Tabular-data

November 8, 2023

0.1 Tabular Data and Graphing lesson

tabular data is generally presented in rows and colums. You might have analyzed this type of data in a spreadsheet program before.

[1]:
```
import numpy
```

[2]:
```
help(numpy.genfromtxt)
```

```
Help on function genfromtxt in module numpy:

genfromtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
skip_header=0, skip_footer=0, converters=None, missing_values=None,
filling_values=None, usecols=None, names=None, excludelist=None, deletechars="
!#$%&'()*+,-./:;<=>?@[\\]^{|}~", replace_space='_', autostrip=False,
case_sensitive=True, defaultfmt='f%i', unpack=None, usemask=False, loose=True,
invalid_raise=True, max_rows=None, encoding='bytes', *, like=None)
    Load data from a text file, with missing values handled as specified.

    Each line past the first `skip_header` lines is split at the `delimiter`
    character, and characters following the `comments` character are discarded.

    Parameters
    ----------
    fname : file, str, pathlib.Path, list of str, generator
        File, filename, list, or generator to read.  If the filename
        extension is ``.gz`` or ``.bz2``, the file is first decompressed. Note
        that generators must return bytes or strings. The strings
        in a list or produced by a generator are treated as lines.
    dtype : dtype, optional
        Data type of the resulting array.
        If None, the dtypes will be determined by the contents of each
        column, individually.
    comments : str, optional
        The character used to indicate the start of a comment.
        All the characters occurring on a line after a comment are discarded.
    delimiter : str, int, or sequence, optional
        The string used to separate values.  By default, any consecutive
        whitespaces act as delimiter.  An integer or sequence of integers
```

```
    can also be provided as width(s) of each field.
skiprows : int, optional
    `skiprows` was removed in numpy 1.10. Please use `skip_header` instead.
skip_header : int, optional
    The number of lines to skip at the beginning of the file.
skip_footer : int, optional
    The number of lines to skip at the end of the file.
converters : variable, optional
    The set of functions that convert the data of a column to a value.
    The converters can also be used to provide a default value
    for missing data: ``converters = {3: lambda s: float(s or 0)}``.
missing : variable, optional
    `missing` was removed in numpy 1.10. Please use `missing_values`
    instead.
missing_values : variable, optional
    The set of strings corresponding to missing data.
filling_values : variable, optional
    The set of values to be used as default when the data are missing.
usecols : sequence, optional
    Which columns to read, with 0 being the first.  For example,
    ``usecols = (1, 4, 5)`` will extract the 2nd, 5th and 6th columns.
names : {None, True, str, sequence}, optional
    If `names` is True, the field names are read from the first line after
    the first `skip_header` lines. This line can optionally be preceded
    by a comment delimiter. If `names` is a sequence or a single-string of
    comma-separated names, the names will be used to define the field names
    in a structured dtype. If `names` is None, the names of the dtype
    fields will be used, if any.
excludelist : sequence, optional
    A list of names to exclude. This list is appended to the default list
    ['return','file','print']. Excluded names are appended with an
    underscore: for example, `file` would become `file_`.
deletechars : str, optional
    A string combining invalid characters that must be deleted from the
    names.
defaultfmt : str, optional
    A format used to define default field names, such as "f%i" or "f_%02i".
autostrip : bool, optional
    Whether to automatically strip white spaces from the variables.
replace_space : char, optional
    Character(s) used in replacement of white spaces in the variable
