

src/src/face.js

```
1  /**
2   * @file Describes ADSODA face
3   * @author Jeff Bigot <jeff@raktres.net> after Greg Ferrar
4   * @class Face
5   */
6
7  import {
8    constantAdd,
9    projectVector,
10   positionPoint,
11   intersectHyperplanes,
12   isCornerEqual,
13   getCoordinate,
14   getConstant,
15   moizeAmongIndex,
16   normalize,
17   multiplyMatrices,
18   subtract,
19   matnorm,
20   matdot,
21   flip
22 } from './halfspace.js'
23 import { NDObject } from './ndobject.js'
24 import * as P from './parameters'
25
26 /**
27  * 3D specific function
28  * @param {*} x
29  * @param {*} y
30  */
31 function cross3d (x, y) {
32   const [x1, x2, x3] = Array.from(x)
33   const [y1, y2, y3] = Array.from(y)
34   return [
35     x2 * y3 - x3 * y2,
36     x3 * y1 - x1 * y3,
37     x1 * y2 - x2 * y1
38   ]
39 }
40
41 class Face extends NDObject {
42   /**
43    * create a face * This is the interface to the Halfspace class. An Halfspace
44    is half of an n-space. it is described by the equation of the bounding
45    hyperplane. A point is considered to be inside the halfspace if the left side of
46    the equation is greater than the right side when the coordinates of the point
47    (vector) are plugged in. The first n coefficients can also be viewed as the
48    normal vector, and the last as a constant which determines which of the possible
49    parallel hyperplanes in n-space this one is.<br> A halfspace is represented by
50    the equation of the bounding hyperplane, so a Hyperplane is really the same as a
51    Halfspace.
```

```
44 * @constructor Face
45 * @param {*} vector
46 */
47 constructor (vector) {
48   super('Face')
49   this.equ = normalize(vector.map(parseFloat))
50   this.touchingCorners = []
51   this.adjacentRefs = new Set()
52   this.dim = this.equ.length - 1
53 }
54
55 /**
56 * @returns JSON face description
57 */
58 exportToJSON () {
59   return `{ "face" : ${JSON.stringify(this.equ)} }`
60 }
61
62 /**
63 *
64 * @param {*} json
65 */
66 static importFromJSON (json) {
67   return new Face(json.face)
68 }
69
70 /**
71 * @returns text face description
72 */
73 logDetail () {
74   return `Face name : ${this.name} \n --- halfspace : ${
75     this.equ
76   } \n --- touching corners ${JSON.stringify(
77     this.touchingCorners
78   )} \n --- nb of adjacent faces : ${this.adjacentRefs.length} `
79 }
80
81 /**
82 *
83 */
84 eraseTouchingCorners () {
85   this.touchingCorners.length = 0
86 }
87
88 /**
89 * @returns face this
90 */
91 eraseAdjacentFaces () {
92   this.adjacentRefs.clear()
93 }
94
95 /**
96 * translate the face following the given vector.<br>
```

```
97     * Translation doesn't change normal vector, Just the constant term need to be
    changed.
98     * new constant = old constant - dot(normal, vector)<br>
99     * @param {*} vector the vector indicating the direction and distance to
    translate this halfspace.
100     * @todo vérifie que mutation nécessaire
101     * @returns face this
102     */
103     translate (vector) {
104         // Given a halfspace
105         //
106         //  $a_1x_1 + \dots + a_nx_n + k = 0$ 
107         //
108         // We can translate by vector (v1, v2, ..., vn) by substituting (xi - vi)
    for
109         // all xi, yielding
110         //
111         //  $a_1(x_1-v_1) + \dots + a_n(x_n-v_n) + k = 0$ 
112         //
113         // This simplifies to
114         //
115         //  $a_1x_1 + \dots + a_nx_n + (k - a_1v_1 - \dots - a_nv_n) = 0$ 
116         //
117         // So all we have to do is change the constant term. This is as expected,
118         // since translating should not change the normal vector (the first n-1
    terms).
119         //
120
121         const dot = matdot(this.equ.slice(0, -1), vector)
122         constantAdd(this.equ, dot)
123         return this
124     }
125
126     /**
127     * This method applies a matrix transformation to this Halfspace.
128     * @param {matrix} matrix the matrix of the transformation to apply.
129     * @todo vérifie que mutation nécessaire
130     * @returns face this
131     */
132     transform (matrix) {
133         //
134         // The normal of the tranformed halfspace can be found with a simple matrix
135         // multiplication.
