; ligne < tailleCommune; ++ligne)
504             {
505                 if (ligne != lignePivot)
506                 {
507                     combinaison_lignes(matriceTemp, ligne, (-1)*matriceTemp->
508                         contenu[ligne][colonne], lignePivot);
509                 }
510             }
511         }
512
513         // recopie de la matrice inverse
514
515         matrice* matriceSortie = creer_matrice(tailleCommune, tailleCommune);
516         for (int ligne = 0; ligne < tailleCommune; ++ligne)
517         {
518             for (int colonne = 0; colonne < tailleCommune; ++colonne)
519             {
520                 matriceSortie->contenu[ligne][colonne] = matriceTemp->contenu[
521                     ligne][colonne+tailleCommune];
522             }
523         }
524
525         supprimer_matrice(matriceTemp);
526         return matriceSortie;
527     }
528
529     matrice* affichage_matrice(matrice* matriceEntree)
530     {
531         printf("Affichage:\n");
532     }

```

```

531     for (int ligne = 0; ligne < matriceEntree->lignes; ++ligne)
532     {
533         printf("|");
534         for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
535         {
536             printf("_{%f}_", matriceEntree->contenu[ligne][colonne]);
537         }
538         printf("\\n");
539     }
540 }

```



## 2 code Ocaml : Affichage

### 2.1 Affichage\_FEM\_3D.ml

```
1
2 (*Pour Windows*)
3 (*
4 #load "graphics.cma";;
5 Graphics.open_graph "800x600";;
6 open Graphics;;
7 open_graph "720x1280";;
8 *)
9
10 (*Pour Linux*)
11 #use "topfind";;
12 #require "graphics";;
13 open Graphics;;
14 let hauteur=720 and largeur=1280;;
15 open_graph "";;
16 resize_window largeur hauteur;;
17
18 (*————Types, variables et fonctions outils————*)
19
20
21 type point = {x: float; y: float; z: float};;
22 type vecteur = {vx: float; vy: float; vz: float};;
23
24
25 let x0 = ref (float_of_int (size_x()/2))
26 and y0 = ref (float_of_int (size_y()/2))
27 and zoom = ref 150.;;
28
29
30 let base = ref ({vx = 1.; vy = 0.; vz = 0.},
31 {vx = 0.; vy = 1.; vz = 0.},
32 {vx = 0.; vy = 0.; vz = 1.});;
33
34
35 let vecteur pt1 pt2 = {vx = (pt2.x -. pt1.x); vy = (pt2.y -. pt1.y); vz = (pt2
    .z -. pt1.z)};;
36
37
38 let produit_scalaire vct1 vct2 = vct1.vx *. vct2.vx +. vct1.vy *. vct2.vy +.
    vct1.vz *. vct2.vz;;
39
40
41 let norme vct = sqrt(vct.vx**2. +. vct.vy**2. +. vct.vz**2.);;
42
43
44 let unitaire vct = {vx = (vct.vx /. (norme vct));
45                                     vy = (vct.vy /. (norme vct))
46                                     ;
47                                     vz = (vct.vz /. (norme vct))
48                                     };;
49
50 let produit_vectoriel vct1 vct2 =
51     {vx = (vct1.vy *. vct2.vz -. vct1.vz *. vct2.vy);
52       vy = (vct1.vz *. vct2.vx -. vct1.vx *. vct2.vz);
```

