main

January 8, 2024

1 Interconversion of cartesian and internal coordinates

Internal coordinates are an popular method for defining molecular structures. Their utility lies in providing a definition of geometry which is independent of a molecule's position in cartesian space, and thus easily comparable. Additionally, internal coordinates directly contain relevant about molecules (i.e., bond lengths and bond angles), allowing certain properties to be fixed as constant during a quantum chemistry calculation (e.g. enforcing symmetry by fixing bond angles).1

The most common format for specifying internal coordinates is the Z-matrix, which consists of one row per atom in a molecule, with the following basic format:

Atom label
Atom label | *Bond length*
Atom label | *Bond length* | *Bond angle*
Atom label | *Bond length* | *Bond angle* | *Dihedral angle*
Atom label | *Bond length* | *Bond angle* | *Dihedral angle*
…
(Continue until all atoms listed)

where each bond/angle entry consists of a numerical value alongside the index of an additional atom that makes up the bond/angle. The first three lines contain fewer entries as less information is needed to uniquely identify these positions. The format is best illustrated using an example; below is a Z-matrix specifying the geometry of an ethane molecule:

```
C
C  1  1.51
H  1  1.09  2  110.6
H  1  1.09  2  110.6  3  120.0
H  1  1.09  2  110.6  4  120.0
H  2  1.09  1  110.6  5  60.0
H  2  1.09  1  110.6  5  300.0
H  2  1.09  1  110.6  5  180.0
```

There is, therefore, often a need within computational chemistry to convert between cartesian and internal coordinates. For example, a conversion from cartesian to internal is needed when a molecule has been drawn by hand and the user wants to create an input file for a quantum chemistry calculation. Conversely, calculation output files containing internal coordinates need to be converted to cartesian space in order to visualise the molecule. This notebook will outline a simple Python program for performing such a conversion.

To begin with, import required packages:

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import itertools # For generating pair combinations of lists
```

Next, we need to import some reference data - to infer bonds from cartesian coordinates, distance between atoms need to be compared with their atomic radii (covalent radii in this case, as van der Waals radii contain some outlier values e.g. for Hydrogen).2 To translate molecules to the centre of a cartesian coordinate system later on, atomic masses are needed for calculating a molecule's centre of mass. The Python dictionaries below were generated using RDKit (see the file **generatingData.py** for details).

[ ]:
```
covRadii =  {'H': 0.23, 'He': 0.93,
             'Li': 0.68, 'Be': 0.35, 'B': 0.83, 'C': 0.68, 'N': 0.68,
                  'O': 0.68, 'F': 0.64, 'Ne': 1.12,
             'Na': 0.97, 'Mg': 1.1, 'Al': 1.35, 'Si': 1.2, 'P': 0.75,
                  'S': 1.02, 'Cl': 0.99, 'Ar': 1.57,
             'K': 1.33, 'Ca': 0.99, 'Sc': 1.44, 'Ti': 1.47, 'V': 1.33,
                  'Cr': 1.35, 'Mn': 1.35, 'Fe': 1.34, 'Co': 1.33, 'Ni': 1.5,
                  'Cu': 1.52, 'Zn': 1.45, 'Ga': 1.22, 'Ge': 1.17, 'As': 1.21,
                  'Se': 1.22, 'Br': 1.21, 'Kr': 1.91,
             'Rb': 1.47, 'Sr': 1.12, 'Y': 1.78, 'Zr': 1.56, 'Nb': 1.48,
                'Mo': 1.47, 'Tc': 1.35, 'Ru': 1.4, 'Rh': 1.45, 'Pd': 1.5,
                'Ag': 1.59, 'Cd': 1.69, 'In': 1.63, 'Sn': 1.46, 'Sb': 1.46,
                'Te': 1.47, 'I': 1.4, 'Xe': 1.98}