136         const coords = multiplyMatrices(matrix, this.equ.slice(0, -1))
137
138         // console.error('trans', this.equ, coords )
139         // The constant is more difficult. Here I have solved this
140         // by finding a point on the original halfspace (by checking axes for
    intersections)
141         // and transforming that point as well. The transformed point lies on the
142         // transformed halfspace, so the constant term can be computed by plugging
    the
```

```
143 // transformed point into the equation of the transformed halfspace (the
coefficients
144 // being the coordinates of the transformed normal and the constant unknown)
and
145 // solving for the constant.
146 //
147
148 // get non 0 coordinate
149 const coordindex = this.equ.findIndex(x => Math.abs(x) > P.VERY_SMALL_NUM)
150 /* let max = 0
151 let coordindex = false
152 for (let index = 0; index < this.equ.length - 1; index++) {
153   if (this.equ[index] > max) {
154     max = this.equ[index]
155     coordindex = index
156   }
157 }
158 */
159 // TODO vérifier si utilisation not small_value x!=0
160 // const intercept = -getConstant(this.equ) / getCoordinate(this.equ,
coordindex)
161 const intercept = -getConstant(this.equ) / getCoordinate(this.equ,
coordindex) // max
162 // At this point we have found a point on the halfspace. This point is
163 // (0, 0, ..., intercept, ..., 0, 0), where intercept is the ith coordinate
164 // and all other coordinates are 0. Since this is a highly sparse and
165 // predictable vector. We will NOT actually plug all these coordinates
166 // into a Vector and use matrix multiplication; rather, we will take
167 // advantage of the fact that multiplication by such a vector yields
168 // a vector which is simply the ith column of m multiplied by intercept.
169 // We skip another step by plugging the coordinates of this product
170 // directly into the transformed equation.
171 //
172
173 let sum = 0
174 const n = coords.length
175 for (let i = 0; i < n; i++) {
176   sum += matrix[i][coordindex] * intercept * coords[i]
177 }
178 this.equ = [...coords, -sum]
179 return this
180 }
181
182 /**
183 *
184 * @param {*} axe
185 * @returns boolean if it is a backface
186 */
187 isBackFace (axe) {
188   return this.orientation(axe) < 0
189 }
190
191 /**
```

```
192 *
193 * @param {*} axe
194 * @returns number sign of coef
195 */
196 orientation (axe) {
197     const val = this.equ[axe]
198     if (val < -P.VERY_SMALL_NUM) return -1
199     if (val > P.VERY_SMALL_NUM) return 1
200     return 0
201 }
202
203 /**
204 *
205 * @param {*} point
206 * @param {*} axe
207 * @returns boolean if point is valid to be used for order
208 * TODO: comprendre pourquoi ce n'est pas utilisé
209 */
210 validForOrder (point, axe) {
211     return !this.pointInsideFace(point) && this.orientation(axe) !== 0
212 }
213
214 /**
215 * This method negates all terms in the equation of this halfspace. This
216 * flips the normal without changing the boundary halfplane.
217 * @todo évaluer l'impact de l'utilisation de ...
218 */
219 flip () {
220     this.equ = flip(this.equ)
221 }
222
223 /**
224 *
225 * @param {*} corner
226 * @todo rationaliser avec suffixCorner
227 * @returns boolean true if corner is added
228 */
229 suffixTouchingCorners (corner) {
230     const exist = [...this.touchingCorners].find(corn =>
231         isCornerEqual(corn, corner)
232     )
233     if (!exist) {
234         this.touchingCorners = [...this.touchingCorners, corner]
235         return true
236     } else {
237         return false
238     }
239 }
240
241 /**
242 * @returns boolean true if it is a real face ie number of corners > dimension
243 */
244 isRealFace () {
```

```
245     return this.touchingCorners.length >= this.dim
246 }
247
248 /**
249  *This method returns true if point is inside the Halfspace or on the
boundary. This method treats point as a point (not a vector).
250  * @param {*} point the point to check
251  * @return boolean true if point is inside or on halfspace
252  * @todo rename containsPoint
253  */
254 // inclue la frontière
255 isPointInsideOrOnFace (point) {
256     //
257     // The point is on the inside side or the boundary of the halfspace if
258     //
259     //  $a_1 x_1 + a_2 x_2 + \dots + a_n x_n + k \leq 0$ 
260     //
261     // where all  $a_i$  are the same as in the equation of the hyperplane.
