src/src/face.js

```
/**
 * @file Describes ADSODA face
 * @author Jeff Bigot <jeff@raktres.net> after Greg Ferrar
 * @class  Face
 */

import {
  constantAdd,
  projectVector,
  positionPoint,
  intersectHyperplanes,
  isCornerEqual,
  getCoordinate,
  getConstant,
  moizeAmongIndex,
  normalize,
  multiplyMatrices,
  subtract,
  matnorm,
  matdot,
  flip
} from './halfspace.js'
import { NDObject } from './ndobject.js'
import * as P from './parameters'

/**
 * 3D specific function
 * @param {*} x
 * @param {*} y
 */
function cross3d (x, y) {
  const [x1, x2, x3] = Array.from(x)
  const [y1, y2, y3] = Array.from(y)
  return [
    x2 * y3 - x3 * y2,
    x3 * y1 - x1 * y3,
    x1 * y2 - x2 * y1
  ]
}

class Face extends NDObject {
  /**
   * create a face * This is the interface to the Halfspace class.  An Halfspace is half of an n-space.  it is described by the equation of the bounding hyperplane.  A point is considered to be inside the halfspace if the left side of the equation is greater than the right side when the coordinates of the point (vector) are plugged in.  The first n coefficients can also be viewed as the normal vector, and the last as a constant which determines which of the possible parallel hyperplanes in n-space this one is.<br>  A halfspace is represented by the equation of the bounding hyperplane, so a Hyperplane  is really the same as a Halfspace.
```

```javascript
   * @constructor Face
   * @param {*} vector
   */
  constructor (vector) {
    super('Face')
    this.equ = normalize(vector.map(parseFloat))
    this.touchingCorners = []
    this.adjacentRefs = new Set()
    this.dim = this.equ.length - 1
  }

  /**
   * @returns JSON face description
   */
  exportToJSON () {
    return `{ "face" : ${JSON.stringify(this.equ)} }`
  }

  /**
   *
   * @param {*} json
   */
  static importFromJSON (json) {
    return new Face(json.face)
  }

  /**
   * @returns text face description
   */
  logDetail () {
    return `Face name : ${this.name} \n --- halfspace : ${
      this.equ
    } \n --- touching corners ${JSON.stringify(
      this.touchingCorners
    )} \n --- nb of adjacent faces : ${this.adjacentRefs.length} `
  }

  /**
   *
   */
  eraseTouchingCorners () {
    this.touchingCorners.length = 0
  }

  /**
   * @returns face this
   */
  eraseAdjacentFaces () {
    this.adjacentRefs.clear()
  }

  /**
   * translate the face following the given vector.<br>
   * Translation doesn't change normal vector, Just the constant term need to be changed.
   * new constant = old constant - dot(normal, vector)<br>
   * @param {*} vector the vector indicating the direction and distance to translate this halfspace.
   * @todo vérifrie que mutation nécessaire
   * @returns face this
   */
  translate (vector) {
    // Given a halfspace
    //
    //    a1*x1 + ... + an*xn + k = 0
    //
    //  We can translate by vector (v1, v2, ..., vn) by substituting (xi - vi) for
    //  all xi, yielding
    //
    //    a1*(x1-v1) + ... + an*(xn-vn) + k = 0
    //
    //  This simplifies to
    //
    //    a1*x1 + ... + an*xn + (k - a1*v1 - ... - an*vn) = 0
    //
    //  So all we have to do is change the constant term.  This is as expected,
    //  since translating should not change the normal vector (the first n-1 terms).
    //

    const dot = matdot(this.equ.slice(0, -1), vector)
    constantAdd(this.equ, dot)
    return this
  }

  /**
   *  This method applies a matrix transformation to this Halfspace.
   *  @param {matrix} matrix the matrix of the transformation to apply.