    names. By default, use a '_'.
case_sensitive : {True, False, 'upper', 'lower'}, optional
    If True, field names are case sensitive.
    If False or 'upper', field names are converted to upper case.
    If 'lower', field names are converted to lower case.
unpack : bool, optional
    If True, the returned array is transposed, so that arguments may be
```

```
    unpacked using ``x, y, z = genfromtxt(…)``.  When used with a
    structured data-type, arrays are returned for each field.
    Default is False.
usemask : bool, optional
    If True, return a masked array.
    If False, return a regular array.
loose : bool, optional
    If True, do not raise errors for invalid values.
invalid_raise : bool, optional
    If True, an exception is raised if an inconsistency is detected in the
    number of columns.
    If False, a warning is emitted and the offending lines are skipped.
max_rows : int,  optional
    The maximum number of rows to read. Must not be used with skip_footer
    at the same time.  If given, the value must be at least 1. Default is
    to read the entire file.

    .. versionadded:: 1.10.0
encoding : str, optional
    Encoding used to decode the inputfile. Does not apply when `fname` is
    a file object.  The special value 'bytes' enables backward compatibility
    workarounds that ensure that you receive byte arrays when possible
    and passes latin1 encoded strings to converters. Override this value to
    receive unicode arrays and pass strings as input to converters.  If set
    to None the system default is used. The default value is 'bytes'.

    .. versionadded:: 1.14.0
like : array_like
    Reference object to allow the creation of arrays which are not
    NumPy arrays. If an array-like passed in as ``like`` supports
    the ``__array_function__`` protocol, the result will be defined
    by it. In this case, it ensures the creation of an array object
    compatible with that passed in via this argument.

    .. versionadded:: 1.20.0

Returns
-------
out : ndarray
    Data read from the text file. If `usemask` is True, this is a
    masked array.

See Also
--------
numpy.loadtxt : equivalent function when no data is missing.

Notes
-----
```

```
* When spaces are used as delimiters, or when no delimiter has been given
  as input, there should not be any missing data between two fields.
* When the variables are named (either by a flexible dtype or with `names`),
  there must not be any header in the file (else a ValueError
  exception is raised).
* Individual values are not stripped of spaces by default.
  When using a custom converter, make sure the function does remove spaces.

References
----------
.. [1] NumPy User Guide, section `I/O with NumPy
       <https://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html>`_.

Examples
--------
>>> from io import StringIO
>>> import numpy as np

Comma delimited file with mixed dtype

>>> s = StringIO(u"1,1.3,abcde")
>>> data = np.genfromtxt(s, dtype=[('myint','i8'),('myfloat','f8'),
… ('mystring','S5')], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])

Using dtype = None

>>> _ = s.seek(0) # needed for StringIO example only
>>> data = np.genfromtxt(s, dtype=None,
… names = ['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])

Specifying dtype and names

>>> _ = s.seek(0)
>>> data = np.genfromtxt(s, dtype="i8,f8,S5",
… names=['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])

An example with fixed-width columns

>>> s = StringIO(u"11.3abcde")
```

```
>>> data = np.genfromtxt(s, dtype=None, names=['intvar','fltvar','strvar'],
…     delimiter=[1,3,5])
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('intvar', '<i8'), ('fltvar', '<f8'), ('strvar', 'S5')])

An example to show comments

>>> f = StringIO('''
… text,# of chars
… hello world,11
… numpy,5''')
>>> np.genfromtxt(f, dtype='S12,S12', delimiter=',')
array([(b'text', b''), (b'hello world', b'11'), (b'numpy', b'5')],
  dtype=[('f0', 'S12'), ('f1', 'S12')])
```