```

52         vz = (vct1.vx *. vct2.vy -. vct1.vy *. vct2.vx) };;
53
54 let dans_base pt bse = let vctb1, vctb2, vctb3 = bse and origine = {x = 0.; y
    = 0.; z = 0.} in
55     {x = (produit_scalaire (vecteur origine pt) vctb1);
56      y = (produit_scalaire (vecteur origine pt) vctb2);
57      z = (produit_scalaire (vecteur origine pt) vctb3)};;
58
59
60 let rotation_x vct theta =
61     {vx = vct.vx;
62      vy = vct.vy *. cos theta -. vct.vz *. sin theta;
63      vz = vct.vy *. sin theta +. vct.vz *. cos theta};;
64
65 let rotation_y vct theta =
66     {vx = vct.vx *. cos theta +. vct.vz *. sin theta;
67      vy = vct.vy;
68      vz = vct.vz *. cos theta -. vct.vx *. sin theta};;
69
70 let rotation_base_x theta = let vct1, vct2, vct3 = !base in
71     base := ((rotation_x vct1 theta),
72              (rotation_x vct2 theta),
73              (rotation_x vct3 theta));;
74
75 let rotation_base_y theta = let vct1, vct2, vct3 = !base in
76     base := ((rotation_y vct1 theta),
77              (rotation_y vct2 theta),
78              (rotation_y vct3 theta))
79 ;;
80
81 let projette pt = (int_of_float (!x0 +. !zoom *. pt.x), int_of_float (!y0 +. !
    zoom *. pt.y));;
82
83 let make_point (x,y,z) = { x = x; y=y; z=z};;
84
85 type element = int * int * float * float;; (*indice du noeud1, indice noeud2,
    module young, section*)
86
87 let make_element indice_noeud1 indice_noeud2 mod_young section = (
    indice_noeud1, indice_noeud2, mod_young, section);;
88
89 type item_affichable = Arete of (point*point*int*int) | Noeud of (point*int*
    int);; (*Arete(point de départ, point d'arrivée, epaisseur, couleur) et
    Noeud(point, rayon, couleur)*)
90
91
92 (*COULEURS ET EPAISSEUR DES ITEMS*)
93
94 (*Fonction pour resize des intervalles (proportionnalité)*)
95 let map debut1 fin1 debut2 fin2 x =
96     let t = (x -. debut1) /. (fin1 -. debut1) in
97     (1. -. t) *. debut2 +. t *. fin2
98 ;;
99
100
101 (*Détermine la couleur d'un noeud à l'aide de la norme de la force appliquée
    en ce noeud*)
102 let couleurs_noeuds noeuds forces =
103     let normes_forces = Array.map norme forces in

```

```

104      (*let max_norme_force force1 force2 = max (norme force1) (norme force2)
      *) in
105      let min_norme_force force1 force2 = min (norme force1) (norme force2)
      in *)
106      let max_force = Array.fold_left max normes_forces.(0) normes_forces
107      and min_force = Array.fold_left min normes_forces.(0) normes_forces
108      in
109      let tab_couleurs = Array.map (fun norme_force -> let teinte = int_of_float (
      map min_force max_force 0. 255. norme_force) in rgb teinte 0 (255 - teinte)
      ) normes_forces in
110      tab_couleurs;; (*Censé faire un dégradé du bleu au rouge*)
111
112      (* Calcule l'emplacement des noeuds après application de la force, ajoutant les
      déplacements *)
113      let noeuds_deplaces noeuds déplacements =
114          Array.map2 (fun point vecteur -> make_point ((point.x +.
      vecteur.vx), (point.y +. vecteur.vy), (point.y +. vecteur.vy)
      ) ) noeuds déplacements
115      ;;
116
117      (* Calcule l'épaisseur à afficher des aretes. Attention, renvoie le max et le
      min des sections (unité d'origine) *)
118      let epaisseurs_elements elements =
119          let sections = Array.map (fun (i1,i2,young,section) -> section)
      elements in
120      let max_section = Array.fold_left (fun section accu_section -> max section
      accu_section) 0. sections
121      and min_section = Array.fold_left (fun section accu_section -> min section
      accu_section) infinity sections
122      and min_epaisseur = 2. (*Constantes d'épaisseurs des traits*)
123      in let max_epaisseur = (max_section /. min_section) *. min_epaisseur
124      in
125      let tab_epaisseurs = Array.map (fun section -> int_of_float (map min_section
      max_section min_epaisseur max_epaisseur section) ) sections in
126
127      tab_epaisseurs, min_section, max_section
128      ;;
129
130      (*Crée un tableau des items à afficher (noeuds et arêtes), qui sera trié par
      la cote moyenne ou la cote en fonction de si c'est un poin ou une arete.
      Contient les aretes et noeuds avant et après application des forces*)
131
132      let make_items_affichables elements noeuds forces déplacements=
133          let epaisseurs,_,_ = epaisseurs_elements elements in
134          let couleurs = couleurs_noeuds noeuds forces in
135          let noeuds_depl = noeuds_deplaces noeuds déplacements in
136          let tab_aretes_originelles = Array.map2 (fun (i1,i2,young,section)
      epaisseur -> Arete(noeuds.(i1),noeuds.(i2),epaisseur,rgb 127 127
      127) ) elements epaisseurs
137          and tab_noeuds_originels = Array.map (fun point -> Noeud(point,5,rgb
      127 127 127)) noeuds
138          and tab_aretes_deplacees = Array.map2 (fun (i1,i2,young,section)
      epaisseur -> Arete(noeuds_depl.(i1),noeuds_depl.(i2),epaisseur,
      black) ) elements epaisseurs
139          and tab_noeuds_deplaces = Array.map2 (fun point couleur -> Noeud(point
      ,7,couleur)) noeuds_depl couleurs
140
141      in
142      Array.concat [tab_aretes_originelles;tab_noeuds_originels;
      tab_aretes_deplacees;tab_noeuds_deplaces]

```