   * @todo vérifrie que mutation nécessaire
   * @returns face this
   */
  transform (matrix) {
    //
    //  The normal of the tranformed halfspace can be found with a simple matrix
    //  multiplication.
    const coords = multiplyMatrices(matrix, this.equ.slice(0, -1))

    // console.error('trans', this.equ, coords )
    // The constant is more difficult.  Here I have solved this
    //  by finding a point on the original halfspace (by checking axes for intersections)
    //  and transforming that point as well.  The transformed point lies on the
    //  transformed halfspace, so the constant term can be computed by plugging the
    //  transformed point into the equation of the transformed halfspace (the coefficients
    //  being the coordinates of the transformed normal and the constant unknown) and
    //  solving for the constant.
    //

    // get non 0 coordinate
    const coordindex = this.equ.findIndex(x => Math.abs(x) > P.VERY_SMALL_NUM)
    /* let max = 0
    let coordindex = false
    for (let index = 0; index < this.equ.length - 1; index++) {
      if (this.equ[index] > max) {
        max = this.equ[index]
        coordindex = index
      }
    }
    */
    // TODO vérifier si utilisaton not small_value x!=0
    // const intercept = -getConstant(this.equ) / getCoordinate(this.equ, coordindex)
    const intercept = -getConstant(this.equ) / getCoordinate(this.equ, coordindex) // max
    //  At this point we have found a point on the halfspace.  This point is
    //  (0, 0, ..., intercept, ..., 0, 0), where intercept is the ith coordinate
    //  and all other coordinates are 0.  Since this is a highly sparse and
    //  predictable vector.  We will NOT actually plug all these coordinates
    //  into a Vector and use matrix multiplication; rather, we will take
    //  advantage of the fact that multiplication by such a vector yields
    //  a vector which is simply the ith column of m multiplied by intercept.
    //  We skip another step by plugging the coordinates of this product
    //  directly into the transformed equation.
    //

    let sum = 0
    const n = coords.length
    for (let i = 0; i < n; i++) {
      sum += matrix[i][coordindex] * intercept * coords[i]
    }
    this.equ = [...coords, -sum]
    return this
  }

  /**
   *
   * @param {*} axe
   * @returns boolean if it is a backface
   */
  isBackFace (axe) {
    return this.orientation(axe) < 0
  }

  /**
```

```
   *
   * @param {*} axe
   * @returns number sign of coef
   */
  orientation (axe) {
    const val = this.equ[axe]
    if (val < -P.VERY_SMALL_NUM) return -1
    if (val > P.VERY_SMALL_NUM) return 1
    return 0
  }

  /**
   *
   * @param {*} point
   * @param {*} axe
   * @returns boolean if point is valid to be used for order
   * TODO: comprendre pourquoi ce n'est pas utilisé
   */
  validForOrder (point, axe) {
    return !this.pointInsideFace(point) && this.orientation(axe) !== 0
  }

  /**
   * This method negates all terms in the equation of this halfspace.  This
   * flips the normal without changing the boundary halfplane.
   * @todo évaluer l'impact de l'utilisation  de ...
   */
  flip () {
    this.equ = flip(this.equ)
  }

  /**
   *
   * @param {*} corner
   * @todo rationaliser avec suffixCorner
   * @returns boolean true if corner is added
   */
  suffixTouchingCorners (corner) {
    const exist = [...this.touchingCorners].find(corn =>
      isCornerEqual(corn, corner)
    )
    if (!exist) {
      this.touchingCorners = [...this.touchingCorners, corner]
      return true
    } else {
      return false
    }
  }

  /**
   * @returns boolean true if it is a real face ie number of corners > dimension
   */
  isRealFace () {
    return this.touchingCorners.length >= this.dim
  }

  /**
   *This method returns true if point  is inside the Halfspace or on the boundary.  This method treats point as a point (not a vector).
   * @param {*} point the point to check
   * @return boolean true if point is inside or on halfspace
   * @todo rename containsPoint
   */
  // inclue la frontière
  isPointInsideOrOnFace (point) {
    //
    //  The point is on the inside side or the boundary of the halfspace if
    //
    //  a x  + a x  + ... + a x + k  <=  0
    //   1 1    2 2          n n
    //
    //  where all ai are the same as in the equation of the hyperplane.
