Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Two useful Python tools - dimpy and tablefile for data analysis applications

Dr. Dwaipayan Deb (dwaipayandeb@yahoo.co.in)
Ramanuj Gupta Degree College

Method Article

Keywords: Python, data-analysis, arrays, list, data-file

Posted Date: December 5th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-1004075/v2

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Two useful Python tools - *dimpy* and *tablefile* for data analysis applications

Dwaipayan Deb

Department of Physics, Ramanuj Gupta Degree College, Iskcon Mandir Road, Ambikapatty, Silchar -788004, Assam, India

Center of Advanced Studies and Innovation Lab, 18/27 Kali Mohan Road, Tarapur, Silchar-788003, Assam, India

Abstract

Python's list array is more powerful than arrays in other languages like C, C++, Fortran, or Java. However, in some cases it becomes tedious and complicated to construct a multidimensional 'list' type array in Python. A Python tool namely '*dimpy*' is discussed in this paper which can easily generate any multidimensional 'list' type array in python. Another Python package called *tablefile* for reading and analysing column-wise data from a data-file is also discussed. How these two tools may be useful and reduce steps of programming is shown by using some mathematics and physics related problems.

Keywords: Python, data-analysis, arrays, list, data-file

1. Introduction

Python has become popular among scientists for its simplicity and flexibility to enhance functionality by adding open-source packages in the program. It is new as compared to other languages and still evolving. However, there are various tools contributed by Python developers in the PyPi repository that can be added to a python program to enhance its performance. There are certain applications, where user experience may be improved by introducing new classes or functions within the program. For example, N-dimensional arrays with variable parameters are frequently required in various applications. An array in Python is in general defined by a 'list'. Other specific array types include 'tuple', 'set' and 'dict'. Although, 'list' is more powerful array type than its similar counterpart in other

languages due to its flexibility to include variable of any type, it becomes a bit complicated to predefine a 'list' type array of larger dimension in Python without assigning an initial value to the elements. Those who have proficiency in languages like Fortran, C, C++, Java, VB.NET, etc. may know how to create arrays of any number of dimensions of a given variable type (i.e. either float, int, or str) in a more or less similar manner. If the number of dimensions or elements is very large it may be problematic to give values to individual elements while declaring a list variable in Python. However, one may use the Numpy package (Harris et al. 2020, Oliphant et al. 2007, Cai et al. 2005), Sympy package (Meurer et al. 2017), Awkward arrays in Python (Pivarski et al. 2020), or *array* class of Python itself, which can create arrays similar to other languages, but none of those give the output arrays in the form of inbuilt 'list' class of Python, whereas 'list' type arrays may be advantageous in many cases (elaborated in next section). To achieve this goal and to make the 'list' type arrays of multiple dimensions in a way similar to arrays in other languages, the Python tool *dimpy* has been developed. Its installation, use, and application with examples have been discussed in this article.

Apart from the above, another useful tool has been discussed in this paper that may be advantageous to those who work with tabulated data in a text file. It is often required to read data from a data-file in which data are presented in a tabulated format consisting of a set of columns. The columns are usually separated by a delimiter which may be a comma, tab, blank space, or any other character. The users of the FORTRAN language can very easily read such data in a very simple way because it has been designed to do so. Although reading a file is simpler in Python as compared to other languages, but there are still some issues that must be addressed for ease of application and to make it popular among scientists and researchers. In this aim, the Python tool *tablefile* was developed which reads data from a data-file in a more convenient manner. In addition to that, it performs some elementary analytical tasks such as - averaging, summation, standard deviation, the maximum and minimum value from the data columns to assist data analysts in their work.

This paper elaborates the advantage, working, and the use of the above two open-source Python tools with some applications to show how these tools in Python code may be advantageous to solve problems in science.

2. Scope of improvement

(i) *Arrays in Python*: The ways to create a real-valued array variable in different languages are summarized in Table-1. Other types (i.e. integer or string) of arrays can be created similarly.

Table-1

Language	Command/Code
Fortran95	REAL A(4,6,9) ! Real 3D array 0f 4×6×9 elements A(1,3,7)=45.6

C/C++	int main()
	{
	float A [8][7]; // Real 2D array 0f 8×7 elements
	A[2][3]=45.6;
	return 0;
	}

Java	public class Main
	{
	public static void main(String[] args) {
	double a[][]=new double[4][5]; // Real 2D array 0f 4×5 elements
	a[1][3]=45.6;
	}
	}

VB.NET	Dim C(10, 20,12) As Single 'Real 3D array 0f 10×20×7 elements C(1,19)=45.6

In Python, there are several ways to generate an array. Most of the methods need that each of the elements must be given a value by the programmer when it is created. The closest approach in Python which is analogous to the other languages as shown in Table-1 may be the empty() function of the *numpy* package which is implemented as shown below

```
>>> import numpy as np
>>> A=np.empty([3,4,6], dtype=float)  # Creates a 3D array of 3×4×6 elements
>>> type(A)
<class 'numpy.ndarray'>
```

Here ‘dtype’ maybe float, int, or str depending on the requirement. It can be seen that the type of array A is not ‘list’ but ‘numpy.ndarray’. Therefore, the operations that are generally acceptable in a list variable of Python are not applicable in this case. For example, if we have two list variables x and y, one consisting of 3 elements and the other of 5 elements, the operation x+y gives a new list consisting of 3+5=8 elements which include all the members of x and y. This operation cannot be conducted on 'numpy.ndarray' variables. In this case, the summation operator (+) is applicable between only arrays of similar dimensions, and it performs arithmetic addition between the numbers corresponding to similar index values. If ‘dtype’ is str, then this operation is not valid. Similarly, if during runtime of the program it is required to add an element to the array, it will be not possible in ‘numpy.ndarray’ type variable, but can be done with the append() function if the array type is ‘list’. Another advantage of list arrays over 'numpy.ndarray' is that they can include elements of mixed data types. That is, an array consisting of some string elements, some integers, and some float variables is allowed in a list class variable, which is not the case in ‘numpy.ndarray’.

While 'numpy.ndarray' may be advantageous in many cases, list type arrays may be in many other applications. However, list type variables must be given some initial values to the individual elements when they are created. If the array is of many dimensions, it becomes a tiresome task to give each initial input individually and to construct many complicated nested lists.

Hence, to create a list array similar to most of the programming languages and numpy.empty() function in Python, the *dimpy* package was developed. The use and working are described in the subsequent section.

(ii) *Reading data from a file*: A basic FORTRAN code to read data from a data-file maybe

```
C     PROGRAM FOR READING DATA FROM FILE ‘FILE.TXT’
```

```
      CHARACTER*3 C2
      OPEN(1, FILE='C:/.../FILE.TXT')
 10   READ(1,*,END=20) C1,C2,C3
      PRINT*, C1,C2,C3
      GOTO 10
 20   CONTINUE
      END
```

The above code reads the data fields from some file 'FILE.TXT' in which 1st and 3rd columns are read as float whereas 2nd column is read as a string of maximum length 3. This code is simple, but leads to runtime error if the data format in the file do not match with what is being read (like that is shown in Fig.1).

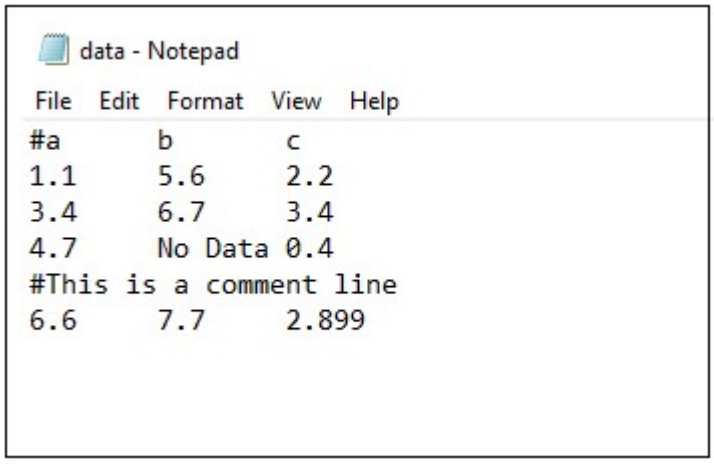

Fig1. Screenshot of the file 'data.txt' shown in Notepad. The first line is a header, and below this values are separated by tabs.

Fig.1 shows an input data-file where the fields are separated by \t (i.e. tab). In between numeric data, there are strings too. In Python, it's easy to read such data-files as a list. A Python code to read data from this file in Python maybe

```
>>> f1=open("c:/..../data.txt", "+r")
>>> dat=f1.readlines()
>>> print(dat)
['3.4\t6.7\t3.4\n', '4.7\tNo Data\t0.4\n', '#This is a comment line\n', '6.6\t7.7
\t2.899']
```

It may be noticed that 'dat' is a 'list' in which each line is a member in the form of a string. Data analysts would like to split the fields and store them as float values and in a more sequential format so that lines and column values can be easily

accessed. That would require writing a few lines of code more in Python to finally get the result. To simplify this task, *tablefile* package was developed which has been described in the next section.

3. Installation and Use

(i) The *dimpy* package: To install, in cmd or terminal use

```
$ python3 -m pip install dimpy
```

An example of a 3-dimensional list array consisting of 2×3×2 elements is shown below

```
>>> from dimpy import *
>>> A=dim(2,3,2)
>>> type(A)
<class 'list'>
>>> print(A)
[[[0, 0], [0, 0], [0, 0]], [[0, 0], [0, 0], [0, 0]]]
```

The default value assigned to the elements is 0 which can be changed to anything (whether an integer, float or string) by the use of function dfv().

```
>>> dfv(A,2.8)
>>> print(A)
[[[2.8, 2.8], [2.8, 2.8], [2.8, 2.8]], [[2.8, 2.8], [2.8, 2.8], [2.8, 2.8]]]
```

In principle, we can construct an equivalent list without the use of any package as shown below

```
>>> B= [[[0]*2]*3]*2
>>> print(B)
[[[0, 0], [0, 0], [0, 0]], [[0, 0], [0, 0], [0, 0]]]
```

But there is a catch. If we assign a value to a particular element, then it's multiplied in each sub-element

```
>>> B[1][2][0]=5
>>> print(B)
[[[5, 0], [5, 0], [5, 0]], [[5, 0], [5, 0], [5, 0]]]
```

But list array created with *dimpy* does not suffer from this limitation and we can give individual inputs to individual elements

```
>>> A[0][2][1]= 123
>>> A[1][2][0]=”STRING”
>>> print(A)
 [[[2.8, 2.8], [2.8, 2.8], [2.8, 123]], [[2.8, 2.8], [2.8, 2.8], ['STRING', 2.8]]
]
```

It should be noted that the array type of A is the ‘list’ and therefore it allows mixed variable types within it.

If one wishes to convert the type of A from the ‘list’ to a new array C of type ‘numpy.ndarray’, one may use C=npary(A) function, provided the *numpy* package is already installed in the system.

```
>>> C=npary(A)
>>> type(C)
<class 'numpy.ndarray'>
>>> print(C)
[[['2.8' '2.8']
  ['2.8' '2.8']
  ['2.8' '123']]

 [['2.8' '2.8']
  ['2.8' '2.8']
  ['STRING' '2.8']]]
```

The point to note here is that as one of the elements in A is a string, all the elements will be converted to string type in C. Further, as the intrinsic data type of C is now ‘str’, one cannot now assign a ‘float’ or ‘int’ value to a particular element in C as we have done in case of A. In this case, the float or int value will be converted to ‘str’ and then stored in C

```
>>> C[1,0,1]="HELLO"
>>> print(C)
[[['2.8' '2.8']
  ['2.8' '2.8']
  ['2.8' '123']]

 [['2.8' 'HELLO']
  ['2.8' '2.8']
```

```
  ['STRING' '2.8']]]
>>> C[1,0,0]=345
>>> print(C)
[[['2.8' '2.8']
  ['2.8' '2.8']
  ['2.8' '123']]

 [['345' 'HELLO']
  ['2.8' '2.8']
  ['STRING' '2.8']]]
```

But remember that, the converse is not true and will through ValueError if tried. That means if C is 'int' or 'float' type then one cannot assign an 'str' data to a given element, because 'str' to 'int' or 'float' conversion is not allowed.

(ii) The *tablefile* package: To install from cmd or terminal, enter

```
$ python3 -m pip install tablefile
```

An example code to read column-wise data from 'data.txt' as shown in Fig.1 maybe

```
>>> from tablefile import *
>>> f1=file("C:/.../data.txt","\t") # Last argument is the column separator in th
e file
>>> lines=f1.read() # Reads the lines
>>> print(lines)
[[1.1, 5.6, 2.2], [3.4, 6.7, 3.4], [4.7, 'No Data', 0.4], [6.6, 7.7, 2.899]]
```

If the data-file separator is one or more blank-space, then one may not specify it at the 2nd argument of file() function. That is, in this case, we may write

```
>>> f1=file("C:/.../data.txt") # If column separator is a blank-space
```

It can be seen that the output of 'lines' is a list that is already divided into lines and columns. The fields that cannot be converted to float remain string and lines starting with '#' have been considered as a comment line and therefore skipped.

While reading data from a table, sometimes we are interested to get all the column values in a list element rather than lines. This can be done by

```
>>> cols=f1.read("c/l") # Here "c/l"says that output should be column/line forma,
i.e. first index of 'cols' will indicate column whereas second index row.
```

```
>>> print(cols)
[[1.1, 3.4, 4.7, 6.6], [5.6, 6.7, 'No Data', 7.7], [2.2, 3.4, 0.4, 2.899]]
>>> print(cols[0][1]) # First column second row element
3.4
```

In addition to reading the data-file in sequential format, *tablefile* also gives some additional features as shown below

```
>>> # Column-wise operations
>>> average=f1.read("av")
>>> sum=f1.read("sm")
>>> std=f1.read("sd") # Standard deviation for a population (ie large N)
>>> stds=f1.read("sds") # Standard deviation for a sample (ie small N)
>>> max=f1.read("mx")
>>> min=f1.read("mn")
>>>print("Average=",average,"Sum=",sum,"Sigma_population=",std,"Sigma_sample=",st
ds, "Maximum=",max,"Minimum=",min)

Average= [3.95, 6.67, 2.22475] Sum= [15.78, 20.0, 8.899] Sigma_population= [2.00,
0.858, 1.136] Sigma_sample= [2.310, 1.050, 1.312]
Maximum= [6.6, 7.7, 3.4] Minimum= [1.1, 5.6, 0.4]
```

Here in each case above, the first element corresponds to the first column, the second element to the second column, and so on. All the strings in the columns that cannot be converted to 'float' will be neglected during the calculation.

If we want to do the statistical calculations above on a particular list – whether it is a line or column or any other list containing some numbers, we can do that using *tablefile* functions as follows

```
>>> # Operations applicable to any list
>>> List1=[2,6,8,"No Data",10,"Data Error",45] # Any list
>>> Avg=av(List1) # Average
>>> Sum=sm(List1) # Sum
>>> Std=sd(List1)  # Standard Deviation for population
>>> Stds=sds(List1)  # Standard Deviation for a sample
>>> Max=mx(List1) # Maximum
>>> Min=mn(List1) # Minimum
>>>print("Average=",Avg,"Sum=",Sum,"Sigma_population=",Std,"Sigma_sample=",Stds,"
Maximum=",Max,"Minimum=",Min)

Average=14.2 Sum=71 Sigma_population= 15.62562 Sigma_sample= 17.46997 Maximum=
45 Minimum= 2
```

Note that in the all above cases the string elements in List1 were neglected during calculation and they did not through any error. Python's inbuilt functions sum(), max(), min() and NumPy's functions such as numpy.std() and numpy.average() can do the same task, but all these will through errors due to the presence of string elements in the list.

To convert the values of a list by means of an equation we may use tablefile's convert function as shown below

```
>>> List1=[2,6,8,"No_Data",10,"Data_Error",45]
>>> List2=convert(List1,"log(x**2+6)") # Converting values by an equation
>>> print(List2)

[2.302585092994046, 3.7376696182833684, 4.248495242049359, 'No_Data',
4.663439094112067, 'Data_Error', 7.616283561580385]
```

4. Some Physical Examples

In this section, some physical examples are shown which demonstrates how we can use the above packages to solve various problems of mathematics and physics

(i) ***Hessian Matrix Problem:*** Hessian matrix is an $n \times n$ matrix defined by

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Where f is a function of n independent variables given by $f \rightarrow f(x_1, x_2, x_3, \ldots . x_n)$. The above matrix is a collection of all possible partial second derivatives of the function f and it helps find the critical points (i.e. local maxima, minima, or saddle point) of the function f in an n-dimensional space. When H is evaluated at a given point, the determinant, eigenvalues, and eigenvectors of H give some important information about the nature of f such as Gaussian curvature, principal curvatures,

and principal directions, and therefore use of Hessian matrix in physics is diverse (e.g. see Lindh et al. 1995, Zhu et al. 2018 and Dias et al. 2016).

In principle, a matrix with variable elements can be defined with the help of the Array function of the Sympy package, but it is not practically useful in cases like this one. Let us see why – suppose we have a function of five variables, so we would require a Hessian matrix of $5 \times 5 = 25$ elements.

Our logical approach would be to first define an array of $5 \times 5 = 25$ elements with dummy values and then assign each of them corresponding double derivative in two nested loops as shown below

```
from sympy import *
from math import pi
n=5  # The number of variables involved
x1,x2,x3,x4,x5=symbols('x1,x2,x3,x4,x5') # Defines the symbolic variables
x=[x1,x2,x3,x4,x5]
f=x1**2+x1*x2+cos(x1*x2)+x3+x4*x5**0.5 # Defines the function

H=Array([[x1,x2,x3,x4,x5],[ x1,x2,x3,x4,x5],[ x1,x2,x3,x4,x5],[ x1,x2,x3,x4,x5],[
x1,x2,x3,x4,x5]]) # Defines the dummy array for Hessian matrix

for i in range(n):
    for j in range(n):
        H[i][j]=diff(diff(f,x[j]),x[i])
print(H)
```

Since we are dealing with symbolic mathematics, dummy values are given as variables in H and not numerical values to avoid errors while replacing them with derivatives within the nested loop*. When we run this program we get an error at the output like this –

```
Traceback (most recent call last):
  File "d:\My Folder\PythonWorkshop\Hessian.py", line 13, in <module>
    H[i][j]=diff(diff(f,x[j]),x[i])
  File "C:\Users\Dwaipayan Deb\AppData\Local\Programs\Python\Python39\lib\site-
packages\sympy\tensor\array\dense_ndim_array.py", line 147,
in __setitem__
    raise TypeError('immutable N-dim array')
TypeError: immutable N-dim array
```

A straight forward approach that may be considered to construct the 5×5 Hessian matrix using Array function would be to write the array like this –

```
H=Array([[diff(diff(f,x1),x1),diff(diff(f,x1),x2),diff(diff(f,x1),x3),diff(diff(f
,x1),x4),diff(diff(f,x1),x5)],[diff(diff(f,x1),x1),diff(diff(f,x1),x2),diff(diff(
f,x1),x3),diff(diff(f,x1),x4),diff(diff(f,x1),x5)],[diff(diff(f,x1),x1),diff(diff
(f,x1),x2),diff(diff(f,x1),x3),diff(diff(f,x1),x4),diff(diff(f,x1),x5)],[diff(dif
f(f,x1),x1),diff(diff(f,x1),x2),diff(diff(f,x1),x3),diff(diff(f,x1),x4),diff(diff
(f,x1),x5)],[diff(diff(f,x1),x1),diff(diff(f,x1),x2),diff(diff(f,x1),x3),diff(dif
f(f,x1),x4),diff(diff(f,x1),x5)]])
```

But, one would hardly prefer to follow this approach. Instead of this, the problem can be easily handled by the use of *dimpy* as follows –

```
from sympy import *
from math import pi
from dimpy import *
x1,x2,x3,x4,x5=symbols('x1,x2,x3,x4,x5') # Defines the symbolic variables
x=[x1,x2,x3,x4,x5]
f=x1**2+x1*x2+cos(x1*x2)+x3+x4*x5**0.5 # Defines the function
n=5 # Number of rows or columns
H=dim(n,n)
for i in range(n):
    for j in range(n):
        H[i][j]=diff(diff(f,x[j]),x[i])
print(H)
```

When we run the above program, following output is obtained

```
[[-x2**2*cos(x1*x2) + 2, -x1*x2*cos(x1*x2) - sin(x1*x2) + 1, 0, 0, 0], [-
x1*x2*cos(x1*x2) - sin(x1*x2) + 1, -
x1**2*cos(x1*x2), 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0.5*x5**(-
0.5)], [0, 0, 0, 0.5*x5**(-0.5), -0.25*x4*x5**(-1.5)]]
```

* This could still be questioned by saying – ‘what if one of the double partial derivatives is a constant?’

As we have now constructed our Hessian matrix, we can easily findout determinant, eigenvalues and eigenvectors at any point under consideration by the use of inbuilt Matrix operations of Sympy

```
H=Matrix(H).subs([(x1,1),(x2,pi),(x3,1),(x4,0),(x5,3)]) # Converts H into a Sympy
 Matrix and then evaluates H at point (x1,x2,x3,x4,x5)=(1,3.1416,1,0,3)
```

```
print(det(H)) # Determinant of H
print(H.eigenvals()) # Eigenvalues of H
print(H.eigenvects()) # Eigenvectors of H
```

(ii) Test for Symplectic Condition (Canonical Transformation): In classical mechanics, we use the 'symplectic condition' to test whether a transformation of coordinates in *phase space* is *canonical* or not Goldstein 1998.

The mathematical approach to test canonicality of an n particle system in *phase space* demands the following condition be satisfied:

$M.J.\tilde{M} = J \vee equivalently \quad \tilde{M}.J.M = J$ ---------(1)

Where

$$M_{ij} = \frac{\partial \zeta_i}{\partial \eta_j}, \qquad i,j=1,2,3,.....,2n \zeta_i = \begin{pmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_n \\ P_1 \\ P_2 \\ \vdots \\ P_n \end{pmatrix}_{2n}, \eta_i = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \\ p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}_{2n}$$

Or,
$$M = \begin{pmatrix} \frac{\partial Q_1}{\partial q_1} & \frac{\partial Q_1}{\partial q_2} & \cdots & \frac{\partial Q_1}{\partial p_n} \\ \frac{\partial Q_2}{\partial q_1} & \frac{\partial Q_2}{\partial q_2} & \cdots & \frac{\partial Q_2}{\partial p_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial P_n}{\partial q_1} & \frac{\partial P_n}{\partial q_2} & \cdots & \frac{\partial P_n}{\partial p_n} \end{pmatrix}_{2n \times 2n}$$

$\tilde{M}$ is the transpose matrix of M, and, J is an $2n \times 2n$ anti-symmetric matrix given by $J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix}$ where 0 is a $n \times n$ *null matrix* or *zero matrix* (i.e. a matrix whose all the elements are zero) and I is an $n \times n$ *unit matrix*

For a two or three-particle system though the test is not difficult to carry out manually, but, if the number of particles in the system is large we must do it programmatically. Let us solve a problem by using the Python program for a two-particle system which can be extended to any number of particles system with little modification.

Problem: Prove that the following transformation is canonical

$Q_1=q_1$ $P_1= p_1 – 2p_2$

$Q_2=q_2$ $P_2= –2q_1 – q_2$

(This is the first part of Exercise No. 8 of Chapter 9 in Goldstein 1998)

From the programming point of view, this problem is similar to the Hessian Matrix problem discussed above and the corresponding Python program is given below

```
from sympy import *
from dimpy import *

n=2 # Number of Particles
q1,q2,p1,p2=symbols('q1,q2,p1,p2') # Symbolic variables are declared
Q1=q1
Q2=p2
P1=p1-2*p2
P2=-2*q1-q2
X=[q1,q2,p1,p2]
Y=[Q1,Q2,P1,P2]

M=dim(2*n,2*n)
J=dim(2*n,2*n)
for i in range(2*n):
    for j in range(2*n):
        M[i][j]=diff(Y[i],X[j]) # Jacobian matrix M
        if i<n and i==(j-n):   # Antisymmetric matrix J
            J[i][j]=1
        elif i>=n and j==(i-n):
            J[i][j]=-1
M=Matrix(M) # Array M is converted into a sympy matrix
J=Matrix(J) # Array J is converted into a sympy matrix
if M.T*J*M==J: # M.T is the tanspose of M
    print("The symplectic condition is satisfied")
else:
    print("The test was not successful")
```

When we run this program, we obtain the message "`The symplectic condition is satisfied`" at the output. Although the above program deals with a two-particle system, our program can be easily used for any number of particles system by introducing necessary parameters and equations in the first few lines.

The `Array()` function can not be used in the above program due to the same reason as in the Hessian matrix case, and therefore *dimpy* is the only option.

(iii) Analysis of Astronomical data: Hipparcos catalog (Perryman et al. 1997) is an astronomical database in the form of an ASCII table (i.e. can be opened by any text editor like Notepad). It contains various astronomical data for 118218 stars in each line tabulated in 77 columns called Fields (excluding the Field 0). Deb and Chakrabory 2014 used a FORTRAN program to read 13 among these 77 columns in their work to identify stars with incorrect spectral classification. As the first step to investigate further on those wrongly classified stars (Table-2, Deb and Chakraborty 2014), we might be interested to list the following information- (1) star identifier number (2) location of the stars (equatorial coordinates), (3) their distance from our sun, (4) the color temperature, (5) uncertainty in color temperature data and (6) the absolute magnitude M_v (which represents luminosity of the star relative to our sun).

The following fields are to be read from the Hipparcos catalog:

H1: Hipparcos identifier number

H8 & H9: Equatorial coordinates of the stars (R.A. and DEC.)

H11: Trigonometric parallax in milli-arc-sec (π_H)

H37: Jhonson (B–V) index

H38: Standard Error in (B–V)

(See https://heasarc.gsfc.nasa.gov/W3Browse/all/hipparcos.html for further information)

M_v can be read from the 4th column of Table-2 of Deb and Chakraborty 2014.

We can calculate the distance *d* of the stars in *parsec* (1 parsec=206265AU) using the relation

$$d = \frac{10^3}{\pi_H}$$

The color temperature T of the star can be calculated from Jhonson (B–V) index (Karttunen et al. 2007) with the help of the following equation

$$log_{10}(T) = \frac{14.551 - (B - V)}{3.684}$$

To do the task programmatically we first copy the data in Table-2 of Deb and Chakraborty 2014 to a text file and name it 'data_deb.txt'. The full Hipparcos catalog is available to download from http://cdsarc.u-strasbg.fr/ftp/cats/I/239/ in the form of an ASCII table with a file name 'hip_main.dat'. We will use Python code to extract/compare data from the above two files and finally construct the required list.

The task to do: First of all, we need to read columns 1, 4, and 8 from 'data_deb.txt'. Stars with presumably incorrect spectral classification are tagged 'Unchanged' in the 8th column. Corresponding data of 1st column is the Hipparcos identifier number. We can now search for this number with our program in the Hipparcos catalog file 'hip_main.dat' in its H1 field and then read corresponding other fields to access the necessary data. We can then perform necessary calculations and finally print the data at the output.

Without adding any package to our program, the code will look like this –

```
f1=open("D:/data_deb.txt",'r+')
data_deb=f1.readlines()
f2=open("D:/hip_main.dat", "r+")
data_hip=f2.readlines()
hiplist=[]
RA_DEC=[]
d_parsec=[]
log10T=[]
log10T_Error=[]
Mv=[]
for data in data_deb:
    data_list=data.split("\t") # Data-file separator is <Tab> ('\t' in Python) in
'data_deb.txt'
    if data_list[7]=="Unchanged" or data_list[7]=="Unchanged\n":  # There are
some lines with <Enter> ('\n' in Python) after 'Unchanged'
        hiplist.append(int(data_list[0])) # Records Hippracos identifier
        for data2 in data_hip:
            data2_list=data2.split("|") # Data-file separator is '|' in
'hip_main.dat'
```

```
            if int(data_list[0])==int(data2_list[1]):
                # float conversion is required because element types are str in
all cases
                # but there may be some data with NO entry (i.e. blank spaces)
                # these will lead to error while convertig to float values
                # therefore try-except statement is used
                try:
                    RA_DEC.append([float(data2_list[8]),float(data2_list[9])]) #
recording co-ordinates
                except:
                    RA_DEC.append("No_data")
                try:
                    d_parsec.append((10**3)/float(data2_list[11])) # recording
distance in parsec
                except:
                    d_parsec.append("No_data")
                try:
                    log10T.append((14.551-float(data2_list[37]))/3.684) # record-
ing (B-V) index
                except:
                    log10T.append("No_data")
                try:
                    log10T_Error.append(float(data2_list[38])/3.684) # recording
(B-V) Error
                except:
                    log10T_Error.append("No_data")
                try:
                    Mv.append(float(data_list[3])) # recording 4th column of
'data_deb.txt' (index=0 for 1st column)
                except:
                    Mv.append("No_data")
for i in range(len(hiplist)):
      print(hiplist[i],'\t',RA_DEC[i],'\t',d_parsec[i],'\t',log10T[i],'\t',log10
T_Error[i],'\t',Mv[i])
```

On the other hand, if we import *tablefile* package to the code, the same task can be performed with the following code-

```
from tablefile import *
f1=file("D:/data_deb.txt",'\t')
lines_deb=f1.read() # reads 'data_deb.txt' in default line/column format
f2=file("D:/hip_main.dat", "|")
cols_hip=f2.read("c/l") # reads 'hip_main.dat' in column/line format
hiplist=[]
```

```
RA_DEC=[]
pi_H=[]
B_V=[]
B_V_Error=[]
Mv=[]
for line in lines_deb:
    index=0
    if (line[7]=="Unchanged\n" or line[7]=="Unchanged"): # There are some lines
with <Enter> ('\n' in Python) after 'Unchanged'
        hiplist.append(int(line[0])) # records Hippracos identifier
        Mv.append(line[3]) # records Mv values(4th column of 'data_deb.txt')
        for column in cols_hip[1]:
            if line[0]==column:
                RA_DEC.append([cols_hip[8][index],cols_hip[9][index]]) # record-
ing co-ordinates
                pi_H.append(cols_hip[11][index]) # recording parallax angle
                B_V.append(cols_hip[37][index]) # recording (B-V)
                B_V_Error.append(cols_hip[38][index]) # recording error in(B-V)
            index+=1
d_parsec=convert(pi_H,'10**3/x') # converting parallax to distance in parsec
log10T=convert(B_V,'(14.551-x)/3.684') # converting (B-V) to log scale color tem-
perature T
log10T_Error=convert(B_V_Error,'x/3.684') # converting (B-V) errors to error in
logT scale
for i in range(len(hiplist)):
      print(hiplist[i],'\t',RA_DEC[i],'\t',d_parsec[i],'\t',log10T[i],'\t',log10
T_Error[i],'\t',Mv[i])
```

If we compare the above two codes, it can be seen that with the use of *tablefile* package in our program we can make it simpler as well as concise. Both the codes are properly commented on so that readers can easily understand the steps. (See the last paragraph for file availability information)

(iv) Analysis of Experimental Data: The temperature dependence of an ohmic conductor is given by

$$R_T = R_o[1 + \alpha(T - T_o)]$$

Where R_T is the resistance at temperature T, R_o is the resistance at a reference temperature T_o which is generally 0^oC, and α is the temperature coefficient of resistance for the material. The experimental determination of α needs temperature v/s resistance data for a wide range of temperatures. Fig.2 shows a computerized

experimental setup for the determination of α. The Arduino microcontroller automatically collects data for the temperature of the oil bath (which is at equilibrium temperature with the resistance), the current through the circuit, and the voltage across the resistance R with the help of respective sensors and sends it to the computer through a USB connection which can be logged to a text file utilizing a serial data read software like Terminal or Putty. Fig.3 shows the screenshot a of typical output data-set from the microcontroller which is programmed to send 50 observations at the interval of 20 milliseconds, then waits 5 minutes for the temperature change, and then repeats the process. Here it can be seen that some entries are due to serial data read errors and hence can not be converted to a floating-point number during calculation and therefore would lead to errors unless it is handled separately in the program. Complete removal of the lines containing errors may not be suggested because it will delete some correct data too. But if we use inbuilt functions of *tablefile* package then this limitation can be overcome without adding extra steps to the code. Two codes producing identical results – with and without importing *tablefile* are shown below. It can be seen that the use of *tablefile* makes the code simpler and shorter. The output is shown in Fig.4.

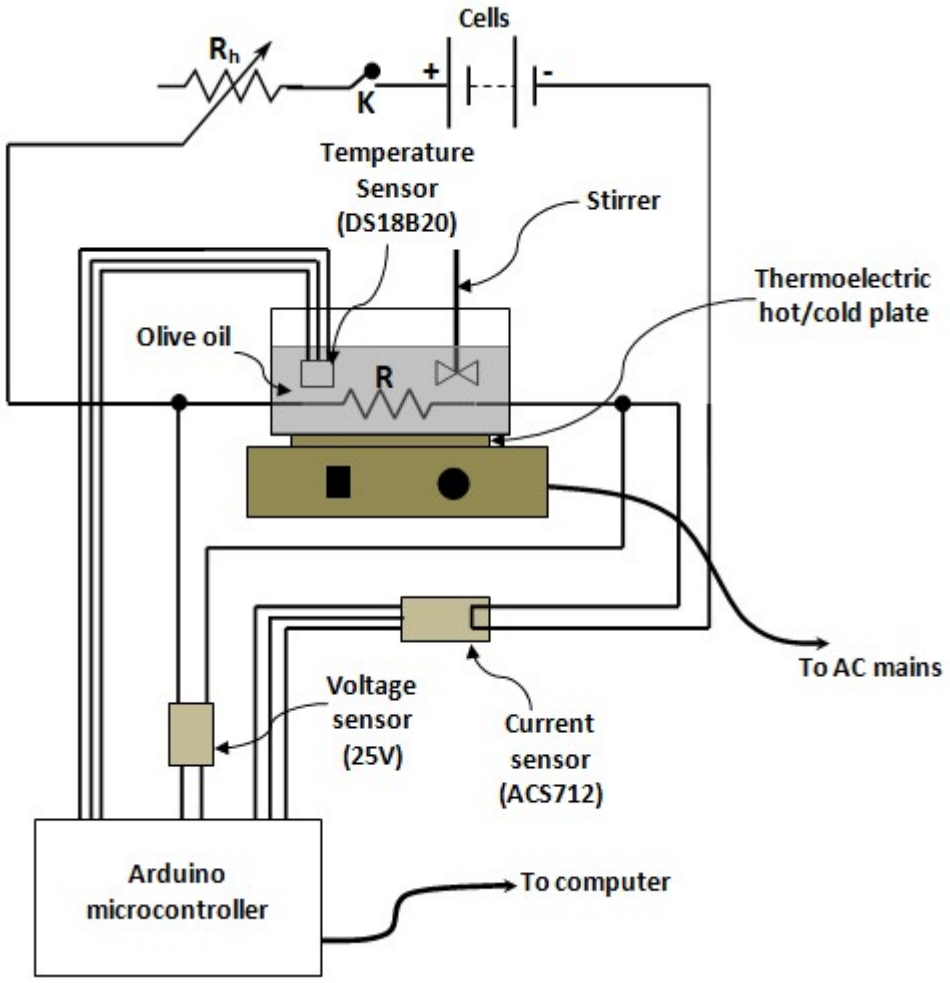

Fig 2. Microcontroller-based experimental setup for the determination of the temperature coefficient of resistance.

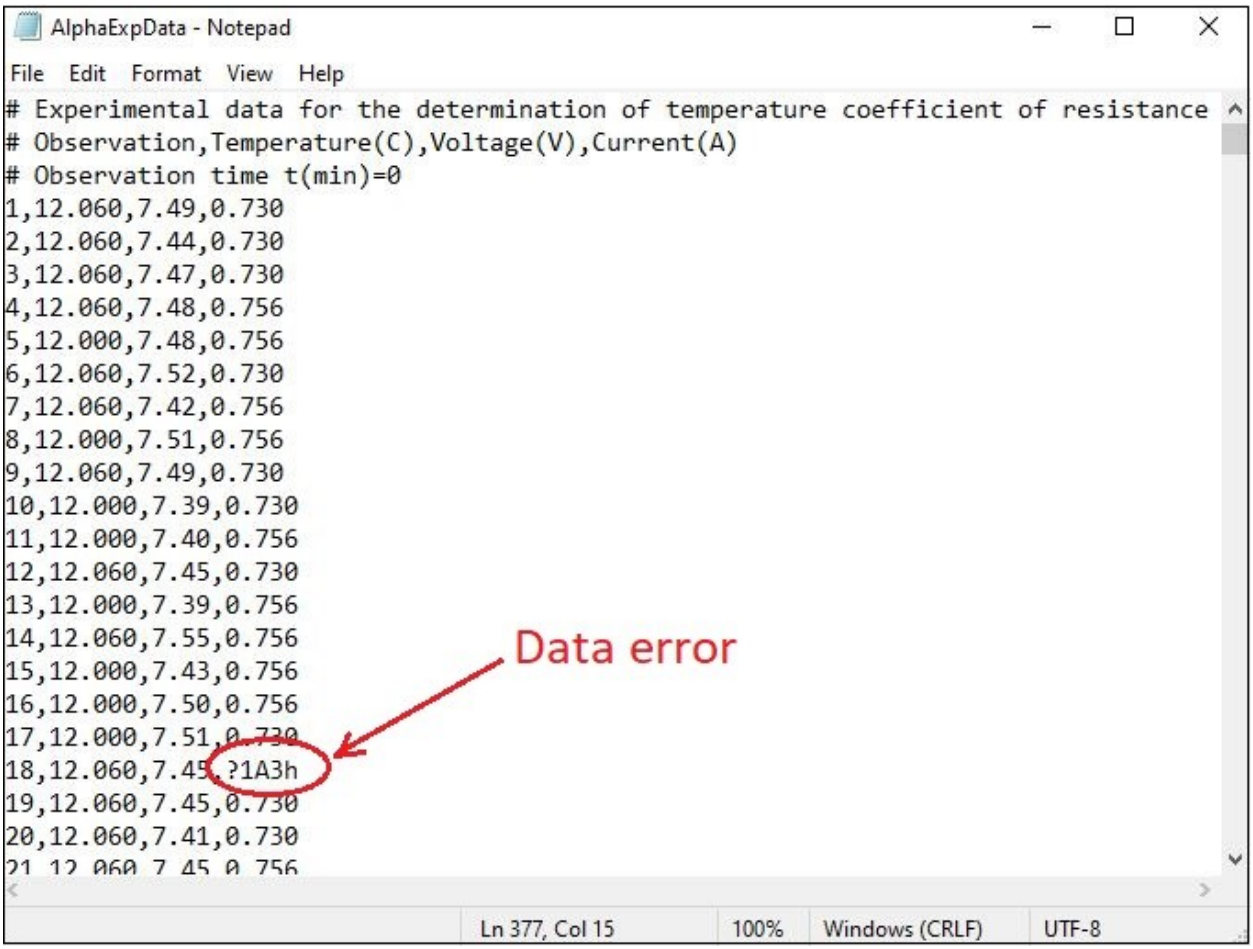

Fig.3 Screenshot of a typical output data-set from the temperature coefficient of resistance determination experiment. There are 50 observations at a given time. Data at different time are separated by the string '# Observation time t(min) = <time>'. It can be seen that a column may contain unknown strings due to a data read error. See the text for file availability.

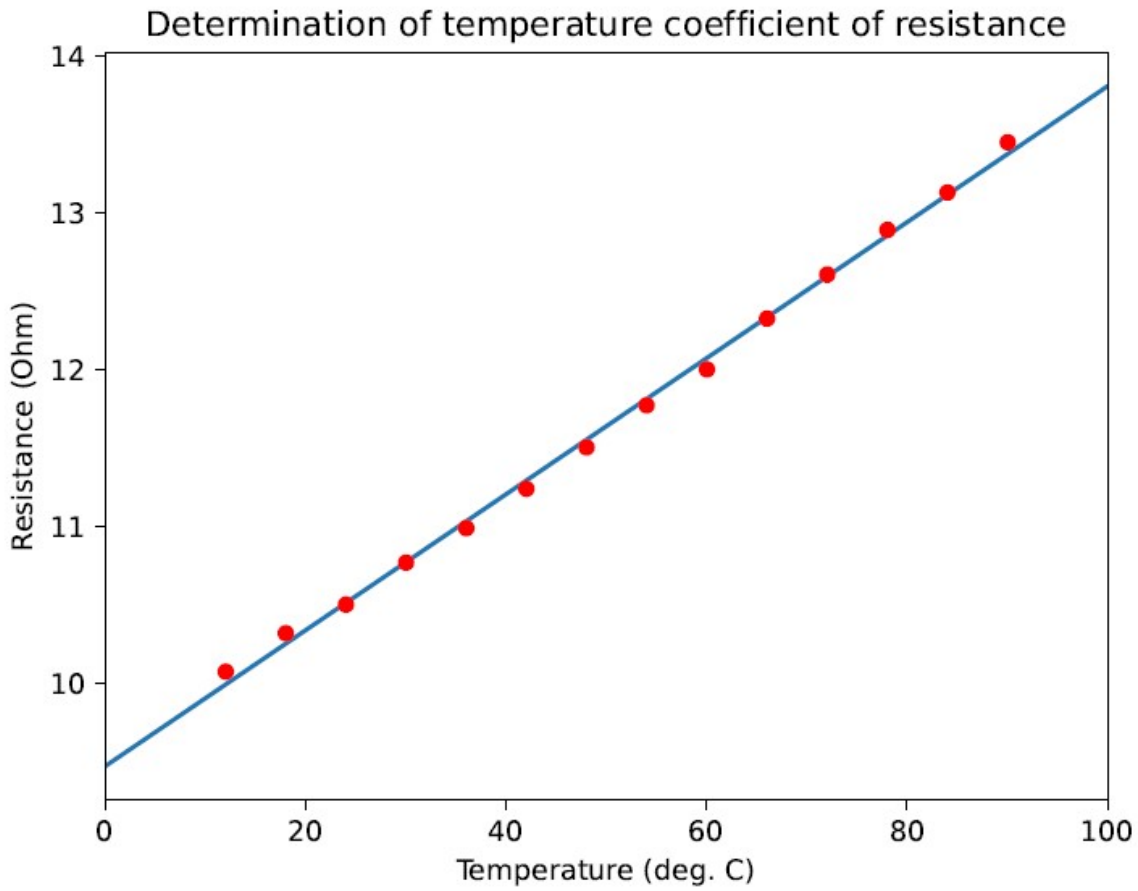

Fig.4 Output of python codes as described in the text produces this temperature v/s resistance plot with a fitted straight line. Error bars are shorter than the size of red circles and hence not visible in this picture. From the (slope/intercept) of the fitted line we get the value of α = 0.0045874/°C.

Code 1: Without *tablefile*

```
import matplotlib.pyplot as plt
from math import *
import numpy as np
from statistics import mean,stdev

f1=open("D:/Programs/AlphaExpData.txt",'r+')
lines=f1.readlines() # Reading lines as a list of strings (columnwise reading not
possible)
```

```
#----------Converting the list of lines to list of columns------------------
cols=[[],[],[],[]] # creating an empty 4 column list (there are 4 columns in our
data table)
i=0
for line in lines:
    if not line[0]=="#":
        line=line.split(',')
        for i in range(len(line)):
            try:
                cols[i].append(float(line[i]))
            except:
                cols[i].append("Error")
#-------------------------------------------------------------------------
N_col=4 # Number of columns in the data
N=50 # Number of observations at a time
j=1
Result=[] # A temporary list to hold T_av, T_err, V_av, V_err, I_av and I_err at
a given time
temperature=[] # List for holding final average temperature data
resistance=[] # List for holding final resistance data
temperature_err=[] # List for holding temperature standard error data
resistance_err=[] # List for holding resistance uncertainty data
for i in range(0,len(cols[j]),N):
    for j in range(1,N_col):
        col_data=cols[j][i:i+N]
        if 'Error' in col_data:
            col_data.remove('Error') # Removing the string 'Error' from the list
        Result.append(mean(col_data)) # Calculation of T_av (for j=1), V_av (for
j=2) or I_av (for j=3)
        Result.append(stdev(col_data)/sqrt(N)) # Standard errors T_err, V_err or
I_err
    data1=cols[2][i:i+N] # Column 3 (j=2) contains voltage data
    if 'Error' in data1:
        data1.remove('Error')
    V_av= mean(data1) # Average voltage
    V_err=stdev(data1)/sqrt(N) # Standard error of voltage
    data2=cols[3][i:i+N] # Column 4 (j=4) contains current data
    if 'Error' in cols[3][i:i+N]:
        data2.remove('Error')
    I_av= mean(data2) # Average current
    I_err=stdev(data2)/sqrt(N) # Standard error of current
    R=V_av/I_av  # Resistance
    Result.append(R)
    R_delta=R*((V_err/V_av)+(I_err/I_av))  # Uncertainty in R measurement
    Result.append(R_delta)
```

```
    temperature.append(Result[0])
    temperature_err.append(Result[1]) # Standard Error in T measurement
    resistance.append(Result[6])
    resistance_err.append(Result[7]) # Uncertainty in R measurement R_delta
    print(Result)
    Result=[]  # The list is made empty to hold data at new time
#print(temperature_err,resistance_err)
m,b=np.polyfit(temperature,resistance,1) # Fitting a line of slope m and inter-
cept b to the data
print("Alpha_fitted=",m/b)
plt.xlim(0,100) # Limits of x_axis range for plot
plt.title("Determination of temperature coefficient of resistance")
plt.xlabel("Temperature (deg. C)")
plt.ylabel("Resistance (Ohm)")
x=np.linspace(0,100) # defining range of x values
plt.plot(x,m*x+b) # plotting the fitted line
# Plotting with error-bars
plt.errorbar(temperature,resistance,xerr=temperature_err,yerr=resistance_err,
color="red",fmt='o',markersize=5)
plt.show()
```

Code 2: With the use of *tablefile*

```
from tablefile import *
import matplotlib.pyplot as plt
from math import *
import numpy as np

f1=file("D:/Programs/AlphaExpData.txt",',')
cols=f1.read("c/l") # Reading data column-wise

N_col=4 # Number of columns in the data
N=50 # Number of observations at a time
j=1
Result=[] # A temporary list to hold T_av, T_err, V_av, V_err, I_av and I_err at
a given time
temperature=[] # List for holding final average temperature data
resistance=[] # List for holding final resistance data
temperature_err=[] # List for holding temperature standard error data
resistance_err=[] # List for holding resistance uncertainty data
for i in range(0,len(cols[j]),N):
    for j in range(1,N_col):
        Result.append(av(cols[j][i:i+N])) # Calculation of T_av (for j=1), V_av
(for j=2) or I_av (for j=3)
```

```
        Result.append(sds(cols[j][i:i+N])/sqrt(N)) # Standard errors T_err, V_err
or I_err
    V_av=av(cols[2][i:i+N]) # Average of voltage data
    V_err=sds(cols[2][i:i+N])/sqrt(N) # Standard error of voltage data
    I_av=av(cols[3][i:i+N]) # Average of current data
    I_err=sds(cols[3][i:i+N])/sqrt(N) # Standard error of current data
    R=V_av/I_av # Calculation of resistance
    Result.append(R)
    R_delta=R*((V_err/V_av)+(I_err/I_av))  # Uncertainty in R measurement
    Result.append(R_delta)
    temperature.append(Result[0])
    temperature_err.append(Result[1]) # Standard Error in T measurement
    resistance.append(Result[6])
    resistance_err.append(Result[7]) # Uncertainty in R measurement R_delta
    print(Result)
    Result=[]
#print(temperature_err,resistance_err)
m,b=np.polyfit(temperature,resistance,1) # Fitting a line of slope m and inter-
cept b to the data
print("Alpha_fitted=",m/b)
plt.xlim(0,100) # Limits of x_axis range for plot
plt.title("Determination of temperature coefficient of resistance")
plt.xlabel("Temperature (deg. C)")
plt.ylabel("Resistance (Ohm)")
x=np.linspace(0,100) # defining range of x values
plt.plot(x,m*x+b) # plotting the fitted line
# Plotting with error-bars
plt.errorbar(temperature,resistance,xerr=temperature_err,yerr=resistance_err,
color="red",fmt='o',markersize=5)
plt.show()
```

Working and theory behind the codes: In the codes, we first read the four columns from the data-file and place them in a two-dimensional list array called ‘cols’ where the first index would indicate column number and the second would indicate the serial number of the data. In the data-file, the 1st column gives the serial number of observations, the 2nd column gives temperature data, the 3rd gives voltage data, and 4th that of current data, and the corresponding data can be accessed by setting the first index of ‘cols’ equal to 0,1,2 and 3 respectively. We divide each column into groups of 50 data sets and then take their average and sample standard deviations. This gives average temperature (T_{av}), voltage (V_{av}),

and current (I_{av}) at different observation times and their corresponding sample standard deviations T_{sds}, $V_{sds,}$ and I_{sds} respectively. Corresponding standard errors can be calculated from $T_{err}=T_{sds}/\sqrt{N}$, $V_{err}=V_{sds}/\sqrt{N}$, and $I_{err}=V_{sds}/\sqrt{N}$ where N is the sample size (which is 50 in this case). Now resistance R can be computed from $R=V_{av}/I_{av}$ and uncertainty in R can be obtained from the well-known relation

$$\delta R = R \times \left(\frac{\delta V}{V_{av}} + \frac{\delta I}{I_{av}}\right)$$

Here $\delta V=V_{err}$ and $\delta I=I_{err}$ are standard errors in V and I measurements respectively. The temperature and resistance data and their uncertainties are stored in four separate lists which are used to plot a graph as shown in Fig.4. A least-square fit with the help of the 'numpy.polyfit()' function is also shown in Fig.4. The (slope/intercept) ratio of this line gives the value of α.

Codes for all the four examples described in this section and the related files are available for download at https://github.com/DwaipayanDeb/dimpy-tablefile-examples.git

5. Discussion

This paper has discussed the limitations of array formation and reading data from files in the present form of Python and introduces two new tools which eliminate these issues and improve the user experience for scientific calculations and analysis. The *tablefile* package simplifies reading tabulated data from a file with any kind of field separator, and also some inbuilt functions are provided to perform basic calculations like summation, averaging, standard deviations, etc. On the other hand, *dimpy* package can generate Python 'list' type arrays of any number of dimensions with any number of elements. Two physical examples in each case are given to show how these tools may simplify Python programming in physics. With the use of *dimpy* we may greatly simplify and enhance the calculations within arrays of two or more dimensions especially for symbolic operations like differentiation and integration. These examples also demonstrate how calculus can be applied to the matrix elements within nested loops that would be a difficult or time-consuming process without the help of *dimpy* in a Python program. Although given examples deal with two-dimensional arrays, a similar procedure may be

applied to perform calculus operations on arrays of any number of dimensions (as in tensors). On the other hand, we see that reading data from huge files like the Hipparcos catalog or a file containing experimental data is simplified with the use of *tablefile*. Also, inbuilt functions like convert(), av(), sd(), etc. are more efficient in the sense that they do not through run time error if the input list argument contains one or more string elements.

6. Declarations:

Funding: No funding was received for this work

Conflict of interest/Competing interests: No conflict of interest/ competing interest applicable

Availability of data and material: Data reported in this work are available to be used by anyone without restrictions.

Code availability: Available as supplementary at researchsquare as well as at github link https://github.com/DwaipayanDeb/dimpy-tablefile-examples.git

Acknowledgment:

I wish to thank Dr. Surajit Sen of Gurucharan College, Silchar, for his encouragement to write this paper.

References

[1] Harris, C.R., Millman, K.J., van der Walt, S.J. *et al.* Array programming with NumPy. *Nature* 585**,** 2020, 357–362 https://doi.org/10.1038/s41586-020-2649-2

[2] Jim Pivarski, Peter Elmer and David Lange, Awkward Arrays in Python, C++, and Numba, *EPJ Web of Conferences* 245, 2020, 05023 https://doi.org/10.1051/epjconf/202024505023

[3] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR,

Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A., SymPy: symbolic computing in Python. *PeerJ Computer Science* 3:e103, 2017, https://doi.org/10.7717/peerj-cs.103

[4] T.E. Oliphant, Python for Scientific Computing, *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10-20, May-June 2007, doi: 10.1109/MCSE.2007.58.

[5] Xing Cai, Hans Petter Langtangen and Halvard Moe, On the performance of the Python programming language for serial and parallel scientific computations, *Scientific Programming* 13 ,2005, 31–56, https://doi.org/10.1155/2005/619804

[6] Roland Lindh, Anders Bernhardsson, Gunnar Karlström, Per-Åke Malmqvist, On the use of a Hessian model function in molecular geometry optimizations**,** 1995 241(4),423-428, https://doi.org/10.1016/0009-2614(95)00646-L

[7] Caoxiang Zhu *et al*. Hessian matrix approach for determining error field sensitivity to coil deviations**,** 2018**,** *Plasma Phys. Control. Fusion***,**60**,**054016 https://doi.org/10.1088/1361-6587/aab6cb

[8] Mafalda Dias, Jonathan Frazer, and M. C. David Marsh, Simple Emergent Power Spectra from Complex Inflationary Physics, 2016, Phys. Rev. Lett. 117, 141303

[9] Herbert Goldstein, Classical Mechanics, 1998, 2nd Edition, p 391-431

[10] Deb, D., Chakraborty, C., Verification of the Spectral Classification of Stars Using the Hipparcos Catalogue, 2014, *Publications of the Astronomical Society of Australia (PASA)*, 31, e046

[11] Perryman, M. A. C., et al. , The Hipparcos Catalogue, 1997A&A, 323, L49

[12] Hannu Karttunen, Pekka Kroger, Heikki Oja, Markku Poutanen, Karl Johan Donner, Fundamental Astronomy, 2007, ISBN 978-3-540-34143-7 5th Edition, Springer Berlin Heidelberg New York

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- Programs.zip