masses = {'H': 1.008, 'He': 4.003,
          'Li': 6.941, 'Be': 9.012, 'B': 10.812, 'C': 12.011, 'N': 14.007,
                'O': 15.999, 'F': 18.998, 'Ne': 20.18,
          'Na': 22.99, 'Mg': 24.305, 'Al': 26.982, 'Si': 28.086, 'P': 30.974,
                'S': 32.067, 'Cl': 35.453, 'Ar': 39.948,
          'K': 39.098, 'Ca': 40.078, 'Sc': 44.956, 'Ti': 47.867, 'V': 50.944,
                'Cr': 51.996, 'Mn': 54.938, 'Fe': 55.845, 'Co': 58.933,
                'Ni': 58.693, 'Cu': 63.546, 'Zn': 65.39, 'Ga': 69.723,
                'Ge': 72.61, 'As': 74.922, 'Se': 78.96, 'Br': 79.904, 'Kr': 83.
 ↪8,
          'Rb': 85.468, 'Sr': 87.62, 'Y': 88.906, 'Zr': 91.224, 'Nb': 92.906,
                'Mo': 95.94, 'Tc': 98.0, 'Ru': 101.07, 'Rh': 102.906,
                'Pd': 106.42, 'Ag': 107.868, 'Cd': 112.412, 'In': 114.818,
                'Sn': 118.711, 'Sb': 121.76, 'Te': 127.6, 'I': 126.904, 'Xe':␣
 ↪131.29}
```

For the conversion from cartesian to internal coordinates, we start with a .xyz file containing cartesian coordinates (see the file **generatingData.py** for details of how this was constructed). For the example of ethane, this file has the following form:

```
8

C -0.7542955261841914 0.009248801338110227 -0.05032591112472543
C 0.754295369574497 -0.009248626144117736 0.05032603249850831
H -1.0930938498328102 0.9032333744783438 -0.5823282907037728
H -1.1142314282525152 -0.87083083686362 -0.5915524137472471
H -1.2060317502185025 0.00944997760576202 0.9461453378660798
H 1.0930944206981041 -0.9032345644374347 0.5823266188659942
H 1.1142308350468102 0.8708300589856497 0.5915540019771358
H 1.2060319291685875 -0.009448184962715272 -0.9461453756319684

```

Thus the file needs to first be read into the python program. For human-readability of the code, it is convenient to write a simple "molecule" class in Python, which can be assigned attributes such as atoms and bonds. This class will also have a function which iterates over atoms and bonds to find each atom's nearest neighbours, and a function to print the molecule to check that everything is looking as it should.

[ ]:
```python
class molecule:
    def __init__(self, inputXYZ = None):
        self.atoms = self.getAtoms(inputXYZ)
        self.bonds = self.getBonds(self.atoms)
        self.getNeighbours()

    # Create two subclasses, for atoms and bonds
    class atom:
        def __init__(self, atomType = '', coords = [], index = None):
            self.atomType = atomType
            self.coords = coords
            self.index = index
            self.neighbours = []

        def updateNeighbours(self, inList):
            self.neighbours = inList

        def printAtom(self):
            print('Atom:', self.atomType, self.coords, self.index)

    class bond:
        def __init__(self, bondPosition = [], bondLength = 0):
            self.bondPosition = bondPosition
            self.bondLength = bondLength

        def printBond(self):
            print('Bond:', self.bondPosition, self.bondLength)

    def getAtoms(self, inputXYZ):
```

```
        with open(inputXYZ,'r') as f:
            outList = []
            lines = f.readlines()[2:] # Skipping comment line
            for index, line in enumerate(lines):
                atom,x,y,z = line.split()
                outList.append(self.atom(atomType = atom,
                                         coords = np.
↪array([float(x),float(y),float(z)]),
                                         index = index))
        return outList

    def getBonds(self, atoms):
        bondsList = []
        # For each combination of atoms:
        for pair in itertools.combinations(atoms,2): # the 2 is for generating␣
↪pair combinations
            # Calc distance by taking norm of subtracted vectors
            atom1, coords1, index1 = pair[0].atomType, np.array(pair[0].
↪coords), pair[0].index
            atom2, coords2, index2 = pair[1].atomType, np.array(pair[1].
↪coords), pair[1].index
            distance = np.linalg.norm(coords1 - coords2)
            # Accessing library of covalent radii which was loaded in earlier
            # Also, factor of 1.3 is commonplace in bond detection algorithms
            # E.g. in this paper: https://doi.org/10.1002/
↪(SICI)1096-987X(19960115)17:1%3C49::AID-JCC5%3E3.0.CO;2-0
            threshold = 1.3 * (covRadii[atom1] + covRadii[atom2])
            if (distance < threshold):
                bondsList.append(self.bond(bondPosition = [index1,index2],
                                           bondLength = distance))
        return bondsList

