Algorithm 1 Functions for Algorithm 1

function SPATIALJOIN(AIS, Polygons,type=completely within)
  Test every position from (AIS.longitude, AIS.latitude) to every polygon in Polygon.spatialGeometry and join(left) polygon attributes of matched values into the AIS dataframe as a new column polygonName.
  Parameters:
  positions←(AIS.logitude,AIS.latitude). Coordinates converted to spatial Point geometry.
  polygons←Polygons.polygonGeometry. Set of spatial polygon geometry from DB column polygon Geometry.
  return AISSpatialMatch
end function
function ANCHORCLUSTER(AISSpatialMatch)
  Creates clusters by DBSCAN from anchorage polygons positions. A unique number is assigned only to clusters. Distance matrix by Euclidean distance.
  Parameters:
  positions iterator←(AISSpatialMatch.longitude,AISSpatialMatch.latitude)
  nameOfPolygon←AISSpatialMatch.polygonName[0]
  tidalStreamSpeed←1.6kts if nameOfPolygon is North Anchorage OR 2.2kts if nameOfPolygon is South Anchorage.
  minPts←3.Minimum number of positions to generate a cluster
  $\epsilon \leftarrow \frac{10 \times tidalStreamSpeed}{3600}$. Decimal degrees maximum distance. Transformation of tidal stream speed (NM/hr) to 10 minutes maximum expected movement.
  return set of unique cluster ID's and None values for non cluster records
end function
function POLYGONVISITID(AISSpatialMatch.polygonName)
  Identifies breaks in the sequence of names within database. Models a vessel shifting polygons and assign a uniqueID to every polygon visit and gaps between polygons if they exist.
  Parameters:
  name←AISSpatialMatch.polygonName. Set of sorted(time) names from DB column polygonName.
  return set of continuous uniqueID's
end function
function INDEXOFFIRSTINCANAL(Index of anchor cluster last value,anchorName)
  Finds the first value inside the Canal after the entry point
  Parameters:
  entryPoint←31°19'.68N if anchorName is North Anchorage o.w 29°55'.91N
  return index of first position inside the canal
end function
function LINESTRING((AIS.longitude,AIS.latitude),LandPolygons)
  Uses an ordered sequence of longitudes and latitudes to build a spatial line. Lines crossing land [SpatialJoin((AIS.longitude,AIS.latitude),LandPolygons,type=Intersect)] or with turns larger than 90° returns **None**
  Parameters:
  spatialPoints←(AIS.longitude,AIS.latitude). Linestring from ordered set of spatial points generated from AIS coordinates.
  landPolygons←LandPolygon.polygonGeometry. Spatial polygon.
  return spatial line
end function
function ROUTEMERGE(lineString,ManualAccessRoutes)
  Test if lineString intercepts any of the AccessRoutes and merge in case of match. No match returns **None**.
  Parameters:
  lineString. Spatial lineString.
  accessRoutes←ManualAccessRoutes.lineStringGeometry. Spatial lineString.
  return merged lineString
end function
function AEDBSCANLINESMATRIX(AccessRoutes)
  DBSCAN clustering on lineStrings based on a distance matrix calculated with the Symmetrized Segment-Path Distance (SPDD) method. ϵ distance calculated as per AEDBSCAN method.
  Parameters:
  lineString←AccessRoutes.lineStringGeometry. Spatial lineString
  minPts←3. Minimum number of lines to generate a cluster
  return set of unique clusterID
end function

