src/src/halfspace.js

```
/**
 * @file Describes ADSODA halfspace
 * @author Jeff Bigot <jeff@raktres.net> after Greg Ferrar
 * @module halfspace
 * */

import moize from 'moize'
import * as P from './parameters'

// |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// | Halfspace.cp
// |
// | This is the implementation of 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.  The first n coefficients can also be viewed as the normal vector.
// |_______________________________________________________________________________________

// dot product of 2 vectors
// function matdot (ar1, ar2) { return ar1.reduce((acc, component, index) => acc + component * ar2[index], 0)}
/**
 * multiplication de deux matrices
 * @param {*} m1
 * @param {*} m2
 */
export function multiplyMatrices (m1, m2) {
  const result = []
  for (let i = 0; i < m2.length; i++) {
    let res = 0
    for (let j = 0; j < m1[0].length; j++) {
      res += m2[j] * m1[i][j]
    }
    result.push(res)
  }
  return result
}

/**
 * flix
 * @param {*} equ
 */
export function flip (equ) {
  return equ.map(coord => -coord)
}

/**
 * soustraction de deux vecteurs
 * on suppose que les deux vecteurs ont la même taille
 * @param {*} a
 * @param {*} b
 */
```

```
export function subtract (a, b) {
  const res = []
  for (let j = 0; j < a.length; j++) {
    res.push(a[j] - b[j])
  }
  return res
}

/**
 * calcule la norme d'un vecteur
 * @param {*} a
 */
export function matnorm (a) {
  let res = 0
  for (let j = 0; j < a.length; j++) {
    res += a[j] * a[j]
  }
  return Math.sqrt(res)
}

/**
 * produit scalaire de deux vecteurs
 * @param {*} vector1
 * @param {*} vector2
 */
export function matdot (vector1, vector2) {
  let result = 0
  for (let i = 0; i < vector1.length; i++) {
    result += vector1[i] * vector2[i]
  }
  return result
}

/**
 * normalise un descripteur d'hyperplan
 * @param {*} HS
 */
export function normalize (HS) {
  let sum = 0
  for (let index = 0; index < HS.length - 1; index++) {
    sum += HS[index] * HS[index]
  }
  const length = Math.sqrt(sum)
  return HS.map(x => x / length)
}

/**
 * TODO: détailler
 * @param {*} matrix
 * @returns echelon matrix
 * TODO: plutôt faire le contrôle de petit dans le controle du while
 */
export function echelon (matrix) {
  const nbrows = matrix.length
  const nbcolumns = matrix[0].length
  // TODO: ne devrait pas être nécessaire !!!
  const outmatrix = matrix.map(row => row.map(x => Math.abs(x) < P.VERY_SMALL_NUM ? 0 : x))

  let lead = 0
  for (let k = 0; k < nbrows; k++) {
    if (nbcolumns <= lead) return outmatrix

    let i = k
    // TODO: voir pour introduire < P.VERY_SMALL_NUM
    while (outmatrix[i][lead] === 0) {
      i++
      if (nbrows === i) {
        i = k
        lead++
        if (nbcolumns === lead) return outmatrix
      }
    }
    const irow = outmatrix[i]
    const krow = outmatrix[k]
    // (outmatrix[i] = krow), (outmatrix[k] = irow)
    outmatrix[i] = krow
    outmatrix[k] = irow

    let val = outmatrix[k][lead]
    // commence à lead et non à 0. Ou alors à k ?
    for (let j = k; j < nbcolumns; j++) {
      const out = outmatrix[k][j] / val
      outmatrix[k][j] = Math.abs(out) < P.VERY_SMALL_NUM ? 0 : out
    }

    for (let l = 0; l < nbrows; l++) {
      if (l === k) continue
      val = outmatrix[l][lead]
      // commencer à lead et non à 0. Ou alors à k ?