    //  The following code evaluates the left side of this inequality.
    //

    return positionPoint(this.equ, point) > -P.VERY_SMALL_NUM
  }

  /**
     * This method returns true point is inside the halfspace.  Points which
          lie on or very close to the bounding hyperplane are considered to be
          outside the halfspace.  This method treats point as a point (not a
          vector).
     * @param {*} point the point to check
    * @returns boolean true if point is inside halfspace

     */
  isPointInsideFace (point) {
    //
    //  The point is on the inside side of the halfspace if
    //
    //  a x  + a x  + ... + a x + k  <  0
    //   1 1    2 2          n n
    //
    //  where all ai are the same as in the equation of the hyperplane.
    //  The following code evaluates the left side of this inequality.
    //

    return positionPoint(this.equ, point) > P.VERY_SMALL_NUM
  }

  isPointOnFace (point) {
    const pos = positionPoint(this.equ, point)
    return pos > -P.VERY_SMALL_NUM && pos < P.VERY_SMALL_NUM
  }

  /**
   *
   * @param {*} axe
   * @returns number factor
   */
  pvFactor (axe) {
    return this.equ[axe]
  }

  /**
   *
   * @param {*} adjaFace
   * @param {*} axe
   * @returns face face
   */
  intersectionsIntoFace (adjaFace, axe) {
    const aF = adjaFace.pvFactor(axe)
    const tF = this.pvFactor(axe)
    const aEq = adjaFace.equ.map(x => x * tF)
    const tEq = this.equ.map(x => x * aF)
    const diffEq = subtract(tEq, aEq)

    const aTC = [...adjaFace.touchingCorners]
    const tTC = [...this.touchingCorners]
    const outPoint = tTC.find(point => !aTC.find(pt => pt === point))
    if (!outPoint) return false

    const outPointProj = projectVector(outPoint, axe)
    let diffEqProj = projectVector(diffEq, axe)
    if (positionPoint(diffEqProj, outPointProj) > P.VERY_SMALL_NUM) {
      diffEqProj = flip(diffEqProj)
    }

    const nFace = new Face(diffEqProj)
    nFace.name = `proj de ${this.equ} selon ${axe} `
    return nFace
    // TODO return false si pas bon
  }

  /**
   *
   * @param {*} adjaFace
   * @param {*} axe
   * @returns face face
   * TODO: ne plus utiliser des faces, seulement des HP
   */
  intersectionIntoSilhouetteFace (adjaFace, axe) {
    const aF = adjaFace.pvFactor(axe)
    const tF = this.pvFactor(axe)
    const aEq = [...adjaFace.equ].map(x => x * tF)
    const tEq = [...this.equ].map(x => x * aF)

    const diffEq = subtract(tEq, aEq)
    // const diffEq = normalize(subtract(tEq, aEq))
    const aTC = [...adjaFace.touchingCorners]
    // const tTC = [...this.touchingCorners]
    const _t = this
    // looking for a point in solid, but not on main face
    // for exemple, a touching corner of the adjacent face
    // not common with main face
    // const outPoint = aTC[0];
    // const outPoint = aTC.find(point => !tTC.find(pt => isCornerEqual(pt, point)))
    const outPoint = aTC.find(point => !_t.isPointOnFace(point))
    if (!outPoint) return false

    const nFace = new Face(diffEq)
    // flip the face if point is not inside
    if (!nFace.isPointInsideFace(outPoint)) {
      nFace.flip()
    }

    nFace.name = `proj de ${this.equ} selon ${axe} `
    return nFace
  }

  /**
   *
   * @param {*} adjaFace
   * @param {*} axe
   */
  intersectionCutIntoSilhouetteFace (adjaFace, axe) {
    const aF = adjaFace.pvFactor(axe)
    const tF = this.pvFactor(axe)
    const aEq = [...adjaFace.equ].map(x => x * tF)
    const tEq = [...this.equ].map(x => x * aF)
    // TODO: normalize dans Face
    const diffEq = subtract(aEq, tEq)
    const nFace = new Face(diffEq)
    nFace.name = `cut de ${this.equ} pour ${axe} = 0 `
    return nFace
  }

  /**
   *
   * @param {*} axe
   * @returns array array of faces
   * TODO: utliser liste de référence dansr adj faces
   * TODO: mettre plutot dans sihouette
   * TODO: pourquoi passer par des faces ?
