Annexe TIPE

Étude et optimisation d'un outil d'ingénierie du bâtiment

Thomas CREUSET

numéro de candidat : 11909

Contents

1 code C : calculateur **2**
- 1.1 standard_lib.h . . . 2
- 1.2 module_matrice.h . . . 2
- 1.3 module_matrice.c . . . 3
- 1.4 module_matrice.c . . . 13

2 code Ocaml : Affichage **25**
- 2.1 Affichage_FEM_3D.ml . . . 25

1 code C : calculateur

1.1 standard_lib.h

```c
/* -Importations- */

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

1.2 module_matrice.h

```c
/* ~Matrices~ */

/* -Importations- */

#include "standard_lib.h"


/* -Types et structures- */

typedef double valeur;

struct matrice_s {
    int lignes;
    int colonnes;
    valeur** contenu;
};

typedef struct matrice_s matrice;;

/* -Déclarations fonctions (f) et procédures (p)- */

matrice* creer_matrice(int lignes, int colonnes);
// f - crée une matrice de dimension 'lignes'x'colonnes' initialisé à 0 (valeur par default du type 'valeur' à changer si ce dernier change)

void supprimer_matrice(matrice* matriceEntree);
// p - vide la mémoire utilisée par la matrice 'matriceEntree'

matrice* sous_matrice(matrice* matriceEntree, int ligneDepart, int colonneDepart, int nbreLignes, int nbreColonnes);
// f - créer la sous matrice comme spécifiée

matrice* add_matrice(matrice* matriceA, matrice* matriceB);
// f - additionne les matrices 'matriceA' et 'matriceB' de manière non destructive

matrice* soustract_matrice(matrice* matriceA, matrice* matriceB);
// f - soustrait la matrice 'matriceB' à la matrice 'matriceB' de manière non destructive

matrice* mult_matrice(matrice* matriceA, matrice* matriceB);
// f - multiplie les matrices 'matriceA' et 'matriceB' de manière non destructive

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 colonne1, int colonne2);
53 // p - échange les colonnes d'indice 'colonne1' 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          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* soustract_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,
            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, "calcule 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 possèdant 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 transoposé 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] =
```

```
                                matriceEntree->contenu[ligne][colonne];
283                             colonneTemp++;
284                         }
285                     }
286
287                     ligneTemp++;
288                 }
289             }
290
291             det += signe * matriceEntree->contenu[idMeilleur][colonneEnCours]
                    * det_matrice(matriceTemp);
292         }
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 >=
            matriceEntree->lignes || ligne2 < 0)
309     {
310         fprintf(stderr, "échange des lignes impossible:\n");
311         fprintf(stderr, "\t-> les lignes 'ligne1' et 'ligne2' doivent exister
                .\n");
312         fprintf(stderr, "\t\t'ligne1': {%d}\n", ligne1);
313         fprintf(stderr, "\t\t'ligne2': {%d}\n", ligne2);
314         exit(EXIT_FAILURE);
315     }
316
317     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
318     {
319         valeur stockageTemp                         = matriceEntree->contenu[
                ligne1][colonne];
320         matriceEntree->contenu[ligne1][colonne] = matriceEntree->contenu[
                ligne2][colonne];
321         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, "calcule 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, "calcule 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][
                    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 
512     // recopie de la matrice inverse
513 
514     matrice* matriceSortie = creer_matrice(tailleCommune, tailleCommune);
515     for (int ligne = 0; ligne < tailleCommune; ++ligne)
516     {
517         for (int colonne = 0; colonne < tailleCommune; ++colonne)
518         {
519             matriceSortie->contenu[ligne][colonne] = matriceTemp->contenu[
                    ligne][colonne+tailleCommune];
520         }
521     }
522 
523     supprimer_matrice(matriceTemp);
524     return matriceSortie;
525 }
526 
527 matrice* affichage_matrice(matrice* matriceEntree)
528 {
529     printf("Affichage:\n");
530 
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 }
```

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* soustract_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,
          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, "calcule 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 possèdant 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 transoposé 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] =
                                matriceEntree->contenu[ligne][colonne];
283                             colonneTemp++;
284                         }
285                     }
286 
287                     ligneTemp++;
288                 }
289             }
290 
291             det += signe * matriceEntree->contenu[idMeilleur][colonneEnCours]
                * det_matrice(matriceTemp);
292         }
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 >=
        matriceEntree->lignes || ligne2 < 0)
309     {
310         fprintf(stderr, "échange des lignes impossible:\n");
311         fprintf(stderr, "\t-> les lignes 'ligne1' et 'ligne2' doivent exister
            .\n");
312         fprintf(stderr, "\t\t'ligne1': {%d}\n", ligne1);
313         fprintf(stderr, "\t\t'ligne2': {%d}\n", ligne2);
314         exit(EXIT_FAILURE);
315     }
316 
317     for (int colonne = 0; colonne < matriceEntree->colonnes; ++colonne)
318     {
319         valeur stockageTemp                         = matriceEntree->contenu[
            ligne1][colonne];
320         matriceEntree->contenu[ligne1][colonne] = matriceEntree->contenu[
            ligne2][colonne];
321         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, "calcule 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, "calcule 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][
                    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
512     // recopie de la matrice inverse
513