```

143 ;;
144
145 (*Fonction auxiliaire pour tracer une arete*)
146 let trace_arete point1 point2 epaisseur couleur =
147     let pt1 = dans_base point1 !base
148     and pt2 = dans_base point2 !base
149     in
150         let x1,y1 = projette pt1
151         and x2,y2 = projette pt2 in
152             set_color couleur;
153             set_line_width epaisseur;
154             moveto x1 y1;
155             lineto x2 y2;
156 ;;
157
158 (*Fonction auxiliaire pour tracer un noeud*)
159 let trace_noeud point rayon couleur =
160     let epaisseur_trait = max 1 (int_of_float(float_of_int (rayon) *. 0.2)
161     ) in
162     let pt = dans_base point !base in
163     let x,y = projette pt in
164     set_color couleur;
165     fill_circle x y rayon;
166     set_color black;
167
168     set_line_width epaisseur_trait;
169     draw_circle x y rayon
170 ;;
171 (*-----Algorithme du Peintre-----*)
172
173
174 (*Profondeur d'un point dans la direction z*)
175 let cote pt = let proj = dans_base pt !base in proj.z;;
176
177 (*Profondeur pour une arete*)
178 let cote_moyenne (point1, point2) = (cote point1 +. cote point2)/. 2.;;
179
180 (*Tri des items pour l'algo du peintre*)
181 let tri tab clef = let taille = (Array.length tab) - 1 in
182     for i = 1 to taille do
183         let j = ref i and check = clef tab.(i) and temp = tab.(i) in
184         while !j > 0 && clef (tab.(!j-1)) > check do
185             tab.(!j) <- tab.(!j-1);
186             j := !j - 1;
187         done;
188         tab.(!j) <- temp;
189     done
190 ;;
191
192 let tri_items items_a_afficher =
193     let clef_tri item = match item with
194         | Arete(point1, point2, epaisseur, couleur_arete) -> cote_moyenne
195             (point1, point2)
196         | Noeud(p, rayon, couleur_noeud) -> cote p
197     in
198     tri items_a_afficher clef_tri
199 ;;

```

```

200
201 (*Affichage des items dans le bon ordre*)
202 let peintre_items items_a_afficher = let taille = Array.length
    items_a_afficher in
203     tri_items items_a_afficher;
204     for i = taille-1 downto 0 do
205         (*print_string "Traçage de l'item n° : ";print_int i;
            print_newline();*)
206         let item= items_a_afficher.(i) in
207         match item with
208         | Noeud(point,rayon,couleur) -> trace_noeud point rayon
            couleur
209         | Arete(point1,point2,epaisseur,couleur) -> trace_arete point1
            point2 epaisseur couleur
210     done
211 ;;
212
213
214 (*———Récupération des données dans un fichier extérieur———*)
215
216
217 (*Type tableau dynamique pour faciliter la récupération des données*)
218 type 'a tableau_dynamique = {mutable support: 'a array;
219
                                     mutable
                                     taille
                                     :
                                     int
                                     };;

220
221 let make_td element = {
222     support = Array.make 16 element; taille = 0};;
223
224 let ajoute td valeur =
225     if td.taille <> Array.length td.support then
226         begin
227             td.support.(td.taille) <- valeur;
228             td.taille <- td.taille + 1;
229         end
230     else
231         begin
232             let new_support = Array.make (td.taille*2) valeur in
233             for i = 0 to (td.taille-1) do
234                 new_support.(i) <- td.support.(i);
235             done;
236             td.support <- new_support;
237             td.taille <- td.taille + 1;
238         end;;
239
240
241 (*Fonction qui lit le fichier contenant les données et qui renvoie les
    tableaux contenant :
242     -les noeuds (indiqués par i)
243     -le déplacement des noeuds (déplacement du noeud i à l'indice i)
244     -les forces appliquées au noeud i
245     -les elements, ie (indice_noeud1,indice_noeud2,module_young,section)
    *)

```