262     // The following code evaluates the left side of this inequality.
263     //
264     //
265     return positionPoint(this.equ, point) > -P.VERY_SMALL_NUM
266 }
267
268 /**
269  * This method returns true point is inside the halfspace. Points which
270  * lie on or very close to the bounding hyperplane are considered to be
271  * outside the halfspace. This method treats point as a point (not a
272  * vector).
273  * @param {*} point the point to check
274  * @returns boolean true if point is inside halfspace
275  */
276
277 isPointInsideFace (point) {
278     //
279     // The point is on the inside side of the halfspace if
280     //
281     //  $a_1 x_1 + a_2 x_2 + \dots + a_n x_n + k < 0$ 
282     //
283     // where all  $a_i$  are the same as in the equation of the hyperplane.
284     // The following code evaluates the left side of this inequality.
285     //
286     //
287     //
288     return positionPoint(this.equ, point) > P.VERY_SMALL_NUM
289 }
290
291 isPointOnFace (point) {
292     const pos = positionPoint(this.equ, point)
293     return pos > -P.VERY_SMALL_NUM && pos < P.VERY_SMALL_NUM
294 }
295
296
```

```
297 /**
298  *
299  * @param {*} axe
300  * @returns number factor
301  */
302 pvFactor (axe) {
303   return this.equ[axe]
304 }
305
306 /**
307  *
308  * @param {*} adjaFace
309  * @param {*} axe
310  * @returns face face
311  */
312 intersectionsIntoFace (adjaFace, axe) {
313   const aF = adjaFace.pvFactor(axe)
314   const tF = this.pvFactor(axe)
315   const aEq = adjaFace.equ.map(x => x * tF)
316   const tEq = this.equ.map(x => x * aF)
317   const diffEq = subtract(tEq, aEq)
318
319   const aTC = [...adjaFace.touchingCorners]
320   const tTC = [...this.touchingCorners]
321   const outPoint = tTC.find(point => !aTC.find(pt => pt === point))
322   if (!outPoint) return false
323
324   const outPointProj = projectVector(outPoint, axe)
325   let diffEqProj = projectVector(diffEq, axe)
326   if (positionPoint(diffEqProj, outPointProj) > P.VERY_SMALL_NUM) {
327     diffEqProj = flip(diffEqProj)
328   }
329
330   const nFace = new Face(diffEqProj)
331   nFace.name = `proj de ${this.equ} selon ${axe}`
332   return nFace
333   // TODO return false si pas bon
334 }
335
336 /**
337  *
338  * @param {*} adjaFace
339  * @param {*} axe
340  * @returns face face
341  * TODO: ne plus utiliser des faces, seulement des HP
342  */
343 intersectionIntoSilhouetteFace (adjaFace, axe) {
344   const aF = adjaFace.pvFactor(axe)
345   const tF = this.pvFactor(axe)
346   const aEq = [...adjaFace.equ].map(x => x * tF)
347   const tEq = [...this.equ].map(x => x * aF)
348
349   const diffEq = subtract(tEq, aEq)
```

```

350     // const diffEq = normalize(subtract(tEq, aEq))
351     const aTC = [...adjaFace.touchingCorners]
352     // const tTC = [...this.touchingCorners]
353     const _t = this
354     // looking for a point in solid, but not on main face
355     // for exemple, a touching corner of the adjacent face
356     // not common with main face
357     // const outPoint = aTC[0];
358     // const outPoint = aTC.find(point => !tTC.find(pt => isCornerEqual(pt,
point)))
359     const outPoint = aTC.find(point => !_t.isPointOnFace(point))
360     if (!outPoint) return false
361
362     const nFace = new Face(diffEq)
363     // flip the face if point is not inside
364     if (!nFace.isPointInsideFace(outPoint)) {
365         nFace.flip()
366     }
367
368     nFace.name = `proj de ${this.equ} selon ${axe} `
369     return nFace
370 }
371
372 /**
373  *
374  * @param {*} adjaFace
375  * @param {*} axe
376  */
377 intersectionCutIntoSilhouetteFace (adjaFace, axe) {
378     const aF = adjaFace.pvFactor(axe)
379     const tF = this.pvFactor(axe)
380     const aEq = [...adjaFace.equ].map(x => x * tF)
381     const tEq = [...this.equ].map(x => x * aF)
382     // TODO: normalize dans Face
383     const diffEq = subtract(aEq, tEq)
384     const nFace = new Face(diffEq)
385     nFace.name = `cut de ${this.equ} pour ${axe} = 0 `
386     return nFace
387 }
388
389 /**
390  *
391  * @param {*} axe
392  * @returns array array of faces
393  * TODO: utiliser liste de référence dansr adj faces
394  * TODO: mettre plutot dans sihouette
395  * TODO: pourquoi passer par des faces ?