514     matrice* matriceSortie = creer_matrice(tailleCommune, tailleCommune);
515     for (int ligne = 0; ligne < tailleCommune; ++ligne)
516     {
517         for (int colonne = 0; colonne < tailleCommune; ++colonne)
518         {
519             matriceSortie->contenu[ligne][colonne] = matriceTemp->contenu[
                    ligne][colonne+tailleCommune];
520         }
521     }
522
523     supprimer_matrice(matriceTemp);
524     return matriceSortie;
525 }
526
527 matrice* affichage_matrice(matrice* matriceEntree)
528 {
529     printf("Affichage:\n");
530
```

```
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  (*Pour Windows*)
2  #load "graphics.cma";;
3  Graphics.open_graph "800x600";;
4  open Graphics;;
5  open_graph "720x1280";;
6
7
8  (*Pour Linux
9  #use "topfind";;
10 #require "graphics";;
11 open Graphics;;
12 let hauteur=720 and largeur=1280;;
13 open_graph "";;
14 resize_window largeur hauteur;;*)
15
16
17 (*-------------Types, variables et fonctions outils--------------*)
18
19 type point   = {x: float;  y: float;  z: float};;
20 type vecteur = {vx: float; vy: float; vz: float};;
21
22 let x0   = ref (float_of_int (size_x()/2))
23 and y0   = ref (float_of_int (size_y()/2))
24 and zoom = ref 150.;;
25
26 let base = ref ({vx = 1.; vy = 0.; vz = 0.},
27 {vx = 0.; vy = 1.; vz = 0.},
28 {vx = 0.; vy = 0.; vz = 1.});;
29
30 let vecteur pt1 pt2 = {vx = (pt2.x -. pt1.x); vy = (pt2.y -. pt1.y); vz = (pt2
       .z -. pt1.z)};;
31
32 let produit_scalaire vct1 vct2 = vct1.vx *. vct2.vx +. vct1.vy *. vct2.vy +.
       vct1.vz *. vct2.vz;;
33
34 let norme vct = sqrt(vct.vx**2. +. vct.vy**2. +. vct.vz**2.);;
35
36 let unitaire vct = {vx = (vct.vx /. (norme vct));
37                                                    vy = (vct.vy /. (norme vct))
                                                       ;
38                                                    vz = (vct.vz /. (norme vct))
                                                       };;
39
40 let produit_vectoriel vct1 vct2 =
41         {vx = (vct1.vy *. vct2.vz -. vct1.vz *. vct2.vy);
42          vy = (vct1.vz *. vct2.vx -. vct1.vx *. vct2.vz);
43          vz = (vct1.vx *. vct2.vy -. vct1.vy *. vct2.vx)};;
44
45 let dans_base pt bse = let vctb1, vctb2, vctb3 = bse and origine = {x = 0.; y
       = 0.; z = 0.} in
46         {x = (produit_scalaire (vecteur origine pt) vctb1);
47          y = (produit_scalaire (vecteur origine pt) vctb2);
48          z = (produit_scalaire (vecteur origine pt) vctb3)};;
49
50 let rotation_x vct theta =
```

```
51          {vx = vct.vx;
52           vy = vct.vy *. cos theta -. vct.vz *. sin theta;
53           vz = vct.vy *. sin theta +. vct.vz *. cos theta};;
54
55  let rotation_y vct theta =
56          {vx = vct.vx *. cos theta +. vct.vz *. sin theta;
57           vy = vct.vy;
58           vz = vct.vz *. cos theta -. vct.vx *. sin theta};;
59
60  let rotation_base_x theta = let vct1, vct2, vct3 = !base in
61          base := ((rotation_x vct1 theta),
62                                  (rotation_x vct2 theta),
63                                  (rotation_x vct3 theta));;
64
65  let rotation_base_y theta = let vct1, vct2, vct3 = !base in
66          base := ((rotation_y vct1 theta),
67                                  (rotation_y vct2 theta),
68                                  (rotation_y vct3 theta));;
69
70  let projette pt = (int_of_float (!x0 +. !zoom *. pt.x), int_of_float (!y0 +. !
        zoom *. pt.y));;
71
72  let make_point (x,y,z) = { x = x; y=y; z=z;};;
73
74  type element = int * int * float * float;; (*indice du noeud1, indice noeud2,
        module young, section*)
75
76  let make_element indice_noeud1 indice_noeud2 mod_young section = (
        indice_noeud1,indice_noeud2,mod_young,section);;
77
78  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)*)
79
80
81  (*COULEURS ET EPAISSEUR DES ITEMS*)
82
83  (*Fonction pour resize des intervalles (proportionnalité)*)
84  let map debut1 fin1 debut2 fin2 x =
85    let t = (x -. debut1)/.(fin1 -. debut1) in
86    (1. -. t)*. debut2 +. t*.fin2;;
87
88  (*Détermine la couleur d'un noeud à l'aide de la norme de la force appliquée
        en ce noeud*)
89  let couleurs_noeuds noeuds forces =
90          let normes_forces = Array.map norme forces in
91          (*let max_norme_force force1 force2 = max (norme force1) (norme force2
                ) in
92          let min_norme_force force1 force2 = min (norme force1) (norme force2)
                in *)
93    let max_force  = Array.fold_left max normes_forces.(0) normes_forces
94    and min_force = Array.fold_left min normes_forces.(0) normes_forces
95  in
96  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
97  tab_couleurs;; (*Censé faire un dégradé du bleu au rouge*)
98
99  (*Calcule l'emplacement des noeuds après application de la force, ajoutant les
```

```
    déplacements*)
100 let noeuds_deplaces noeuds deplacements =
101                 Array.map2 (fun point vecteur ->
102                 print_string "Coordonnées : "; print_float point.x; print_string
                        " "; print_float point.y; print_string " "; print_float
                        point.z;
103                 print_newline();
104                 print_string "Deplacement : "; print_float vecteur.vx;
                        print_string " "; print_float vecteur.vy; print_string " ";
                        print_float vecteur.vz;
105                 print_newline();
106                 print_newline();
107                 make_point ((point.x +. vecteur.vx),(point.y +. vecteur.vy),(
                        point.z +. vecteur.vz)) ) noeuds deplacements ;;
108
109 (*Calcule l'épaisseur à afficher des aretes. Attention, renvoie le max et le
    min des sections (unité d'origine)*)
110 let epaisseurs_elements elements =
111         let sections = Array.map (fun (i1,i2,young,section) -> section)
                elements in
112 let max_section = Array.fold_left (fun section accu_section -> max section
    accu_section) 0. sections
113 and min_section = Array.fold_left (fun section accu_section -> min section
    accu_section) infinity sections
114 and min_epaisseur = 2. (*Constantes d'épaisseurs des traits*)
115 in let max_epaisseur = (max_section/. min_section) *. min_epaisseur
116 in
117 let tab_epaisseurs = Array.map (fun section -> int_of_float (map min_section
    max_section min_epaisseur max_epaisseur section) ) sections in
118 tab_epaisseurs,min_section,max_section;;
119
120 (*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.
121           Contient les aretes et noeuds avant et après application des forces*)
122 let make_items_affichables elements noeuds forces deplacements=
123         let epaisseurs,_,_ = epaisseurs_elements elements in
124         let couleurs = couleurs_noeuds noeuds forces in
125         let noeuds_depl = noeuds_deplaces noeuds deplacements in
126         let tab_aretes_originelles = Array.map2 (fun (i1,i2,young,section)
                epaisseur -> Arete(noeuds.(i1),noeuds.(i2),epaisseur,rgb 127 127
                127) ) elements epaisseurs
127         and tab_noeuds_originels = Array.map (fun point -> Noeud(point,3,rgb
                127 127 127)) noeuds
128         and tab_aretes_deplacees = Array.map2 (fun (i1,i2,young,section)
                epaisseur -> Arete(noeuds_depl.(i1),noeuds_depl.(i2),epaisseur,
                black) ) elements epaisseurs
129         and tab_noeuds_deplaces = Array.map2 (fun point couleur -> Noeud(point
                ,7,couleur)) noeuds_depl couleurs
130         in
131         Array.concat [tab_aretes_originelles;tab_noeuds_originels;
                tab_aretes_deplacees;tab_noeuds_deplaces];;
132
133 (*Fonction auxiliaire pour tracer une arete*)
134 let trace_arete point1 point2 epaisseur couleur =
135         let pt1 = dans_base point1 !base
136         and pt2 = dans_base point2 !base
137   in
138         let x1,y1 = projette pt1
139         and x2,y2 = projette pt2 in
```

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

```
                    point2 epaisseur couleur
194         done;;
195
196
197 (*-----Récuperation des données dans un fichier extérieur----------*)
198
199 (*Type tableau dynamique pour faciliter la récupération des données*)
200 type 'a tableau_dynamique = {mutable support: 'a array;
201                                                                 mutable
                                                                     taille
                                                                     :
                                                                     int
                                                                     };;
202
203 let make_td element = {
204         support = Array.make 16 element; taille = 0};;
205
206 let ajoute td valeur =
207         if td.taille <> Array.length td.support then
208                 begin
209                 td.support.(td.taille) <- valeur;
210                 td.taille <- td.taille + 1;
211                 end
212         else
213                 begin
214                 let new_support = Array.make (td.taille*2) valeur in
215                 for i = 0 to (td.taille-1) do
216                         new_support.(i) <- td.support.(i);
217                 done;
218                 td.support <- new_support;
219                 td.taille <- td.taille + 1;
220                 end;;
221
222 (*Fonction qui lit le fichier contenant les données et qui renvoie les
        tableaux contenant :
223          -les noeuds (indicés par i)
224          -le déplacement des noeuds (deplacement du noeud i à l'indice i)
225          -les forces appliquées au noeud i
226          -les elements, ie (indice_noeud1,indice_noeud2,module_young,section)
                *)
227 let lecture_fichier nomFichier =
228         let fichier = open_in nomFichier in
229         let point_generique = make_point (0.,0.,0.) in
230         let element_generique = make_element 0 0 0. 0.  in
231         let deplacement_generique = vecteur point_generique point_generique in
232         let force_generique = vecteur point_generique point_generique in
233         let noeuds  = make_td point_generique in
234         let deplacements = make_td deplacement_generique in
235         let forces = make_td force_generique in
236         let elements = make_td element_generique in
237
238         let ligne = ref (input_line fichier) in
239         let nb_noeuds, nb_elements = Scanf.sscanf !ligne "%d;%d" (fun n1 n2 ->
                (n1,n2)) in
240
241         for i = 0 to nb_noeuds-1 do
```

```
242                 ligne:= input_line fichier;
243                 let x,dx,fx = Scanf.sscanf !ligne "%f;%f;%f" (fun x dx fx-> (x
                        ,dx,fx)) in
244                 ligne:= input_line fichier;
245                 let y,dy,fy = Scanf.sscanf !ligne "%f;%f;%f" (fun y dy fy -> (
                        y,dy,fy)) in
246                 ligne:= input_line fichier;
247                 let z,dz,fz = Scanf.sscanf !ligne "%f;%f;%f" (fun z dz fz-> (z
                        ,dz,fz)) in
248
249                         ajoute noeuds (make_point (x,y,z));
250                         ajoute deplacements (vecteur point_generique (
                            make_point (dx,dy,dz)));
251                         ajoute forces (vecteur point_generique (make_point (fx
                            ,fy,fz)));
252         done;
253         for i = 0 to nb_elements-1 do
254                 ligne:=input_line fichier;
255                 let element = Scanf.sscanf !ligne "%d;%d;%f;%f" (fun i1 i2
                        module_young section -> (i1,i2,module_young,section)) in
256                         ajoute elements element;
257         done;
258         close_in fichier;
259         let coupe_tableau_dyn tab = Array.sub (tab.support) 0 (tab.taille)  in
260   coupe_tableau_dyn noeuds,coupe_tableau_dyn deplacements,coupe_tableau_dyn
          forces ,coupe_tableau_dyn elements;;
261
262 (*Boucle pour afficher la structure et la faire tourner à l'aide du clavier*)
263 let en_sync_items items_a_afficher =
264         auto_synchronize false;
265         display_mode false;
266         peintre_items items_a_afficher;
267
268                 while true do
269                         let event = wait_next_event [Key_pressed] in let key =
                                event.key in
270                         if key = 's' then y0 := !y0 -. 5.;
271                         if key = 'z' then y0 := !y0 +. 5.;
272                         if key = 'q' then x0 := !x0 -. 5.;
273                         if key = 'd' then x0 := !x0 +. 5.;
274                         if key = 'o' then rotation_base_y (0.05);
275                         if key = 'l' then rotation_base_y (-0.05);
276                         if key = 'k' then rotation_base_x (-0.05);
277                         if key = 'm' then rotation_base_x (0.05);
278                         if key = 'a' then zoom := !zoom +. 5.;
279                         if key = 'e' then zoom := !zoom -. 5.;
280                         clear_graph ();
281                         peintre_items items_a_afficher;
282                         synchronize ();
283                 done;;
284
285 (*Fonction main : récupere les tableaux et lance la fonction en_sync_items.*)
286 let main () =
287 let noeuds,deplacements,forces,elements = lecture_fichier "resultat.txt" in
288 print_string "Nombre d'éléments : ";
289 print_int (Array.length elements); print_newline();
290
291 (*let noeuds2 = noeuds_deplaces noeuds deplacements in
292 let tous_noeuds = Array.append noeuds noeuds2 in
```

```
293 let tous_elements = Array.append elements elements in
294 *)
295 let items_a_afficher = make_items_affichables elements noeuds forces
        deplacements in
296 (*affiche_aretes_elements elements noeuds;*)
297 (*set_line_width 10;
298 lineto (size_x()/2) (size_y()/2); *)
299 synchronize ();
300 en_sync_items items_a_afficher;;
301
302 main();;
```