    def getNeighbours(self):
        ''' For each atom in molecule, compute its neighbours, and add as␣
↪property to the atom object '''
        for atom in self.atoms:
            atomNeighbours = []
            for bond in self.bonds:
                bondMembers = set(bond.bondPosition)
                if atom.index in bondMembers:
                    bondMembers.remove(atom.index)
                    atomNeighbours.append(list(bondMembers)[0])
            atom.updateNeighbours(atomNeighbours)

    def printMolecule(self):
        for atom in self.atoms:
```

```
            atom.printAtom()
            if len(atom.neighbours) > 0:
                print('Neighbours: ',atom.neighbours)
        for bond in self.bonds:
            bond.printBond()

testMol = molecule(inputXYZ = 'xyzFiles/CH3CH3.xyz')
testMol.printMolecule()
```

Executing the above cell should print a list of atoms, nearest neighbours, and bonds for ethane. Next, to construct internal coordinates from this information, we first need to define two short functions for calculating the bond angle and dihedral angle from three-dimensional vectors:

[ ]:
```
def calcAngle(i,j,k):
    ji = i-j
    jk = k-j
    cos = np.dot(ji, jk) / (np.linalg.norm(ji)*np.linalg.norm(jk))
    angle = np.degrees(np.arccos(cos))
    return angle

def calcDihedral(i,j,k,l):
    i = i.coords
    j = j.coords
    k = k.coords
    l = l.coords

    # Source for the following calculation method:
    # https://stackoverflow.com/a/34245697
    # (Fast and numerically stable method for calculating dihedrals)
    ij = -1.0*(j-i)
    jk = k-j
    kl = l-k
    jk /= np.linalg.norm(jk)
    n1 = np.cross(ij,jk)
    n2 = np.cross(jk,kl)
    n1 /= np.linalg.norm(n1)
    n2 /= np.linalg.norm(n2)
    v = ij - np.dot(ij, jk)*jk
    w = kl - np.dot(jk, jk)*jk
    x = np.dot(v,w)
    y = np.dot(np.cross(jk, v), w)
    dihed = np.degrees(np.arctan2(y,x))

    if dihed < 0:
        dihed += 360 # Transforming range [-180,180] to [0,360]
    return dihed
```

```
# A simple example of how these functions work
print(calcAngle(np.array([0,1,0]),
                np.array([1,0,0]),
                np.array([0,0,1])))
```

Using the above components, a function can be written to iterate through atoms and write a Z-matrix.

It turns out that in many cases it is sufficient to iterate through atoms in the order specified in an .xyz file. However, this is not always the case, and the code below will fail if the atoms are not specified in an order which follows a path along the backbone of a molecule. This is because the code imposes the constraint that bonds and bond angles must follow along actual bonds - in principle, however, this constraint is not needed in order to specify the positions of a list of atoms in space, as the Z-matrix does not contain complete bonding information in any case (e.g. for cyclic molecules, not all bonds will be represented). A more complete program for solving this problem (beyond the scope of this notebook) would need to include an algorithm for finding pathways through a molecule.3

The below function uses the molecule object to construct a Z-matrix and write this into the form of a text file.

[ ]:
```
def writeZMat(mol, outFileName):
    atomsToAdd = mol.atoms.copy()
    addedAtoms = []

    # Defining some nested functions specific to the Z-matrix writing task
    # These are used later in the "write file" block
    def findAtomToAdd():
        '''
        Search for an atom bonded to a previously added atom
        Return that atom + the one it was bonded to
        '''
        for atom in atomsToAdd:
            for anchorAtom in addedAtoms:
                if atom.index in anchorAtom.neighbours:
                    return atom, anchorAtom

    def findAtomForAngle(at, anchorAt):
        '''
        Return an atom that forms an angle with the two atoms inputted
        '''
        for i in anchorAt.neighbours:
            if i != at.index:
                return mol.atoms[i]

    def findAtomForDihed(at, anchorAt, angleAt):
        dihedAt = None
        found = False
```

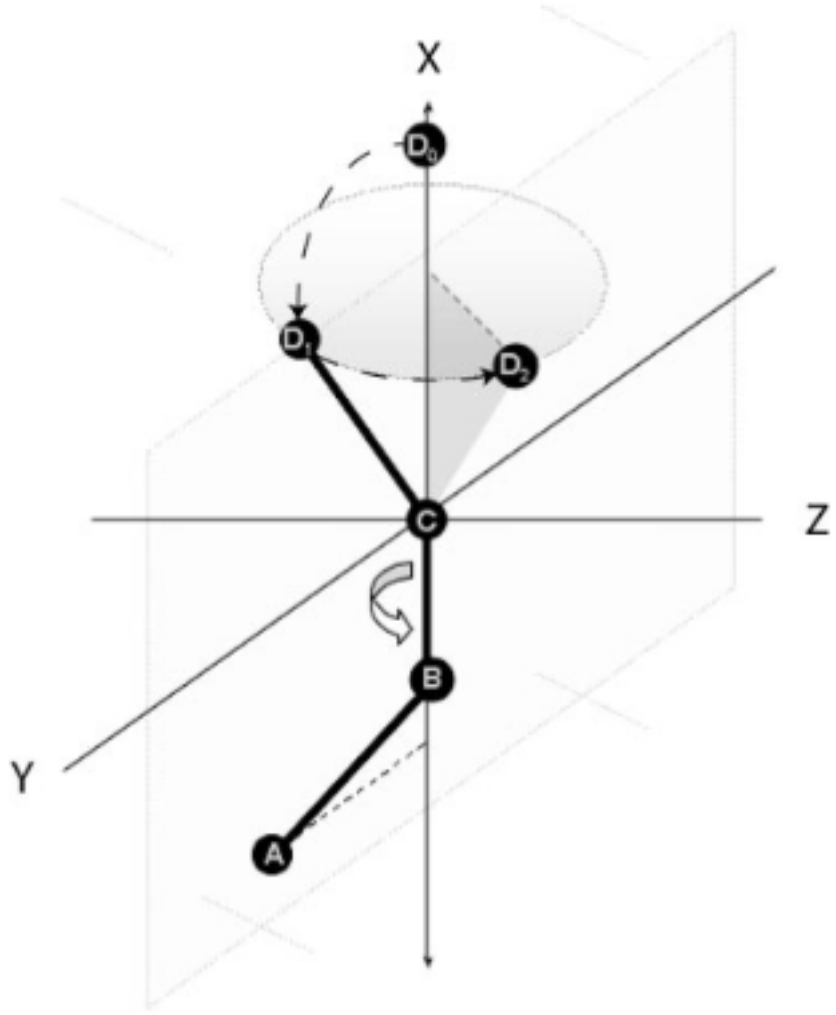
```
        for i in angleAt.neighbours:
            # If atom is not equal to current atom/anchor atom, and is among␣
↪those already added
            if (i != at.index) and (i != anchorAt.index) and (mol.atoms[i] in␣
↪addedAtoms):
                dihedAt = mol.atoms[i]
                found = True

        # If there are no atoms available to construct a "conventional"␣
↪dihedral angle, use one that is bonded to angleAtom,
        # effectively constructing a virtual bond e.g. between two hydrogens in␣
↪NH3
        if found == False:
            for i in anchorAt.neighbours:
                if (i != at.index) and (i != anchorAt.index) and (mol.atoms[i]␣
↪in addedAtoms):
                    dihedAt = mol.atoms[i]
                    found = True

        if found == True:
            return dihedAt
        else:
            print('Failed to find atom for dihedral angle')

    def noteAdded(atom):
        atomsToAdd.remove(atom)
        addedAtoms.append(atom)

    # Write file
    with open(outFileName, 'w') as f:
        # 1st line - going to make assumption that first atom listed is␣
↪acceptable place to start
        at1 = atomsToAdd[0]
        b1 = None # Don't need bonds and angles yet
        a1 = None
        f.write(at1.atomType)
        noteAdded(at1)

        # 2nd line - find something bonded to atom1
        at2 = findAtomToAdd()[0]
        b2 = np.linalg.norm(at1.coords - at2.coords) # Anchor atom has to be at1
        a2 = None
        f.write('\n' + at2.atomType + ' '
                + '1' + ' ' + '{:.6f}'.format(b2))
        noteAdded(at2)
```

```
        # 3rd line - need atom, bond, angle
        at3, anchorAt3 = findAtomToAdd()
        b3 = np.linalg.norm(at3.coords - anchorAt3.coords)
        angleAt3 = findAtomForAngle(at3, anchorAt3)
        a3 = calcAngle(at3.coords, anchorAt3.coords, angleAt3.coords)
        f.write('\n' + at3.atomType + ' '
                + str(anchorAt3.index+1) + ' ' + '{:.6f}'.format(b3) + ' '
                + str(angleAt3.index+1) + ' ' + '{:.6f}'.format(a3))
        noteAdded(at3)

        # Rest of atoms
        while atomsToAdd:
            at, anchorAt = findAtomToAdd()
            b = np.linalg.norm(at.coords - anchorAt.coords)
            angleAt = findAtomForAngle(at, anchorAt)
            ang = calcAngle(at.coords, anchorAt.coords, angleAt.coords)

            try:
                dihedAt = findAtomForDihed(at, anchorAt, angleAt)
                dihed = calcDihedral(at, anchorAt, angleAt, dihedAt)
                f.write('\n' + at.atomType + ' '
                        + str(anchorAt.index+1) + ' ' + '{:.6f}'.format(b) + ' '
                        + str(angleAt.index+1) + ' ' + '{:.6f}'.format(ang) + '␣
↪'
                        + str(dihedAt.index+1) + ' ' + '{:.6f}'.format(dihed))␣
↪# Format to 6 sig figs
            except:
                print('Failed to find atom for dihedral angle')

            noteAdded(at)

        print('Completed Z-matrix construction')

writeZMat(mol = testMol,
          outFileName = 'outZMat.txt')
```

This should write a file that looks as follows:

```
C
C 1 1.512058
H 1 1.094084 2 110.567945
H 1 1.094084 2 110.567965 3 120.000026
H 1 1.094084 2 110.567953 4 119.999985
H 2 1.094084 1 110.567970 5 60.000138
H 2 1.094084 1 110.567950 5 300.000084
H 2 1.094085 1 110.567964 5 180.000091
9
```

Now, to check whether this is a sensible Z-matrix, we can write a function for the reverse conversion from internal to cartesian coordinates. This does not need to involve a molecule object, as the transformation can simply be done by placing each atom in the Z-matrix successively. Placing atoms requires some mathematical transformations of coordinates to rotate them according to bond angles and dihedral angles - for example, if placing an atom D with reference to previous atoms A, B and C, one can place a point D0 along the B-C bond axis according to the new bond length, rotate it in the A-B-C plane according to the bond angle, and rotate around the B-C bond axis according to the dihedral angle: 4



The function below takes a text file containing a Z-matrix as input, and writes a .xyz file containing the corresponding cartesian coordinates. It translates and rotates the coordinates to put the centre of mass at the origin and align the molecule with its principal rotation axes.5

[ ]:
```
def ZMatToXYZ(inputZMat, outFileName):
    # Read input ZMat
    with open(inputZMat,'r') as f:
        ZMat = []
        for line in f.readlines():
            if '\n' in line:
                line = line.replace('\n','') # Remove trailing '\n' from each␣
↪line
            line = line.split()
            for idx,item in enumerate(line):
                if item[0].isnumeric():
                    line[idx] = float(item) # Convert strings of numbers to␣
↪numbers
            ZMat.append(line)
```

```
    atomsList = [] # Creating temporary separate list of atom symbols, for ease of coordinate transformations
    rawCoords = []

    # First atom
    atomsList.append(ZMat[0][0])
    rawCoords.append([0., 0., 0.])

    # Second atom
    atomsList.append(ZMat[1][0])
    rawCoords.append([0., 0., ZMat[1][2]]) # Place along z axis with distance of 1st bond length

    # Some functions for determining positions of next atoms
    def rotate(vector,axis,angle):
        '''
        Rodrigues-Gibbs formula for rotating a vector around an axis by a given angle (from reference 4)
        '''
        return vector * np.cos(angle) + np.cross(axis, vector * np.sin(angle)) + axis * np.dot(vector, axis) * (1-np.cos(angle))

    def calcCartesian(aCoords,bCoords,cCoords,bondLength,ang,dihed):
        '''
        Calculates cartesian coordinates of atom D given atoms A, B and C (see reference 4)
        '''
        AB = bCoords - aCoords
        BC = bCoords - cCoords # It turns out the code works when defining it this way around - perhaps a typo in the original paper

        bc = BC / np.linalg.norm(BC)
        n = np.cross(AB,bc) / np.linalg.norm(np.cross(AB,bc))

        d0 = bondLength * bc # Initial position is C-D bond length * direction of BC bond (treating C as origin) - translate later
        d1 = rotate(d0, n, ang)
        d2 = rotate(d1, bc, dihed)

        dCoords = cCoords + d2 # Translate origin from C to 0,0,0

        return dCoords

    # Add rest of atoms
    for idx, line in enumerate(ZMat[2:]):
```

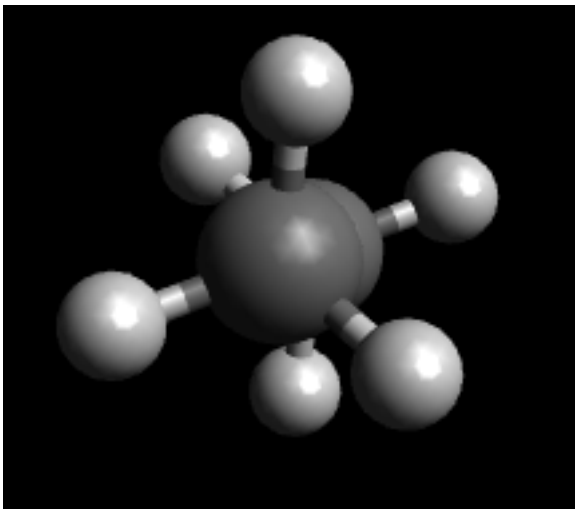
```
        # Term the previous three atoms A, B and C
        cIdx = int(line[1]) - 1
        bIdx = int(line[3]) - 1
        cCoords = np.array(rawCoords[cIdx])
        bCoords = np.array(rawCoords[bIdx])
        bondLength = line[2]
        ang = np.radians(line[4])

        if idx == 0: # If this is the 3rd atom
            aCoords = np.array([0,1,0])
            dihed = 0
        else:
            aIdx = int(line[5]) - 1
            aCoords = np.array(rawCoords[aIdx])
            dihed = np.radians(line[6])

        dCoords = calcCartesian(aCoords,bCoords,cCoords,bondLength,ang,dihed)

        atomsList.append(line[0])
        rawCoords.append([dCoords[0], dCoords[1], dCoords[2]]) # Atom type, 3x␣
↪coords

    # Next, want to translate + rotate the arbitrary coords that have been␣
↪determined

    # Calculate centre of mass and move this to the origin
    massesList = np.array([masses[atomType] for atomType in atomsList])
    transposedCoords = np.swapaxes(rawCoords,0,1)
    # Centre of mass for x,y,z components (same result as calculating for␣
↪components together)
    centreOfMass = [(component*massesList).sum() / massesList.sum() for␣
↪component in transposedCoords] # For x,y,z components
    translCoords = np.array(rawCoords) - centreOfMass

    # Calculate inertia matrix and rotate molecule onto its principal rotation␣
↪axes (see reference 5)
    I_xx = sum([m * (coords[1]**2 + coords[2]**2) for m,coords in␣
↪zip(massesList,rawCoords)])
    I_yy = sum([m * (coords[0]**2 + coords[2]**2) for m,coords in␣
↪zip(massesList,rawCoords)])
    I_zz = sum([m * (coords[0]**2 + coords[1]**2) for m,coords in␣
↪zip(massesList,rawCoords)])
    I_xy = sum([(m * coords[0] * coords[1]) for m,coords in␣
↪zip(massesList,rawCoords)])
    I_xz = sum([(m * coords[0] * coords[2]) for m,coords in␣
↪zip(massesList,rawCoords)])
```

```
    I_yz = sum([(m * coords[1] * coords[2]) for m,coords in␣
 ↪zip(massesList,rawCoords)])
    I = np.array([[I_xx, -I_xy, -I_xz],
                  [-I_xy, I_yy, -I_yz],
                  [-I_xz, -I_yz, I_zz]])
    evals, evectors = np.linalg.eig(I)
    rotMatrix = np.linalg.inv(evectors)
    rotCoords = np.matmul(rotMatrix,translCoords.T).T

    with open(outFileName,'w') as f:
        nAtoms = len(ZMat) # Convention is to write number of atoms in 1st line
        f.write(str(nAtoms))
        f.write('\n')
        for atom,coords in zip(atomsList,rotCoords):
            line = atom + ' ' + ' '.join(str(item) for item in coords)
            f.write('\n'+line)

    print('Completed cartesian coordinates construction')

ZMatToXYZ(inputZMat = 'outZMat.txt',
          outFileName = 'outXYZ.txt')
```

Now, if we create a molecule object from the reconstructed coordinates, and write a quick function to plot it, we can see the reconstructed molecule:

[ ]:
```
reconstructedMol = molecule(inputXYZ = 'outXYZ.txt')

def plotMolecule(mol):
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    for atom in mol.atoms:
        ax.text(atom.coords[0],atom.coords[1],atom.coords[2],str(atom.atomType))
    for idx,bond in enumerate(mol.bonds):
        idx1, idx2 = bond.bondPosition
        coords1, coords2 = mol.atoms[idx1].coords, mol.atoms[idx2].coords
        plotVals = list(zip(coords1,coords2)) # transpose to get separate x, y,␣
 ↪z
        ax.plot(*plotVals,c='k')
    plt.show()

plotMolecule(reconstructedMol)
```

Inputting the coordinates into any molecular structure editor makes it easier to see that the bond angles are indeed correct (the image below is a screenshot from copy-and-pasting the coordinates in outXYZ.txt into Avogadro):



The below cell contains some optional code for other example molecules, including one larger one, for which this notebook can perform coordinate conversions.

Some notable limitations of this program include:

- molecules which contain a dihedral angle of 180 degrees (fix by either specifying dummy atoms or defining extended dihedral angle)6
- .xyz files which are not ordered in a way that provides a path along bonds in the molecule (fix by either implementing a path-finding algorithm, or could brute-force a solution by shuffling atoms until a Z-matrix can be constructed)
- files describing a transition state geometry i.e. no path available through all atoms along bonds (fix by creating a virtual bond between fragments)

[ ]:
```
# testMol2 = molecule(inputXYZ = 'xyzFiles/acetate.xyz')
# writeZMat(testMol2, outFileName = 'testMol2_ZMat')
# ZMatToXYZ('testMol2_ZMat',outFileName = 'testMol2_reconstructed.xyz')
# plotMolecule(molecule('testMol2_reconstructed.xyz'))

# The following large molecule containing 39 atoms also works, although␣
 ↪difficult to view in matplotlib:

# testMol3 = molecule(inputXYZ = 'xyzFiles/gs_150_1.xyz')
# writeZMat(testMol3, outFileName = 'testMol3_ZMat')
# ZMatToXYZ('testMol3_ZMat',outFileName = 'testMol3_reconstructed.xyz')
# plotMolecule(molecule('testMol3_reconstructed.xyz'))
```

1.0.1 References

1 Constructing Z-matrices, https://gaussian.com/zmat/, (accessed January 2024).

2 Molecular File Format Descriptions, http://www.ccl.net/chemistry/resources/messages/1996/10/21.005-dir/index.html, (accessed January 2024).

3 r2z (a python class for building a ZMatrix from a RDKit molecule), https://github.com/wutobias/r2z, (accessed January 2024).

4 Parsons, J., Holmes, J. B., Rojas, J. M., Tsai, J., & Strauss, C. E. (2005). Practical conversion from torsion space to Cartesian space for in silico protein synthesis. Journal of computational chemistry, 26(10), 1063-1068. https://doi.org/10.1002/jcc.20237

5 Principal rotation axes and principal moments of inertia, https://pythoninchemistry.org/ch40208/comp_chem_methods/moments_of_inertia.html (accessed January 2024).

6 Domenichini, G., & Dellago, C. (2023). Molecular Hessian matrices from a machine learning random forest regression algorithm. The Journal of Chemical Physics, 159(19). https://doi.org/10.1063/5.0169384