Algorithm 2 Suez Canal modelling

```
Inputs:
    A dataframe with columns {index,vesselID,longitude,latitude,time,draught}; AIS
    A dataframe with columns {polygonName, polygonGeometry}; CanalPolygons
    A dataframe with columns {polygonName, polygonGeometry}; LandPolygons
    A dataframe with columns {linestringName,lineStringGeometry}; ManualAccessRoutes
    Default Parameters:
    boxNorthLimit ←31°36'N, canal north limit
    boxEastLimit ←32°48'E, canal east limit
    boxSouthLimit ←29°26'N, canal south limit
    boxWestLimit ←32°06'E, canal west limit
    timeToAccess←2
init {AccessRoutes, FilteredAccessRoutes}
for AIS.GroupBy(vesselID) do
    if ANY (AIS.latitude≥boxSouthLimit AND AIS.latitude≤boxNorthLimit AND then
      AIS.longitude≥boxWestLimit AND AIS.longitude≤boxEastLimit)
        init {AISSpatialMatch,AISSpatialMatch.subgroup, AISSpatialMatch.clusterID}
        AISSpatialMatch←call SPATIALJOIN(AIS,CanalPolygons)
        AISSpatialMatch.subgroup←call POLYGONVISITID(AISSpatialMatch.polygonName)
        filter AISSpatialMatch.polygonName in AnchorName OR AISSpatialMatch.polygonName in AccessName
        for AISSpatialMatch.GroupBy(subgroup) do
            AISSpatialMatch.clusterID←call ANCHORCLUSTER((AISSpatialMatch.longitude, AISSpatial-
            Match.latitude), AISSpatialMatch.polygonName)
        end for
        for AISSpatialMatch.GroupBy(clusterID) do
            init {indexLeaveAnchor, timeLeaveAnchor, anchorName, indexFirstCanal, timeFirstCanal, lineString}
            indexLeaveAnchor←AISSpatialMatch[-1][index]
            timeLeaveAnchor←AISSpatialMatch.index=indexLeaveAnchor.time
            anchorName←AISSpatialMatch.index=indexLeaveAnchor.anchorName
            indexFirstCanal←call INDEXOFFIRSTINCANAL(indexLeaveAnchor,anchorName)
            timeFirstCanal←AIS[indexFirstCanal][time]
            if timeFirstCanal-timeLeaveAnchor≤timeToAccess then
                filter AIS[indexLeaveAnchor to indexFirstCanal]
                lineString←call LINESTRING((AIS.longitude,AIS.latitude),LandPolygons)
                lineString←call ROUTEMERGE(lineString,ManualAccessRoutes)
                if lineString≠None then
                    AccessRoutes←AccessRoutes∪{(lineString, anchorName, AIS[-1][draught])}
                end if
            end if
        end for
    end if
end for
for AccessRoutes.GroupBy(anchorName) do
    init {AccessRoutes.clusterID}
    AccessRoutes.clusterID←call AEDBSCANLINESMATRIX(AccessRoutes)
    filter AccessRoutes.clusterID≠None
    FilteredAccessRoutes←FilteredAccessRoutes∪{AccessRoutes}
end for
```

Algorithm 3 Functions for Algorithm 2

```
function SpatialJoin(AIS, Polygons,type=completely within)
    Test every position from (AIS.longitude, AIS.latitude) to every polygon in Polygon.spatialGeometry and
    join(left) polygon attributes of matched values into the AIS dataframe as a new column polygonName.
    Parameters:
    positions←(AIS.logitude,AIS.latitude). Coordinates converted to spatial Point geometry.
    polygons←Polygons.polygonGeometry. Set of spatial polygon geometry from DB column polygon Geometry.
    return AISSpatialMatch
end function
function PolygonVisitID(AISSpatialMatch.polygonName)
    Identifies breaks in the sequence of names within database. Models a vessel shifting polygons and assign
    a uniqueID to every polygon visit and gaps between polygons if they exist.
    Parameters:
    name←AISSpatialMatch.polygonName. Set of sorted(time) names from DB column polygonName.
    return set of continuous uniqueID's
end function
function PolygonVisitTimeID(AISSpatialMatch.time,AISSpatialMatch.subgroup)
    Identifies breaks in time within same subgroups. Starts labelling subgroups every time the threshold is
    passed
    Parameters:
    timeThreshold←96. Hours.
    subgroup←AISSpatialMatch.subgroup. Labels grouping subgroups from DB column subgroup.
    time←AISSpatialMatch.time. Timestamp from DB column time.
    return set of continuous uniqueID's
end function
function SubgroupsMerge(AISSpatialMatch.subgroup1,AISSpatialMatch.subgroup2)
    Recognizes from comparing values of both columns whether a change exist in any of the columns and
    creates a new subgroup column with new ID's.
    Parameters:
    spatialSubgroup←AISSpatialMatch.subgroup1. Label for change of polygon. From DB column subgroup1.
    timeSubgroups←AISSpatialMatch.subgroup2. Label for time break. From DB column subgroup2.
    return set of continuous uniqueID's
end function
function SubgroupsCutOff(TransitIndices.polygonName)
    Could be thought as the opposite of PolygonVisitID. Identifies breaks in the sequence and groups rows
    between the breaks.
    Parameters:
    name←TransitIndices.polygonName. Set of sorted(time) names from DB column polygonName.
    return set of continuous uniqueID's
end function
```