   */
  silhouette (axe, faces) {
    if (this.isBackFace(axe)) {
      return false
    }
    // TODO: vérifier si newface utile et remplacer par _this
    const newFace = new Face(this.equ)
    newFace.touchingCorners = [...this.touchingCorners]
    const _t = this
    const silFaces = []
    this.adjacentRefs.forEach(element => {
      // Just keep backface to get visible edge ;
      if (faces[element].isBackFace(axe)) {
        // TODO: ne plus utiliser des faces, seulement des HP
        const nface = newFace.intersectionIntoSilhouetteFace(faces[element], axe)
        if (nface) {
          silFaces.push(nface)
        }
      }
    })
    return silFaces
  }

  cutSilhouette (axe, faces) {
    const silFaces = []
    const _t = this
    this.adjacentRefs.forEach(element => {
      // contrairement à une silhouette normale, on ne prend que la silhouette de la face
      const nface = _t.intersectionCutIntoSilhouetteFace(faces[element], axe)
      if (nface) {
        silFaces.push(nface)
      }
    })
    return silFaces
  }

  /**
   * 3D specific function
   *
   */
  orderedCorners () {
    const corners = [...this.touchingCorners]
    const ci = corners[0]
    const vequ = this.equ.slice(0, -1)
    const vref = subtract(ci, corners[1])
    return corners
      .map(corner => [order3D(corner, vequ, ci, vref), corner])
      .sort(function (a, b) {
        return a[0] - b[0]
      })
      .map(el => el[1])
  }

  /**
     *This method returns true if point is inside all the specified halfspaces.
     *     Points which lie on or very close to the bounding hyperplane are
     *     considered to be outside the halfspace.  This method treats point as
     *     a point (not a vector).
     * @param {*} faces
     * @param {*} point the point to check
     * @returns boolean true if point is inside all faces

     */

  static isPointInsideFaces (faces, point) {
    return faces.every(face => face.isPointInsideFace(point))
  }

  /**
   *
   * @param {*} faces
   * @returns ?
   */
  static facesIntersection (faces) {
    const hyps = faces.map(face => face.equ)
    return intersectHyperplanes(hyps)
  }

  /**
   *
   * @param {*} faces
   * @param {*} facesrefs
   * @returns ?
   */
  static facesRefIntersection (faces, refs) {
    const hyps = refs.map(ref => faces[ref].equ)
    return intersectHyperplanes(hyps)
  }

  /**
   *
   * @param {*} faces
   * @param {*} refs
   * @param {*} corner
   * TODO: travailler sur les index plutôt que sur les faces
   */
  static updateAdjacentFacesRefs (faces, refs, corner) {
    // TODO: pas au bon endroit
    refs.forEach(ref => faces[ref].suffixTouchingCorners(corner))
    const grouprefs = moizeAmongIndex(refs.length, 2, 2)
    grouprefs.forEach(groupref => {
      faces[refs[groupref[0]]].adjacentRefs.add(refs[groupref[1]])
      faces[refs[groupref[1]]].adjacentRefs.add(refs[groupref[0]])
    })
  }
}

/**
 * 3D specific function
 * @param {*} point1
 * @param {*} halfspace
 * @param {*} pointref
 * @param {*} vectorref
 * @returns angle
 */
function order3D (point1, halfspace, pointref, vectorref) {
  const v1 = subtract(point1, pointref)
  const crossP = cross3d(vectorref, v1)
  const norm = matnorm(crossP)
  const dotP = matdot(vectorref, v1)
  const theta = Math.atan2(norm, dotP)
  const sign = matdot(crossP, halfspace)
  if (sign < 0) { // TODO: very small ou 0 ?
    return -theta
  } else {
    return theta
  }
}
export { Face }
```