```

246 let lecture_fichier nomFichier =
247     let fichier = open_in nomFichier in
248     let point_generique = make_point (0.,0.,0.) in
249     let element_generique = make_element 0 0 0. 0. in
250     let deplacement_generique = vecteur point_generique point_generique in
251     let force_generique = vecteur point_generique point_generique in
252     let noeuds = make_td point_generique in
253     let déplacements = make_td deplacement_generique in
254     let forces = make_td force_generique in
255     let elements = make_td element_generique in
256
257     let ligne = ref (input_line fichier) in
258     let nb_noeuds, nb_elements = Scanf.sscanf !ligne "%d;%d" (fun n1 n2 ->
        (n1,n2)) in
259
260     for i = 0 to nb_noeuds-1 do
261         ligne:= input_line fichier;
262         let x,dx,fx = Scanf.sscanf !ligne "%f;%f;%f" (fun x dx fx-> (x
            ,dx,fx)) in
263         ligne:= input_line fichier;
264         let y,dy,fy = Scanf.sscanf !ligne "%f;%f;%f" (fun y dy fy -> (
            y,dy,fy)) in
265         ligne:= input_line fichier;
266         let z,dz,fz = Scanf.sscanf !ligne "%f;%f;%f" (fun z dz fz-> (z
            ,dz,fz)) in
267
268             ajoute noeuds (make_point (x,y,z));
269             ajoute déplacements (vecteur point_generique (
                make_point (dx,dy,dz)));
270             ajoute forces (vecteur point_generique (make_point (fx
                ,fy,fz)));
271     done;
272     for i = 0 to nb_elements-1 do
273         ligne:=input_line fichier;
274         let element = Scanf.sscanf !ligne "%d;%d;%f;%f" (fun i1 i2
            module_young section -> (i1,i2,module_young,section)) in
275             ajoute elements element;
276     done;
277     close_in fichier;
278     let coupe_tableau_dyn tab = Array.sub (tab.support) 0 (tab.taille) in
279     coupe_tableau_dyn noeuds,coupe_tableau_dyn déplacements,coupe_tableau_dyn
        forces ,coupe_tableau_dyn elements
280 ;;
281
282
283
284 (*Boucle pour afficher la structure et la faire tourner à l'aide du clavier*)
285 let en_sync_items items_a_afficher =
286     auto_synchronize false;
287     display_mode false;
288     peintre_items items_a_afficher;
289
290     while true do
291         let event = wait_next_event [Key_pressed] in let key =
            event.key in
292         if key = 's' then y0 := !y0 -. 5.;
293         if key = 'z' then y0 := !y0 +. 5.;
294         if key = 'q' then x0 := !x0 -. 5.;
295         if key = 'd' then x0 := !x0 +. 5.;

```

```

296         if key = 'o' then rotation_base_y (0.05);
297         if key = 'l' then rotation_base_y (-0.05);
298         if key = 'k' then rotation_base_x (-0.05);
299         if key = 'm' then rotation_base_x (0.05);
300         if key = 'a' then zoom := !zoom +. 5.;
301         if key = 'e' then zoom := !zoom -. 5.;
302         clear_graph ();
303         peindre_items items_a_afficher;
304         synchronize ();
305     done
306 ;;
307
308
309 (*Fonction main : recupere les tableaux et lance la fonction en_sync_items.*)
310 let main () =
311     let noeuds,deplacements,forces,elements = lecture_fichier "resultat.txt" in
312     print_string "Nombre_d'éléments:_";
313     print_int (Array.length elements); print_newline();
314
315     (*let noeuds2 = noeuds_deplaces noeuds deplacements in
316     let tous_noeuds = Array.append noeuds noeuds2 in
317     let tous_elements = Array.append elements elements in
318     *)
319     let items_a_afficher = make_items_affichables elements noeuds forces
320         deplacements in
321     (*affiche_aretes_elements elements noeuds;*)
322     (*set_line_width 10;
323     lineto (size_x()/2) (size_y()/2); *)
324     synchronize ();
325     en_sync_items items_a_afficher;
326 ;;
327 main();;
```