      for (let j = k; j < nbcolumns; j++) {
        const nval = outmatrix[l][j] - val * outmatrix[k][j]
        outmatrix[l][j] = Math.abs(nval) < P.VERY_SMALL_NUM ? 0 : nval
      }
    }
    lead++
  }
  return outmatrix
}

/**
 *
 * @param {*} matrix
 * @returns matrix
 * TODO: vérifier si on doit controler une valeur petite ou une valeur nulle
 */
/*
export function nonZeroRows (matrix) {
  return matrix.filter(
    row => row.find(val => val !== 0)
  )
}
*/
```

```javascript

/**
 * TODO: décrire
 * @param {*} matrix
 * @returns vector solution of the system
 * TODO: remplacer map
 */
export function solution (matrix) {
  const mat1 = echelon([...matrix])
  if (!mat1) return false
  if (mat1.length < mat1[0].length - 1) return false
  for (let index = 0; index < mat1[0].length - 1; index++) {
    if (mat1[index][index] === 0) return false
  }
  // TODO: vérifier si c'est vraiment nécessaire
  // const mat2 = nonZeroRows(mat1)
  const last = mat1.map(el => -el.slice(-1)[0])
  return last
}

/**
 * Get constant value of the halfspace
 */
export function getConstant (halfspace) {
  return halfspace[halfspace.length - 1]
}

/**
 *
 * @param {*} halfspace
 * @param {*} i
 */
export function getCoordinate (halfspace, i) {
  return halfspace[i]
}

/**
 * add a constant to the halfspace constant
 * @param {*} u
 * @param {*} x
 */
export function constantAdd (u, x) {
  u[u.length - 1] -= x
}

/**
 *
 * @param {*} halfspace
 * @param {*} point
 */
export function positionPoint (halfspace, point) {
  return matdot(halfspace, [...point, 1])
}

/**
 *
 * @param {*} vector
 * @param {*} axe
 * @todo replace axe with a vector
 */
export function projectVector (vector, axe) {
  return [...vector.slice(0, axe), ...vector.slice(axe + 1, vector.length)]
}

/**
 *
 * @param {*} hyperplanes
 * @returns the intersection of hyperplanes, false if no solution found
 * @todo verify that the left part of the matrix is an identity matrix
 */
export function intersectHyperplanes (hyperplanes) {
  const dimension = hyperplanes[0].length - 1
  const result = solution(hyperplanes)
  return result.length === dimension ? result : false
}

/**
 * return every compositions of index elements from l size wich size is
 * between a and b
 * @param {*} l nb of elements
 * @param {*} a
 * @param {*} b
 * @returns array of compositions
 */
export function amongIndex (l, a, b) {
  const extract = [[]]
  const ref = new Array(l)
  for (let i = 0; i < l; i++) {
    ref[i] = i
  }
  for (let i = 0; i < l; i++) {
    const le = extract.length
    for (let j = 0; j < le; j++) {
      extract.push(extract[j].concat(ref[i]))
    }
  }
  const res = extract.filter(sub => sub.length >= a && sub.length <= b)
  res.sort(function (a, b) { return b.length - a.length })
  return res
}

/**
 *
 */
export const moizeAmongIndex = moize(amongIndex)

/**
 *
 * @param {*} corner1
 * @param {*} corner2
 */
export function isCornerEqual (corner1, corner2, diff = P.VERY_SMALL_NUM) {
  if (corner1 instanceof Array && corner2 instanceof Array) {
    for (let i = 0; i < corner1.length; i++) {
      if (Math.abs(corner1[i] - corner2[i]) > diff) {
        return false
      }
    }
    return true
  } else {
    return Math.abs(corner1 - corner2) < diff
  }
}
/**
 * compare two HP
 * @param {*} hp1
 * @param {*} hp2
 * @param {*} diff
 */
export function isHPEqual (hp1, hp2, diff = P.VERY_SMALL_NUM) {
  return isCornerEqual(normalize(hp1), normalize(hp2), diff)
}
/**
 * @param {*} corner1
 * @param {*} corner2
 */
export function vectorFromPoints (corner1, corner2) {
  return subtract(corner1, corner2)
}

export function findnormal (pointsArray) {
  const mat = pointsArray.map(el => el.concat(1))
  const ech = echelon(mat)
  const isnull = ech.find((element, idx) => {
    return element[idx] === 0
  })
  if (isnull) { return false }
  const res = ech.map(el => el.slice(-1)[0])
  // trouve la derniere valeur en utilisant un point
  res.push(-matdot(res, pointsArray[0]))
  return res
}

```