Algorithm 4 Transit raw database mapping

```
Inputs:
    A dataframe with columns {index,vesselID,longitude,latitude,time,draught,speed}; AIS
    A dataframe with columns {polygonName, polygonGeometry}; CanalPolygons
    Default Parameters:
    portVisitTest←96. Hours theshold for port visit and returning to anchorage
    anchorShiftTest←5. Hours threshold for anchorage shifting validation
init {MappingIndices}
for AIS.GroupBy(vesselID) do
    init {AISSpatialMatch}
    AISSpatialMatch←call SpatialJoin(AIS,CanalPolygons)
    if ANY (AISSpatialMatch.polygonName) then
        init {AISSpatialMatch.subgroup1,AISSpatialMatch.subgroup2,AISSpatialMatch.visit,TransitIndices,TransitIndices.same}
        AISSpatialMatch.subgroup1←call PolygonVisitID(AISSpatialMatch.polygonName)
        filter AISSpatialMatch.polygonName≠None
        AISSpatialMatch.subgroup2←call PolygonVisitTimeID(AISSpatialMatch.time,AISSpatialMatch.subgroup1)
        AISSpatialMatch.visit←call SubgroupsMerge(AISSpatialMatch.subgroup1,AISSpatialMatch.subgroup2)
        drop{AISSpatialMatch.subgroup1, AISSpatialMatch.subgroup2}
        for AISSpatialMatch.GroupBy(visit) do
            TransitIndices←TransitIndices∪AISSpatialMatch[0]
        end for
        TransitIndices.same←call PolygonVisitID(TransitIndices.polygonName)
        for TransitIndices.Groupby(same) do
            if TransitIndices.same length>1 then
                for i= 0 to TransitIndices.same length-2 do
                    init {indexFirst,indexSecond,indexLastOfFirst}
                    indexFirst←TransitIndices[i][index]
                    indexSecond←TransitIndices[i+1][index]
                    filter AISSpatialMatch.index[indexFirst to indexSecond]
                    if ANY (AISSpatialMatch.polygonName in AnchorName) then
                        drop TransitIndices.index=indexFirst
                    else
                        indexLastOfFirst← AISSpatialMatch.Groupby(valid).valid[0].index[-1]
                        if (AISSpatialMatch.index=indexLastOfFirst.time-
                          AISSpatialMatch.index=indexSecond.time)<anchorShiftTest
then
                            drop TransitIndices.index=indexSecond
                        end if
                    end if
                end for
            end if
        end for
        drop TransitIndices.same
        init{TransitIndices.equal}
        TransitIndices.equal←call SubgroupsCutOff(TransitIndices.polygonName)
        for TransitIndices.Groupby(equal) do
            if TransitIndices.equal length > 2 then
                TransitIndices.equal←call PolygonVisitTimeID(TransitIndices.time,TransitIndices.equal)
            end if
        end for
        filter TransitIndices.equal length = 2
        for TransitIndices.Groupby(equal) do
            init {indexFirst, indexSecond, validIDSecond, indexEnd}
            indexFirst←TransitIndices[0][index]
            indexSecond←TransitIndices[-1][index]
            validIDSecond←AISSpatialMatch.index=indexSecond.visit[0]
            indexEnd←AISSpatialMatch.visit=validIDSecond.index[-1]
            MappingIndices←MappingIndices∪{AISSpatialMatch[0][vesselID],indexFirst,indexEnd}
        end for
    end if
end for
```

Algorithm 5 Functions for Algorithm 3

function SPATIALJOIN(AIS, Polygons,type=completely within)
  Test every position from (AIS.longitude, AIS.latitude) to every polygon in Polygon.spatialGeometry and join(left) polygon attributes of matched values into the AIS dataframe as a new column polygonName.
  Parameters:
  positions←(AIS.logitude,AIS.latitude). Coordinates converted to spatial Point geometry.
  polygons←Polygons.polygonGeometry. Set of spatial polygon geometry from DB column polygon Geometry.
  return AISSpatialMatch
end function
function REFINEDANCHORCLUSTER(AISTransit,anchorName)
  1.Creates clusters by DBSCAN from anchorage polygons positions. A unique number is assigned only to clusters. Distance matrix by Euclidean distance.
  2.Remove outliers by calculating a threshold from μ+1σ on AISTransit.speed on valid clusters.
  3.Merge clusters within 5 hours from each other AND with no port visit in between. o.w keep the last cluster.
  4.Validates the cluster by verifying if a position exists 30 minutes before and after the cluster.
  Parameters:
  positions iterator←(AISTransit.longitude,AISTransit.latitude)
  speed iterator←AISTransit.speed
  tidalStreamSpeed←1.6kts if anchorName is North Anchorage o.w 2.2kts.
  minPts←3.Minimum number of positions to generate a cluster
  $\epsilon \leftarrow \frac{10 \times tidalStreamSpeed}{3600}$. Decimal degrees maximum distance. Transformation of tidal stream speed (NM/hr) to 10 minutes maximum expected movement.
  return set of unique cluster ID and None values for non cluster records
end function
function ENTRYINTERPOLATION(AISTransit, AccessRoutesEntry,entryPoint)
  1. Create a lineString from last in AISTransit.clusterID until first position after entryPoint.
  2. Get closest line(SPDD distance) between lineString and AccessRouteEntry and transfer AISTransit.time from lineString to closest point in closest line.
  3. Trim line from time of last in anchor cluster until first timestamp after entryPoint.
  4. Calculate distances from first in line (last in anchor cluster) to every timestamp and entryPoint.
  5. Interpolate with distances and timestamps at entryPoint
  Parameters:
  position←(AISTransit.longitude,AISTransit.latitude). Coordinates converted to spatial Point geometry.
  lineStrings←AccessRoutesEntry.lineStringsGeometry. Set of spatial lineStrings.
  entryPoint. Spatial point.
  return time at entry point
end function
function EXITINTERPOLATION(AISTransit, AccessRoutesExit, exitPoint)
  1. Reverse lines at AccesRoutesExit.
  2. Create a lineString from position before of exitPoint until position after exitPoint.
  3. Get closest line (SPDD distance) between lineString and AccessRoutesExit and transfer AISTransit.time from lineString to closest point in closest line.
  4. Trim line from time of position before of exitPoint until time of first position after exitPoint.
  5. Calculate distances from first in line (before exitPoint) to exitPoint and first position after exitPoint.
  6. Interpolate with distances and timestamps at exitPoint.
  Parameters:
  position←(AISTransit.longitude,AISTransit.latitude). Coordinates converted to spatial Point geometry.
  lineStrings←AccessRoutesExit.lineStringsGeometry. Set of spatial lineStrings.
  exitPoint. Spatial point.
  return time at exit point
end function

Algorithm 6 Data generation

```
Inputs:
    A dataframe with columns {index,vesselID,longitude,latitude,time,draught,speed}; AIS
    A dataframe with columns {polygonName, polygonGeometry}; CanalPolygons
    A dataframe with columns {anchorName,lineStringDraught,lineStringGeometry}; FilteredAccessRoutes
    A dataframe with columns {vesselID,indexFirst,indexEnd}; MappingIndices
    Default Parameters:
    northAccessEntryPoint←31°19'.68N
    southAccessEntryPoint←29°55'.91N
    southAccessRoute←{South Access}
    northAccessRoute←{North Access East, North Access West, Said Container Access}
    southAnchorNames←{IC-5C,S.Green Isl.,1H-2H,E1-E12,E13-E21,W1-W14, BV}
    northAnchorNames←{North Anchorage}
init ExportDB
for MappingIndices do
    init {AISTransit, AISTransit.clusterID, firstAnchorageName, lastAnchorageName, entryPoint, exitPoint,
    vesselDraughtEntry, vesselDraughtExit, AccessRoutesEntry, AccessRoutesExit, timeAtEntry,timeAtExit,
    anchoringTime}
    filter AIS[indexFirst to indexEnd]
    AISTransit←call SpatialJoin(AIS, CanalPolygons)
    if ANY(AISTransit.polygonName in northAnchorNames) AND ANY(AISTransit.polygonName in
      southAnchorNames) AND ANY(AISTransit.polygonName in northAccessRoute) AND ANY
      (AISTransit.polygonName in southAccessRoute) then
        firstAnchorageName←AISTransit[0][polygonName]
        lastAnchorageName←AISTransit[-1][polygonName]
        if firstAnchorageName=North Anchorage then
            entryPoint←northAccessEntryPoint
            exitPoint←southAccessEntryPoint
        else
            entryPoint←southAccessEntryPoint
            exitPoint←northAccessEntryPoint
        end if
        AISTransit.clusterID←call RefinedAnchorCluster(AISTransit,firstAnchorageName)
        vesselDraughtEntry←AISTransit[0][draught]
        vesselDraughtExit←AISTransit[-1][draught]
        AccessRoutesEntry←filter FilteredAccessRoutes.anchorName=firstAnchorageName AND
        FilteredAccessRoutes.draught≥vesselDraughtEntry
        timeAtEntry←call EntryInterpolation(AISTransit, AccessRoutesEntry, entryPoint)
        AccessRoutesExit←filterFilteredAccessRoutes.anchorName=lastAnchorageName AND
        FilteredAccessRoutes.draught≥vesselDraughtExit
        timeAtExit←call ExitInterpolation(AISTransit, AccessRoutesExit, exitPoint)
        filter AISTransit.clusterID≠None
        anchoringTime←AISTransit[-1][time]-AISTransit[0][time]
        ExportDB←ExportDB∪{AISTransit[0][vesselID], anchoringTime, timeAtEntry, timeAtExit, firstAnchorageName}
    end if
end for
```