396  */
397 silhouette (axe, faces) {
398     if (this.isBackFace(axe)) {
399         return false
400     }
401     // TODO: vérifier si newface utile et remplacer par _this

```



```
402     const newFace = new Face(this.equ)
403     newFace.touchingCorners = [...this.touchingCorners]
404     const _t = this
405     const silFaces = []
406     this.adjacentRefs.forEach(element => {
407         // Just keep backface to get visible edge ;
408         if (faces[element].isBackFace(axe)) {
409             // TODO: ne plus utiliser des faces, seulement des HP
410             const nface = newFace.intersectionIntoSilhouetteFace(faces[element], axe)
411             if (nface) {
412                 silFaces.push(nface)
413             }
414         }
415     })
416     return silFaces
417 }
418
419 cutSilhouette (axe, faces) {
420     const silFaces = []
421     const _t = this
422     this.adjacentRefs.forEach(element => {
423         // contrairement à une silhouette normale, on ne prend que la silhouette de
424         la face
425         const nface = _t.intersectionCutIntoSilhouetteFace(faces[element], axe)
426         if (nface) {
427             silFaces.push(nface)
428         }
429     })
430     return silFaces
431 }
432
433 /**
434  * 3D specific function
435  */
436 orderedCorners () {
437     const corners = [...this.touchingCorners]
438     const ci = corners[0]
439     const vequ = this.equ.slice(0, -1)
440     const vref = subtract(ci, corners[1])
441     return corners
442         .map(corner => [order3D(corner, vequ, ci, vref), corner])
443         .sort(function (a, b) {
444             return a[0] - b[0]
445         })
446         .map(el => el[1])
447 }
448
449 /**
450  * This method returns true if point is inside all the specified halfspaces.
451  * Points which lie on or very close to the bounding hyperplane are
452  * considered to be outside the halfspace. This method treats point as
453  * a point (not a vector).
```

```
454     * @param {*} faces
455     * @param {*} point the point to check
456     * @returns boolean true if point is inside all faces
457
458     */
459
460     static isPointInsideFaces (faces, point) {
461         return faces.every(face => face.isPointInsideFace(point))
462     }
463
464     /**
465     *
466     * @param {*} faces
467     * @returns ?
468     */
469     static facesIntersection (faces) {
470         const hyps = faces.map(face => face.equ)
471         return intersectHyperplanes(hyps)
472     }
473
474     /**
475     *
476     * @param {*} faces
477     * @param {*} facesrefs
478     * @returns ?
479     */
480     static facesRefIntersection (faces, refs) {
481         const hyps = refs.map(ref => faces[ref].equ)
482         return intersectHyperplanes(hyps)
483     }
484
485     /**
486     *
487     * @param {*} faces
488     * @param {*} refs
489     * @param {*} corner
490     * TODO: travailler sur les index plutôt que sur les faces
491     */
492     static updateAdjacentFacesRefs (faces, refs, corner) {
493         // TODO: pas au bon endroit
494         refs.forEach(ref => faces[ref].suffixTouchingCorners(corner))
495         const grouprefs = moizeAmongIndex(refs.length, 2, 2)
496         grouprefs.forEach(groupref => {
497             faces[refs[groupref[0]]].adjacentRefs.add(refs[groupref[1]])
498             faces[refs[groupref[1]]].adjacentRefs.add(refs[groupref[0]])
499         })
500     }
501 }
502
503 /**
504 * 3D specific function
505 * @param {*} point1
506 * @param {*} halfspace
```

```
507 * @param {*} pointref
508 * @param {*} vectorref
509 * @returns angle
510 */
511 function order3D (point1, halfspace, pointref, vectorref) {
512   const v1 = subtract(point1, pointref)
513   const crossP = cross3d(vectorref, v1)
514   const norm = matnorm(crossP)
515   const dotP = matdot(vectorref, v1)
516   const theta = Math.atan2(norm, dotP)
517   const sign = matdot(crossP, halfspace)
518   if (sign < 0) { // TODO: very small ou 0 ?
519     return -theta
520   } else {
521     return theta
522   }
523 }
524 export { Face }
525
```