[3]:
```
filepath = 'data/distance_data_headers.csv'

distances = numpy.genfromtxt(fname=filepath, delimiter=',', dtype='unicode')
```

[4]:
```
print(distances)
```

```
[['Frame' 'THR4_ATP' 'THR4_ASP' 'TYR6_ATP' 'TYR6_ASP']
 ['1' '8.9542' '5.8024' '11.5478' '9.9557']
 ['2' '8.6181' '6.0942' '13.9594' '11.6945']
 …
 ['9998' '8.6625' '7.7306' '9.5469' '10.3063']
 ['9999' '9.2456' '7.8886' '9.8151' '10.7564']
 ['10000' '8.8135' '7.917' '9.9517' '10.7848']]
```

[5]:
```
headers = distances[0]
```

[6]:
```
print(headers)
```

```
['Frame' 'THR4_ATP' 'THR4_ASP' 'TYR6_ATP' 'TYR6_ASP']
```

[7]:
```
data = distances[1:]
print(data)
```

```
[['1' '8.9542' '5.8024' '11.5478' '9.9557']
 ['2' '8.6181' '6.0942' '13.9594' '11.6945']
 ['3' '9.0066' '6.0637' '13.0924' '11.3043']
 …
 ['9998' '8.6625' '7.7306' '9.5469' '10.3063']
 ['9999' '9.2456' '7.8886' '9.8151' '10.7564']
 ['10000' '8.8135' '7.917' '9.9517' '10.7848']]
```

[8]:
```
data = data.astype(float)
```

[9]:
```
print(data)
```

```
[[1.00000e+00 8.95420e+00 5.80240e+00 1.15478e+01 9.95570e+00]
 [2.00000e+00 8.61810e+00 6.09420e+00 1.39594e+01 1.16945e+01]
 [3.00000e+00 9.00660e+00 6.06370e+00 1.30924e+01 1.13043e+01]
 …
 [9.99800e+03 8.66250e+00 7.73060e+00 9.54690e+00 1.03063e+01]
 [9.99900e+03 9.24560e+00 7.88860e+00 9.81510e+00 1.07564e+01]
 [1.00000e+04 8.81350e+00 7.91700e+00 9.95170e+00 1.07848e+01]]
```

[10]:
```
# to access a specific element of an array, array_name[row,column]
```

[11]:
```
print(data[2,1])
```

```
9.0066
```

[12]:
```
point1 = data[0,1]
point2 = data[1,0]

point1 = 8.95420
point2 = 2.00000

print(f'point1 is {point1}')
```

```
point1 is 8.9542
```

[13]:
```
print(f'point2 is {point2}')
```

```
point2 is 2.0
```

[14]:
```
small_data = data[0:10, 0:3]
print(small_data)
```

```
[[ 1.      8.9542  5.8024]
 [ 2.      8.6181  6.0942]
 [ 3.      9.0066  6.0637]
 [ 4.      9.2002  6.0227]
 [ 5.      9.1294  5.9365]
 [ 6.      9.0462  6.2553]
 [ 7.      8.8657  5.9186]
 [ 8.      9.3256  6.2351]
 [ 9.      9.4184  6.1993]
 [10.      9.06    6.0478]]
```

[15]:
```
set1 = small_data[5,:]
set2 = small_data[:,1:]
```

```
print(set1)
```

```
[6.     9.0462 6.2553]
```

[16]:
```
print(set2)
```

```
[[8.9542 5.8024]
 [8.6181 6.0942]
 [9.0066 6.0637]
 [9.2002 6.0227]
 [9.1294 5.9365]
 [9.0462 6.2553]
 [8.8657 5.9186]
 [9.3256 6.2351]
 [9.4184 6.1993]
 [9.06   6.0478]]
```

[17]:
```
# to take an average numpy.mean(data_set)
thr4_atp_data = data[:,1]
print(thr4_atp_data)
```

```
[8.9542 8.6181 9.0066 … 8.6625 9.2456 8.8135]
```

[18]:
```
average = numpy.mean(thr4_atp_data)
```

[19]:
```
print(average)
```

```
10.876950930000001
```

[20]:
```
average = numpy.mean(data[:,1])
```

[21]:
```
print(average)
```

```
10.876950930000001
```

[22]:
```
# range for loop does an opporation a particular number of times
# range(start,end)
```

[25]:
```
# how do i find the number of columns?
# using skills you already know, think about how you could find the number of␣
 ↪columns.
num_columns = len(headers)
num_columns = len(data[200,:])
print(num_columns)
```

```
5
```

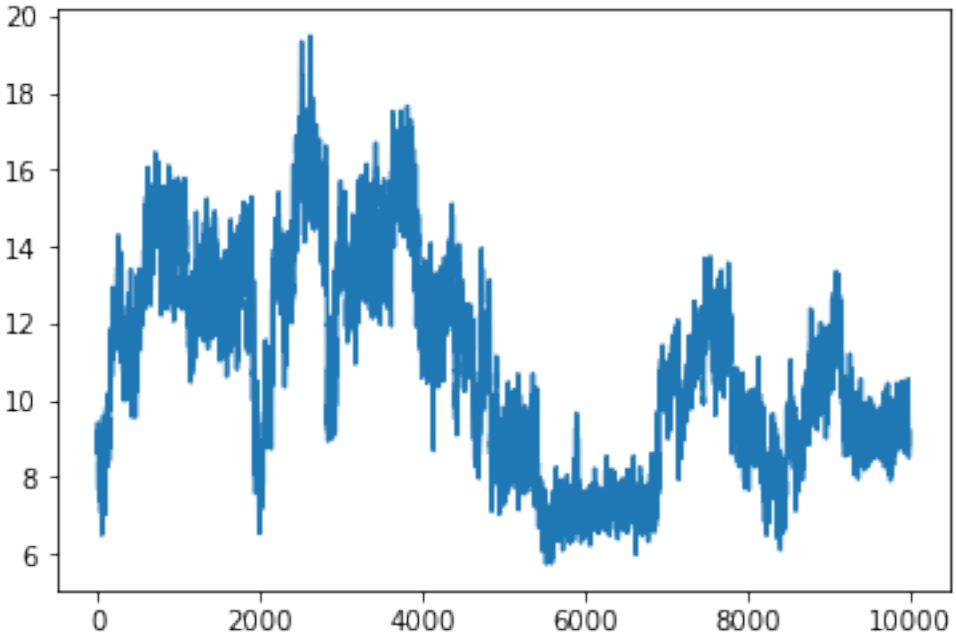
[31]:
```
for i in range(1,num_columns):
    data_column = data[:,i]
    average = numpy.mean(data_column)
    print(f'{headers[i]}:{average:.4f}')
```

```
THR4_ATP:10.8770
THR4_ASP:7.3423
TYR6_ATP:11.2098
TYR6_ASP:10.9934
```

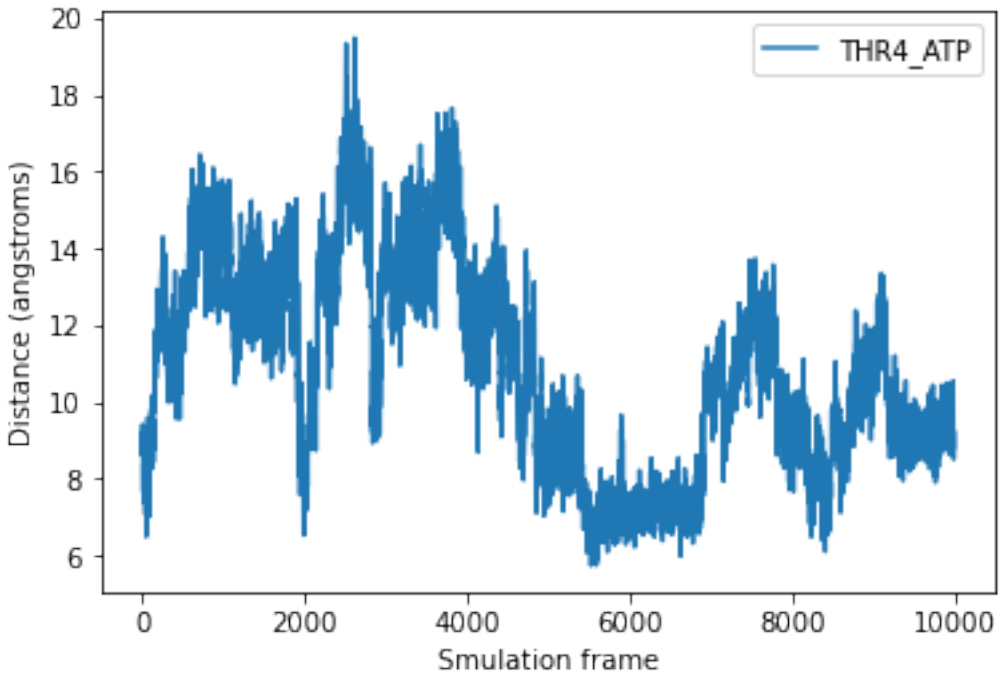
[33]:
```
import matplotlib.pyplot as plt
```

[37]:
```
plt.figure()
xdata = data[:,0]
ydata = data[:,1]
plt.plot(data[:,0], data[:,1])
```

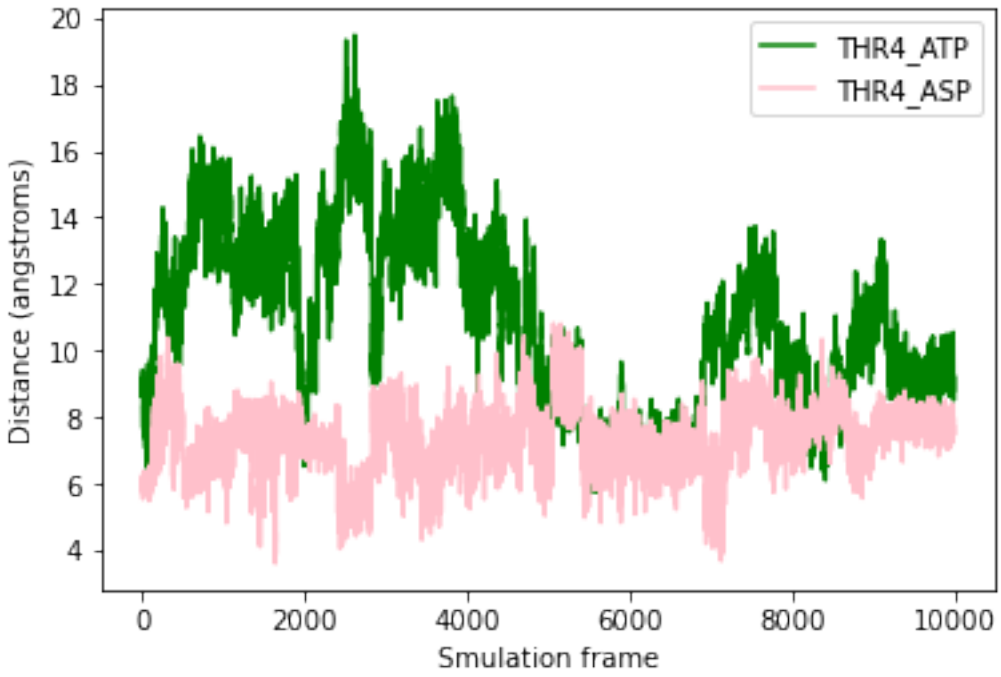
[37]: [<matplotlib.lines.Line2D at 0x7effeb73eca0>]



[44]:
```
plt.figure()
plt.plot(data[:,1], label=F'{headers[1]}')
plt.xlabel('Smulation frame')
plt.ylabel('Distance (angstroms)')
plt.legend()

plt.savefig(F'{headers[1]}.png', dpi=300)
```



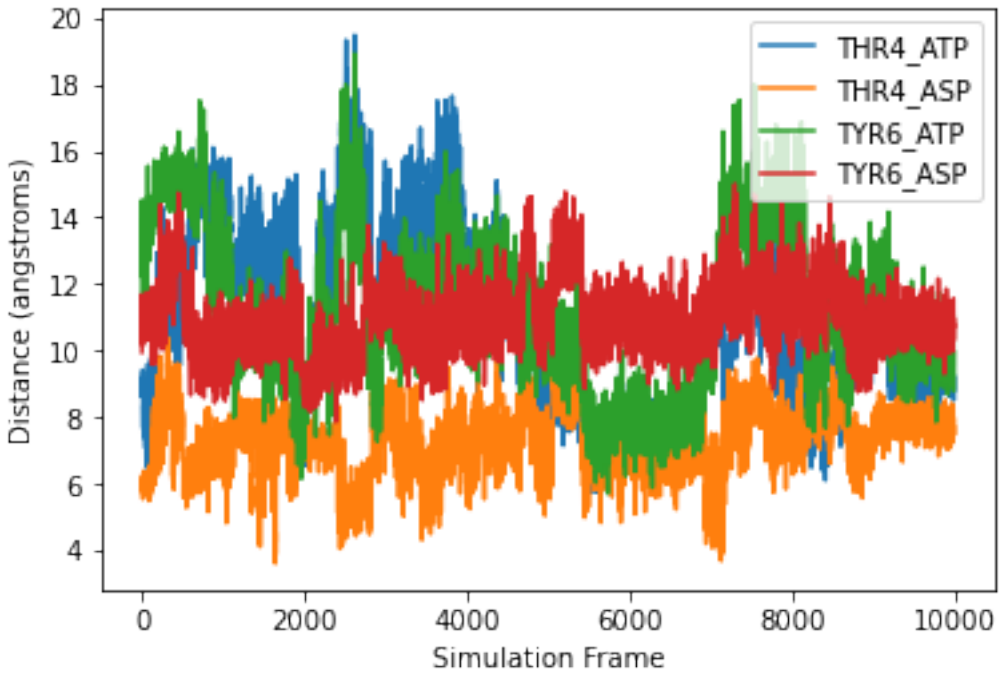
[54]:
```
plt.figure()
plt.plot(data[:,1], label=F'{headers[1]}', color='green')
plt.plot(data[:,2], label=F'{headers[2]}', color='pink')
plt.xlabel('Smulation frame')
plt.ylabel('Distance (angstroms)')
plt.legend()

plt.savefig(F'two_samples.png', dpi=300)
```



[55]:
```
# using the example abouve, use a range for loop to make a plot with all the␣
 ↪coloumns of data
```

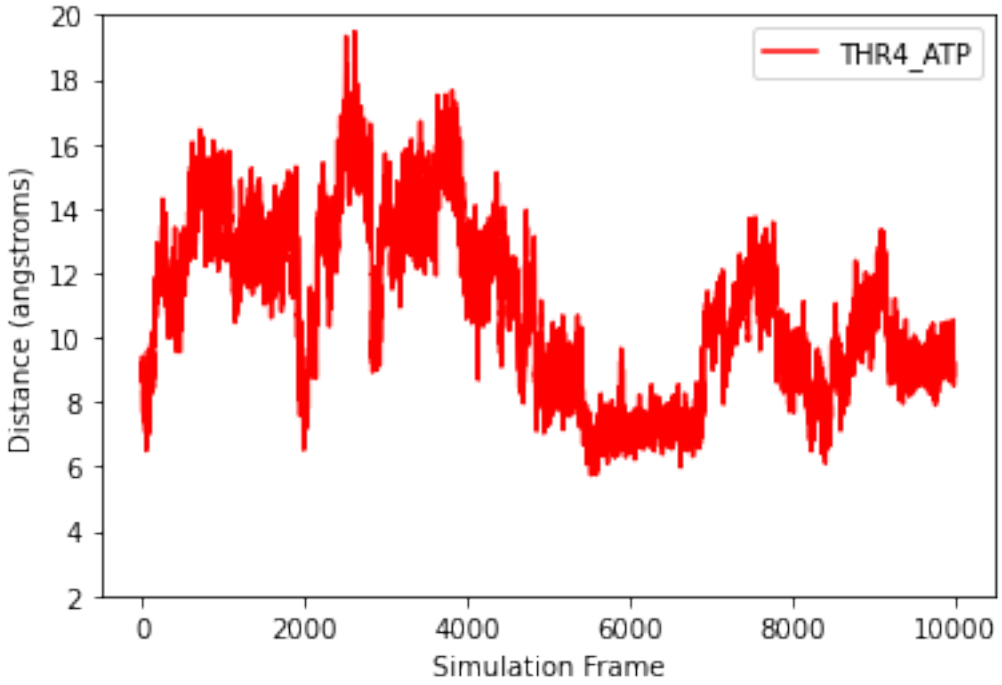
[57]:
```
plt.figure()
for i in range(1,num_columns):
    plt.plot(data[:,i], label=F'{headers[i]}')

plt.legend()
plt.xlabel('Simulation Frame')
plt.ylabel('Distance (angstroms)')
```

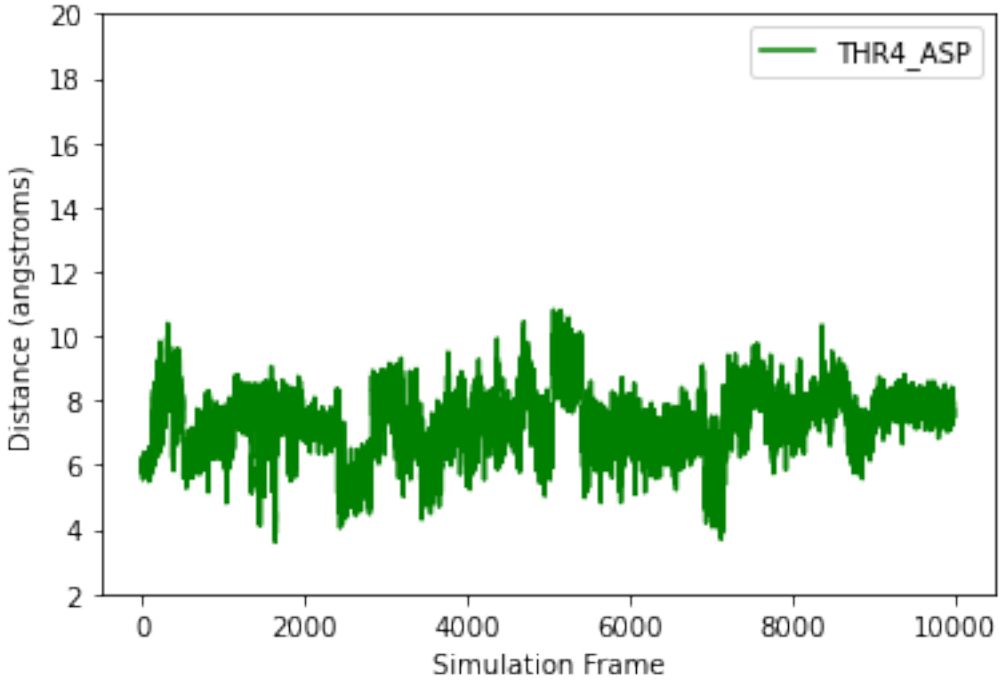
[57]:
```
Text(0, 0.5, 'Distance (angstroms)')
```

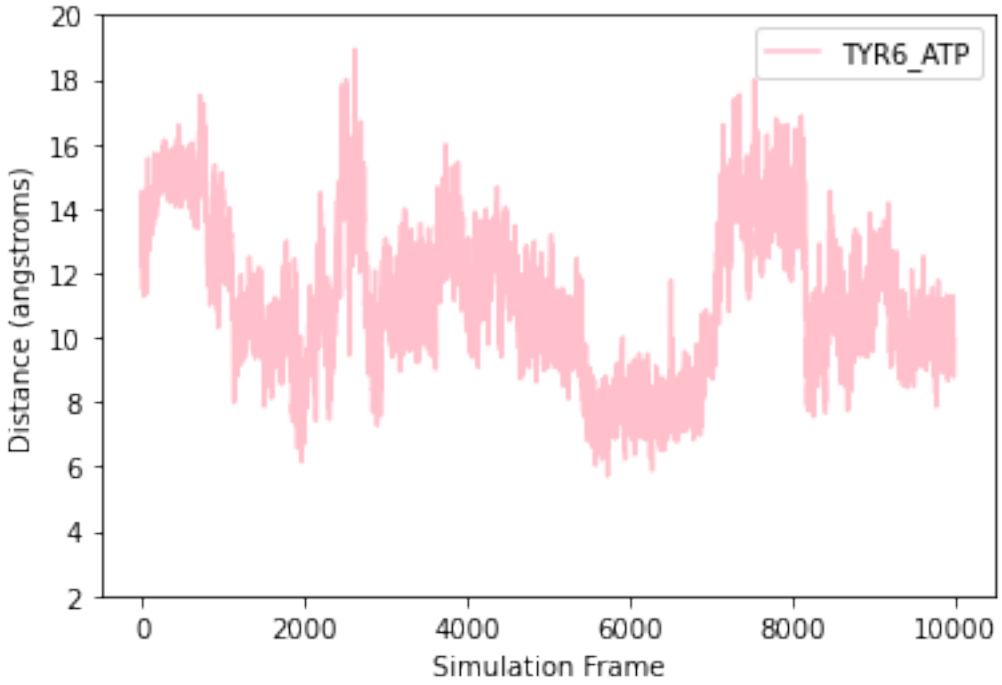


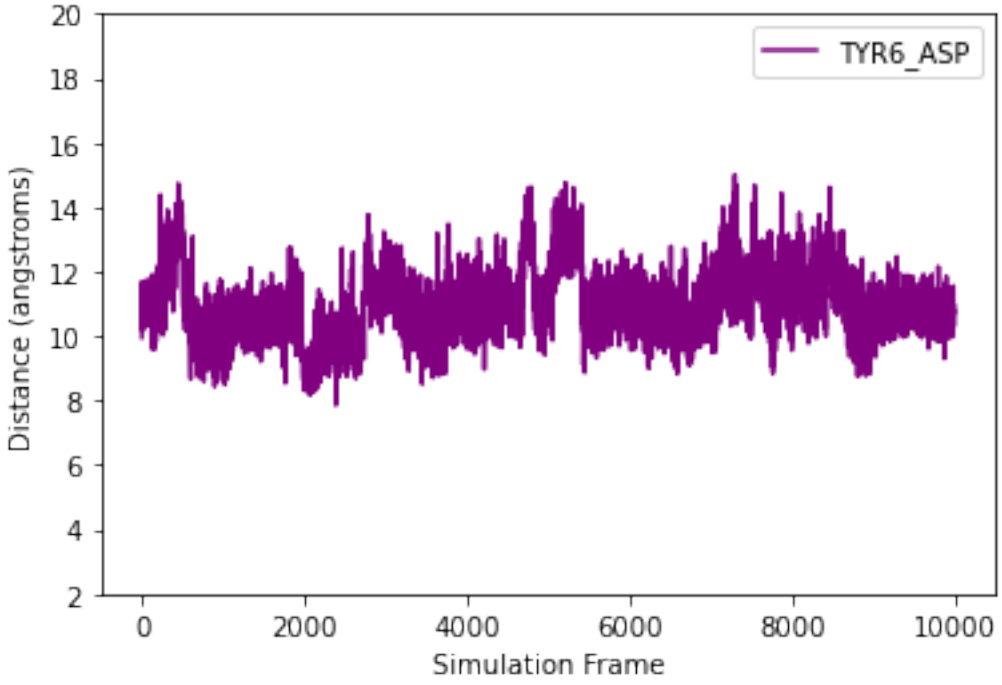
[58]:
```
# think about how you would change the for loop to make every data set plot on
↪its own graph.
```

[65]:
```
colors = ['','red', 'green', 'pink', 'purple']
for i in range(1,num_columns):
    plt.figure()
    plt.plot(data[:,i], label=F'{headers[i]}', color=colors[i])
    plt.ylim(2,20)
    plt.legend()
    plt.xlabel('Simulation Frame')
    plt.ylabel('Distance (angstroms)')
```


THR4_ATP
Distance (angstroms)
Simulation Frame
20
18
16
14
12
10
8
6
4
2
0
2000
4000
6000
8000
10000


THR4_ASP
Distance (angstroms)
Simulation Frame
20
18
16
14
12
10
8
6
4
2
0
2000
4000
6000
8000
10000


TYR6_ATP
Distance (angstroms)
Simulation Frame
20
18
16
14
12
10
8
6
4
2
0
2000
4000
6000
8000
10000


TYR6_ASP
Distance (angstroms)
Simulation Frame
20
18
16
14
12
10
8
6
4
2
0
2000
4000
6000
8000
10000

[ ]: