Determining rigid body transformation using the SVD algorithm

Marcos Duarte
Laboratory of Biomechanics and Motor Control (http://demotu.org/ (http://demotu.org/))
Federal University of ABC, Brazil

Ideally, three non-collinear markers placed on a moving rigid body is everything we need to describe its movement (translation and rotation) in relation to a fixed coordinate system. However, in practical situations of human motion analysis, markers are placed on the soft tissue of a deformable body and this generates artifacts caused by muscle contraction, skin deformation, marker wobbling, etc. In this situation, the use of only three markers can produce unreliable results. It has been shown that four or more markers on the segment followed by a mathematical procedure to calculate the 'best' rigid-body transformation taking into account all these markers produces more robust results (Söderkvist & Wedin 1993; Challis 1995; Cappozzo et al. 1997).

One mathematical procedure to calculate the transformation with three or more marker positions envolves the use of the singular value decomposition (http://en.wikipedia.org/wiki/Singular_value_decomposition) (SVD) algorithm from linear algebra. The SVD algorithm decomposes a matrix $\mathbf{M}$ (which represents a general transformation between two coordinate systems) into three simple transformations: a rotation $\mathbf{V^T}$, a scaling factor $\mathbf{S}$ along the rotated axes and a second rotation $\mathbf{U}$:

$$\mathbf{M = U\ S\ V^T}$$

And the rotation matrix is given by:

$$\mathbf{R = U\ V^T}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are both orthonormal (det = $\pm$1).

For example, if we have registered the position of four markers placed on a moving segment in 100 different instants and the position of these same markers during, what is known in Biomechanics, a static calibration trial, we would use the SVD algorithm to calculate the 100 rotation matrices (between the static trials and the 100 instants) in order to find the Cardan angles for each instant.

The function `svdt.py` (its code is shown at the end of this text) determines the rotation matrix (R) and the translation vector (L) for a rigid body after the following transformation: $B = R * A + L + err$. Where A and B represent the rigid body in different instants and err is an aleatory noise. A and B are matrices with the marker coordinates at different instants (at least three non-collinear markers are necessary to determine the 3D transformation).
The matrix A can be thought to represent a local coordinate system (but A it's not a basis) and matrix B the global coordinate system. The operation $P_g = R * P_l + L$ calculates the coordinates of the point P_l (expressed in the local coordinate system) in the global coordinate system (P_g).

Let's test the `svdt` function:

In [2]:

```
# Import the necessary libraries
import numpy as np
import sys
sys.path.insert(1, r'./../functions')
```

In [3]:

```
from svdt import svdt

# markers in different columns (default):
A = np.array([0,0,0, 1,0,0,  0,1,0,  1,1,0])  # four markers
B = np.array([0,0,0, 0,1,0, -1,0,0, -1,1,0])  # four markers

R, L, RMSE = svdt(A, B)

print('Rotation matrix:\n', np.around(R, 4))
print('Translation vector:\n', np.around(L, 4))
print('RMSE:\n', np.around(RMSE, 4))

# markers in different rows:
A = np.array([[0,0,0], [1,0,0], [ 0,1,0], [ 1,1,0]])  # four markers
B = np.array([[0,0,0], [0,1,0], [-1,0,0], [-1,1,0]])  # four markers

R, L, RMSE = svdt(A, B, order='row')

print('Rotation matrix:\n', np.around(R, 4))
print('Translation vector:\n', np.around(L, 4))
print('RMSE:\n', np.around(RMSE, 4))
```

```
Rotation matrix:
 [[ 0. -1.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]
Translation vector:
 [0. 0. 0.]
RMSE:
 0.0
Rotation matrix:
 [[ 0. -1.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]
Translation vector:
 [0. 0. 0.]
RMSE:
 0.0
```

For the matrix of a pure rotation around the z axis, the element in the first row and second column is $-sin\gamma$, which means the rotation was 90^o, as expected.

A typical use of the `svdt` function is to calculate the transformation between A and B ($B = R * A + L$), where A is the matrix with the markers data in one instant (the calibration or static trial) and B is the matrix with the markers data of more than one instant (the dynamic trial).
Input A as a 1D array `[x1, y1, z1, ..., xn, yn, zn]` where `n` is the number of markers and B a 2D array with the different instants as rows (like in A).

The output R has the shape `(3, 3, tn)`, where `tn` is the number of instants, L the shape `(tn, 3)`, and $RMSE$ the shape `(tn)`. If `tn` is equal to one, the outputs have the same shape as in `svdt` (the last dimension of the outputs above is dropped).

Let's show this case:

In [4]:

```
A = np.array([1,0,0, 0,1,0,  0,0,1])
B = np.array([0,1,0, -1,0,0, 0,0,1])
B = np.vstack((B, B))  # simulate two instants (two rows)

R, L, RMSE = svdt(A, B)

print('Rotation matrix:\n', np.around(R, 4))
print('Translation vector:\n', np.around(L, 4))
print('RMSE:\n', np.around(RMSE, 4))
```

```
Rotation matrix:
 [[[-0. -1. -0.]
  [ 1. -0. -0.]
  [-0. -0.  1.]]

 [[-0. -1. -0.]
  [ 1. -0. -0.]
  [-0. -0.  1.]]]
Translation vector:
 [[0. 0. 0.]
 [0. 0. 0.]]
RMSE:
 [0. 0.]
```

References

- Cappozzo A, Cappello A, Della Croce U, Pensalfini F (1997) Surface-marker cluster design criteria for 3-D bone movement reconstruction (http://www.ncbi.nlm.nih.gov/pubmed/9401217). IEEE Trans Biomed Eng., 44:1165-1174.
- Challis JH (1995). A procedure for determining rigid body transformation parameters (http://www.ncbi.nlm.nih.gov/pubmed/7601872). Journal of Biomechanics, 28, 733-737.
- Söderkvist I, Wedin PA (1993) Determining the movements of the skeleton using well-configured markers (http://www.ncbi.nlm.nih.gov/pubmed/8308052). Journal of Biomechanics, 26, 1473-1477.

Function `svdt.py`

In [ ]:

```
# %load ./../functions/svdt.py
#!/usr/bin/env python

"""Calculates the transformation between two coordinate systems using SVD."""

__author__ = "Marcos Duarte, https://github.com/demotu/BMC"
__version__ = "1.0.1"
__license__ = "MIT"

import numpy as np


def svdt(A, B, order='col'):
    """Calculates the transformation between two coordinate systems using SVD.

    This function determines the rotation matrix (R) and the translation vector
    (L) for a rigid body after the following transformation [1]_, [2]_:
    B = R*A + L + err.
    Where A and B represents the rigid body in different instants and err is an
    aleatory noise (which should be zero for a perfect rigid body). A and B are
    matrices with the marker coordinates at different instants (at least three
    non-collinear markers are necessary to determine the 3D transformation).

    The matrix A can be thought to represent a local coordinate system (but A
    it's not a basis) and matrix B the global coordinate system. The operation
    Pg = R*Pl + L calculates the coordinates of the point Pl (expressed in the
    local coordinate system) in the global coordinate system (Pg).

    A typical use of the svdt function is to calculate the transformation
    between A and B (B = R*A + L), where A is the matrix with the markers data
    in one instant (the calibration or static trial) and B is the matrix with
    the markers data for one or more instants (the dynamic trial).

    If the parameter order='row', the A and B parameters should have the shape
    (n, 3), i.e., n rows and 3 columns, where n is the number of markers.
    If order='col', A can be a 1D array with the shape (n*3, like
    [x1, y1, z1, ..., xn, yn, zn] and B a 1D array with the same structure of A
    or a 2D array with the shape (ni, n*3) where ni is the number of instants.
    The output R has the shape (ni, 3, 3), L has the shape (ni, 3), and RMSE
    has the shape (ni,). If ni is equal to one, the outputs will have the
    singleton dimension dropped.

    Part of this code is based on the programs written by Alberto Leardini,
    Christoph Reinschmidt, and Ton van den Bogert.

    Parameters
    ----------
    A   : Numpy array
        Coordinates [x,y,z] of at least three markers with two possible shapes:
        order='row': 2D array (n, 3), where n is the number of markers.
        order='col': 1D array (3*nmarkers,) like [x1, y1, z1, ..., xn, yn, zn].

    B   : 2D Numpy array
        Coordinates [x,y,z] of at least three markers with two possible shapes:
        order='row': 2D array (n, 3), where n is the number of markers.
        order='col': 2D array (ni, n*3), where ni is the number of instants.
        If ni=1, B is a 1D array like A.

    order : string
```

```
        'col': specifies that A and B are column oriented (default).
        'row': specifies that A and B are row oriented.

    Returns
    -------
    R   : Numpy array
        Rotation matrix between A and B with two possible shapes:
        order='row': (3, 3).
        order='col': (ni, 3, 3), where ni is the number of instants.
        If ni=1, R will have the singleton dimension dropped.

    L   : Numpy array
        Translation vector between A and B with two possible shapes:
        order='row': (3,) if order = 'row'.
        order='col': (ni, 3), where ni is the number of instants.
        If ni=1, L will have the singleton dimension dropped.

    RMSE : array
        Root-mean-squared error for the rigid body model: B = R*A + L + err
        with two possible shapes:
        order='row': (1,).
        order='col': (ni,), where ni is the number of instants.

    See Also
    --------
    numpy.linalg.svd

    Notes
    -----
    The singular value decomposition (SVD) algorithm decomposes a matrix M
    (which represents a general transformation between two coordinate systems)
    into three simple transformations [3]_: a rotation Vt, a scaling factor S
    along the  rotated axes and a second rotation U: M = U*S*Vt.
    The rotation matrix is given by: R = U*Vt.

    References
    ----------
    .. [1] Soderkvist, Kedin (1993) Journal of Biomechanics, 26, 1473-1477.
    .. [2] http://www.kwon3d.com/theory/jkinem/rotmat.html.
    .. [3] http://en.wikipedia.org/wiki/Singular_value_decomposition.


    Examples
    --------
    >>> import numpy as np
    >>> from svdt import svdt
    >>> A = np.array([0,0,0, 1,0,0,  0,1,0,  1,1,0])  # four markers
    >>> B = np.array([0,0,0, 0,1,0, -1,0,0, -1,1,0])  # four markers
    >>> R, L, RMSE = svdt(A, B)
    >>> B = np.vstack((B, B))  # simulate two instants (two rows)
    >>> R, L, RMSE = svdt(A, B)
    >>> A = np.array([[0,0,0], [1,0,0], [ 0,1,0], [ 1,1,0]])  # four markers
    >>> B = np.array([[0,0,0], [0,1,0], [-1,0,0], [-1,1,0]])  # four markers
    >>> R, L, RMSE = svdt(A, B, order='row')
    """

    A, B = np.asarray(A), np.asarray(B)
    if order == 'row' or B.ndim == 1:
        if B.ndim == 1:
            A = A.reshape(int(A.size/3), 3)
            B = B.reshape(int(B.size/3), 3)
```

```
        R, L, RMSE = svd(A, B)
    else:
        A = A.reshape(int(A.size/3), 3)
        ni = B.shape[0]
        R = np.empty((ni, 3, 3))
        L = np.empty((ni, 3))
        RMSE = np.empty(ni)
        for i in range(ni):
            R[i, :, :], L[i, :], RMSE[i] = svd(A, B[i, :].reshape(A.shape))

    return R, L, RMSE


def svd(A, B):
    """Calculates the transformation between two coordinate systems using SVD.

    See the help of the svdt function.

    Parameters
    ----------
    A   : 2D Numpy array (n, 3), where n is the number of markers.
        Coordinates [x,y,z] of at least three markers
    B   : 2D Numpy array (n, 3), where n is the number of markers.
        Coordinates [x,y,z] of at least three markers

    Returns
    -------
    R    : 2D Numpy array (3, 3)
         Rotation matrix between A and B
    L    : 1D Numpy array (3,)
         Translation vector between A and B
    RMSE : float
         Root-mean-squared error for the rigid body model: B = R*A + L + err.

    See Also
    --------
    numpy.linalg.svd
    """

    Am = np.mean(A, axis=0)           # centroid of m1
    Bm = np.mean(B, axis=0)           # centroid of m2
    M = np.dot((B - Bm).T, (A - Am))  # considering only rotation
    # singular value decomposition
    U, S, Vt = np.linalg.svd(M)
    # rotation matrix
    R = np.dot(U, np.dot(np.diag([1, 1, np.linalg.det(np.dot(U, Vt))]), Vt))
    # translation vector
    L = B.mean(0)  - np.dot(R, A.mean(0))
    # RMSE
    err = 0
    for i in range(A.shape[0]):
        Bp = np.dot(R, A[i, :]) + L
        err += np.sum((Bp - B[i, :])**2)
    RMSE = np.sqrt(err/A.shape[0]/3)

    return R, L, RMSE
```

In [ ]: