Computational Geosciences

An educational project funded by the

Faculty of Science and Technology

University of Stavanger, Norway

2021

Preface

Welcome to the Computational Geosciences resource at the University of Stavanger, Norway (UiS). Computational Geosciences is not a new subject. A course in Computational Geology was offered for the first time at the University of South Florida in 1996 (Vacher, 2000). Computational Geosciences makes connections between mathematics, computation and geology. It promotes a mathematical problem-solving disposition (Vacher, 2000).

This resource is designed based on the same principles and with emphasis on problem-solving. However, the access to data and the tools used to visualize data have improved tremendously over the last 20 years. Today, a geologist carrying a mobile device in the field has access to a collection of sensors collecting data in real time (magnetometer, accelerometers, gravity, GPS, etc.), and accurate databases of topography, aerial photos, and satellite imagery. Such information not only supports the geologist in the field, but also allows her to test hypotheses and take decisions. Computation is greatly facilitated by high-level programming languages (e.g. Matlab and Python) that focus on visualizing and solving problems, rather than on coding details. The digital era is here, and to analyze the large number of data associated with it, we need math and computing.

To develop this resource, we put together an interesting group of faculty from the Departments of Energy Resources (IER) and Mechanical and Structural Engineering (IMBM) at the University of Stavanger, with expertise in Geographic Information Science (GIS, Lisa Watson, IER), Geophysics (Wiktor Weibull, IER), Structural Geology (Nestor Cardozo, IER), and Fluid Mechanics (Knut Giljarhus, IMBM). Master students from Computational Engineering (Angela Hoch), Offshore Engineering (Adham Amer), and Geosciences (Vania Mansoor and Linda Olsen) were instrumental. They wrote our scattered code into functions and notebooks, solve the exercise problems, and help editing the resource in Overleaf.

Python is the programming language of choice. The resource consists of ten chapters covering an introduction to computation in Geosciences and Python (chapter 1), understanding location (chapter 2), orientation and display of geologic features (chapter 3), coordinate systems and vectors (chapter 4), coordinate transformations (chapter 5), tensors (chapter 6), stress (chapter 7), strain (chapter 8), elasticity (chapter 9), and the inverse problem (chapter 10). Each chapter describes shortly the basic theory before going directly into

applications and problems. Exercises at the end of each chapter are essential to master the material. These exercises are not trivial and may require additional research. They are designed to test knowledge of the material.

Much of the material is based on the book Structural Geology Algorithms: Vectors and Tensors (Allmendinger et al., 2012), and the lab manual: Modern Structural Practice (Allmendinger, 2019). However, we have also included additional GIS and Geophysics topics. The resource is focused on our areas of expertise, but we hope you can use and further develop the material for other areas.

You can download or subscribe to changes in the resource using its git repository. We hope you enjoy this resource and learn from it, as well as use it for teaching. This will be the measure of our success. We also hope to spark enough interest for users to contribute to the resource with additional material and more chapters in the future. Finally, we are grateful to the Faculty of Science and Technology at the University of Stavanger for sponsoring this project.

Accessing the resource material

The best way to access the resource is by cloning or downloading its git repository. This repository contains a pdf of this book and a folder called source with the notebooks, functions and data in their corresponding folders. The notebooks follow this directory structure. We recommend you use the same structure when running them.

References

Allmendinger, R.W., Cardozo, N. and Fisher, D.M. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Allmendinger, R.W. 2019. Modern Structural Practice: A structural geology laboratory manual for the 21st century. [Online]. [Accessed January, 2020].

Vacher, H.L. 2000. A Course in Geological-Mathematical Problem Solving. Journal of Geoscience Education 48, 478-481.

Contents

1 Computation in Geosciences **9**

1.1 Solving problems by computation 9

1.2 Why Python? 10

1.3 Installing Python 11

1.4 A first introduction to Python 11

1.4.1 Basics 12

1.4.2 Conditionals 12

1.4.3 Loops 13

1.4.4 Functions and modules 14

1.4.5 Mathematics 15

1.4.6 Plotting 19

1.5 Exercises 20

References 21

2 Understanding location **23**

2.1 Locations 23

2.2 Geodesy basics 23

2.2.1 Scales 24
2.2.2 Authalic Sphere 25
2.2.3 Ellipsoidal Earth 26
2.2.4 Geoid 27
2.3 Projections 27
2.3.1 Distortions 28
2.4 Reference systems and datums 32
2.5 Conversion vs. transformation 36
2.6 Exercises 44
References 44

3 Geologic features **47**

3.1 Primitive objects: Lines and planes 47
3.2 Lines and planes orientations 47
3.2.1 Planes: Strike and dip 48
3.2.2 Lines: Trend and plunge or rake 49
3.2.3 The pole to the plane 50
3.2.4 Instruments used in the field 51
3.2.5 Uncertainties in orientations 53
3.3 Displaying geologic features 56
3.3.1 Maps 57
3.3.2 Stereonets 59
3.3.3 Plotting lines and poles in a stereonet 63

3.4 Exercises 66

References 67

4 Coordinate systems and vectors **69**

4.1 Coordinate systems 69

4.2 Vectors 70

4.2.1 Vector components, magnitude, and unit vectors 70

4.2.2 Vector operations 72

4.3 Geologic features as vectors 75

4.3.1 From spherical to Cartesian coordinates 75

4.3.2 From Cartesian to spherical coordinates 78

4.4 Applications 80

4.4.1 Mean vector 80

4.4.2 Angles, intersections, and poles 85

4.4.3 Three point problem 89

4.5 Uncertainties 92

4.6 Exercises 95

References 98

5 Transformations **101**

5.1 Transforming coordinates and vectors 101

5.1.1 Coordinate transformations 101

5.1.2 Transformation of vectors 103

5.1.3 A simple transformation: From **ENU** to **NED** 104

5.2 Applications . . . 105
5.2.1 Stratigraphic thickness . . . 105
5.2.2 Outcrop trace of a plane . . . 112
5.2.3 Down-Plunge projection . . . 115
5.2.4 Rotations . . . 121
5.2.5 Plotting great and small circles using rotations . . . 125
5.3 Exercises . . . 133
References . . . 138

6 Tensors **141**
6.1 Basic characteristics of a tensor . . . 141
6.2 Principal axes of a tensor . . . 143
6.3 Tensors as vector operators . . . 144
6.4 Tensor transformations . . . 145
6.4.1 The Mohr Circle . . . 146
6.5 The orientation tensor . . . 148
6.5.1 Best-fit fold axis . . . 149
6.5.2 Line distributions . . . 157
6.5.3 Best-fit plane . . . 158
6.6 Exercises . . . 162

7 Stress **165**
7.1 The stress tensor . . . 165
7.1.1 Cauchy's law . . . 167

7.1.2 Stress transformation 170
7.1.3 Principal axes of stress 172
7.2 Mohr Circle for stress 177
7.2.1 Special states of stress 179
7.3 Mean and deviatoric stress 180
7.4 Applications 181
7.4.1 Normal and shear tractions on a plane 181
7.4.2 The Mohr Circle for stress in 3D 188
7.5 Exercises 194

8 Strain **197**

8.1 Deformation and strain 197
8.2 Deformation and displacement gradients 200
8.3 Infinitesimal strain 204
8.3.1 Mohr Circle for infinitesimal strain 207
8.3.2 Applications of Infinitesimal Strain 209
8.4 Finite strain 225
8.4.1 Mohr Circle for finite strain 229
8.4.2 2D finite strain from displacement data 232
8.5 Progressive strain 234
8.5.1 Pure shear 235
8.5.2 Simple shear 239
8.5.3 General shear 242

8.6 Exercises 247

Chapter 1

Computation in Geosciences

1.1 Solving problems by computation

Geology is an interpretive and historical science (Frodeman, 1995). We observe, collect, analyze, and interpret data (what) to tell a story (why). To collect data, we need to take measurements. All measurements have some uncertainty, and therefore uncertainty and error propagation are very important in geosciences, and they are a recurring topic in this resource.

For the last 50 years or more, the methods geoscientists have used to visualize, analyze and interpret data are mostly graphical. For example, in structural geology, students typically learn two types of graphical constructions: orthographic and spherical projections (stereonets) (Ragan, 2009). Although these methods are great to visualize and solve geometrical problems in three-dimensions, they are not amenable to computation, and therefore applying these methods to large datasets with thousands of entries is impractical. Plane and spherical trigonometry allow deriving formulas (e.g. apparent dip formula) for computation (Ragan, 2009). However, these formulas give little insight about the problems. They are just formulas associated with complex geometric constructions, which bear no relation to each other, and which are difficult to combine to solve more complicated problems.

It turns out that many of the most interesting problems in geosciences can be solved using linear algebra, and more specifically vectors and tensors (Allmendinger et al., 2012). Linear algebra also happens to be the language of data and computation. The main purpose of this resource is to show how

to solve problems in geosciences using computation. There are several advantages of following this approach. It will enhance your mathematical and computational skills, as well as promote your geological-mathematical problem solving disposition. In today's digital age, these skills are very useful.

1.2 Why Python?

The choice of programming language is important. While computer languages such as C or C++ are ideal to work with large datasets and computer-intensive operations, they involve a steep learning curve associated with their syntax, compilation, and execution (Jacobs et al., 2016). These coding details have little to do with the problem-solving approach of this resource. Interpretive languages such as Python, R or Matlab are a better choice because of their simpler syntax, and the interpretation and execution of commands as they are called (no need for compilation). In addition, these languages have access to an integrated development environment (IDE) that facilitates writing and debugging programs, and to many standard libraries that perform advanced tasks such as matrix operations and data visualization. Thus, Python, R or Matlab are "scientific packages" rather than just programming languages.

In this resource, the language of choice is Python. Besides the reasons above, Python has the following advantages:

- Python can be learned quickly. It typically involves less code than other languages and its syntax is easier to read.
- Python comes with robust standard libraries for arrays and mathematical functions (NumPy), visualization (PyPlot), and scientific computing (SciPy).
- Python is one of the most popular programming languages, with a large base of developers and users. It is used by every major technology company and it is almost a skill you must have in your CV to land a job as a geoscientist.
- Because of its large developers base, Python has access to a large amount of additional libraries, including several libraries for geosciences. We make use of some of these libraries.

- Python can be installed easily through a single distribution that includes all the standard libraries and provides access to additional libraries (see next section).

- Last but not least, Python is free and open source. This is probably why Python is more popular than its commercial counterpart Matlab.

1.3 Installing Python

We recommend installing Python using the free Anaconda distribution. This distribution includes Python as well as many other useful applications, including Jupyter, which is the system we use to write the notebooks in this resource. Anaconda can be easily installed on any major operating system, including Windows, macOS, or Linux.

The installation process is quite straightforward. From the Anaconda individual edition page, go to the Download section. Windows and macOS users just need to download the Graphical installer, run it, and follow the steps to install Anaconda. Linux users need to download the installer and type a set of commands in a terminal window. Further instructions can be found in the Anaconda installation section.

1.4 A first introduction to Python

In this section, we use our first Jupyter notebook to learn the basics of Python. Clone or download the resource git repository. Open Anaconda and then launch Jupyter Notebook. This will open a browser with a list of files and folders in your home directory. Navigate to the folder source/notebooks in the repository and open the notebook ch1. Alternatively, follow the notebook in the sections below. Surprisingly, few lines of code are required to introduce key topics such as conditionals, loops, functions, array mathematics, and plotting. This shows the power of Python.

1.4.1 Basics

A notebook is divided into computational units called *cells*. Cells can contain text such as this one or Python code. Below is a cell with some typical Python statements. Try changing the variables and re-run the cell. To run a cell, either click the *Run* button, or type *Ctrl+Enter* (*cmd+Enter* in a Mac).

```
a = 2
b = 9.0
c = a + b
print('The sum is: ', c)

# This is just a comment

name = 'Donald'
print('Hello, my name is', name)
```

Output:
The sum is: 11.0
Hello, my name is Donald

There are some other useful shortcuts you should know. To run a cell and move to the next cell, type *Shift+Enter*. To run a cell and insert a new cell below, type *Alt+Enter*. You can use the arrow keys to move quickly between cells. To run all the cells of a notebook, choose the *Cell* → *Run all* menu.

1.4.2 Conditionals

A conditional is used to perform different operations depending on a conditional statement. In Python, this is expressed in the following way:

```
a = 3
b = 5
if a > b:
    print('a is bigger than b')
elif a < b:
    print('a is smaller than b')
else:
    print('a is equal to b')
```

Output:
a is smaller than b

Try changing the values of a and b to see how the output changes. Also, note that Python cares about white spaces, so there must be a tab indent or 4 spaces for each operation in the if statement. You can also use the boolean operators *and*, *or*, and *not* in the conditional statement:

```
age = 30
if age > 18 and age < 34:
    print('You are a young adult')

if age < 18 or age > 80:
    print('You are not allowed to drive a car')
```

Output:
You are a young adult

1.4.3 Loops

A loop is used to execute a group of statements multiple times. For instance, to print all numbers from 1 to 10 divisible by 3, we can use a *for* loop together with an *if* statement, and the modulus operator %:

```
print('Number divisible by three:')
for i in range(1, 11):
    if i % 3 == 0:
        print(i)
```

Output:
Numbers divisible by three:
3
6
9

range is a Python function that iterates from the given first number up to the second number (but not including it). If we only give one number, the iteration will go from zero up to (but not including) the given number. There are more examples of *for* loops later in this notebook.

1.4.4 Functions and modules

If we have written a useful piece of code, we often want to use it again without copying and pasting the code multiple times. To do this, we use functions and modules. For instance, if we want to convert an angle from degrees to radians, we can use the following formula:

$$\alpha_{\text{radians}} = \alpha_{\text{degrees}} \frac{\pi}{180} \tag{1.1}$$

To put this into a callable function, we use the *def* keyword:

```
def deg_to_rad(angle_degrees):
    pi = 3.141592
    return angle_degrees*pi/180.0

angle_degrees = 45.0
print('Radians', deg_to_rad(angle_degrees))
```

Output:
Radians 0.785398

We can also include code from other places. This is useful to make your own library of functions that you can then use in other notebooks. This is the modus operandi of this resource. We will implement and use functions to solve interesting problems in geosciences. Using a text editor, create a file called *mylib.py* and put it in the same folder the notebook is. In the file, write a function to convert from radians to degrees:

```
def rad_to_deg(angle_radians):
    pi = 3.141592
    return angle_radians*180/pi
```

We can then import in the notebook the code from the file and use it like this:

```
try:
    import mylib
    angle_radians = 0.785398
```

```
    print('Degrees', mylib.rad_to_deg(angle_radians))

except ModuleNotFoundError:
    print('Create a file called mylib.py')
```

Output:
Degrees 45.0

Note: If you make a change in *mylib.py*, the changes will not be immediately available in the notebook and it needs to be restarted. To circumvent this, we can use the following commands to always reload imported modules:

```
%load_ext autoreload
%autoreload
```

1.4.5 Mathematics

To use Python as an environment for numerical mathematics, it is useful to use the NumPy library for arrays and matrices, and the Matplotlib for plotting. See the links in the *Help* menu for more information on these libraries. The following two lines import these libraries:

```
import numpy as np
import matplotlib.pyplot as plt
```

To define an array, we use the NumPy *array* function:

```
a = np.array( [1, 2, 3, 4] )
print(a)
```

Output:
[1 2 3 4]

To access an array element, we use brackets with the index of the element. A very important difference compared to Matlab is that in Python, the first element has index zero (like most other programming languages). We can also use negative indices to access values starting from the end of the array.

```
print(a[0], a[2])
print(a[-1])
```

Output:
1 3
4

Slicing is a very useful feature to extract subarrays. For instance:

```
print(a[2:])
print(a[1:3])
```

Output:
[3 4]
[2 3]

Matrices are defined as multi-dimensional arrays:

```
a_matrix = np.array( [[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]] )
b_matrix = np.array( [[2, 4],
                      [3, 5],
                      [5, 7]] )
print(a_matrix)
print(b_matrix)
```

Output:
[[1 2 3]
[4 5 6]
[7 8 9]]
[[2 4]
[3 5]
[5 7]]

We can get the number of rows and columns of the matrix from the *shape* variable:

```
nrow, ncol = b_matrix.shape
print('b has {} rows and {} columns'.format(nrow, ncol))
```

Output:
b has 3 rows and 2 columns.

Let us make a function to multiply two matrices. Consider a $n \times m$ matrix $\mathbf{A}$ and a $m \times p$ matrix $\mathbf{B}$. The formula to multiply these matrices can be written as:

$$\mathbf{C} = \mathbf{AB} = \sum_{k=1}^{m} A_{ik} B_{kj} \tag{1.2}$$

for $i = 1,...,n$ and $j = 1,...,p$. Here $\mathbf{C}$ will be a $n \times p$ matrix. To implement this formula, we need to use a triple-nested loop, as shown in the function below:

```
def matrix_multiply(A,B):
    n, m = A.shape
    nrow_B, p = B.shape

    # Check that matrices are conformable
    if not nrow_B == m:
          print('Error, the number of columns in A must be
    equal to the number of rows in B!')
          return -1
     # Initialize C using the numpy zeros function
     C = np.zeros((n,p))
     for i in range(n):
         for j in range (p):
             for k in range (m):
                 C [i,j] = C[i,j] + A[i,k]*B[k,j]
     return C

print(matrix_multiply(a_matrix, b_matrix))
```

Output:
[[23. 35.]
[53. 83.]
[83. 131.]]

Verify by hand calculation that the above result is correct. Remember, the element in the first row and first column of **C** is equal to the sum of the product of the elements in the first row of **A** times the elements in the first column of **B**, and so on. What happens if you try the multiplication **BA**? Try it.

Although the function above is elegant, it is not very efficient. The NumPy library contains super-optimized code for common operations such as matrix multiplication. The NumPy *dot* function can be used for matrix multiplication. Let's repeat the matrix multiplication above using the *dot* function:

```
C = np.dot(a_matrix, b_matrix)
print(C)
```

Output:
[[23 35]
[53 83]
[83 131]]

When working with large matrices, there is a significant impact on the runtime. To illustrate this, let's generate two 100 x 100 matrices and time how long it takes to multiply them. The *%%timeit* command will run the cell a number of times and output the average time spent per run. The NumPy *random.rand* function generates the arrays and fill them with random numbers.

```
%%timeit
N = 100
A = np.random.rand(N,N)
B = np.random.rand(N,N)
C = matrix_multiply(A,B)
```

Output:
625 ms ± 3.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

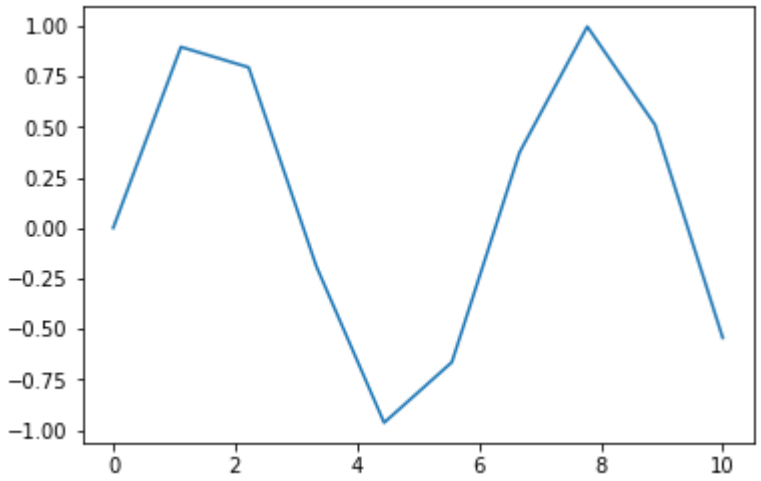
Change the multiplication function from *matrix_multiply* to *np.dot* and note the difference in runtime. On a standard computer, our *matrix_multiply* function uses ≈ 600 milliseconds per loop, while NumPy *dot* function uses ≈ 200 microseconds. The NumPy *dot* function is a staggering 3000 times faster!

1.4.6 Plotting

Arrays can be easily plotted using the Matplotlib *plot* command. Below we plot the sinusoidal function. We use the NumPy *linspace* function to generate an array with equally spaced values between the start and end point, and the NumPy *sin* function to take the sine of the array. The *plot* command plots the data and the semicolon after it removes any message output. With a low number of points, the curve is actually jagged. Increase the number of points n in the *linspace* command to get a smoother curve. Try values of n = 100, 1000, and 10000.

```
# The linspace command gives us an equally spaced array
# The syntax is:
# linspace(start_point, end_point, number_of_points)
n = 10
x = np.linspace(0, 10, n)
y = np.sin(x)
plt.plot(x, y);
```

Output:

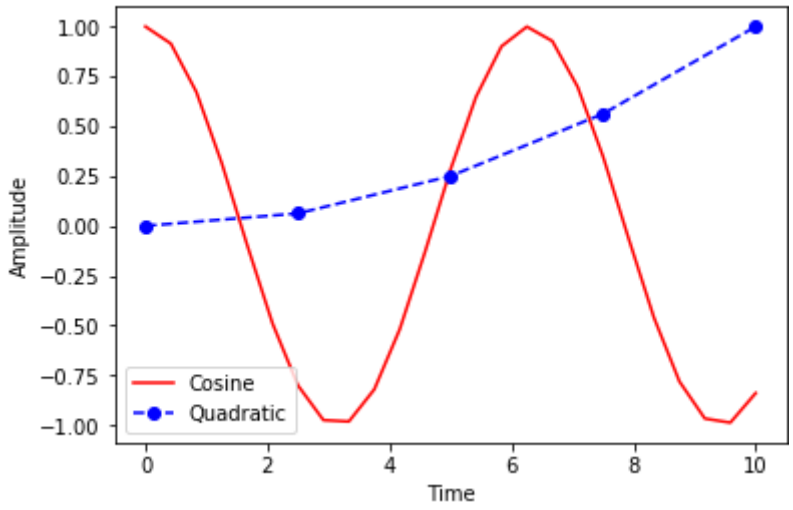


We end with a slightly more advanced plot, showing how to change line style and markers, and add axes labels and a legend. The NumPy *cos* function takes the cosine of the array, and *xlabel*, *ylabel* and *legend* are all Matplotlib commands to add labels to the axes and a legend to the graph.

t

```
n = 25
x = np.linspace(0, 10,25)
y = np.cos(x)
# plot cosine function as a red line
plt.plot(x, y, 'r')
x = np.linspace(0,10,5)
y = 0.01*x**2
# plot quadratic funcion as blue dashed line with dots
plt.plot(x,y,'bo--')
# label axes
plt.xlabel('Time')
plt.ylabel('Amplitude')
# Add legend
plt.legend(['Cosine', 'Quadratic']);
```

Output:



1.5 Exercises

1. Write a program that prints each number from 1 to 20 on a new line. For each multiple of 3, print "Fizz" instead of the number. For each multiple of 5, print "Buzz" instead of the number. For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number. The correct answer is: 1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz. *Hint:* You will need to use a loop and conditionals to solve this problem.

2. Write a function to convert an angle from degrees to radians or from radians to degrees. The function should accept two inputs: the angle, and a flag to tell the function whether the angle should be converted from degrees to radians (flag = 0) or from radians to degrees (flag = 1). *Hint:* In the function, you should use a conditional to either convert the angle from degrees to radians, or from radians to degrees.

3. Given two 3 × 3 matrices **A** = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] and **B** = [[5, 7, 2], [3, 5, 1], [2, 4, 3]], compute:

   (a) the sum of the matrices (**A** + **B**),
   (b) The difference of the matrices (**A** - **B**),
   (c) The product of the matrices (**AB**),
   (d) The square root of matrix **A**,
   (e) The sum of all elements of matrix **B**,
   (f) The column sum of matrix **A**,
   (g) The row sum of matrix **A**,
   (h) The transpose of matrix **A** ($\mathbf{A}^T$),
   (i) The product $\mathbf{A}\mathbf{A}^T$. What is this product equal to?

   Hint: Look at the functions *add, subtract, dot, sqrt, sum* and *transpose* in the NumPy library.

4. The apparent dip α of a plane is given by the following equation:

$$\tan\alpha = \tan\delta \sin\beta \tag{1.3}$$

   where δ is the true dip of the plane, and β is the structural bearing (Fig. 3.1b). We will talk about this equation latter in chapter 3.

   (a) Make a function to compute the apparent dip α from the true dip δ and structural bearing β. For the function, these angles should be entered in radians.
   (b) Use this function to make a graph of apparent dip α (0 to 90°, vertical axis) versus the structural bearing β (0 to 90°, horizontal axis), for values of true dip δ of 10, 20, 30, 40, 50, 60, 70, and 80°.

   The graph should look like Figure 1.1 below:

Hint: You will need to import the NumPy and Matplotlib libraries to solve this problem. The problem is intendedly hard, don't give up.

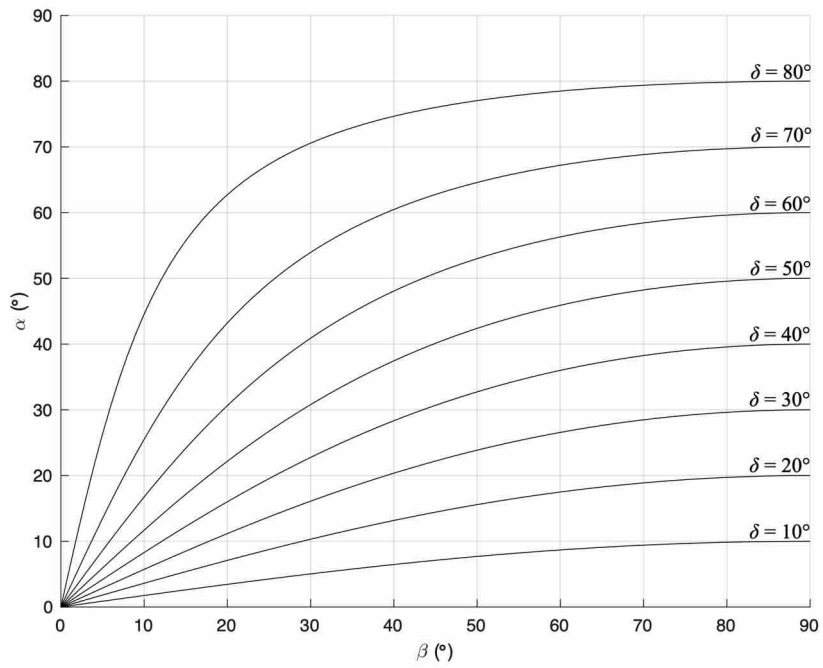


Figure 1.1: Apparent dip α as function of section bearing β and true dip δ.

References

Allmendinger, R.W., Cardozo, N. and Fisher, D.W. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Frodeman, R. 1995. Geological reasoning: Geology as an interpretive and historical science. GSA Bulletin 107, 960-968.

Jacobs, C.T., Gorman, G.J., Rees, H.E. and Craig, L.E. 2016. Experiences with efficient methodologies for teaching computer programming to geoscientists. Journal of Geological Education 64, 183-198.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Chapter 2

Understanding location

2.1 Locations

Understanding where we are located or where an object of interest is located is extremely important in geosciences. Geographic information science is the study of geographic information; it includes theory and concepts and provides methods to combine and analyze spatial data (Watson, 2017). Geographic information systems is the software and technology that supports the application of geographic information theory (Watson, 2017). GIS is an acronym used for either geographic information systems or geographic information science.

Geosciences are Earth-based and so location-based. The geographic aspect is highly applicable for the geosciences. The basic usage of GIS is cartographic – making maps. In this chapter, you will become familiar with some of the basic concepts in defining locations on the Earth. These concepts are also used for defining locations on other planets as well.

2.2 Geodesy basics

Geodesy is "the branch of mathematics dealing with the shape and area of the Earth or large portions of it" (Lexico, 2019); however, the discipline is expanding to include other planets. We need geodesy to model the Earth

and make calculations because the Earth is not a perfect sphere nor flat.

2.2.1 Scales

In cartography, we use scales to describe how the real world length has been reduced to fit on a page or screen. Usually, map scales are described as ratios, such as 1:50,000. When we describe the scale, we say it is either small or large. The description of small or large refers to the scale, not the size of the area. Therefore, if we describe a scale as small, it means the fraction described by the scale ratio is small; this in turn means the map covers a large area. The inverse is true for large scales; the fraction described by the scale is large and in turn covers a small area (Kraak and Ormeling, 2003, Lisle et al., 2011). There is not an official categorical differentiation between small and large scales. Generally speaking, small scale maps cover regions, countries, and continents, while large scale maps cover neighborhoods, towns, or counties. The following example illustrates this.

Example 1: Understanding Map Scales

A map has a scale of 1:1,000,000. Would you refer to this as a small or large scale?

1. First, rewrite this as a fraction: 1/1,000,000
2. Is this a small fraction or a large fraction? This is a relatively small fraction, so it is a small scale.

Figure 2.1 is an example of a map with a scale of 1:1,000,000.

A map has a scale of 1:10,000. Would you refer to this as a small or large scale?

1. First, rewrite this as a fraction: 1/10,000
2. Is this a small fraction or a large fraction? This is a relatively large fraction, so it is a large scale.

Figure 2.2 is an example of a map with a scale of 1:10,000.

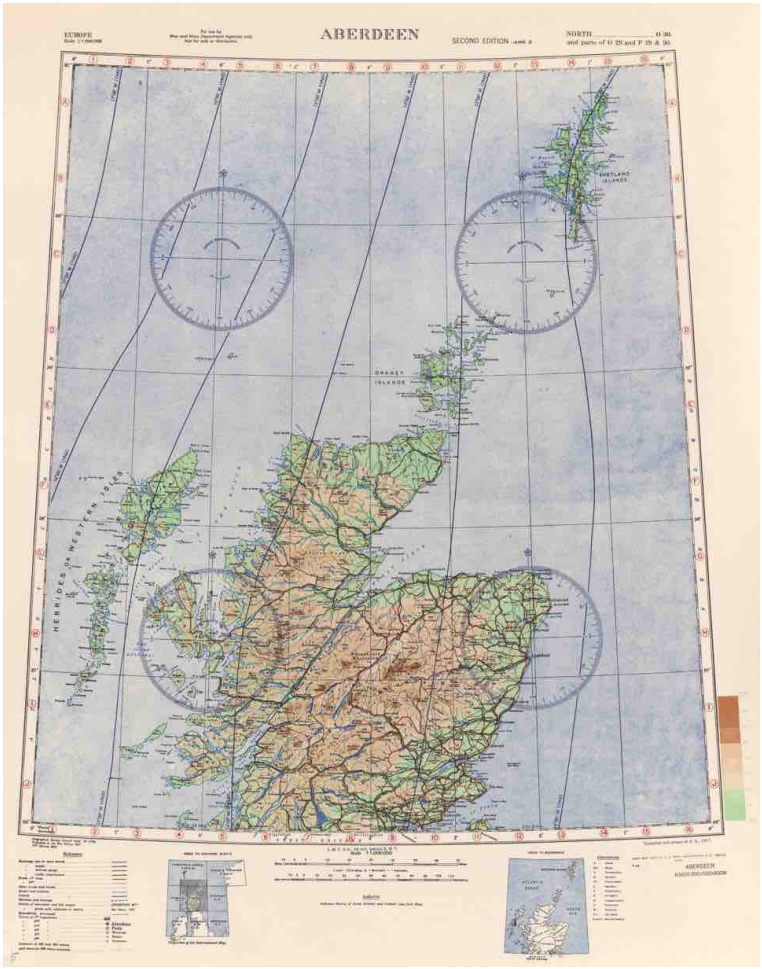


Figure 2.1: A historical map of northern Scotland at the scale of 1:1,000,000 (Geographic Section - General Staff, 1941).

2.2.2 Authalic Sphere

The authalic sphere is a sphere used for the basic surface for mapping (Fig. 2.3); its surface area is the same as the ellipsoid (Robinson et al., 1995). Based on the WGS-84 ellipsoid, the Earth has a radius of 6371 km and a circumference of 40,030.2 km. The radius is an often-used constant in geodesy (Robinson et al., 1995). The authalic sphere is used in small scale mapping (small scale covers a large area) because the difference between the authalic sphere and the ellipsoid is minimum over large areas (Robinson et al., 1995).

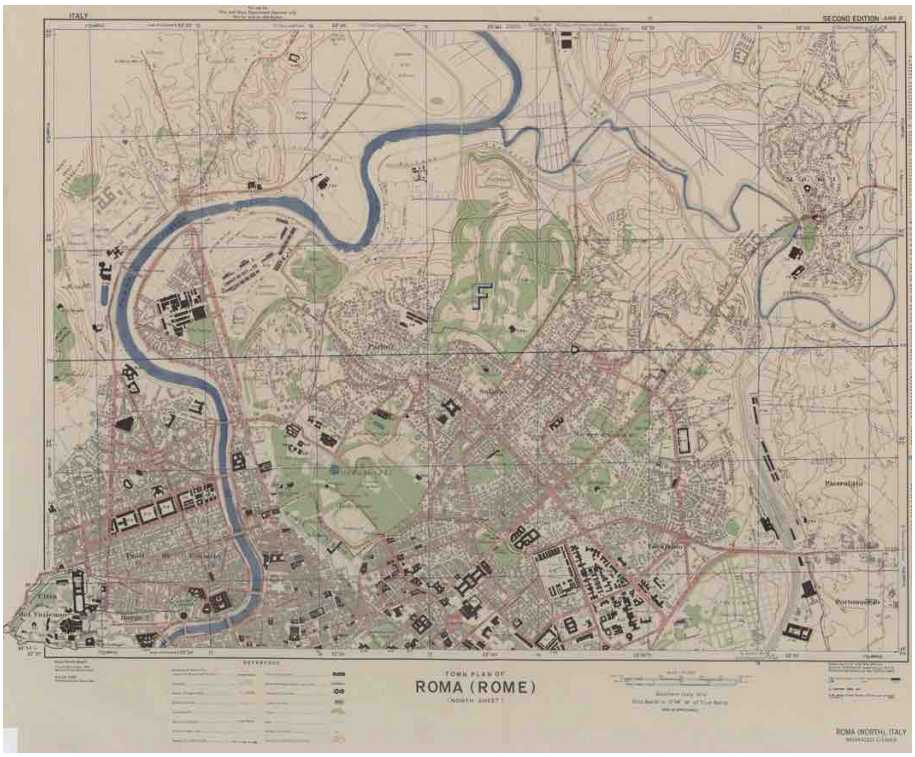


Figure 2.2: A historical map of Rome, Italy at the scale of 1:10,000 (C.I.U. and War Office, 1944).

2.2.3 Ellipsoidal Earth

Due to gravity, the Earth flattens at the poles. In a cross-section, the Earth looks like an oblate ellipsoid (Fig. 2.3) (Robinson et al., 1995). Oblateness refers to flatness. When mapping over small areas (large scale mapping), the oblateness of the ellipsoid must be taken into account. The GPS network uses the WGS-84 ellipsoid (Robinson et al., 1995).

There are varying ellipsoidal measurements on different continents and times. These continental differences are due to gravity. Temporal differences are due to technological accuracy. The WGS-84 ellipsoid is based on satellite observations and is accepted as being highly accurate (Robinson et al., 1995).

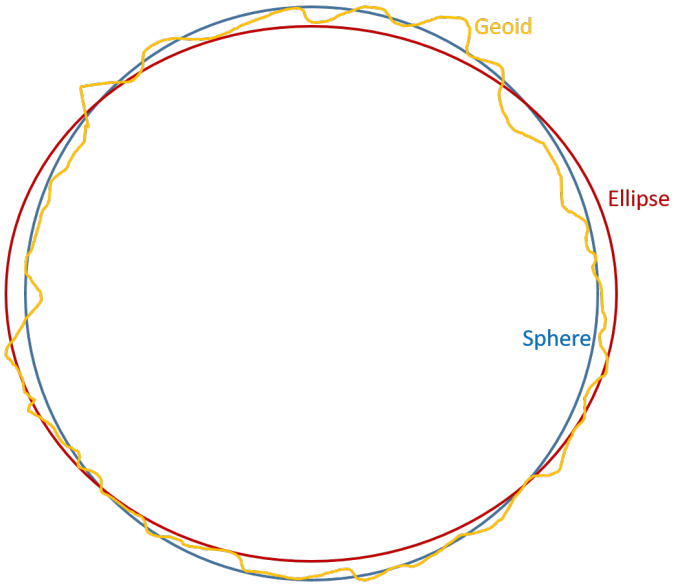


Figure 2.3: Highly stylized comparison of sphere, ellipse, and geoid.

2.2.4 Geoid

Geoid means Earth-like and is in 3D. It is based on an equipotential gravity surface. The geoid follows the mean sea level in oceans and hypothetical sea-level canals on the continents (Robinson et al., 1995). Due to geology (rock density) and topography, the geoid deviates from the ellipsoid (Fig. 2.3). The geoid is a "reference surface for ground surveyed horizontal and vertical positions" (Robinson et al., 1995).

2.3 Projections

A projection is a mathematical equation to transfer a region, of whatever size, of the round Earth onto a flat surface. Projections are used because distance and surface area calculations are more difficult on a sphere. A flat map can show greater detail than a sphere and is more transportable. Imagine how large a globe you would need to sufficiently show the streets in your neighborhood! We need projections to transform our 3D ellipsoidal Earth onto a flat map. Projections may be based on the authalic sphere, ellipsoid, or geoid.

Before proceeding, take a moment to look over an informational pictographic by the U.S. Geological Survey describing different types of projection.

2.3.1 Distortions

All projections have distortions that vary by projection type (i.e. transverse Mercator vs. Miller cylindrical – see pictograph mentioned in previous section). Selecting a projection depends on discipline, size of area, orientation of area, regional standards, map purpose, and map scale. There are many resources for determining which projection you should use. Large-scale mapping uses conformal projection because angles measured on the ground are the same as those in the map (Iliffe and Lott, 2008). Four types of distortion are: area, shape, direction, and distance. The Tissot's Indicatrix is a graphic device to show the distortion at a point (Robinson et al., 1995). We will investigate this phenomenon using the Python library Cartopy. To install this library, follow the steps below:

Installing Cartopy

We will use the Cartopy library to visualize projection distortions using the Tissot's Indicatrix. We will make a special Cartopy Environment in Anaconda. This is because Cartopy dependencies are lower than some other libraries we'll be using. This happens from time to time when we use open-source libraries.

1. Open Anaconda Navigator.
2. In the left panel, click on "Environments".
3. In the middle panel, click "Create" to create a new environment
4. Name this environment: ch2cartopy
5. Select Python version 3.7
6. Click "Create". This will take a few minutes.
7. We need to install Cartopy. In the right panel in the search field, type: cartopy

8. You will get the message "0 packages available matching cartopy" since Cartopy is not installed.
9. In the right panel, change "Installed" to "Not installed"
10. Cartopy now shows up[1]. Click the check box next to "cartopy".
11. In the screen lower right corner, click "Apply"
12. Please wait while Cartopy is collected and then click "Apply"
13. The Cartopy library and any dependencies will be installed or updated. This may take a few minutes.
14. Click on "Home"
15. Install the console of your choice for the new environment. Click on "Install" under Jupyter Notebook, for example.
16. Click on "Launch"

Example 2: Tissot's Indicatrix

This example will introduce you to understanding distortions in projections. As you change the projection name, a different mathematical equation will be used to portray the round Earth in a flat presentation. Pay particular attention to the size, shape, and spacing of the ellipses describing the distortion. The Tissot's Indicatrix quickly and easily visualizes the changes in area and spatial relationships between different projections.

The notebook ch2-1 contains this example. If you just launched Jupyter Notebook as indicated above, open the notebook. If you closed Anaconda, follow these steps:

1. Open Anaconda Navigator
2. Click on "Environments"
3. Choose "ch2cartopy"
4. Click "Home"

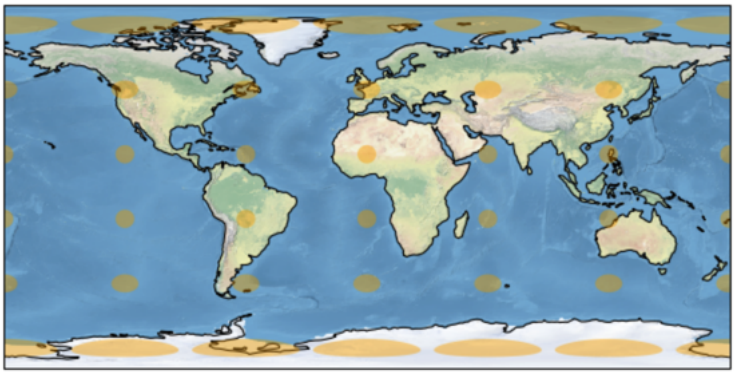
[1] If cartopy does not show up, you may need to press "Update index" in the right panel.

5. Launch Jupyter Notebook
6. Open the notebook ch2-1

This example starts with the Plate Carree projection (Cartopy, 2018b). Run the code below:

```
# Import cartopy and matplotlib
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

# Hide warnings
import warnings
warnings.simplefilter('ignore')

# Figure
plt.figure(figsize=(10, 5))

# Plate Carree projection
ax = plt.axes(projection=ccrs.PlateCarree())

# Make the map global rather than have it zoom in to
# the extents of any plotted data
ax.set_global()

# Earth image
ax.stock_img()
# Coastlines
ax.coastlines()

# Tissot's indicatrix: Orange ellipses
ax.tissot(facecolor='orange', alpha=0.4)
plt.show()
```

Output:

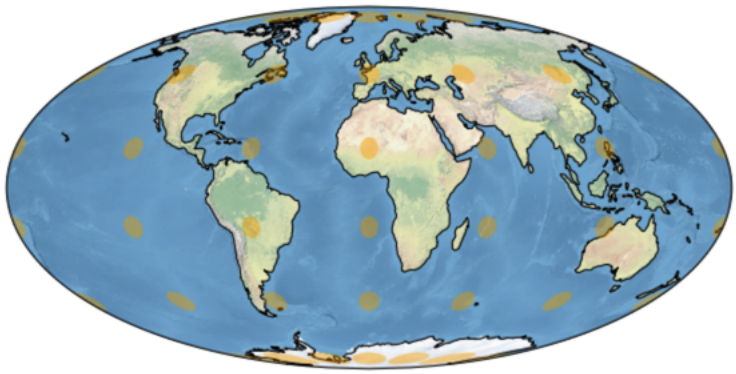


The Tissot's Indicatrix is symbolized by the orange ellipses. Closer to the poles, the ellipses become more oblate; while closer to the Equator, they are more circular. The Plate Carrée projection is a specific form of the Equidistant Cylindrical projection. Plate Carrée has the latitude of origin at the Equator.

Change the code to use the Mollweide projection:

```
# Figure
plt.figure(figsize=(10, 5))

# Mollweide projection
ax = plt.axes(projection=ccrs.Mollweide())

# Make the map global rather than have it zoom in to
# the extents of any plotted data
ax.set_global()

ax.stock_img()
ax.coastlines()

ax.tissot(facecolor='orange', alpha=0.4)
plt.show()
```

Output:



Notice how the sizes of the ellipses are very similar throughout the map. All of the ellipses are more rounded and circular regardless of their position, as compared to the Plate Carrée projection. Mollweide is often used for world maps.

Cartopy has a list of projections that are included in the library (Cartopy, 2018a). Change the code above to project the map in Azimuthal Equidistant.

2.4 Reference systems and datums

A coordinate reference system is a coordinate system that has been referenced to a datum. A datum is the location used for a reference point from which spatial measurements are made. There are geographic and Cartesian coordinate systems. Coordinates are for specific locations on the Earth. They can be expressed as geographic using latitude and longitude. Latitude are parallels that are evenly spaced and longitude are meridians that converge at the poles. These are measured in degrees. Cartesian coordinates are expressed in x and y and may have units that are meters, feet, or kilometers, for example. Coordinates only have meaning when the coordinate system and datum are known (Iliffe and Lott, 2008, Robinson et al., 1995).

Example 3: Defining the coordinate reference system

In any GIS program, including spatial libraries and code, the user must ensure that the coordinate reference system is defined for the spatial data. The GIS software will make assumptions, sometimes erroneous, if the coordinate reference system is not properly defined. In this example, we will see a demonstration of these assumptions and how to prevent them. This example is from the SciTools tutorials to understand Cartopy (Cartopy, 2018c).

In Cartopy, there are two keywords that you must understand in order to properly display your data. The "projection" argument is used for display of your data. This only affects the map or plot. It does not define the coordinate reference system of the data itself. The "transform" argument, on the other hand, defines the coordinate reference system. The best practice is to define both of these. We will investigate the error that occurs when the best practice is not followed and compare this to when the best practice is followed. The notebook ch2-2 contains this example.

First we will create some dummy data on a regular latitude/longitude grid:

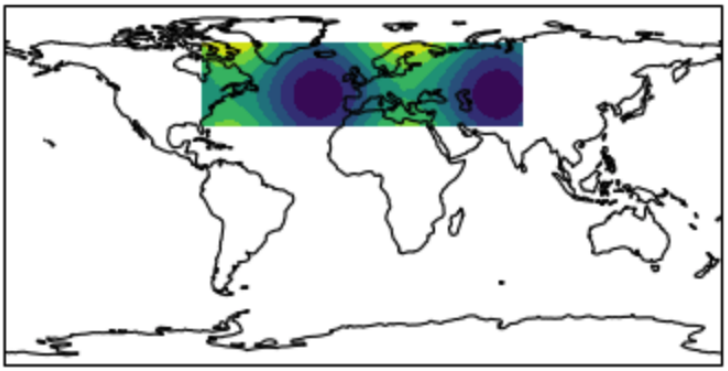
```
import numpy as np

lon = np.linspace(-80, 80, 25)
lat = np.linspace(30, 70, 25)
lon2d, lat2d = np.meshgrid(lon, lat)

```

```
data = np.cos(np.deg2rad(lat2d) * 4) + np.sin(np.deg2rad(
    lon2d) * 4)
```

In order to demonstrate the error before "best practice", we will create a map using the Plate Carree projection but only specify the "projection" argument. Remember that the best practice requires both projection and transform arguments to be defined.

```
# Import cartopy and matplotlib
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

# The projection keyword determines how the plot will look
plt.figure(figsize=(6, 3))
ax = plt.axes(projection=ccrs.PlateCarree())
ax.set_global()
ax.coastlines()

# didn't use transform, but looks ok...
ax.contourf(lon, lat, data)
plt.show()
```

Output:

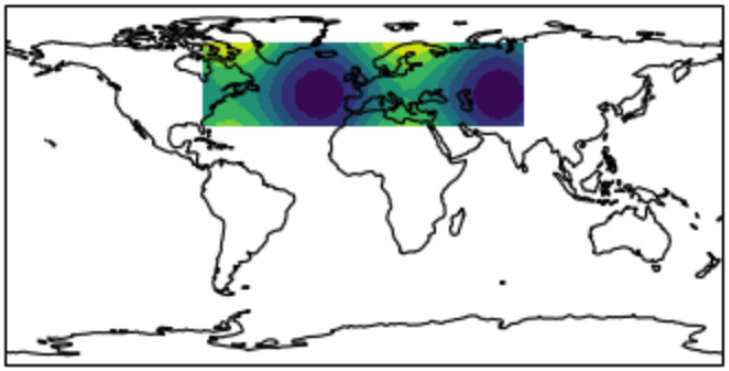


In this case, the data just happen to fall in the correct location. Now, we will define the data coordinate reference system (first line of code below) and add the transform argument to the plot (second last line of code below).

```
# The data are defined in lat/lon coordinate system,
# so PlateCarree() is the appropriate choice:
```

```
data_crs = ccrs.PlateCarree()

# The projection keyword determines how the plot will look
plt.figure(figsize=(6, 3))
ax = plt.axes(projection=ccrs.PlateCarree())
ax.set_global()
ax.coastlines()

# use transform
ax.contourf(lon, lat, data, transform=data_crs)
plt.show()
```

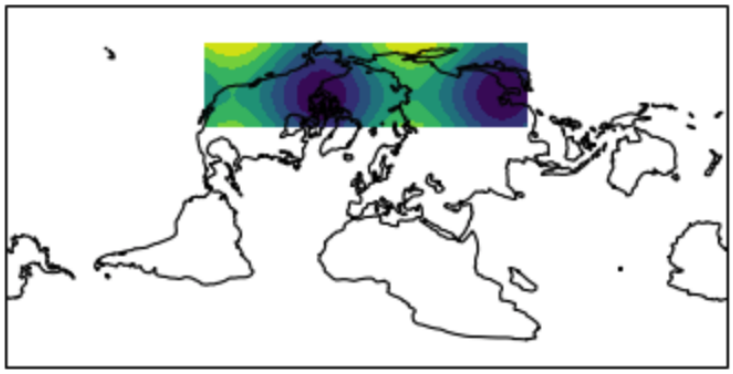
Output:



You will notice that our plot remains unchanged. The assumption, as stated previously, is that if the coordinate reference system of the data is undefined, it is the same as the map (or plot) projection. In the example above, this has been the case. Let us now investigate what happens when changing the projection of the map without defining the coordinate reference system of the data. We now define the projection to "RotatedPole" and omit the transform argument to see what happens:

```
# Now we plot a rotated pole projection
projection = ccrs.RotatedPole(pole_longitude=-177.5,
    pole_latitude=37.5)
plt.figure(figsize=(6, 3))
ax = plt.axes(projection=projection)
ax.set_global()
ax.coastlines()

ax.contourf(lon, lat, data)  # didn't use transform, uh oh!
plt.show()
```

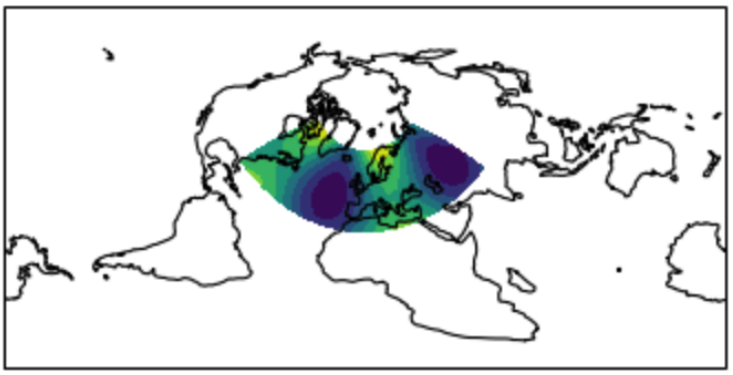
Output:



In this case, we see that the country boundaries have rotated and changed shape, however, the data did not move with the the country boundaries. We made a wrong assumption in the data definition. Therefore we need to define the transform argument:

```
# A rotated pole projection again...
projection = ccrs.RotatedPole(pole_longitude=-177.5,
    pole_latitude=37.5)
plt.figure(figsize=(6, 3))
ax = plt.axes(projection=projection)
ax.set_global()
ax.coastlines()

# ...but now using the transform argument
ax.contourf(lon, lat, data, transform=data_crs)
plt.show()
```

Output:



Now the data are correctly projected. Get in the habit of always defining the coordinate reference system and the map plot projection. It will save headaches and misunderstandings of the data.

Here is another script using an entirely different projection and additional plotting parameters. Can you figure out what the script will produce before running the cell? Notice the presence of the Matplotlib function *subplot*. What do you think this function does?

```
# We can choose any projection we like...
projection = ccrs.InterruptedGoodeHomolosine()
plt.figure(figsize=(6, 7))
ax1 = plt.subplot(211, projection=projection)
ax1.set_global()
ax1.coastlines()
ax2 = plt.subplot(212, projection=ccrs.NorthPolarStereo())
ax2.set_extent([-180, 180, 20, 90], crs=ccrs.PlateCarree())
ax2.coastlines()

# ...as long as we provide the correct transform,
# the plot will be correct
ax1.contourf(lon, lat, data, transform=data_crs)
ax2.contourf(lon, lat, data, transform=data_crs)
plt.show()
```

Now that you have worked through the exercise, you should have an understanding of why it is important to fully define the coordinate reference system and the projection when creating a map.

2.5 Conversion vs. transformation

When working with data collected from several sources or in different coordinate reference systems, the data must be redefined to have the same coordinate reference system and datum. Consistent coordinate reference systems for data in a map is important because there may be spatial differences between the coordinate reference systems creating locational errors. A coordinate conversion is when the coordinate reference systems have the same datum. A coordinate transformation is when the coordinate reference systems have different datums.

When making a map, all of the data in the map should have the same coordinate reference system definition. Software include many definitions and transformations but refer to Snyder (1987) and International Association of Oil and Gas Producers (2018) for projection formulae (Iliffe and Lott, 2008). In Example 2, the Tissot's Indicatrix ellipses did not change coordinate reference systems. The method used to plot the ellipses changed. In Example 3, we also only changed the projection of the map, not the coordinate reference system. In the following examples, we will make coordinate conversions and transformations. For this purpose, we will use the pyproj Python library and examples from the pyproj documentation (Whitaker, 2019).

Installing pyproj

A version of pyproj was installed with Cartopy, but you need the latest version which has different dependencies than Cartopy. You will create a new environment.

1. Open Anaconda Navigator.
2. In the left panel, click on "Environments"
3. In the middle panel, click "Create" to create a new environment
4. Name this environment: ch2pyproj
5. Select Python version 3.7
6. Click "Create". This will take a few minutes
7. We need to install pyproj. In the right panel in the search field, type: pyproj
8. You will get the message "0 packages available matching pyproj" since pyproj is not installed.
9. In the right panel, change "Installed" to "Not installed"
10. pyproj now shows up. Click the check box next to "pyproj".
11. In the screen lower right corner, click "Apply"
12. Please wait while pyproj is collected and then click "Apply"

13. The pyproj library and any dependencies will be installed or updated. This may take a few minutes.

14. Click on "Home"

15. Install the console of your choice for the new environment. Click on "Install" under Jupyter Notebook, for example.

16. Click on "Launch"

Example 4: Coordinate conversion

The notebook ch2-3 contains this example. If you just launched Jupyter Notebook as indicated above, open the notebook. If you closed Anaconda, follow these steps:

1. Open Anaconda Navigator

2. Click on "Environments"

3. Choose "ch2pyproj"

4. Click "Home"

5. Launch Jupyter Notebook

6. Open the notebook ch2-3

We have a coordinate pair defined in decimal degrees of latitude and longitude. The longitude is -120.108°and latitude is 34.3611666°. We want to make a coordinate conversion from latitude and longitude to Universal Transverse Mercator, where the point is defined by east and north coordinates in meters. To learn more about Universal Transverse Mercator (UTM), refer to (Snyder, 1987). In the code, we use the pyproj *Proj* function. We can only use *Proj* when making a coordinate conversion (i.e. the same datum):

```
# Import pyproj
from pyproj import Proj

# Construct the projection matrix
```

```
p = Proj(proj='utm',zone=10,ellps='WGS84', preserve_units=
    False)

# Apply the projection to the lat-long point
x,y = p(-120.108, 34.36116666)

print(f'x={x:9.3f}, y={y:11.3f}')
```

Output:
x=765975.641, y=3805993.134

This is the same location but only expressed in east and north coordinates in meters using the UTM coordinate reference system. The datum used is WGS84. We can convert the UTM coordinates back to latitute and longitude by adding two lines of code:

```
# Apply the inverse of the projection matrix
# to the point in UTM
lon,lat = p(x,y,inverse=True)
print(f'lon={lon:8.3f}, lat={lat:5.3f}')
```

Output:
lon=-120.108, lat=34.361

In the two cells of code above, we've truncated the output to only three decimal places, but we can confirm that the inverse conversion arrives at the original pair. Let's now try converting several points of different latitude and longitude using a collection of objects in Python, or tuples. Add the following code:

```
# three points in lat-long
lons = (-119.72,-118.40,-122.38)
lats = (36.77, 33.93, 37.62 )
# Apply the projection to the points
x1,y1 = p(lons, lats)
print(x1,y1)
```

Output:
(792763.8631257227, 925321.5373562573, 554714.3009414743)
(4074377.6167697194, 3763936.9410883673, 4163835.3033114495)

Now, let's do a more advanced exercise: In the cartographic community, an easy way to communicate the coordinate reference system is to use the EPSG Geodetic Parameter Data set. Every coordinate reference system is given a code. This ensures that if someone uses UTM zone 10 North with datum WGS-84 and tells you UTM zone 10, that you do not accidentally use UTM zone 10 North with datum GRS80, for example.

Earlier in this exercise, we defined the UTM zone in the *Proj* function. Here, we will refer to the EPSG code. First, we will take a coordinate pair in longitude and latitude with datum WGS84 and convert it to EPSG:32667. Before proceeding, conduct a quick internet search on what EPSG:32667 means. This is important to understand what we will do next. The first part of the code is:

```
# silence warnings
import warnings
warnings.simplefilter('ignore')

# initial coordinate conversion
p = Proj(init='EPSG:32667', preserve_units=True)
# Apply the conversion to the lat-long point
x,y = p(-114.057222, 51.045)
print(f'x={x:9.3f}, y={y:11.3f} (feet)')
```

Output:
x=-5851386.754, y=20320914.191 (feet)

Let's dissect this as the pyproj code looks quite a bit different. The first part of the function Proj calls EPSG:32667. If you looked up EPSG:32667 online, you found that it is for UTM zone 17 North, but the units are in feet. The default mode for Proj is "preserve_units=False", which forces any unit to meters. However, we want to see the units in US Survey Feet as the projection defines; therefore, we change the argument to True.

Now, suppose we want to see the output in meters. How will you amend the code? Here is what you should add:

```
# Print the coordinate pair in meters
p1 = Proj(init='EPSG:32667', preserve_units=False)
x1,y1 = p1(-114.057222, 51.045)
print(f'x={x1:9.3f}, y={y1:11.3f} (meters)')
```

Output:
x=-1783506.250, y=6193827.033 (meters)

As discussed, you should change "preserve_units=False" and change the unit to be printed from "feet" to "meters". Congratulations! You now have a good understanding of coordinate conversion.

Example 5: Coordinate transformation

We learned earlier that we have a coordinate conversion where a coordinate pair is converted between coordinate reference systems with the same datum. In many instances, the coordinate reference system will also undergo a datum shift – this is a coordinate transformation.

This example is included in the notebook ch2-4. We will use the pyproj *CRS* and *transform* functions. The *CRS* function defines the coordinate reference system while the *transform* function specifies which coordinate reference system is the original and which is the output. *CRS* has the same ability to refer directly to an EPSG code.

The input coordinates are in EPSG:4326, which is a commonly used code. It is the geographic coordinate system with datum WGS84. The output coordinates are EPSG:31984, which is for UTM zone 24 S with datum SIRGAS2000.

```
# Import transform and CRS functions
from pyproj import transform
from pyproj import CRS

# input coordinates
c1 = CRS('EPSG:4326')
# coordinate pair
y1=-10.754283
x1=-39.866132
# output coordinates
c2 = CRS('EPSG:31984')
# Coordinate transformation
x2, y2 = transform(c1, c2, x1, y1)
print(f'x={x2:9.3f}, y={y2:11.3f}')
```

Output:
x=2930179.850, y=5185231.716

Example 6: Transforming several points at once

We have focused our examples on one coordinate pair at a time. The reality is that you will more often have several coordinates to transform at one time. The notebook ch2-5 explains how to do this.

We have a csv file with two columns: longitude and latitude. Each coordinate pair is the center of a volcano around the world. There are 1,509 volcanoes in our dataset. The original coordinate reference system is geographic coordinates with datum WGS84. We want to make a coordinate transformation of these data points to World Mercator. It will take much too long to manually transform these coordinates as we have done in the notebooks before. Therefore, our new code will read the csv file and create a new csv file.

Check that the pathway of *in_path* and *out_path* matches the directory where the csv file is. In this example, the volcanoes file (*volc_longlat.csv*) is in the directory data/ch2-5. Run the code, you will know the process is finished when the message "process completed" and the time of execution are returned:

```
# Import libraries
import csv, pyproj
from functools import partial
from os import listdir, path

# Time the execution of the code
import time
start_time = time.time()

# Remove warnings
import warnings
warnings.simplefilter('ignore')

# Define some constants at the top

lon = 'LONGITUDE' #name of longitude field in original files
lat = 'LATITUDE' #name of latitude field in original files
f_x = 'x' #name of new x value field in new projected files
f_y = 'y' #name of new y value field in new projected files
in_path = path.abspath('../data/ch2-5') #input directory
out_path = path.abspath('../data/ch2-5') #output directory
input_projection = 'EPSG:4326' #WGS84
output_projection = 'EPSG:3395' #World Mercator

# Get CSVs to reproject from input path
```

```
files= [f for f in listdir(in_path) if f.endswith('.csv')]

# Define partial function for use later when reprojecting
project = partial(
    pyproj.transform,
    pyproj.Proj(init=input_projection),
    pyproj.Proj(init=output_projection))

for csvfile in files:
    # Open a writer, appending '_project' onto the base name
    with open(path.join(out_path, csvfile.replace('.csv','
   _project.csv')), 'w') as w:
        # Open the reader
        with open(path.join( in_path, csvfile), 'r') as r:
            reader = csv.DictReader(r, dialect='excel')
            # Create new fieldnames list from reader
            # replacing lon and lat fields with
            # x and y fields
            fn = [x for x in reader.fieldnames]
            fn[fn.index(lon)] = f_x
            fn[fn.index(lat)] = f_y
            writer = csv.DictWriter(w, fieldnames=fn)
            # Write the output
            writer.writeheader()
            for row in reader:
                x,y = (float(row[lon]), float(row[lat]))
                try:
                    # Add x,y keys and remove lon, lat keys
                    # project point
                    row[f_x], row[f_y] = project(x, y)
                    row.pop(lon, None)
                    row.pop(lat, None)
                    writer.writerow(row)
                except Exception as e:
                    # If coordinates are out of bounds,
                    # skip row and print the error
                    print (e)
print('process completed')
end_time = time.time()
print("it took {} seconds to run the code".format(end_time-
   start_time))
```

Output:
process completed
it took 55.04537224769592 seconds to run the code

It takes about 55 seconds to run this code in a standard computer. Check the

newly created csv file and notice that you now have a listing of coordinates in meters. The EPSG definition of the output coordinate reference system is listed under *output_projection*. You can easily change this variable to another EPSG and rerun the script. Notice that the code is written so that every csv file in the directory will undergo a coordinate transformation.

2.6 Exercises

1. So far, you have worked with data sets that have been provided to you. It is an important skill to be able to find data sets online and to prepare them for use in your work.

   The United States Geological Survey (USGS) Earthquake Hazards Program monitors, records, and maintains a global database of earthquake activity. The public can query the archive and download earthquake epicenter data. Go to the Search Earthquake Catalog page of the USGS. Using the Basic Options, download all of the magnitude 4.5+ earthquakes in the last year in the world. In the Output Options, choose a CSV format.

   (a) Modify the notebook ch2-5 to transform the earthquake epicenters from latitude-longitude to World Mercator (or another map projection of your choice). Use this pictographic for further information. The datum of the earthquake epicenters is likely WGS84.

   (b) Plot the earthquake epicenters. Make sure to include the outline of the continents. *Hint*: Look at the notebook ch2-2 for a starting point, and check the Cartopy website for examples of how to plot localities on a map.

   (c) Now modify your plot to color the earthquakes by depth (red are shallow and blue are deep earthquakes), and the size of the points by the earthquake magnitude.

   (d) Add the volcanoes from Example 6 (*volc_longlat.csv*) to the map (use triangles to indicate volcanoes). Do you see any correlation between the earthquake epicenters and the volcanoes?

References

C.I.U. AND WAR OFFICE. 1944. Town Plan of Roma (Rome) (North Sheet), 1:10,000. Washington, D. C.: War Office.

Cartopy. 2018a. Projections [Online]. UK: SciTools. [Accessed 19 November, 2019]

Cartopy. 2018b. Tissot's Indicatrix [Online]. UK: SciTools. [Accessed 19 November, 2019].

Cartopy. 2018c. Understanding the Transform and Projection Keywords [Online]. UK: SciTools. [Accessed 19 November, 2019].

Geographic Section - General Staff. 1941. Aberdeen, 1:1,000,000. Great Britain: War Office.

Iliffe, J. and Lott, R. 2008. Datums and Map Projections: For Remote Sensing, GIS, and Surveying, Dunbeath, Scotland, Whittles.

International Association of Oil and Gas Producers. 2018. Geomatics Guidance Note 7, Part 2 Coordinate conversions and Transformations including Formulas.

Kraak, M.J. and Ormeling, F.J. 2003. Cartography: visualization of geospatial data. Addison Wesley.

Lexico. 2019. Geodesy [Online]. Oxford. [Accessed August, 2019].

Lisle, R. J., Brabham, P. and Barnes, J. W. 2011. Basic Geological Mapping, Chicester, UK, Wiley-Blackwell.

Robinson, A. H., Morrison, J. L., Muehrcke, P. C., Kimerling, A. J. and Guptill, S. C. 1995. Elements of Cartography, New York, Wiley.

Snyder, J. P. 1987. Map Projections: a working manual. Geological Survey Professional Paper. Washington, D. C., U.S.A.: United States Government Printing Office.

Watson, L. 2017. Spatial-based assessment at continental to global scale: case studies in petroleum exploration and ecosystem services. PhD, Utrecht University.

Whitaker, J. 2019. pyproj Transformer Documentation [Online]. [Accessed 7 January, 2020].

Chapter 3

Geologic features

3.1 Primitive objects: Lines and planes

The fundamental geometric features of geology are lines (e.g. a lineation or a fold axis) and planes (e.g. bedding or a foliation). A *line* is the element generated by a moving point. It can be straight or curved. We will treat straight lines here. A *plane* is a flat surface; a line joining two points on the plane lies wholly on its surface, and two intersecting lines on the plane define the plane. This is equivalent to say that three non-collinear points on the plane define the plane (this is the principle of the well-known three-point problem). Obviously, linear features can be curved (e.g. the intersection of bedding with irregular topography), and surfaces can be non-planar (e.g. bedding on a fold). However, even these more complex cases can be expressed as a collection of lines and planes.

3.2 Lines and planes orientations

Two important properties of lines and planes are location (chapter 2) and orientation (this chapter). Lines and planes orientations are measured with respect to the geographic north and the angle downward or upward from the horizontal. We refer to this coordinate system as the spherical coordinate system, and the measurements defining the lines or planes orientations as the spherical coordinates.

3.2.1 Planes: Strike and dip

A plane orientation can be defined by the angle a horizontal line on the plane makes with the geographic north, known as the *strike* and the maximum angle measured downward from the horizontal to the plane, known as the *dip* (Fig 3.1a). The strike is measured as an azimuth, an angle between 0 and 360°(0 = north, 90 = east, 180 = south, 270 = west). The dip is an angle between 0 (horizontal plane) and 90°(vertical plane). The projection of the dip onto the horizontal is known as the *dip direction* and is always 90°from the strike. However, is the dip direction plus or minus 90°the given strike? Which end of the strike line should we use? To avoid ambiguities, we will use a format known as the *right hand rule* (RHR). In the RHR format, one gives the strike such that the dip direction is always the strike plus 90°, i.e. the dip direction is to the right of the strike (Fig. 3.1a).

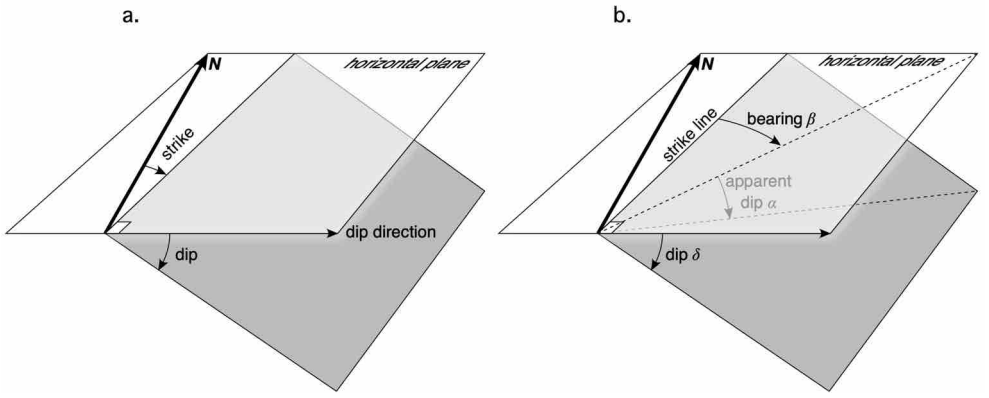


Figure 3.1: **a.** Strike and dip of a plane, **b.** Apparent dip of a plane. Modified from Allmendinger et al. (2012).

It is only along the dip direction that the true dip can be measured, any other direction will give a lower apparent dip (Fig. 3.1b). The relation between the dip (δ) and the apparent dip (α) is given by the equation:

$$\tan\alpha = \tan\delta \sin\beta \tag{3.1}$$

where β is the angle between the strike (horizontal) line on the plane and the vertical section on which the apparent dip is measured (Fig. 3.1b). This is also Eq. 1.3, which we plotted in problem 4 of chapter 1 (Fig. 1.1). You can quickly verify that it works by setting $\beta = 0$ (a cross section parallel to

strike) which gives $\alpha = 0$ (since sin(0) is 0), and $\beta = 90°$(a cross section perpendicular to strike) which gives $\alpha = \delta$ (since sin(90) is 1). This leads to a very important observation: *Any plane on a vertical section parallel to strike looks horizontal (even if it's dipping), and the true dip of the plane can only be observed on a vertical section perpendicular to strike.* This is why we should always visualize planes (bedding, faults, etc.) on cross sections perpendicular to strike.

3.2.2 Lines: Trend and plunge or rake

The orientation of a line is specified by the azimuth of the horizontal projection of the line, or *trend*, and the vertical angle measured downward from the horizontal to the line, or *plunge* (Fig. 3.2a). The plunge has a range between -90 and 90°. Positive plunge indicates lines pointing downwards, and negative plunge lines pointing upwards. To measure the trend and plunge one must determine the vertical plane containing the line. This is quite difficult and often results in errors (section 3.2.5). For this reason, and if the line is on a plane, it is more convenient (and accurate) to measure the angle on the plane between the strike line and the line. This angle is known as the rake or pitch (Fig. 3.2b). To avoid any confusion, the rake should be always measured from the given strike and thus it varies between 0 and 180°.

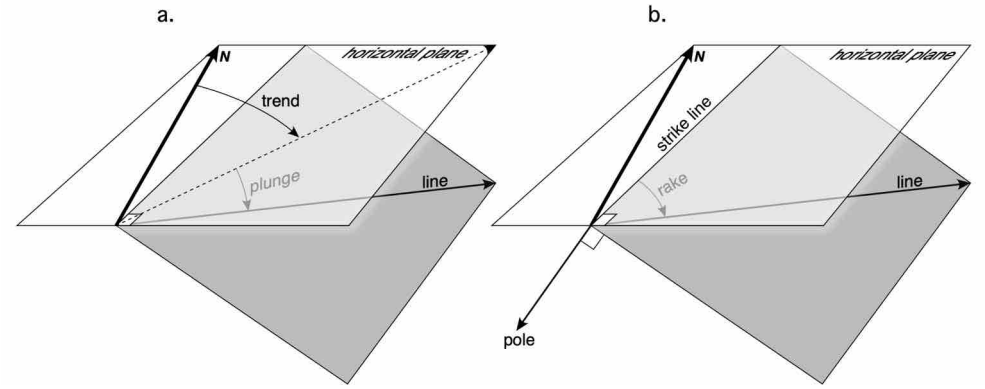


Figure 3.2: **a.** Trend and plunge of a line, **b**. Rake of a line and pole to a plane. Modified from Allmendinger et al. (2012).

3.2.3 The pole to the plane

Any plane can be uniquely represented by its downward normal. This line is known as the pole to the plane (Fig. 3.2b). If we use the RHR format, the orientation of the pole is given by:

$$\begin{aligned} trend\,of\,pole &= strike\,of\,plane - 90^\circ \\ plunge\,of\,pole &= 90^\circ - dip\,of\,plane \end{aligned} \tag{3.2}$$

The pole facilitates analyzing planes graphically and by computation. Our first function *Pole* computes the pole to a plane ($k = 1$) or the plane from its pole ($k = 0$). It is followed by the helper function *ZeroTwoPi* which makes sure azimuths are always between 0 and 360°. Notice that angles (*trd* and *plg*) should be entered in radians, and the plane must follow the RHR format.

```
import math
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def Pole(trd,plg,k):
  '''
  Pole returns the pole to a plane or the plane from a pole

  If k = 0, Pole returns the strike (trd1) and dip (plg1)
  of a plane, given the trend (trd) and plunge (plg)
  of its pole.

  If k = 1, Pole returns the trend (trd1) and plunge (plg1)
  of a pole, given the strike (trd) and dip (plg)
  of its plane.

  NOTE: Input/Output angles are in radians.
  Input/Output strike and dip follow the RHR format

  Pole uses function ZeroTwoPi
  '''
  # Some constants
  east = math.pi/2

  # Eq. 3.2
  # Calculate plane given its pole
  if k == 0:
    if plg >= 0:
      plg1 = east - plg
```

```
      trd1 = ZeroTwoPi(trd + east)
    else: # Unusual case of pole pointing upwards
      plg1 = east + plg
      trd1 = ZeroTwoPi(trd - east)
  # Else calculate pole given its plane
  elif k == 1:
    plg1 = east - plg;
    trd1 = ZeroTwoPi (trd - east)

  return trd1, plg1
```

```
import math

def ZeroTwoPi(a):
  '''
  This function makes sure input azimuth (a)
  is within 0  to 2*pi (b)

  NOTE: Azimuths a and b are input/output in radians

  Python function translated from the Matlab function
  ZeroTwoPi in Allmendinger et al. (2012)
  '''
  b=a
  twopi = 2*math.pi
  if b < 0:
    b += twopi
  elif b >= twopi:
    b -= twopi

  return b
```

3.2.4 Instruments used in the field

Traditionally, geologists use a geological compass/clinometer to measure the orientation of planes and lines in the field. Figure 3.3 shows four of the most common compasses used in geology: the Silva compass (Fig. 3.3a), the Brunton compass (Fig. 3.3b), the Krantz compass (Fig. 3.3c, a less expensive variant of the Freiberg compass), and the Brunton Geo compass (Fig. 3.3d). All these compasses have a magnetic needle that points to the magnetic north (N or white end of the needle), a horizontal level, and a clinometer (an instrument to measure vertical angles). The Silva compass has an azimuth scale that can be rotated to follow the magnetic needle, while

in the other three compasses the azimuth scale is fixed. This is why east-west (E-W) are in the right place in the Silva compass, while they are flipped in the other three compasses (Fig. 3.3a-d).

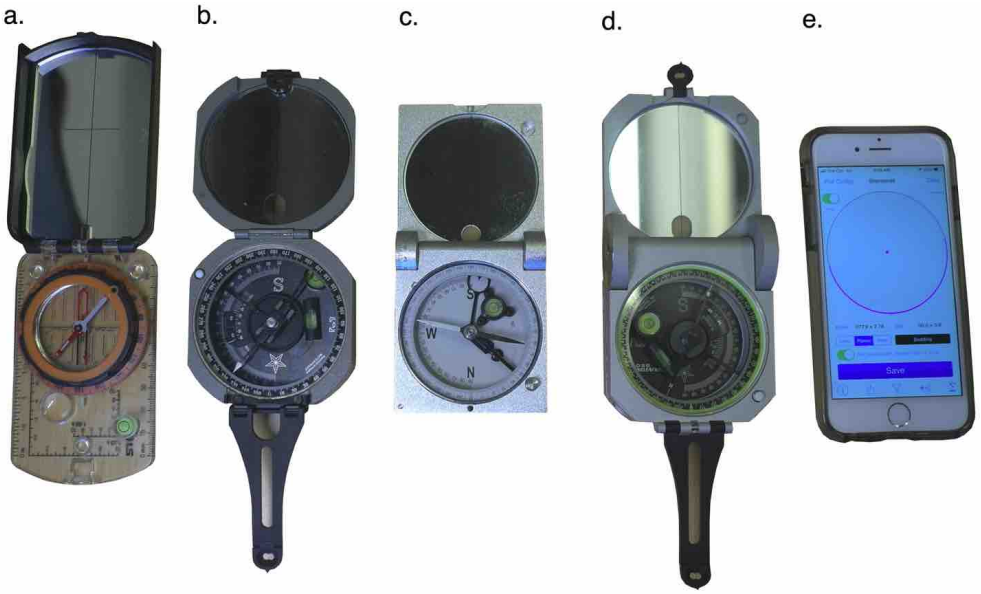


Figure 3.3: **a.** Silva, **b.** Brunton, **c.** Krantz, **d.** Brunton Geo, and **e.** Smartphone with Stereonet Mobile.

The Silva and Brunton compasses are designed to measure strike and dip through two measurements, while the Krantz compass measures dip direction and dip at once. The Brunton Geo compass can work either as a Brunton or Krantz compass, and it has higher precision than the other three compasses (it is also the most expensive). The use of these compasses is well explained in field geology books such as Compton (1985) and Coe (2010). For illustration, Figure 3.4 shows how strike and dip are measured with the Brunton compass (the same principles apply to the Silva compass). Notice that in this measurement, it is crucial to determine when the compass is horizontal (Fig. 3.4a). We will see that this can be a source of error (section 3.2.5).

These days, digital devices in the form of smartphone programs or apps (Fig. 3.3e) are slowly replacing the analog compasses. Smartphones contain instruments such as accelerometers, gyroscopes, and magnetometers, which enable apps such as Stereonet Mobile (Richard Allmendinger) or Fieldmove Clino (Petroleum Experts) to determine the exact orientation of the device in space. Measuring a plane or a line just requires placing the phone on the plane or along the line. Thus, one can capture a large number of measure-

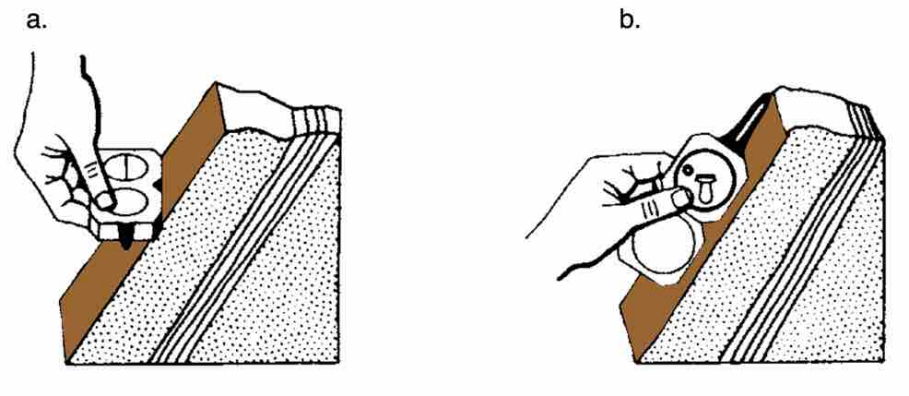


Figure 3.4: Measuring the **a.** strike and **b.** dip of a plane. Modified from Compton (1985).

ments quickly. However, smartphones are very sensitive to nearby magnetic fields and one can easily get spurious results (Novakova and Plavlis, 2017). Smartphones also have access to accurate geographic location (GPS, cell and wireless networks) as well as satellite imagery and raster data such as elevation. They can greatly facilitate mapping in the field.

3.2.5 Uncertainties in orientations

Geological planes and lines are irregular and therefore it is difficult to take exact measurements of them. Every plane or line measurement has an uncertainty (an error). There are different ways to try to reduce this error, either by placing a smooth planar object (e.g. a field notebook) on the plane or along the line, or by sighting the plane or line from the distance (Compton, 1985). Figure 3.5 illustrates the error associated to the strike and dip measurement of a plane. If the compass is not exactly horizontal then a direction other than the strike will be measured. The departure of the compass from the horizontal or operator error (ε_o) will give a strike error (ε_s).

From the three right-triangles and their corresponding equations in Figure 3.5, and by substituting the first two equations for w and l into the third equation for ε_s, one gets the following relation (Woodcock, 1976):

$$\sin \varepsilon_s = \frac{\tan \varepsilon_o}{\tan \delta} \tag{3.3}$$

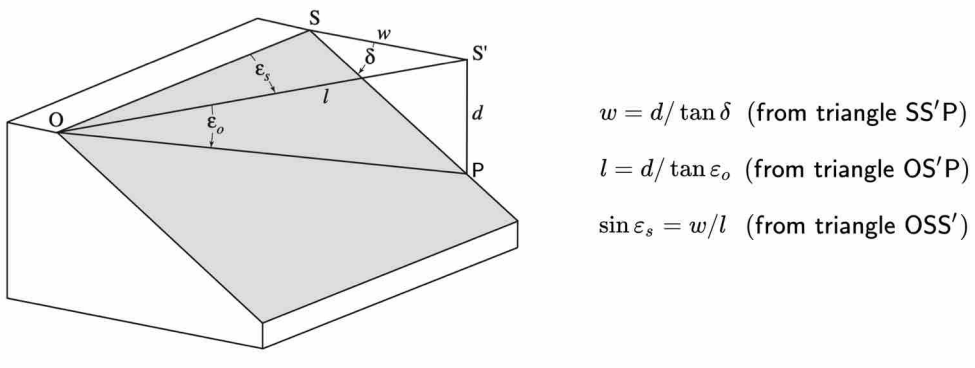


Figure 3.5: Geometrical relations for estimating the strike error ε_s from the operator error (or departure of the compass from the horizontal) ε_o. Modified from Ragan (2009).

where δ is the dip angle of the plane. This equation is plotted in Figure 3.6 for dip angles δ of 0 to 40°and operator errors ε_o of 1 to 5°. It is clear that the strike error ε_s increases with decreasing dip. For a gentle 5°dipping plane, an operator error ε_o of 2°(a compass just 2°off the horizontal) results in a strike error ε_s of about 24°! Thus, one should always be suspicious about the accuracy of strike and dip measurements, particularly if they are from gently dipping planes.

For line measurements, the situation is not better. When measuring the orientation of a line, it is common practice to align the compass in the direction of the horizontal projection of the line, which as anyone who has tried this in the field knows, is quite difficult. There will be an operator error and the measured trend β' will differ from the true trend β (Fig. 3.7a). The trend error ε_t ($|\beta' - \beta|$)in terms of the angle on the plane ε_o which the measured line makes with the true line, is given by the following equations (Woodcock, 1976):

$$\begin{aligned} \tan\varepsilon_t &= \frac{[\tan(r+\varepsilon_o) - \tan(r)]\cos\delta}{1 + [\tan(r+\varepsilon_o)\tan(r)]\cos^2\delta} \quad if\beta' > \beta \\ \tan\varepsilon_t &= \frac{[\tan(r) - \tan(r-\varepsilon_o)]\cos\delta}{1 + [\tan(r)\tan(r-\varepsilon_o)]\cos^2\delta} \quad if\beta' < \beta \end{aligned} \tag{3.4}$$

where r is the rake of the line, and δ is the dip of the plane (Fig. 3.7a).

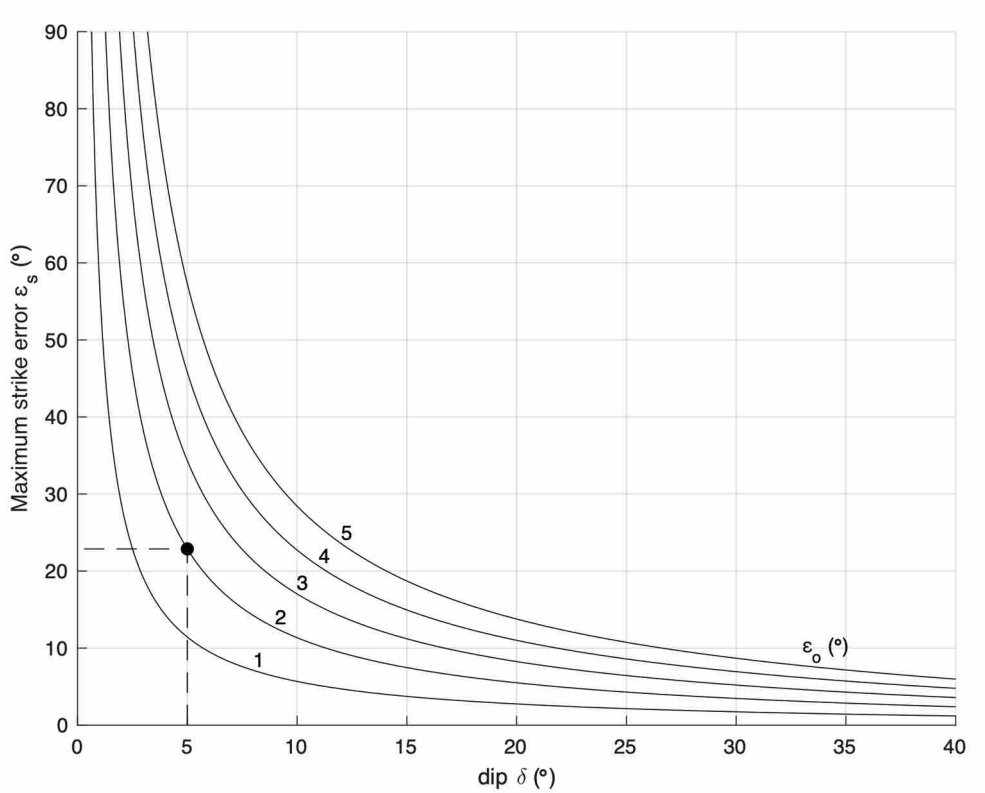


Figure 3.6: Strike error ε_s as a function of dip δ for values of operator error ε_o of 1-5°. The notebook that produced this graph is available from the resource git repository.

These equations are plotted in Figure 3.7b-c for an ε_o of 3°. The trend error is greater for a measured line on the down-dip ($\beta' > \beta$) side of the line (Fig. 3.7b), than for a measured line on the up-dip ($\beta' < \beta$) side of the line (Fig. 3.7c). This means that repeated measurements will not be symmetrically distributed around the true trend β. Also for a given ε_o, the trend error ε_t increases with the dip δ of the plane and the rake r of the line, i.e. a combination of a steep plane and a large rake may result in a large trend error.

Equations 3.3 and 3.4 allow determining the uncertainties associated to the measurement of planes and lines. As we will see in section 4.5, these errors propagate in any computation making use of these orientations.

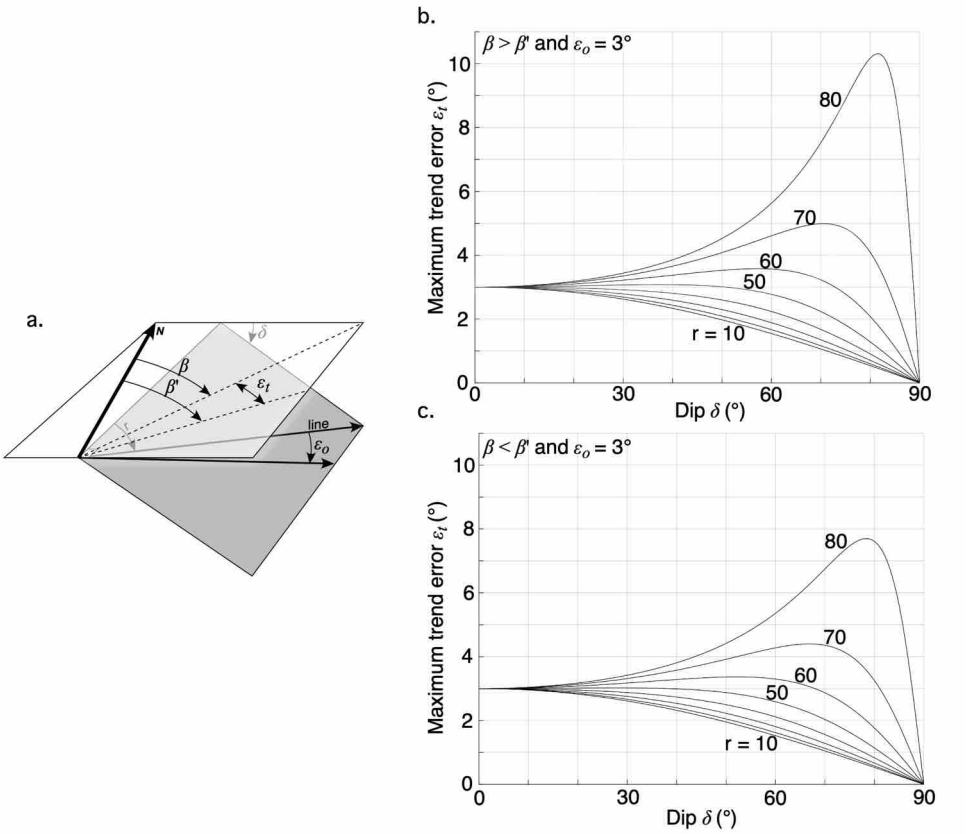


Figure 3.7: **a.** Geometrical relations for estimating the trend error ε_t from the rake r of the line, the dip δ of the plane, and the angle on the plane ε_o between the measured and the true lines. Trend error as function of dip for **b.** a measured line on the down-dip side of the line, and **c.** a measured line on the up-dip side of the line. In b and c, ε_o is 3°. The notebooks that produced graphs b and c are available from the resource git repository.

3.3 Displaying geologic features

There are two fundamental ways geologists display geologic features: maps and stereonets. In maps, we are concerned about the location and orientation of the features, and the spatial relation of one feature to another. In stereonets, we are just concerned with the orientation of the features.

3.3.1 Maps

All maps are a projection of surface or subsurface geologic features onto a horizontal plane. In section 2.3, we looked at the different methods used to project data from the approximately spherical Earth to a flat surface, and the distortions associated to these methods. Geologic features (bedding, faults, the ground surface) are rarely flat, and therefore to display the spatial variation of their elevation (or depth) on maps, we use contours. A contour line is a line joining the points in the map area of equal value for a specific parameter. On a topographic map, for example, contour lines join points of equal elevation on the ground surface. Contour lines should not cross (unless very unusual circumstances) or disappear in the middle of the map (unless the contoured feature is intersected by another). If the difference in value between adjacent contours or contour interval is held constant throughout the map, the gradient (rate of change) of the parameter in a given direction is proportional to the spacing of the contour lines: high gradient is represented by closely spaced contours, and low gradient by widely spaced contours. This is expressed by the following relation:

$$gradient = \arctan \frac{parameter\ change\ between\ contours}{map\ distance\ between\ contours} \tag{3.5a}$$

For a topographic map, this relation becomes:

$$slope\ angle = \arctan \frac{elevation\ change\ between\ contours}{map\ distance\ between\ contours} \tag{3.5b}$$

which is why when choosing the walking path to a high ground area, you should look for the widely spaced contours (unless you are a climber or a goat).

Geologic features are rarely isolated, and they usually have different orientations, so we should expect them to intersect. The intersection of two non-parallel planes (e.g. bedding contacts) is a straight line. In chapter 4, we will see how to determine this type of intersection using vector operations. If one of the surfaces is not planar but is irregular, the intersection is a curved line which is more difficult to determine. One of the most fundamental mapping problem geology students are early confronted with is the intersection of a planar feature (e.g. bedding or a fault) with the irregular land surface.

This is elegantly summarized by the *Rule of V's* (Fig. 3.8).

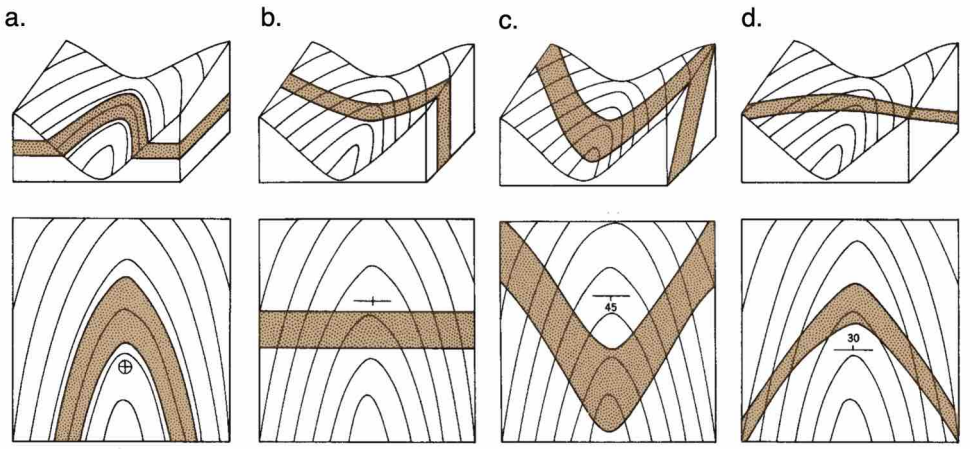


Figure 3.8: Outcrop pattern across a valley of **a.** Horizontal bed, **b.** Vertical bed, **c.** Bed dipping downstream, and **d.** Bed dipping upstream. Modified from Ragan (2009).

The Rule of V's says that when a planar contact crosses a valley, its outcrop pattern will V or curve in the direction that the contact is dipping, but only if the contact if steeper than the slope of the valley, which is normally the case (Fig. 3.8c-d). There are two exceptions: 1. If the contact is horizontal, its outcrop pattern will follow the topographic contours, which makes sense since the contours are the intersection of horizontal planes of different elevation with the ground (Fig. 3.8a), and 2. If the contact is vertical, its outcrop pattern across the valley is a straight line. Vertical planes *ignore* topography.

Determining the outcrop trace of a planar contact on irregular topography is not straightforward. Graphically, this problem involves making elevation contours on the planar contact. These are called structure contours. Then one should look at the locations where the structure contours of the contact have the same elevation than the topographic contours of the land surface. On these locations, the contact outcrops. Finally, one should join these locations with a line, to make the outcrop trace of the contact. Figure 3.9 illustrates this procedure for a plane dipping north and intersecting irregular topography. Notice how in the stream valleys, the outcrop trace of the plane curves to the north, clearly following the Rule of V's.

This graphical approach requires a great deal of patience and drawing skills. Later in section 5.2.2, we sill see that if we know the plane's orientation and one outcrop location, it is possible to project the plane throughout the

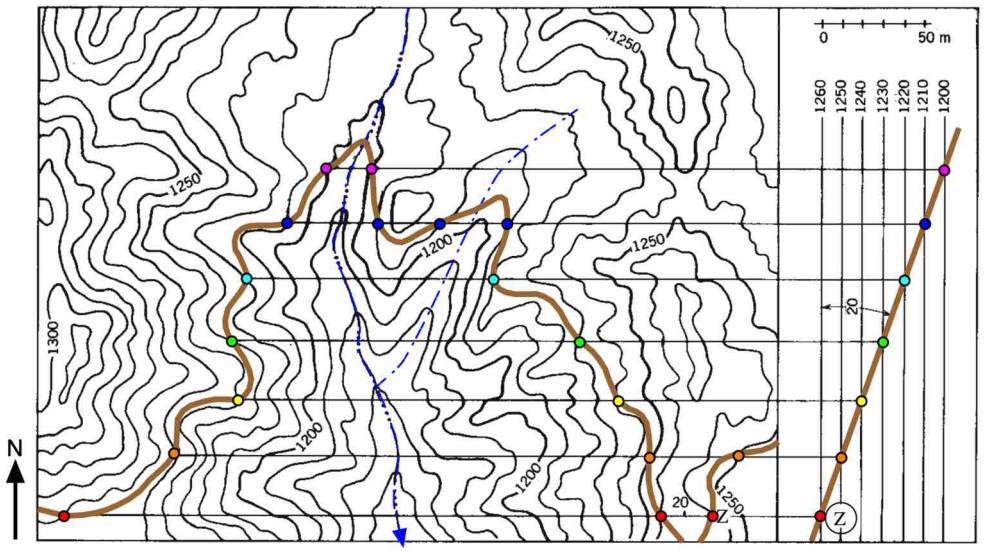


Figure 3.9: Outcrop trace of a plane dipping 20°N. The left figure is the map, and the right figure is a N-S cross section. Color points are the locations where the plane's structure contours have the same elevation than the topographic contours. The line joining these points is the outcrop trace of the plane. Modified from Ragan (2009).

terrain using computation, provided we have a digital elevation model (DEM) of the terrain. This saves a lot of time and it's a great way to quality control mapping, test different hypotheses, and take better decisions in the field.

3.3.2 Stereonets

Spherical projections can be used to represent the orientation of a plane or a line if the plane or line is positioned so that it passes through the center of the sphere. A plane will intersect the sphere along a great circle, and a line will pierce the sphere at a point (Fig. 3.10a). Obviously, it would be inconvenient to carry a sphere everywhere. Fortunately, it is possible to project the sphere onto a plane using, for example, an azimuthal projection (section 2.3).

A *stereonet* or stereographic projection is a special kind of azimuthal projection, where the point source or viewpoint lies on the surface of the sphere, and the projection plane passes through the center of the sphere. In a stere-

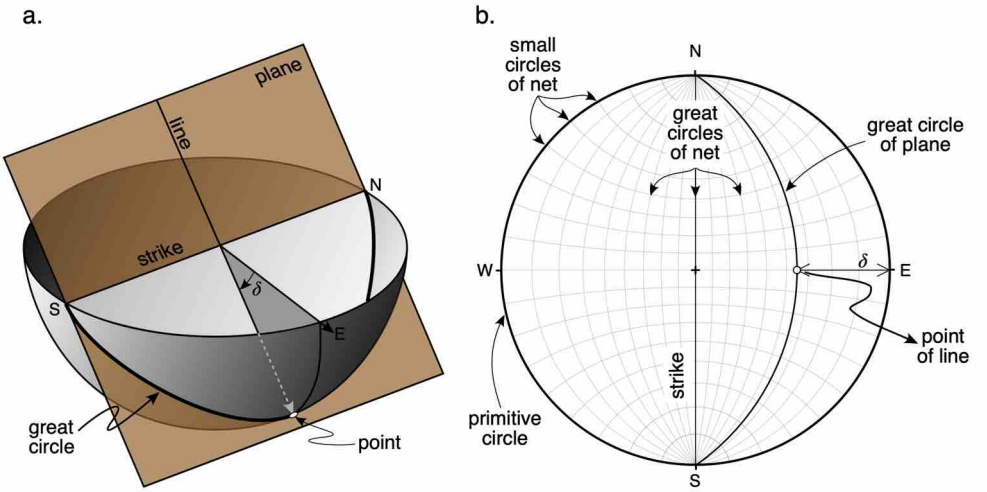


Figure 3.10: **a.** Plane and line intersecting the lower half of a sphere. The rake of the line is 90°and therefore its plunge is equal to the plane's dip δ. **b.** Lower hemisphere stereographic projection of plane and line. Modified from Allmendiger et al. (2012) and Allmendinger (2019).

onet, the viewpoint is at the top of the sphere or zenith, the view direction is downwards, the projection plane is the equatorial plane dividing the sphere into lower and upper hemispheres, and the lower hemisphere (bowl in Fig. 3.10a) is projected. In the stereonet, the rim of the bowl is called the primitive circle and it represents a horizontal plane (Fig. 3.10b). A net consisting of great circles representing N-S striking, 0-90°E and W dipping planes, and small circles representing cones of N-S horizontal axis and 0-90°apical radius opening to the S and N, helps drawing any plane or line (Fig. 3.10b). Several books explain how to do this and solve orientation problems (including rotations) using the stereonet (e.g. Marshak and Mitra, 1988).

For our purpose, it is more important to know how this projection actually works. Figure 3.11a illustrates this on a vertical section passing through the center of the sphere. Any line from the zenith (the top of the sphere) pinches the equatorial plane at one point, and this is the location where the point plots in the stereonet. This is defined by the following equation:

$$x = R \tan\left(45° - \frac{\phi}{2}\right) \tag{3.6}$$

where x is the is the distance of the point from the center of the net, R is

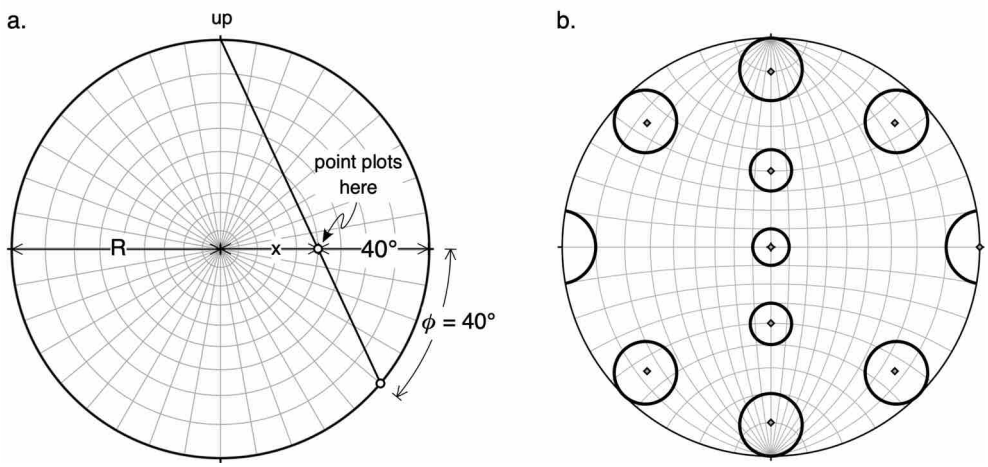


Figure 3.11: **a.** The equal angle stereonet illustrated on a vertical plane passing through the center of the sphere. **b.** Lower hemisphere equal angle projection of small circles of 10°radius but different axis orientations. Modified from Allmendinger et al. (2012).

the radius of the net, and ϕ is the plunge of the line. This method preserves angles perfectly and thus, on the primitive circle, degrees are equally spaced, and a small circle will be a circle anywhere on the net (Fig. 3.11b). This is why this projection is called the equal angle or Wulff stereonet. However, the preservation of angles has a disadvantage: areas are distorted. Thus, for example, a 10°radius small circle will look smaller near the center of the net but larger near the edges (Fig. 3.11b). This poses a problem when trying to assess visually or graphically the density of points plotted on the net.

The equal area or Schmidt net (Fig. 3.12) overcomes this problem. Strictly speaking, this projection is not a stereographic projection because the projection plane is at the bottom of the sphere. The point of intersection of the line and the surface of the lower hemisphere, is projected to the horizontal plane at the lowest point of the sphere, along a circular arc centered at the bottom of the sphere. The x distance of the point is then scaled by a factor of $\sqrt{2}$ to fit the radius R of the net (Fig. 3.12a). This is expressed by the following equation:

$$x = R\sqrt{2}\sin\left(45^\circ - \frac{\phi}{2}\right) \tag{3.7}$$

The tradeoff is that angles are no longer preserved, and small circles are no

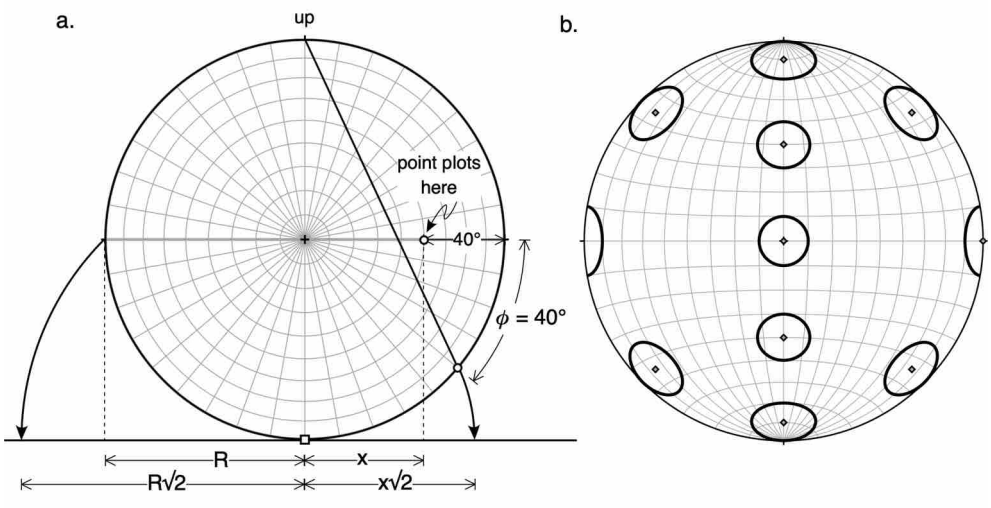


Figure 3.12: **a.** The equal area stereonet illustrated on a vertical plane passing through the center of the sphere. **b.** Lower hemisphere equal area projection of small circles of 10°radius but different axis orientations. Modified from Allmendinger et al. (2012).

longer true circles (Fig. 3.12b). The equal angle or Wulff net is used in problems where visualizing correctly angles on the net is important such as in crystallography and geography, while the equal area or Schmidt net is used in cases where analyzing the concentration of points on the net is important such as in structural analysis.

The function *StCoordLine* computes the coordinates of a line in an equal angle or an equal area net (equations 3.7 and 3.8). Notice that angles (*trd* and *plg*) should be entered in radians.

```
import math
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def StCoordLine(trd,plg,sttype):
  '''
  StCoordLine computes the coordinates of a line
  in an equal angle or equal area stereonet of unit radius

  trd = trend of line
  plg = plunge of line
  sttype = Stereonet type: 0 = equal angle, 1 = equal area
  xp and yp = Coordinates of the line in the stereonet

  NOTE: trend and plunge should be entered in radians
```

```
15
16    StCoordLine uses function ZeroTwoPi
17
18    Python function translated from the Matlab function
19    StCoordLine in Allmendinger et al. (2012)
20    '''
21    # Take care of negative plunges
22    if plg < 0:
23      trd = ZeroTwoPi(trd+math.pi)
24      plg = -plg
25
26    # Some constants
27    piS4 = math.pi/4
28    s2 = math.sqrt(2)
29    plgS2 = plg/2
30
31    # Equal angle stereonet, Eq. 3.6
32    if sttype == 0:
33      xp = math.tan(piS4 - plgS2)*math.sin(trd)
34      yp = math.tan(piS4 - plgS2)*math.cos(trd)
35    # Equal area stereonet, Eq. 3.7
36    elif sttype == 1:
37      xp = s2*math.sin(piS4 - plgS2)*math.sin(trd)
38      yp = s2*math.sin(piS4 - plgS2)*math.cos(trd)
39
40    return xp, yp
```

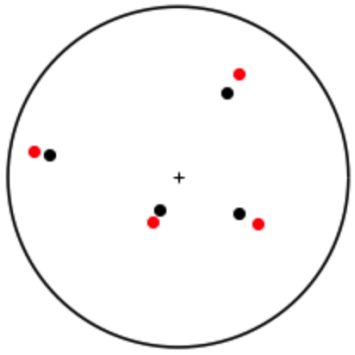
3.3.3 Plotting lines and poles in a stereonet

The notebook ch3 illustrates the use of the *StCoordLine* and *Pole* functions to plot lines and poles to planes on an equal angle or an equal area stereonet. You will get the chance to practice more with these functions in section 3.4.

```
1  # Import libraries
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Import functions Pole and StCoordLine
6  import sys, os
7  sys.path.append(os.path.abspath('../functions'))
8  from Pole import Pole as Pole
9  from StCoordLine import StCoordLine as StCoordLine
10
11 # Plot the following four lines (trend and plunge)
12 # on an equal angle or equal area stereonet
```

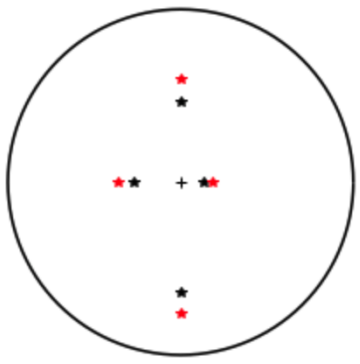
```
lines = np.array([[30, 30],[120, 45],[210, 65],[280, 15]])
pi = np.pi
linesr = lines * pi/180 # lines in radians

# Plot the primitive of the stereonet
r = 1; # unit radius
TH = np.arange(0,360,1)*pi/180
x = r * np.cos(TH)
y = r * np.sin(TH)
plt.plot(x,y,'k')
# Plot center of circle
plt.plot(0,0,'k+')
# Make axes equal and remove them
plt.axis('equal')
plt.axis('off')

# Find the coordinates of the lines in the
# equal angle or equal area stereonet
nrow, ncol = lines.shape
eqAngle = np.zeros((nrow, ncol))
eqArea = np.zeros((nrow, ncol))

for i in range(nrow):
    # Equal angle coordinates
    eqAngle[i,0], eqAngle[i,1] = StCoordLine(linesr[i,0],
   linesr[i,1],0)
    # Equal area coordinates
    eqArea[i,0], eqArea[i,1] = StCoordLine(linesr[i,0],linesr
   [i,1],1)

# Plot the lines
# Equal angle as black dots
plt.plot(eqAngle[:,0],eqAngle[:,1],'ko')
# Equal area as red dots
plt.plot(eqArea[:,0],eqArea[:,1],'ro');
```

Output:



```
# Plot the following four planes (strike and dip, RHR)
# as poles on an equal angle or equal area stereonet
planes = np.array([[0, 30], [90, 50], [180, 15], [270, 65]])
planesr = planes * pi/180 # planes in radians

# Plot the primitive of the stereonet
plt.plot(x,y,'k')
# Plot center of circle
plt.plot(0,0,'k+')
# Make axes equal and remove them
plt.axis('equal')
plt.axis('off')

# Find the coordinates of the poles to the planes in the
# equal angle or equal area stereonet
for i in range(nrow):
    # Compute pole of plane
    trend, plunge = Pole(planesr[i,0],planesr[i,1],1)
    # Equal angle coordinates
    eqAngle[i,0], eqAngle[i,1] = StCoordLine(trend,plunge,0)
    # Equal area coordinates
    eqArea[i,0], eqArea[i,1] = StCoordLine(trend,plunge,1)

# Plot the poles
# Equal angle as black asterisks
plt.plot(eqAngle[:,0],eqAngle[:,1],'k*')
# Equal area as red asterisks
plt.plot(eqArea[:,0],eqArea[:,1],'r*');
```

Output:



3.4 Exercises

1. Modify the notebook that makes Fig. 3.6 to extend the range of dip δ angles from 0 to 90°and the operator error ε_o from 1 to 10°.

2. Modify the notebooks that make Fig. 3.7 b and c for an ε_o of 5°.

3. You can draw a great circle on a stereonet by plotting closely spaced points along the great circle. These are lines on the plane. The following arrays contain the trend and plunge of lines on a plane of orientation 030/40 (strike and dip, RHR format):

   trend = [30, 34, 38, 42, 46, 50, 54, 58, 63, 67, 72, 78, 83, 89, 95, 101, 107, 113, 120, 127, 133, 139, 145, 151, 157, 162, 168, 173, 177, 182, 186, 190, 194, 198, 202, 206, 210]

   plunge = [0, 3, 6, 10, 13, 16, 19, 22, 24, 27, 29, 32, 34, 36, 37, 38, 39, 40, 40, 40, 39, 38, 37, 36, 34, 32, 29, 27, 24, 22, 19, 16, 13, 10, 6, 3, 0]

   Plot these lines on an equal angle and an equal area stereonet. From the resulting great circle, can you guess how a plane of orientation 050/60 (RHR) would look like on the stereonet?

4. You can also draw a small circle on a stereonet by plotting closely spaced points along the small circle. These are lines on the conical surface. The following arrays contain the trend and plunge of lines on a small circle of axis 050/30 (trend and plunge) and radius 20°:

   trend = [50, 53, 57, 60, 63, 66, 68, 70, 72, 73, 73, 73, 72, 70, 68, 64, 60, 55, 50, 45, 40, 36, 32, 30, 28, 27, 27, 27, 28, 30, 32, 34 37, 40, 43, 47, 50]

   plunge = [10, 10, 11, 12, 14, 16, 19, 22, 25, 28, 31, 35, 38, 41, 44, 47, 48, 50, 50, 50, 48, 47, 44, 41, 38, 35, 31, 28, 25, 22, 19, 16 14, 12, 11, 10, 10]

   Plot these lines on an equal angle and an equal area stereonet. What are the differences between the small circle in the equal angle and equal area stereonets?

5. The strike and dip arrays below contain the strike and dip (RHR format) of 50 bedding surfaces in a fold:

   strike = [8, 22, 19, 33, 27, 37, 41, 47, 55, 40, 32, 55, 65, 68, 89, 79, 102, 105, 108, 122, 132, 136, 145, 159, 156, 164, 176, 169, 179, 173, 167, 160, 145, 148, 141, 125, 108, 92, 75, 57, 50, 39, 22, 10, 1, 9, 15, 16, 114, 78]

dip = [75, 79, 68, 72, 61, 46, 50, 67, 51, 66, 55, 42, 49, 58, 54, 45, 35, 49, 63, 45, 52, 66, 52, 59, 76, 64, 72, 83, 78, 88, 72, 81, 73, 62, 50, 63, 42, 48, 56, 62, 50, 65, 76, 87, 81, 68, 74, 83, 56, 37]

Plot the poles to these planes in an equal area stereonet. The resultant diagram is called a *point-*, *scatter-* or $\pi-$ diagram. You can approximate a great circle through the poles. What is the approximate orientation of this great circle? What does the pole to this great circle represent?

References

Allmendinger, R.W., Cardozo, N. and Fisher, D.W. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Allmendinger, R.W. 2019. Modern Structural Practice: A structural geology laboratory manual for the 21st century. [Online]. [Accessed January, 2020].

Coe, A. 2010. Geological Field Techniques. Wiley-Blackwell.

Compton, R.R. 1985. Geology in the field. John Wiley & Sons.

Novakova, L. and Pavlis, T.L. 2017. Assessment of the precision of smart phones and tablets for measurement of planar orientations: A case study. Journal of Structural Geology 97, 93-103.

Marshak, S. and Mitra, G. 1988. Basic Methods of Structural Geology. Prentice Hall.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Woodcock, N.H. 1976. The accuracy of structural field measurements. Journal of Geology 84, 350-355.

Chapter 4

Coordinate systems and vectors

Strike and dip, and trend and plunge, are a convenient way to represent the orientation of planes and lines. However, it is difficult to handle these angles using computation. In this chapter, we will see how to convert linear features (lines and poles to planes) from spherical (trend and plunge) to Cartesian (direction cosines) coordinates, thus representing these features as vectors. This facilitates the analysis of planes and lines using linear algebra and computation, and it will allow us to solve a range of interesting problems using vector operations.

4.1 Coordinate systems

Any point or location in space can be represented by the coordinates of the point with respect to the three orthogonal axes of a Cartesian coordinate system. We will call the three axes of this coordinate system $\mathbf{X_1}$, $\mathbf{X_2}$ and $\mathbf{X_3}$ (Fig. 4.1). In addition, we will follow a right-handed naming convention: If you hold your right hand so that your thumb points in the positive direction of the first axis $\mathbf{X_1}$, your other fingers should curl from the positive direction of the second axis $\mathbf{X_2}$ toward the positive direction of the third axis $\mathbf{X_3}$ (Fig. 4.1). Such a coordinate system is called a right-handed coordinate system.

In geosciences, we use mainly two types of right-handed coordinate systems: An east (**E**), north (**N**), up (**U**) coordinate system (Fig. 4.1a), and a north (**N**), east (**E**), down (**D**) coordinate system (Fig. 4.1b). The **ENU** coor-

dinate system is used in GIS and Geophysics when dealing with elevations (e.g. topography), while the **NED** coordinate system is used in Structural Geology where, by convention, angles measured downwards from the horizontal (e.g. plunge of a downward pointing line) are considered positive. In this chapter, we will use the **NED** coordinate system, but when dealing with topography and elevations, we will use the **ENU** coordinate system.

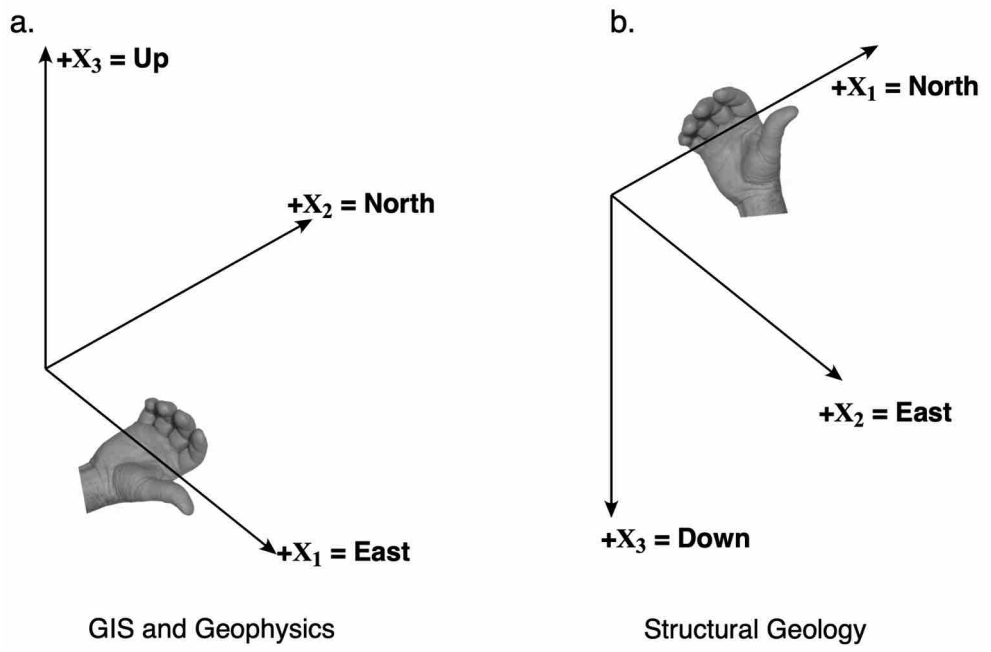


Figure 4.1: Right-handed Cartesian coordinate systems. **a.** The **ENU** coordinate system used when dealing with topography, and **b.** The **NED** coordinate system used in Structural Geology. Modified from Allmendinger et al. (2012).

4.2 Vectors

4.2.1 Vector components, magnitude, and unit vectors

A line from the origin of the Cartesian coordinate system to a point in space is the position *vector* of the point. A *vector* is an object that has both a magnitude and a direction. Displacement, velocity, force, acceleration, and poles to planes, are all vectors. A vector is defined by its three components

with respect to the axes of the Cartesian coordinate system; these are the projections of the vector onto the axes $\mathbf{X_1}$, $\mathbf{X_2}$ and $\mathbf{X_3}$ (Fig. 4.2a).

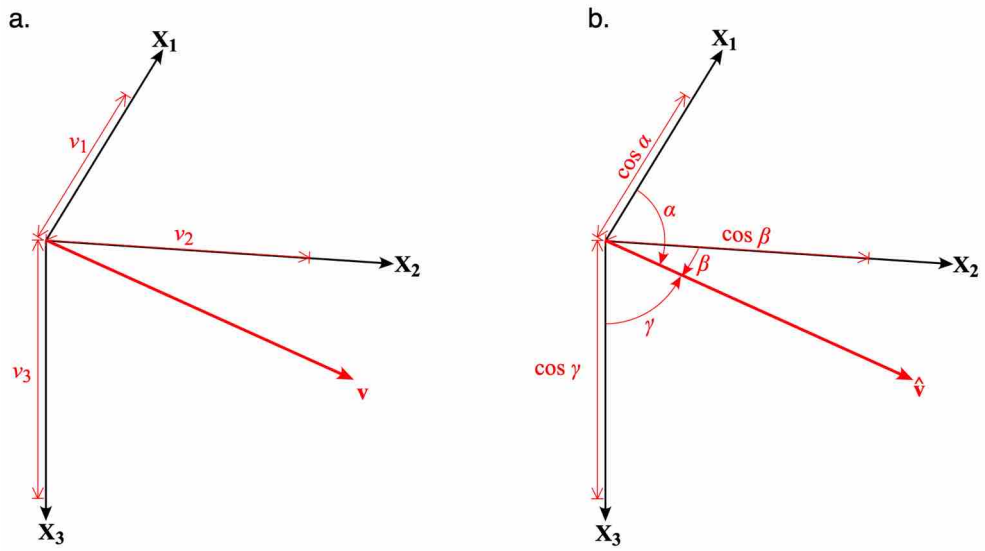


Figure 4.2: **a.** Components of a vector. **b.** Direction cosines of a unit vector. Modified from Allmendinger et al. (2012)

This is expressed by the following equation:

$$\mathbf{v} = [v_1, v_2, v_3] \tag{4.1}$$

We use lower capital letters to denote vectors. The magnitude (length) of a vector can be computed using Pythagoras' theorem:

$$v = (v_1^2 + v_2^2 + v_3^2)^{1/2} \tag{4.2}$$

The result is just a number, a scalar. We use regular, non-capital letters to denote scalars. If we divide each of the components of a vector by its magnitude, the result is a unit vector, a vector with the same orientation but with a magnitude (length) of one (Fig. 4.2b):

$$\hat{\mathbf{v}} = [v_1/v, v_2/v, v_3/v] \tag{4.3}$$

We use a hat to indicate unit vectors. There is a very interesting property of unit vectors; the components of a unit vector are the cosines of the angles the vector makes with the axes of the coordinate system (Fig. 4.2b):

$$\hat{\mathbf{v}} = [\cos\alpha, \cos\beta, \cos\gamma] \tag{4.4}$$

these are called the *direction cosines* of the vector. By convention, $\cos\alpha$ is the direction cosine of the vector with respect to $\mathbf{X_1}$, $\cos\beta$ is the direction cosine of the vector with respect to $\mathbf{X_2}$, and $\cos\gamma$ is the direction cosine of the vector with respect to $\mathbf{X_3}$ (Fig. 4.2b).

In Python, we can use the NumPy *linalg.norm* function to compute the magnitude of a vector and convert it to a unit vector as illustrated in the following notebook ch4-1:

```
# Import numpy
import numpy as np
# Import linear algebra functions
from numpy import linalg as la
# Make vector
v = np.array([1,2,3])
print('Vector: ', v)
# Magnitude of the vector
length = la.norm(v)
print('Magnitude of the vector: ', length)
# Unit vector
v_hat = v / length
print('Unit Vector: ', v_hat)
# Magnitude of unit vector
length = la.norm(v_hat)
print('Magnitude of the unit vector: ', length)
```

Output:
Vector: [1 2 3]
Magnitude of the vector: 3.7416573867739413
Unit Vector: [0.26726124 0.53452248 0.80178373]
Magnitude of the unit vector: 1.0

4.2.2 Vector operations

To multiply a scalar times a vector, just multiply each component of the vector by the scalar:

$$x\mathbf{v} = [xv_1, xv_2, xv_3] \tag{4.5}$$

This operation is useful, for example, to reverse the direction of the vector; just multiply the vector by -1. To add two vectors, just sum their components:

$$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u} = [u_1 + v_1, u_2 + v_2, u_3 + v_3] \tag{4.6}$$

Vector addition is commutative but vector subtraction is not. Vector addition and subtraction obey the parallelogram rule, whereby the resulting vector bisects the two vectors to be added or subtracted (Fig. 4.3a).

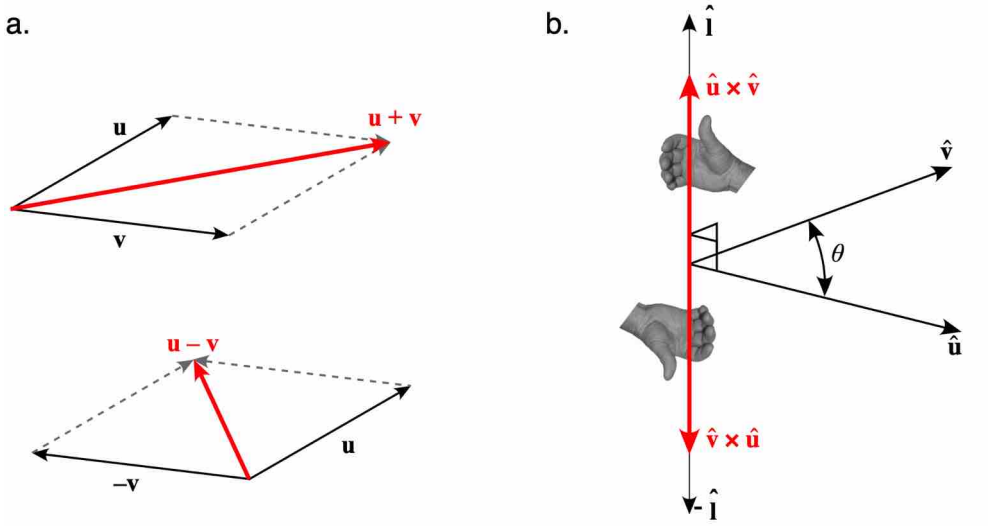


Figure 4.3: **a.** Vector addition and subtraction. **b.** Cross product of two unit vectors. Modified from Allmendinger et al. (2012).

There are two operations that are unique to vectors: the *dot product* and the *cross product*. The result of the dot product is a scalar and is equal to the magnitude of the first vector times the magnitude of the second vector times the cosine of the angle between the vectors:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u} = uv \cos\theta = u_1 v_1 + u_2 v_2 + u_3 v_3 = u_i v_i \tag{4.7}$$

The dot product is commutative. If the two vectors are unit vectors, you can easily see that the dot product is:

$$\hat{\mathbf{u}} \cdot \hat{\mathbf{v}} = \cos\theta = u_1 v_1 + u_2 v_2 + u_3 v_3 \tag{4.8}$$

or in terms of the direction cosines of the vectors:

$$\hat{\mathbf{u}} \cdot \hat{\mathbf{v}} = \cos\theta = cos\alpha_1 \cos\alpha_2 + \cos\beta_1 \cos\beta_2 + \cos\gamma_1 \cos\gamma_2 \tag{4.9}$$

which as we will see later, it is a great way to find the angle between two unit vectors.

The result of the cross product is another vector. This vector is perpendicular to the other two vectors, and has a magnitude that is equal to the product of the magnitude of each vector times the sine of the angle between the vectors:

$$\mathbf{u} \times \mathbf{v} = uv \sin\theta \hat{\mathbf{l}} = [u_2v_3 - u_3v_2, u_3v_1 - u_1v_3, u_1v_2 - u_2v_1] \tag{4.10}$$

The cross product is not commutative. If the vectors are unit vectors, the length of the resulting vector is equal to the sine of the angle between the vectors (Fig. 4.3b). The new vector obeys a right-hand rule; for $\mathbf{u} \times \mathbf{v}$, the fingers curl from $\mathbf{u}$ towards $\mathbf{v}$ and the thumb points in the direction of the resulting vector, and vice versa (Fig. 4.3b).

In Python, these operations are easy to perform using the NumPy library as shown in the following notebook ch4-2:

```
# Import numpy
import numpy as np
# Make vectors
u = np.array([1,2,3])
v = np.array([3,2,1])
print('u = ', u)
print('v = ', v)
# Scalar multiplication of vector
sv = 3 * u
print('3 * u = ', sv)
# Sum of vectors
vsum = u + v
print('u + v = ', vsum)
#  Dot product of vectors
dotp = np.dot(u,v)
print('u . v = ', dotp)
# Cross product of vectors
crossp = np.cross(u,v)
print('u x v = ', crossp)
```

```
Output:
u = [1 2 3]
v = [3 2 1]
3 * u = [3 6 9]
u + v = [4 4 4]
u . v = 10
u x v = [-4 8 -4]
```

4.3 Geologic features as vectors

We have now all the mathematical tools to represent geologic features as vectors. Since we are only interested in the orientation of these features, we will treat lines and poles to planes as unit vectors. We will also use the Structural Geology **NED** coordinate system.

4.3.1 From spherical to Cartesian coordinates

Figure 4.4 shows a line as a unit vector $\hat{\mathbf{v}}$ in the **NED** coordinate system. Clearly, the angle that the line makes with the **D** axis is 90°- *plunge*, therefore:

$$\cos\gamma = \cos(90^\circ - plunge) = \sin(plunge) \tag{4.11a}$$

The horizontal projection of the line is $\cos(plunge)$ (Fig. 4.4). $\cos\alpha$ and $\cos\beta$ are just the **N** and **E** components of this horizontal line (Fig. 4.4):

$$\cos\alpha = \cos(trend)\cos(plunge) \tag{4.11b}$$

$$\cos\beta = \cos(90^\circ - trend)\cos(plunge) = \sin(trend)\cos(plunge) \tag{4.11c}$$

The magnitude and sign of the direction cosines tell us a lot about the orientation of the line (Fig. 4.5). A horizontal line (plunge = 0) has $\cos\gamma = 0$, a downward pointing line (plunge > 0) has $+\cos\gamma$, and if the line is vertical

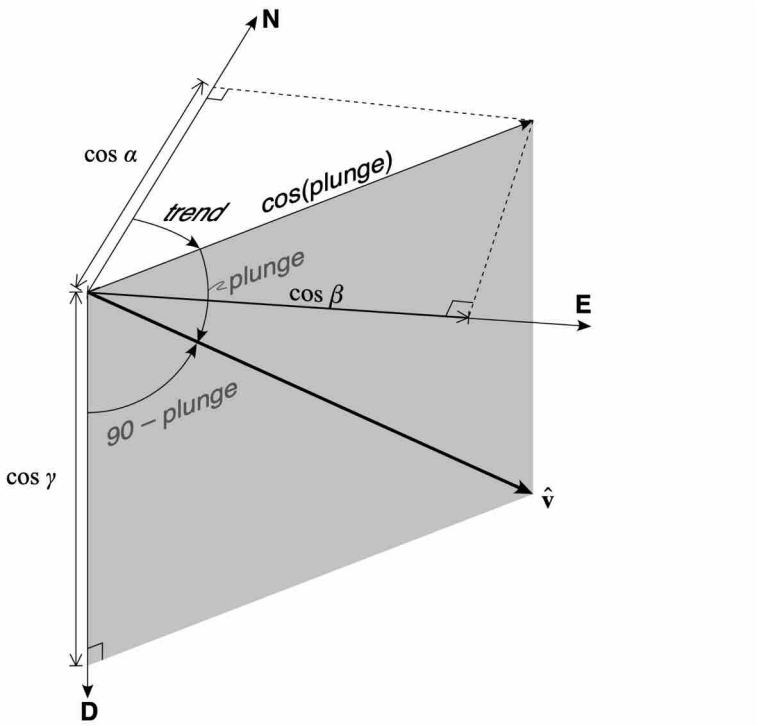


Figure 4.4: Diagram showing the relations between the trend and plunge and the direction cosines in the **NED** coordinate system. Gray plane is the vertical plane in which the plunge is measured. Modified from Allmendinger et al. (2012).

(plunge = 90°) $\cos\gamma = 1$ and the other two direction cosines are 0. A horizontal or downward pointing line (plunge ≥ 0) has $+\cos\alpha$ if it trends NE or NW (first or fourth quadrants), and $+\cos\beta$ if it trends NE or SE (first or second quadrants). If the line trends N or S, $\cos\beta = 0$; and if the line trends E or W, $\cos\alpha = 0$.

To determine the direction cosines of a pole to a plane, we just need to express the trend and plunge of the pole in terms of the strike and dip of the plane assuming a RHR format (Eq. 3.2), and use these in Eq. 4.11. The direction cosines of the pole to the plane are then:

$$\cos\alpha = \cos(strike - 90^\circ)\cos(90^\circ - dip) = \sin(strike)\sin(dip) \qquad (4.12a)$$

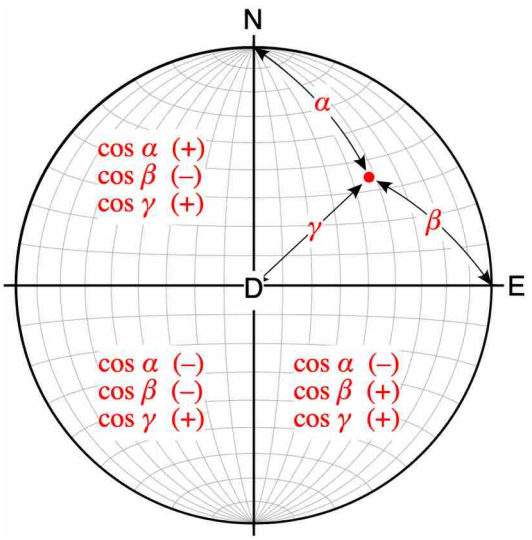


Figure 4.5: Lower hemisphere stereonet showing the sign of the direction cosines in each quadrant. In the NE quadrant, all three direction cosines are positive. Modified from Allmendinger et al. (2012).

$$\cos\beta = \sin(strike - 90^\circ)\cos(90^\circ - dip) = -\cos(strike)\sin(dip) \quad (4.12b)$$

$$\cos\gamma = \sin(90^\circ - dip) = \cos(dip) \quad (4.12c)$$

These equations are summarized in Table 4.1.

Axis	**Direction cosines**	**Lines**	**Poles to planes** (RHR format)
N	$\cos\alpha$	$\cos(trend)\cos(plunge)$	$\sin(strike)\sin(dip)$
E	$\cos\beta$	$\sin(trend)\cos(plunge)$	-$\cos(strike)\sin(dip)$
D	$\cos\gamma$	$\sin(plunge)$	$\cos(dip)$

Table 4.1: Conversion from spherical to Cartesian coordinates

The following function *SphToCart* converts a line ($k = 0$) or a plane ($k = 1$) from spherical to Cartesian coordinates. Notice that the angles (*trd* and *plg*) should be entered in radians:

```python
import math

def SphToCart(trd,plg,k):
  '''
  SphToCart converts from spherical to Cartesian coordinates

  SphToCart(trd,plg,k) returns the north (cn),
  east (ce), and down (cd) direction cosines of a line.

  k: integer to tell whether the trend and plunge of a line
  (k = 0) or strike and dip of a plane in right hand rule
  (k = 1) are being sent in the trd and plg slots. In this
  last case, the direction cosines of the pole to the plane
  are returned

  NOTE: Angles should be entered in radians

  Python function translated from the Matlab function
  SphToCart in Allmendinger et al. (2012)
  '''
  # If line (see Table 4.1)
  if k == 0:
    cn = math.cos(trd) * math.cos(plg)
    ce = math.sin(trd) * math.cos(plg)
    cd = math.sin(plg)
  # Else pole to plane (see Table 4.1)
  elif k == 1:
    cn = math.sin(trd) * math.sin(plg)
    ce = -math.cos(trd) * math.sin(plg)
    cd = math.cos(plg)

  return cn, ce, cd
```

4.3.2 From Cartesian to spherical coordinates

Converting from direction cosines (Cartesian coordinates) to trend and plunge (spherical coordinates) is a little less straightforward. The plunge is easy:

$$plunge = \sin^{-1}(\cos\gamma) \tag{4.13a}$$

The trend can be determined as follows:

$$\frac{\cos\beta}{\cos\alpha} = \frac{\sin(trend)\cos(plunge)}{\cos(trend)\cos(plunge)} = \tan(trend)$$

or:

$$trend = \tan^{-1}\left(\frac{\cos\beta}{\cos\alpha}\right) \tag{4.13b}$$

The problem is that the trend varies from 0 and 360°. For the $\tan^{-1}$ function, there are two possible angles between 0 and 360°. Which one should we use? The answer is to use the signs of the direction cosines to determine in which quadrant the trend lies within. By inspection of Figure 4.5, one can see that:

$$trend = \tan^{-1}\left(\frac{\cos\beta}{\cos\alpha}\right) \;\; if \;\; \cos\alpha > 0 \tag{4.14a}$$

$$trend = 180^\circ + \tan^{-1}\left(\frac{\cos\beta}{\cos\alpha}\right) \;\; if \;\; \cos\alpha < 0 \tag{4.14b}$$

One should also check for the special case of $\cos\alpha = 0$:

$$trend = 90^\circ \; if \; (\cos\alpha = 0 \; and \; \cos(\beta) \geq 0) \tag{4.14c}$$

$$trend = 270^\circ \; if \; (\cos\alpha = 0 \; and \; \cos(\beta) < 0) \tag{4.14d}$$

The following function *CartToSph* converts a line from Cartesian to spherical coordinates. Notice that the trend and plunge are returned in radians:

```
import math
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def CartToSph(cn,ce,cd):
  '''
  CartToSph converts from Cartesian to spherical coordinates

  CartToSph(cn,ce,cd) returns the trend (trd)
  and plunge (plg) of a line for input north (cn),
```

```
  east (ce), and down (cd) direction cosines

  NOTE: Trend and plunge are returned in radians

  CartToSph uses function ZeroTwoPi

  Python function translated from the Matlab function
  CartToSph in Allmendinger et al. (2012)
  '''
  pi = math.pi
  # Plunge
  plg = math.asin(cd) # Eq. 4.13a

  # Trend: If north direction cosine is zero, trend
  # is east or west. Choose which one by the sign of
  # the east direction cosine
  if cn == 0.0:
    if ce < 0.0:
      trd = 3.0/2.0*pi # Eq. 4.14d, trend is west
    else:
      trd = pi/2.0 # Eq. 4.14c, trend is east
  # Else
  else:
    trd = math.atan(ce/cn) # Eq. 4.14a
    if cn < 0.0:
      # Add pi
      trd = trd+pi # Eq. 4.14b
    # Make sure trend is between 0 and 2*pi
    trd = ZeroTwoPi(trd)

  return trd, plg
```

4.4 Applications

4.4.1 Mean vector

An important problem in geosciences is to determine the average or mean vector that represents a group of lines. These lines can be for example poles to bedding, paleocurrent directions, paleomagnetic poles, or slip vectors on a fault surface. This problem can be solved using vector addition. The resultant vector $\mathbf{r}$ of the sum of the N unit vectors representing the lines has components:

$$r_1 = \sum_{i=1}^{N} \alpha_i \qquad r_2 = \sum_{i=1}^{N} \beta_i \qquad r_3 = \sum_{i=1}^{N} \gamma_i \tag{4.15a}$$

where α, β and γ are the direction cosines of the unit vectors. The length of the resultant vector $\mathbf{r}$ is:

$$r = \sqrt{r_1^2 + r_2^2 + r_3^2} \tag{4.15b}$$

and the direction cosines of the unit vector that is parallel to the mean of the individual vectors are:

$$\overline{\alpha} = \frac{r_1}{r} \qquad \overline{\beta} = \frac{r_2}{r} \qquad \overline{\gamma} = \frac{r_3}{r} \tag{4.15c}$$

These direction cosines define the orientation of the mean vector. A measure of how concentrated the individual vectors are or how representative the mean vector is, is given by the *mean resultant length*:

$$\overline{r} = \frac{r}{N} \qquad where \qquad 0 \leq \overline{r} \leq 1 \tag{4.15d}$$

The closer this value is to 1, the better the concentration. The function *CalcMV* calculates the mean vector for a series of lines. It also calculates the Fisher statistics for the mean vector (Fisher et al., 1987), which is the standard way to represent uncertainties in this analysis. Notice that *CalcMV* uses our two previous functions *SphToCart* and *CartToSph* to convert from spherical to Cartesian coordinates, and vice versa.

```
import math
from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph

def CalcMV(T,P):
  '''
  CalcMV calculates the mean vector for a group of lines

  CalcMV(T,P) calculates the trend (trd) and plunge (plg)
  of the mean vector, its mean resultant length (Rave), and
  Fisher statistics (concentration factor (conc), 99 (d99)
```

```
  and 95 (d95) % uncertainty cones); for a series of lines
  whose trends and plunges are stored in the arrays T and P

  NOTE: Input/Output trends and plunges, as well as
  uncertainty cones are in radians

  CalcMV uses functions SphToCart and CartToSph

  Python function translated from the Matlab function
  CalcMV in Allmendinger et al. (2012)
  '''
  # Number of lines
  nlines = len(T)

  # Initialize the 3 direction cosines which contain the
  # sums of the individual vectors
  CNsum = 0.0
  CEsum = 0.0
  CDsum = 0.0

  #Now add up all the individual vectors. Eq. 4.15a
  for i in range(nlines):
    cn,ce,cd = SphToCart(T[i],P[i],0)
    CNsum += cn
    CEsum += ce
    CDsum += cd

  # R is the length of the resultant vector and
  # Rave is the mean resultant length. Eqs. 4.15b and d
  R = math.sqrt(CNsum*CNsum + CEsum*CEsum + CDsum*CDsum)
  Rave = R/nlines
  # If Rave is lower than 0.1, the mean vector is
  # insignificant, return error
  if Rave < 0.1:
    raise ValueError('Mean vector is insignificant')
  #Else
  else:
    # Divide the resultant vector by its length to get
    # the direction cosines of the unit vector
    CNsum = CNsum/R
    CEsum = CEsum/R
    CDsum = CDsum/R
    # Convert the mean vector to the lower hemisphere
    if CDsum < 0.0:
      CNsum = -CNsum
      CEsum = -CEsum
      CDsum = -CDsum
    # Convert the mean vector to trend and plunge
    trd, plg = CartToSph(CNsum,CEsum,CDsum)
```

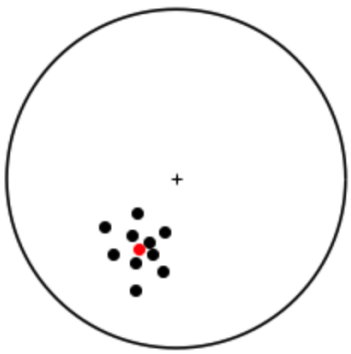
```
    # If there are enough measurements calculate the
    # Fisher statistics (Fisher et al., 1987)
    if R < nlines:
      if nlines < 16:
        afact = 1.0-(1.0/nlines)
        conc = (nlines/(nlines-R))*afact**2
      else:
        conc = (nlines-1.0)/(nlines-R)
    if Rave >= 0.65 and Rave < 1.0:
      afact = 1.0/0.01
      bfact = 1.0/(nlines-1.0)
      d99 = math.acos(1.0-((nlines-R)/R)*(afact**bfact-1.0))
      afact = 1.0/0.05
      d95 = math.acos(1.0-((nlines-R)/R)*(afact**bfact-1.0))

  return trd, plg, Rave, conc, d99, d95
```

The notebook ch4-3 shows the solution to the mean vector problem in Ragan (2009, pp. 147-148), which involves finding the mean orientation of 10 poles to bedding:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt

# Import functions StCoordLine and CalcMV
import sys, os
sys.path.append(os.path.abspath('../functions'))
from StCoordLine import StCoordLine as StCoordLine
from CalcMV import CalcMV as CalcMV

# Arrays T and P contain the trend (T)
# and plunge (P) of the 10 poles
T = np.array([206, 220, 204, 198, 200, 188, 192, 228, 236,
    218])
P = np.array([32, 30, 46, 40, 20, 32, 54, 56, 36, 44])

# Convert T and P from degrees to radians
pi = np.pi
TR = T * pi/180
PR = P * pi/180

# Compute the mean vector and print orientation
# and mean resultant length
trd, plg, Rave, conc, d99, d95 = CalcMV(TR,PR)
print('Mean vector trend = {:.1f}, plunge {:.1f}'.format(trd
    *180/pi,plg*180/pi))
```

```
print('Mean resultant length = {:.3f}'.format(Rave))

# Plot the primitive of the stereonet
r = 1; # unit radius
TH = np.arange(0,360,1)*pi/180
x = r * np.cos(TH)
y = r * np.sin(TH)
plt.plot(x,y,'k')
# Plot center of circle
plt.plot(0,0,'k+')
# Make axes equal and remove them
plt.axis('equal')
plt.axis('off')

# Plot the poles as black points
# on an equal angle stereonet
npoles = len(T)
eqAngle = np.zeros((npoles, 2))
for i in range(npoles):
    # Equal angle coordinates
    eqAngle[i,0], eqAngle[i,1] = StCoordLine(TR[i],PR[i],0)
plt.plot(eqAngle[:,0],eqAngle[:,1],'ko')

# Plot the mean vector as a red point
mvx, mvy = StCoordLine(trd,plg,0)
plt.plot(mvx,mvy,'ro');
```

Output:
Mean vector trend = 208.6, plunge 40.0
Mean resultant length = 0.963



Notice that the mean resultant length is close to 1.0, so that the mean vector (red dot) is a representative orientation of the individual vectors (black dots).

4.4.2 Angles, intersections, and poles

Many interesting problems can be solved using the dot and cross product operations. The dot product can be used to find the angle between two lines or planes, while the cross product can be used to find a plane from two lines or the intersection of two planes. Table 4.2 lists some problems that can be solved using these operations.

Problem	Solution
Angle between two lines	arccos of dot product between lines
Angle between two planes	supplement of arccos of dot product between poles to planes
Intersection of two planes	Cross product of poles to planes
Plane containing two lines	Pole to plane is cross product of lines
Apparent dip of plane	Intersection of plane and vertical section of a given orientation
Plane from two apparent dips	Plane containing the two apparent dips (lines)

Table 4.2: Some problems that can be solved using the dot and cross product operations.

The function *Angles* computes the angle between two lines (*ans0* = *'l'*), the angle between two planes (*ans0* = *'p'*), the plane from two lines (*ans0* = *'a'*), or the intersection of two planes (*ans0* = *'i'*).

```
import math
from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph
from Pole import Pole as Pole

def Angles(trd1,plg1,trd2,plg2,ans0):
  '''
  Angles calculates the angles between two lines,
  between two planes, the line which is the intersection
  of two planes, or the plane containing two apparent dips

  Angles(trd1,plg1,trd2,plg2,ans0) operates on
  two lines or planes with trend/plunge or
  strike/dip equal to trd1/plg1 and trd2/plg2

  ans0 is a character that tells the function what
  to calculate:
```

```
18
19      ans0 = 'a' -> plane from two apparent dips
20      ans0 = 'l' -> the angle between two lines
21
22      In the above two cases, the user sends the trend
23      and plunge of two lines
24
25      ans0 = 'i' -> the intersection of two planes
26      ans0 = 'p' -> the angle between two planes
27
28      In the above two cases the user sends the strike
29      and dip of two planes in RHR format
30
31    NOTE: Input/Output angles are in radians
32
33    Angles uses functions SphToCart, CartToSph and Pole
34
35    Python function translated from the Matlab function
36    Angles in Allmendinger et al. (2012)
37    '''
38    # If planes have been entered
39    if ans0 == 'i' or ans0 == 'p':
40      k = 1
41    # Else if lines have been entered
42    elif ans0 == 'a' or ans0 == 'l':
43      k = 0
44
45    # Calculate the direction cosines of the lines
46    # or poles to planes
47    cn1, ce1, cd1 = SphToCart(trd1,plg1,k)
48    cn2, ce2, cd2 = SphToCart(trd2,plg2,k)
49
50    # If angle between 2 lines or between
51    # the poles to 2 planes
52    if ans0 == 'l' or ans0 == 'p':
53      # Use dot product
54      ans1 = math.acos(cn1*cn2 + ce1*ce2 + cd1*cd2)
55      ans2 = math.pi - ans1
56
57    # If intersection of two planes or pole to
58    # a plane containing two apparent dips
59    if ans0 == 'a' or ans0 == 'i':
60      # If the 2 planes or lines are parallel
61      # return an error
62      if trd1 == trd2 and plg1 == plg2:
63        raise ValueError('Error: lines or planes are parallel')
64      # Else use cross product
65      else:
66        cn = ce1*cd2 - cd1*ce2
```

```
      ce = cd1*cn2 - cn1*cd2
      cd = cn1*ce2 - ce1*cn2
      #Make sure the vector points downe
      if cd < 0.0:
        cn = -cn
        ce = -ce
        cd = -cd
      # Convert vector to unit vector
      r = math.sqrt(cn*cn+ce*ce+cd*cd)
      # Calculate line of intersection or pole to plane
      trd, plg = CartToSph(cn/r,ce/r,cd/r)
      # If intersection of two planes
      if ans0 == 'i':
        ans1 = trd
        ans2 = plg
      # Else if plane containing two dips,
      # calculate plane from its pole
      elif ans0 == 'a':
        ans1, ans2 = Pole(trd,plg,0)

  return ans1, ans2
```

The notebook ch4-4 illustrates the use of the function *Angles* to solve several problems. Let's start with the following problem from Leyshon and Lisle (1996): Two limbs of a chevron fold (A and B) have orientations (RHR) as follows: Limb A = 120/40, Limb B = 250/60. Determine: (a) the trend and plunge of the hinge line of the fold, (b) the rake of the hinge line in limb A, (c) the rake of the hinge line in limb B.

```
import math
pi = math.pi

# Import function Angles
import sys, os
sys.path.append(os.path.abspath('../functions'))
from Angles import Angles as Angles

# Strike and dip of the limbs in radians
str1 = 120 * pi/180 # SW dipping limb
dip1 = 40 * pi/180
str2 = 250 * pi/180 # NE dipping limb
dip2 = 60 * pi/180

# (a) Chevron folds have planar limbs. The hinge
# of the fold is the intersection of the limbs
htrd, hplg = Angles(str1,dip1,str2,dip2,'i')
```

```
print('Hinge trend = {:.1f}, plunge {:.1f}'.format(htrd*180/
    pi,hplg*180/pi))

# The rake of the hinge on either limb is the angle
# between the hinge and the strike line on the limb.
# This line is horizontal and has plunge = 0
plg = 0

# (b) For the SW dipping limb
ang1, ang2 = Angles(str1,plg,htrd,hplg,'l')
print('Rake of hinge in SW dipping limb = {:.1f}'.format(ang1
    *180/pi))

# (c) And for the NE dipping limb
ang1, ang2 = Angles(str2,plg,htrd,hplg,'l')
print('Rake of hinge in NE dipping limb = {:.1f}'.format(ang1
    *180/pi))
```

Output:
Hinge trend = 265.8, plunge 25.3
Rake of hinge in SW dipping limb = 138.4
Rake of hinge in NE dipping limb = 29.5

Let's do another problem from the same book: A quarry has two walls, one trending 002°and the other 135°. The apparent dip of bedding on the faces are 40°N and 30°SE respectively. Calculate the strike and dip of bedding.

```
# The apparent dips are just two lines on bedding
# These lines have orientations:
trd1 = 2 * pi/180
plg1 = 40 * pi/180
trd2 = 135 * pi/180
plg2 = 30 * pi/180

# Calculate bedding from these two apparent dips
strike, dip = Angles(trd1,plg1,trd2,plg2,'a')
print('Bedding strike = {:.1f}, dip {:.1f}'.format(strike
    *180/pi,dip*180/pi))
```

Output:
Bedding strike = 333.9, dip 60.7

And the final problem also from the same book: Slickenside lineations trending 074°occur on a fault with orientation 300/50 (RHR). Determine the plunge of these lineations and their rake in the plane of the fault.

```
# The lineation on the fault is just the intersection
# of a vertical plane with a strike equal to
# the trend of the lineation, and the fault
str1 = 74 * pi/180
dip1 = 90 * pi/180
str2 = 300 * pi/180
dip2 = 50 * pi/180

# Find the intersection of these two planes which is
# the lineation on the fault
ltrd, lplg = Angles(str1,dip1,str2,dip2,'i')
print('Slickensides trend = {:.1f}, plunge {:.1f}'.format(
    ltrd*180/pi,lplg*180/pi))

# And the rake of this lineation is the angle
# between the lineation and the strike line on the fault
plg = 0
ang1, ang2 = Angles(str2,plg,ltrd,lplg,'l')
print('Rake of slickensides = {:.1f}'.format(ang1*180/pi))
```

Output:
Slickensides trend = 74.0, plunge 40.6
Rake of slickensides = 121.8

There are many interesting problems you can solve using the function *Angles*. You will find more problems in section 4.6.

4.4.3 Three point problem

The three point problem is a fundamental problem in geology. It is based on the fact that three non-collinear points on a plane define the orientation of the plane. The graphical solution to this problem is introduced early in the Geosciences Bachelor. It involves finding the strike line (a line connecting two points of equal elevation) on the plane, and the dip from two strike lines (two structure contours) on the plane.

However, there is an easier and more accurate solution to this problem using linear algebra: The three points on the plane define two lines, and the cross product between these lines is parallel to the pole to the plane, from which the orientation of the plane can be estimated. The function *ThreePoint* computes the strike and dip of plane from the east (**E**), north (**N**), and up (**U**) coordinates of three points on the plane:

```
import numpy as np
from numpy import linalg as la
from CartToSph import CartToSph as CartToSph
from Pole import Pole as Pole

def ThreePoint(p1,p2,p3):
  '''
  ThreePoint calculates the strike (strike) and dip (dip)
  of a plane given the east (E), north (N), and up (U)
  coordinates of three non-collinear points on the plane

  p1, p2 and p3 are 1 x 3 arrays defining the location
  of the points in an ENU coordinate system. For each one
  of these arrays the first entry is the E coordinate,
  the second entry the N coordinate, and the third entry
  the U coordinate

  NOTE: strike and dip are returned in radians and they
  follow the right-hand rule format

  ThreePoint uses functions CartToSph and Pole
  '''
  # make vectors v (p1 - p3) and u (p2 - p3)
  v = p1 - p2
  u = p2 - p3

  # take the cross product of v and u
  vcu = np.cross(v,u)

  # make this vector a unit vector
  mvcu = la.norm(vcu) # magnitude of the vector
  uvcu = vcu/mvcu # unit vector

  # make the pole vector in NED coordinates
  p = [uvcu[1], uvcu[0], -uvcu[2]]

  # Make pole point downwards
  if p[2] < 0:
    p = [-elem for elem in p]

  # find the trend and plunge of the pole
```

```
42    trd, plg = CartToSph(p[0],p[1],p[2])
43
44    # find strike and dip of plane
45    strike, dip = Pole(trd, plg, 0)
46
47    return strike, dip
```

Let's use this function in the following example. The geologic map in Fig. 4.6 shows a sequence of sedimentary units dipping south (you can see this by the outcrop V in the valley). Points 1, 2 and 3 are located on the base of unit S and their **ENU** coordinates are:

point1 = [509, 2041, 400]

point2 = [1323, 2362, 500]

point3 = [2003, 2913, 700]

Calculate the strike and dip of the plane. The notebook ch4-5 shows the solution to this problem:

```
1  import numpy as np
2  pi = np.pi
3
4  # Import function ThreePoint
5  import sys, os
6  sys.path.append(os.path.abspath('../functions'))
7  from ThreePoint import ThreePoint as ThreePoint
8
9  # ENU coordinates of the three points
10 p1 = np.array([509, 2041, 400])
11 p2 = np.array([1323, 2362, 500])
12 p3 = np.array([2003, 2913, 700])
13
14 # Compute the orientation of the plane
15 strike, dip = ThreePoint(p1,p2,p3)
16 print('Plane strike = {:.1f}, dip = {:.1f}'.format(strike
       *180/pi,dip*180/pi))
```

Output:
Plane strike = 84.5, dip = 22.5

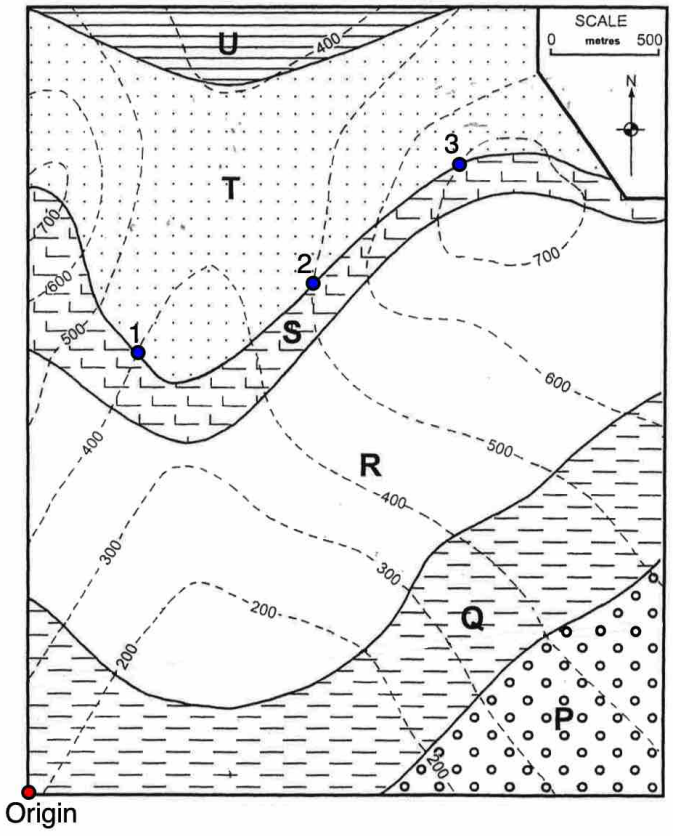


Figure 4.6: Geologic map of sedimentary units dipping south (Bennison et al., 2011). Points 1 to 3 on the base of unit S are used to estimate the orientation of bedding. Notice that the origin is at the map lower left corner.

4.5 Uncertainties

As we saw in section 3.2.5, strike and dip or trend and plunge measurements have errors (Figs. 3.6 and 3.7), and these errors propagate in any computation making use of these angles. Also, as accurate as GPS and elevation measurements are today, they also have errors. Thus, functions *Angles* and *ThreePoint* lack this important element of uncertainty.

Suppose that x, ..., z are measured with uncertainties (or errors) δx, ..., δz and the measured values are used to compute the function $q(x, ..., z)$. If the uncertainties in x, ..., z are independent and random, then the uncertainty (or error) in q is (Taylor, 1997):

$$\delta q = \sqrt{\left(\frac{\partial q}{\partial x}\delta x\right)^2 + ... + \left(\frac{\partial q}{\partial z}\delta z\right)^2} \tag{4.16}$$

This formula is easy to calculate for a few operations, but it can become quite complex for a long chain of operations. Fortunately, there is a Python package that handles calculations with numbers with uncertainties. This package is called *uncertainties* and it was developed by Eric Lebigot. You can find details about the package as well as instructions for installing it in the uncertainties website. If you have the Python *pip* package-management system, you can install the uncertainties package by entering in a terminal:

```
pip install --upgrade uncertainties
```

After this, you can use the uncertainties package in your functions. The functions *AnglesU* and *ThreePointU* in the resource git repository, are the corresponding *Angles* and *ThreePoint* functions with uncertainties. We don't list these functions here, but rather illustrate their use in the following notebook ch4-6.

In the first problem on page 64, the uncertainty in strike is 4°and in dip is 2°. What is the uncertainty in the hinge orientation and its rake on the limbs? This problem can be solved as follows:

```
# Import libraries
import numpy as np
pi = np.pi
import uncertainties as unc

# Import function AnglesU
import sys, os
sys.path.append(os.path.abspath('../functions'))
from AnglesU import AnglesU as AnglesU

# Strike and dip of the limbs in radians
str1 = 120 * pi/180 # SW dipping limb
dip1 = 40 * pi/180
str2 = 250 * pi/180 # NE dipping limb
dip2 = 60 * pi/180

# Errors in radians
ustr = 4 * pi/180 # Error in strike
```

```
udip = 2 * pi/180 # Error in dip

# Create the input values with uncertainties
str1 = unc.ufloat(str1, ustr)  # str1 = str1 +/-ustr
dip1 = unc.ufloat(dip1, udip)  # dip1 = dip1 +/-udip
str2 = unc.ufloat(str2, ustr)  # str2 = str2 +/-ustr
dip2 = unc.ufloat(dip2, udip)  # dip2 = dip2 +/-udip

# (a) Chevron folds have planar limbs. The hinge
# of the fold is the intersection of the limbs
htrd, hplg = AnglesU(str1,dip1,str2,dip2,'i')
print('Hinge trend = {:.1f}, plunge {:.1f}'.format(htrd*180/
    pi,hplg*180/pi))

# The rake of the hinge on either limb is the angle
# between the hinge and the strike line on the limb.
# This line is horizontal and has plunge = 0
plg = unc.ufloat(0, udip)  # plg = 0 +/-udip

# (b) For the SW dipping limb
ang1, ang2 = AnglesU(str1,plg,htrd,hplg,'l')
print('Rake of hinge in SW dipping limb = {:.1f}'.format(ang1
    *180/pi))

# (c) And for the NE dipping limb
ang1, ang2 = AnglesU(str2,plg,htrd,hplg,'l')
print('Rake of hinge in NE dipping limb = {:.1f}'.format(ang1
    *180/pi))
```

Output:
Hinge trend = 265.8+/-3.3, plunge 25.3+/-2.6
Rake of hinge in SW dipping limb = 138.4+/-4.6
Rake of hinge in NE dipping limb = 29.5+/-3.5

In the map of Fig. 4.6, the error in east and north coordinates is 10 m, and in elevation is 5 m. What is uncertainty in the strike and dip of the T-S contact?

```
# Import function ThreePointU
from ThreePointU import ThreePointU as ThreePointU

# ENU coordinates of the three points
# with uncertainties in E-N = 10, and U = 5
p1 = np.array([unc.ufloat(509, 10), unc.ufloat(2041, 10), unc
    .ufloat(400, 5)])
```

```
p2 = np.array([unc.ufloat(1323, 10), unc.ufloat(2362, 10),
    unc.ufloat(500, 5)])
p3 = np.array([unc.ufloat(2003, 10), unc.ufloat(2913, 10),
    unc.ufloat(700, 5)])

# Compute the orientation of the plane
strike, dip = ThreePointU(p1,p2,p3)
print('Plane strike = {:.1f}, dip = {:.1f}'.format(strike
    *180/pi,dip*180/pi))
```

Output:
Plane strike = 84.5+/-3.5, dip = 22.5+/-2.7

4.6 Exercises

Problems 1-3 are from Marshak and Mitra (1988). Solve these problems using the function *Angles*.

1. A fault surface has an orientation (RHR) 190/80. Slickenlines on the fault trend 300°.
   (a) What is the plunge of the lineation?
   (b) What is the rake of the lineation on the fault?
2. A shale bed has an orientation (RHR) 115/42. What is the apparent dip of the bed in the direction 265°?
3. A sandstone bed strikes 140°across a stream. The stream flows down a narrow gorge with vertical walls. The apparent dip of the bed on the walls of the gorge is 095/25. What is the true dip of the bed?
4. In the geologic map of Fig. 4.7, points 1 to 9 have the following **ENU** coordinates:
   point1 = [1580, 379, 400]
   point2 = [1234, 992, 300]
   point3 = [2054, 1753, 400]
   point4 = [448, 1424, 600]
   point5 = [1921, 2195, 500]

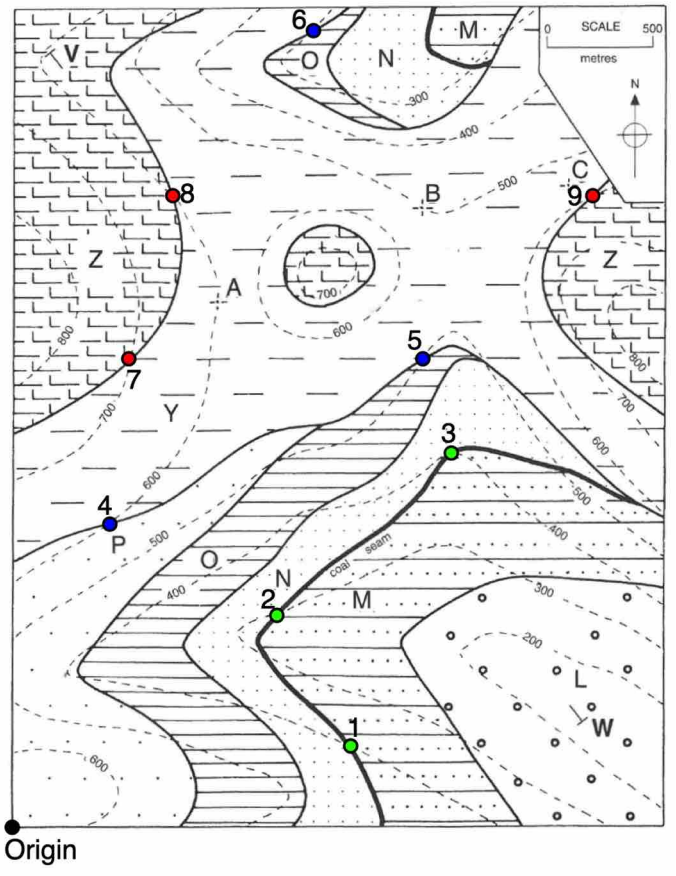


Figure 4.7: Map for exercise 4. This is map 10 of Bennison et al. (2011).

point6 = [1408, 3737, 300]

point7 = [536, 2196, 700]

point8 = [743, 2963, 600]

point9 = [2720, 2963, 600]

(a) Compute the strike and dip of the coal seam (points 1-3).

(b) Compute the strike and dip of the contact where the blue points 4-6 are located. What kind of contact is this? Is the coal seam below or above this contact?

(c) Compute the strike and dip of the contact between units Y and Z (points 7-9). Is this contact below or above the contact in b?

(d) The line of section V-W has a trend of 142°. What is the apparent dip of the three contacts above along the section?

(e) Draw a schematic cross section along line V-W

Hint: Use function *ThreePoint* to solve a, b and c. Use function *Angles* to solve d.

5. The map of Fig. 4.8 shows an area of a reconnaissance survey in the Appalachian Valley and Ridge Province of western Maryland, USA. On the western half of the map, the contact between a shale horizon (B) and a sandstone unit (C) has been located in two areas. Three points on this contact have the following **ENU** coordinates:

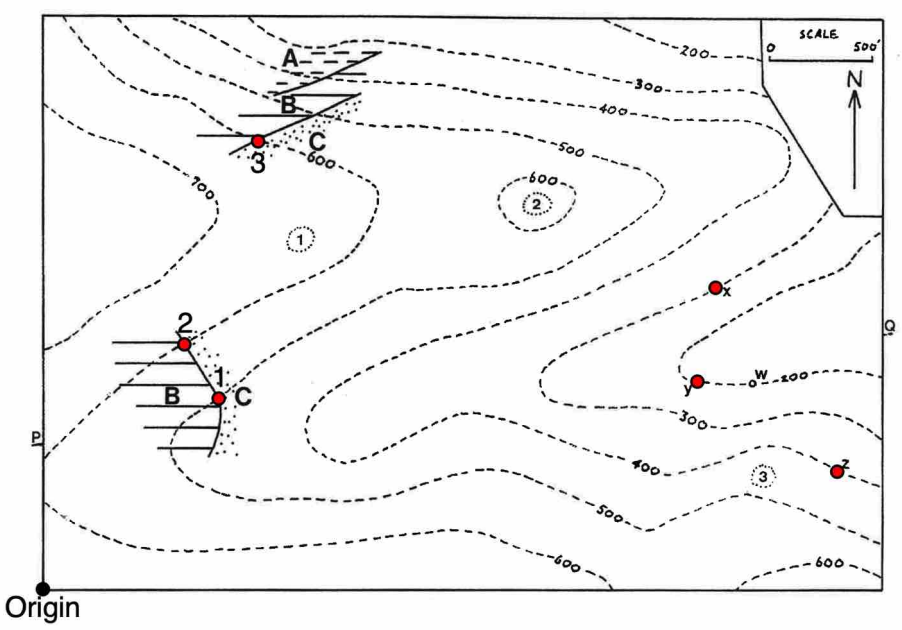


Figure 4.8: Map for exercise 5

point1 = [862, 943, 500]

point2 = [692, 1212, 600]

point3 = [1050, 2205, 600]

On the eastern half of the map, the contact between B and C was found exposed at three locations labelled x, y, and z. The **ENU** coordinates of these points are:

pointx = [3298, 1487, 300]

pointy = [3203, 1031, 200]

pointz = [3894, 590, 400]

(a) Compute the strike and dip of the contact on the western half of the map.

(b) Compute the strike and dip of the contact on the eastern half of the map.

(c) What kind of structure is present on the map?

(d) Compute the intersection of the western and eastern contacts. What does this line represent?

Hint: Use function *ThreePoint* to solve a and b. Use function *Angles* to solve d.

6. In problem 3, the error in azimuth is 5°and in apparent dip is 3°. What is the uncertainty in the true dip of the bed? *Hint*: Use function *AnglesU*.

7. In the map of Fig. 4.7, the east and north coordinates of the points have 15 m error, and the elevations have 5 m error.

(a) What is the uncertainty in the strike and dip of the coal seam?

(b) What is the uncertainty in the strike and dip of the unconformity?

(c) What is the angle between the unconformity and the coal seam and what is the uncertainty in this angle?

Hint: Use functions *ThreePointU* and *AnglesU*.

References

Allmendinger, R.W., Cardozo, N. and Fisher, D.W. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Bennison, G.M., Olver, P.A. and Moseley, K.A. 2011. An Introduction to Geological Structures and Maps, 8th edition. Hodder Education.

Fisher, N.I., Lewis, T. and Embleton, B.J.J. 1987. Statistical analysis of spherical data. Cambridge University Press.

Leyshon, P.R. and Lisle, R.J. 1996. Stereographic Projection Techniques in Structural Geology. Butterworth Heinemann.

Marshak, S. and Mitra, G. 1988. Basic Methods of Structural Geology. Prentice Hall.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Taylor, J.R. 1997. An Introduction to Error Analysis, 2nd edition. University Science Books.

Chapter 5

Transformations

Many problems in geology become simpler when viewed from another perspective. For example, when studying the movement of continents through time because of plate tectonics (Fig. 5.1a), two coordinate systems are required, one in a present-day geographic frame, and another one attached to the continent. Or to analyze a fault (Fig. 5.1b), we need one coordinate system attached to the fault (with one axis parallel to the pole and another to the slickensides), which we want to relate to the more familiar **NED** system. A change in coordinate system is called a coordinate transformation and this is an operation that happens everytime and everywhere. Computer games, flight simulators, and 3D interpretation programs rely heavily on coordinate transformations.

5.1 Transforming coordinates and vectors

5.1.1 Coordinate transformations

A transformation involves a change in the origin and orientation of the coordinate system. We will refer to the new axes as the primed coordinate system, $\mathbf{X}'$, and the old coordinate system as the unprimed system, $\mathbf{X}$ (Fig. 5.2a). Let's assume the origin of the old and new coordinate systems is the same. The change in orientation of the new coordinate system is defined by the angles between the new coordinate axes and the old axes. These angles are marked systematically, the first subscript refers to the new coordinate

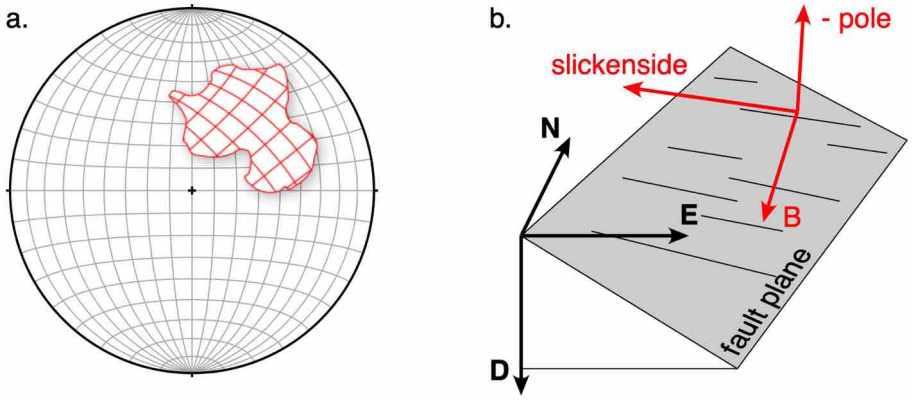


Figure 5.1: Examples of coordinate transformations in geology. **a.** Continental drift, **b.** A fault plane. Red is the coordinate system for analysis, and gray is the geographic coordinate system. Modified from Allmendinger et al. (2012).

axis and the second subscript to the old coordinate axis. For example, θ_{23} is the angle between the $\mathbf{X_2}'$ axis and the $\mathbf{X_3}$ axis (Fig. 5.2a).

To define the transformation, we use the cosines of these angles rather than the angles themselves (Fig. 5.2b). These are the direction cosines of the new axes with respect to the old axes. The subscript convention is exactly the same. For example, a_{23} is the direction cosine of the $\mathbf{X_2}'$ axis with respect to the $\mathbf{X_3}$ axis. There are nine direction cosines that form a 3×3 array, where each row refers to a new axis and each column to an old axis (Fig. 5.2b). This matrix $\mathbf{a}$ of direction cosines is known as the *transformation matrix*, and it is the key element that defines the transformation.

Fortunately, not all the nine direction cosines in the transformation matrix are independent. Since the base vectors of the new coordinate system are unit vectors, their magnitude is 1:

$$\begin{aligned} a_{11}^2 + a_{12}^2 + a_{13}^2 &= 1 \\ a_{21}^2 + a_{22}^2 + a_{23}^2 &= 1 \\ a_{31}^2 + a_{32}^2 + a_{33}^2 &= 1 \end{aligned} \tag{5.1}$$

and because the base vectors of the new coordinate system are perpendicular to each other, the dot product of two of these axes is zero:

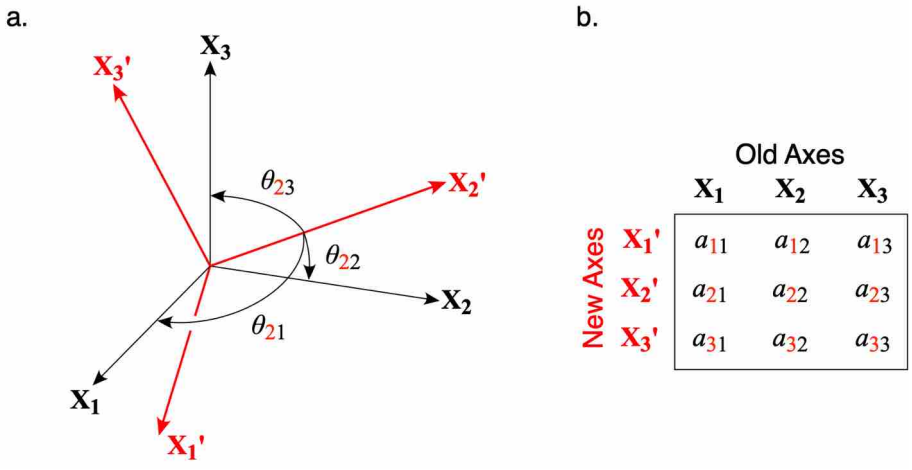


Figure 5.2: **a.** Rotation of a Cartesian coordinate system. The old axes are black, the new axes are primed and red. Only three of the nine possible angles are shown. **b.** Graphic device for remembering how the subscript of the direction cosines relate to the new and the old axes. Modified from Allmendinger et al. (2012).

$$\begin{aligned} a_{21}a_{31} + a_{22}a_{32} + a_{23}a_{33} &= 0 \\ a_{31}a_{11} + a_{32}a_{12} + a_{33}a_{13} &= 0 \\ a_{11}a_{21} + a_{12}a_{22} + a_{13}a_{23} &= 0 \end{aligned} \tag{5.2}$$

Equations 5.1 and 5.2 are known as the *orthogonality relations*. Since we have nine unknowns (i.e. the nine direction cosines) and six equations, there are only three independent direction cosines in the transformation matrix. If we know three of the direction cosines defining the transformation, we can calculate the other six.

5.1.2 Transformation of vectors

Once we know the transformation matrix **a** and the components of a vector in the old coordinate system, we can calculate the components of the vector in the new coordinate system. The equations are fairly simple:

$$\begin{aligned} {v_1}' &= a_{11}v_1 + a_{12}v_2 + a_{13}v_3 \\ {v_2}' &= a_{21}v_1 + a_{22}v_2 + a_{23}v_3 \\ {v_3}' &= a_{31}v_1 + a_{32}v_2 + a_{33}v_3 \end{aligned} \tag{5.3}$$

or using the Einstein summation notation:

$${v_i}' = a_{ij}v_j \tag{5.4}$$

where i is the free suffix, and j is the dummy suffix. Assuming that (i, $j = 1, 2, 3$), Equation 5.4 represents three separate equations (the three indices of i), each one with three terms (the three indices of j). These equations are easy to implement in Python code:

```
for i in range(0,3,1):
    v_new[i] = 0
    for j in range(0,3,1):
        v_new[i] = a[i,j]*v_old[j] + v_new[i]
```

You will see this code snippet repeatedly in the functions of this chapter. Linear algebra is elegant. To convert the vector from the new coordinate system back to the old coordinate system, you just need to do:

$$v_i = a_{ji}{v_j}' \tag{5.5}$$

or in Python code:

```
for i in range(0,3,1):
    v_old[i] = 0
    for j in range(0,3,1):
        v_old[i] = a[j,i]*v_new[j] + v_old[i]
```

5.1.3 A simple transformation: From ENU to NED

There is a simple coordinate transformation that nicely illustrates the theory above: the transformation from an **ENU** to a **NED** coordinate system (Fig.

4.1). It is simple because the angles involved are either 0, 90, or 180°. The direction cosines of the new axes (**NED**) with respect to the old axes (**ENU**), and the transformation matrix **a** is:

$$\mathbf{a} = \begin{pmatrix} \cos 90 & \cos 0 & \cos 90 \\ \cos 0 & \cos 90 & \cos 90 \\ \cos 90 & \cos 90 & \cos 180 \end{pmatrix} \tag{5.6}$$

which simplifies to:

$$\mathbf{a} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{5.7}$$

When we use this matrix in Eq. 5.4, we get:

$$v_1{}' = v_2; \quad v_2{}' = v_1; \quad v_3{}' = -v_3; \tag{5.8}$$

Notice that in this special case **a** is symmetric ($a_{ij} = a_{ji}$), so the elements of **a** are also the direction cosines of **ENU** with respect to **NED**. In the following sections, we will look at more complicated coordinate transformations, but the principles mentioned here will still apply.

5.2 Applications

5.2.1 Stratigraphic thickness

The thickness of a stratigraphic unit is the perpendicular distance between the parallel planes bounding the unit. This is also called the true or stratigraphic thickness (Ragan, 2009). A general problem though is that points on the planes bounding the unit, are commonly given in a geographic (e.g. **ENU**) coordinate system. One therefore must use ortographic projections and trigonometry to determine the stratigraphic thickness of the unit from the points (Ragan, 2009).

An easier approach is to use a transformation. Figure 5.3 illustrates the situation. Points on the top and base of the unit are given in an **ENU** coordinate system. We can transform these points into a coordinate system attached to the bounding planes, where the strike (RHR) of the planes is the first axis, the dip the second axis, and the pole to the planes the third axis. We will call this coordinate system the **SDP** system. The thickness of the unit is just the difference between the **P** coordinate of any point on the top of the unit and the **P** coordinate of any point on the base.

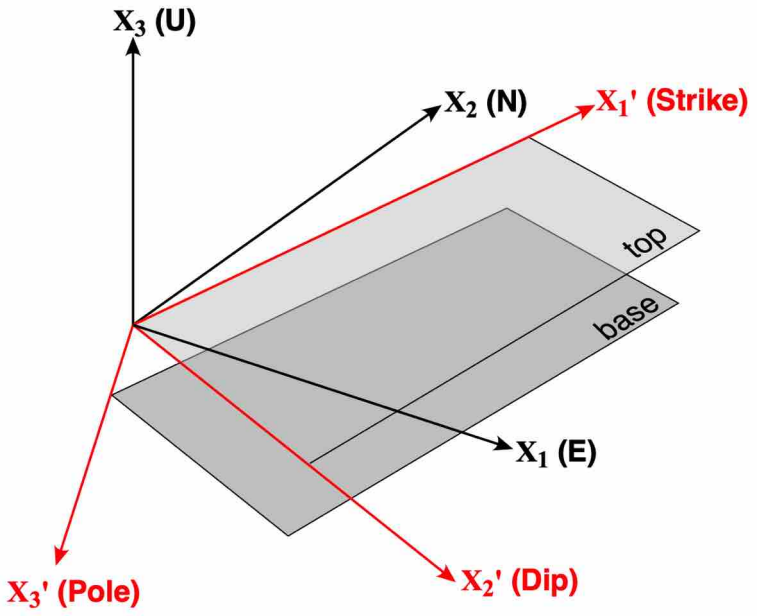


Figure 5.3: Coordinate transformation from an **ENU** to a Strike-Dip-Pole (**SDP**) coordinate system. The thickness of the unit can be calculated by subtracting the **P** coordinates of any point on the top and any point on the base. Modified from Allmendinger (2019).

We can find the elements of the matrix **a** for this transformation using trigonometry. However, we will follow a more didactic approach. We will reference the two **ENU** and **SDP** coordinate systems with respect to the **NED** coordinate system, and use the dot product to determine the direction cosines of **SDP** into **ENU**.

The direction cosines of the **ENU** coordinate system with respect to the **NED** coordinate system are given by Eq. 5.7. The direction cosines of the **SDP** coordinate system with respect to the **NED** coordinate system are given by Table 4.1:

$$\begin{aligned}
\mathbf{S} &= [\cos(strike), \sin(strike), 0] \\
\mathbf{D} &= [\cos(strike + 90)\cos(dip), \sin(strike + 90)\cos(dip), \sin(dip)] \\
\mathbf{P} &= [\cos(strike - 90)\cos(90 - dip), \sin(strike - 90)\cos(90 - dip), \sin(90 - dip)]
\end{aligned}$$

which simplifies to:

$$\begin{aligned}
\mathbf{S} &= [\cos(strike), \sin(strike), 0] \\
\mathbf{D} &= [-\sin(strike)\cos(dip), \cos(strike)\cos(dip), \sin(dip)] \\
\mathbf{P} &= [\sin(strike)\sin(dip), -\cos(strike)\sin(dip), \cos(dip)]
\end{aligned} \tag{5.9}$$

The transformation matrix **a** from the **ENU** to the **SDP** coordinate system has as components the direction cosines of the new **SDP** axes into the old **ENU** axes. From Eq. 4.9, one can see that these are just the dot product between the new and old axes:

$$\mathbf{a} = \begin{pmatrix} \mathbf{S}\cdot\mathbf{E} & \mathbf{S}\cdot\mathbf{N} & \mathbf{S}\cdot\mathbf{U} \\ \mathbf{D}\cdot\mathbf{E} & \mathbf{D}\cdot\mathbf{N} & \mathbf{D}\cdot\mathbf{U} \\ \mathbf{P}\cdot\mathbf{E} & \mathbf{P}\cdot\mathbf{N} & \mathbf{P}\cdot\mathbf{U} \end{pmatrix}$$

which is equal to:

$$\mathbf{a} = \begin{pmatrix} \sin(strike) & \cos(strike & 0 \\ \cos(strike)\cos(dip) & -\sin(strike)\cos(dip) & -\sin(dip) \\ -\cos(strike)\sin(dip) & \sin(strike)\sin(dip) & -\cos(dip) \end{pmatrix} \tag{5.10}$$

So if point 1 is at the top of the unit and has coordinates $[E_1, N_1, U_1]$, and point 2 is at the base of the unit and has coordinates $[E_2, N_2, U_2]$, the **P** coordinates of these points are:

$$\begin{aligned}
P_1 &= -\cos(strike)\sin(dip)E_1 + \sin(strike)\sin(dip)N_1 - \cos(dip)U_1 \\
P_2 &= -\cos(strike)\sin(dip)E_2 + \sin(strike)\sin(dip)N_2 - \cos(dip)U_2
\end{aligned} \tag{5.11}$$

and the thickness of the unit is:

$$t = P_2 - P_1 \tag{5.12}$$

The function *TrueThickness* calculates the thickness of a unit given the strike and dip of the unit, and the **ENU** coordinates of two top and base points:

```
import numpy as np

def TrueThickness(strike,dip,top,base):
  '''
  TrueThickness calculates the thickness (t) of a unit
  given the strike (strike) and dip (dip) of the unit,
  and points at its top (top) and base (base)

  top and base are 1 x 3 arrays defining the location
  of top and base points in an ENU coordinate system.
  For each one of these arrays, the first, second
  and third entries are the E, N and U coordinates

  NOTE: strike and dip should be input in radians
  '''
  # make the transformation matrix from ENU coordinates
  # to SDP coordinates. Eq. 5.10
  sinStr = np.sin(strike)
  cosStr = np.cos(strike)
  sinDip = np.sin(dip)
  cosDip = np.cos(dip)
  a = np.array([[sinStr, cosStr, 0],
  [cosStr*cosDip, -sinStr*cosDip, -sinDip],
  [-cosStr*sinDip, sinStr*sinDip, -cosDip]])

  # transform the top and base points
  # from ENU to SDP coordinates. Eq. 5.4
  topn = np.zeros(3)
  basen = np.zeros(3)
  for i in range(0,3,1):
    for j in range(0,3,1):
      topn[i] = a[i,j]*top[j] + topn[i]
      basen[i] = a[i,j]*base[j] + basen[i]

  # compute the thickness of the unit. Eq. 5.12
  t = np.abs(basen[2] - topn[2])

  return t
```

Let's use this function to determine the thickness of the sedimentary units T to Q in the geologic map of Fig. 5.4. This time we have points at the top and base of these units, and their **ENU** coordinates are:

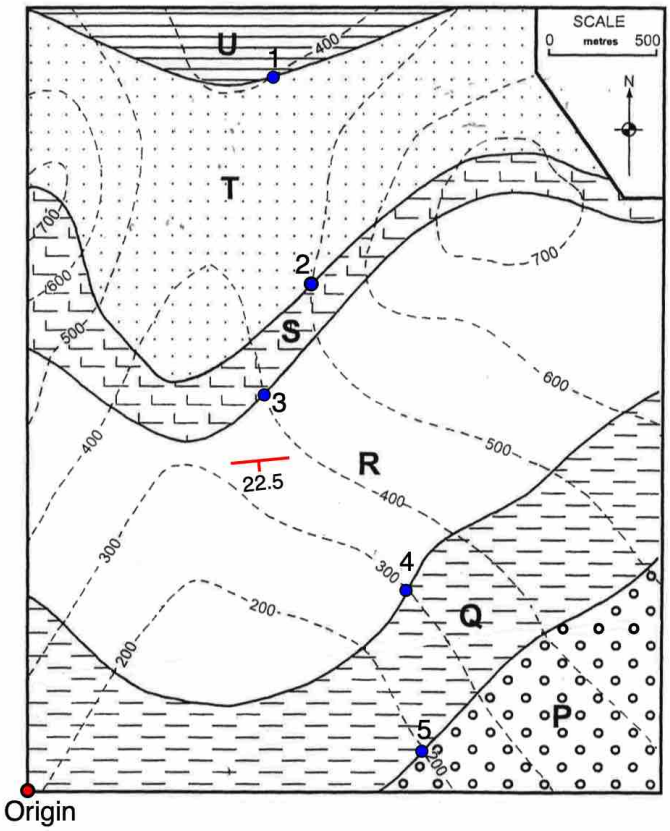


Figure 5.4: Geologic map of sedimentary units with an orientation 84.5/22.5 (RHR) (Bennison et al., 2011). Points at the top and base of units T to Q are used to determine the thickness of the units.

point1 = [1147, 3329, 400]

point2 = [1323, 2362, 500]

point3 = [1105, 1850, 400]

point4 = [1768, 940, 300]

point5 = [1842, 191, 200]

The notebook ch5-1 shows the solution to this problem:

```
import numpy as np
pi = np.pi

# Import function TrueThickness
import sys, os
sys.path.append(os.path.abspath('../functions'))
from TrueThickness import TrueThickness as TrueThickness

# Strike and dip of the unit in radians
strike = 84.5 * pi/180
dip = 22.5 * pi/180

# ENU coordinates of the points
p1 = np.array([1147, 3329, 400])
p2 = np.array([1323, 2362, 500])
p3 = np.array([1105, 1850, 400])
p4 = np.array([1768, 940, 300])
p5 = np.array([1842, 191, 200])

# Compute the thickness of the units
thickT = TrueThickness(strike,dip,p2,p1)
thickS = TrueThickness(strike,dip,p3,p2)
thickR = TrueThickness(strike,dip,p4,p3)
thickQ = TrueThickness(strike,dip,p5,p4)
print('Thickness of unit T = {:.1f} m'.format(thickT))
print('Thickness of unit S = {:.1f} m'.format(thickS))
print('Thickness of unit R = {:.1f} m'.format(thickR))
print('Thickness of unit Q = {:.1f} m'.format(thickQ))
```

Output:
Thickness of unit T = 467.2 m
Thickness of unit S = 94.6 m
Thickness of unit R = 278.6 m
Thickness of unit Q = 195.6 m

What about if there are uncertainties in the strike and dip of the unit, and in the top and base points? The function *TrueThicknessU* in the resource git repository handles this case. Suppose that in the problem above, the uncertainties in strike and dip are 4°and 2°respectively, the uncertainty in east and north coordinates is 10 m, and in elevation is 5 m. The notebook ch5-2 shows the solution to this problem:

```
# Import libraries
import numpy as np
pi = np.pi
import uncertainties as unc

# Import function TrueThicknessU
import sys, os
sys.path.append(os.path.abspath('../functions'))
from TrueThicknessU import TrueThicknessU as TrueThicknessU

# Strike and dip of the unit in radians
strike = 84.5 * pi/180
dip = 22.5 * pi/180

# Strike and dip errors in radians
ustrike = 4 * pi/180
udip = 2 * pi/180

# Create the strike and dip with uncertainties
strike = unc.ufloat(strike, ustrike) # strike +/- ustrike
dip = unc.ufloat(dip, udip) # dip +/- udip

# ENU coordinates of the points
# with uncertainties in E-N = 10, and U = 5
p1 = np.array([unc.ufloat(1147, 10), unc.ufloat(3329, 10),
    unc.ufloat(400, 5)])
p2 = np.array([unc.ufloat(1323, 10), unc.ufloat(2362, 10),
    unc.ufloat(500, 5)])
p3 = np.array([unc.ufloat(1105, 10), unc.ufloat(1850, 10),unc
    .ufloat(400, 5)])
p4 = np.array([unc.ufloat(1768, 10), unc.ufloat(940, 10), unc
    .ufloat(300, 5)])
p5 = np.array([unc.ufloat(1842, 10), unc.ufloat(191, 10), unc
    .ufloat(200, 5)])

# Compute the thickness of the units
thickT = TrueThicknessU(strike, dip, p2, p1)
thickS = TrueThicknessU(strike, dip, p3, p2)
thickR = TrueThicknessU(strike, dip, p4, p3)
thickQ = TrueThicknessU(strike, dip, p5, p4)
print('Thickness of unit T = {:.1f} m'.format(thickT))
print('Thickness of unit S = {:.1f} m'.format(thickS))
print('Thickness of unit R = {:.1f} m'.format(thickR))
print('Thickness of unit Q = {:.1f} m'.format(thickQ))
```

```
Output:
Thickness of unit T = 467.2+/-31.5 m
Thickness of unit S = 94.6+/-20.4 m
Thickness of unit R = 278.6+/-37.0 m
Thickness of unit Q = 195.6+/-27.0 m
```

For the thinnest unit S, the uncertainty in thickness is as much as 20% the thickness of this unit!

5.2.2 Outcrop trace of a plane

The **ENU** to **SDP** transformation is useful to solve another important problem, namely the outcrop trace of a plane on irregular terrain (Fig. 3.9; Allmendinger in press.). If we know the orientation of the plane (strike and dip), the **ENU** coordinates of the location where the plane outcrops, and the topography of the terrain as a digital elevation model (DEM[1]), we can determine the outcrop trace of the plane using computation. The solution is surprisingly simple, the plane outcrops wherever the **P** coordinate of the terrain is equal to the **P** coordinate of the plane's outcrop location.

Let's call the **P** coordinate of the plane's outcrop location P_1, and the **P** coordinate of a point in the DEM grid P_{gp}. The difference between these two is:

$$D = P_1 - P_{gp} \tag{5.13}$$

At each point in the DEM grid, we can calculate and store this difference. The plane will outcrop wherever D is zero. Therefore, to draw the outcrop trace, we just need to contour the D value of zero on the grid. The function *OutcropTrace* computes the value of D on a DEM grid of regularly spaced points defined by **E** (**XG**), **N** (**YG**) and **U** (**ZG**) coordinates. Notice that these arrays must follow the format given by the NumPy *meshgrid* function:

[1] A grid of regularly spaced points in east and north and with elevation data.

```python
import numpy as np

def OutcropTrace(strike,dip,p1,XG,YG,ZG):
  '''
  OutcropTrace estimates the outcrop trace of a plane,
  given the strike (strike) and dip (dip) of the plane,
  the ENU coordinates of a point (p1) where the plane
  outcrops, and a DEM of the terrain as a regular grid
  of points with E (XG), N (YG) and U (ZG) coordinates.

  After using this function, to draw the outcrop trace
  of the plane, you just need to draw the contour 0 on
  the grid XG,YG,DG

  NOTE: strike and dip should be input in radians
    p1 must be an array
    XG and YG arrays should be constructed using
    the Numpy function meshgrid
  '''
  # make the transformation matrix from ENU coordinates to
  # SDP coordinates. We just need the third row of this
  # matrix
  a = np.zeros((3,3))
  a[2,0] = -np.cos(strike)*np.sin(dip)
  a[2,1] = np.sin(strike)*np.sin(dip)
  a[2,2] = -np.cos(dip);

  # Initialize DG
  n, m = XG.shape
  DG = np.zeros((n,m))

  # Estimate the P coordinate of the outcrop point p1
  P1 = a[2,0]*p1[0] + a[2,1]*p1[1] + a[2,2]*p1[2]

  # Estimate the P coordinate at each point of the DEM
  # grid and subtract P1. Eq. 5.13
  for i in range(n):
    for j in range(m):
      DG[i,j] = P1 - (a[2,0]*XG[i,j] + a[2,1]*YG[i,j] + a
    [2,2]*ZG[i,j])

  return DG
```

Let's apply this function to the map of Fig. 5.5. On the western half of this map, the contact between units B and C outcrops at point 2 and has an orientation 020/22 (RHR). On the eastern half of the map, the same contact

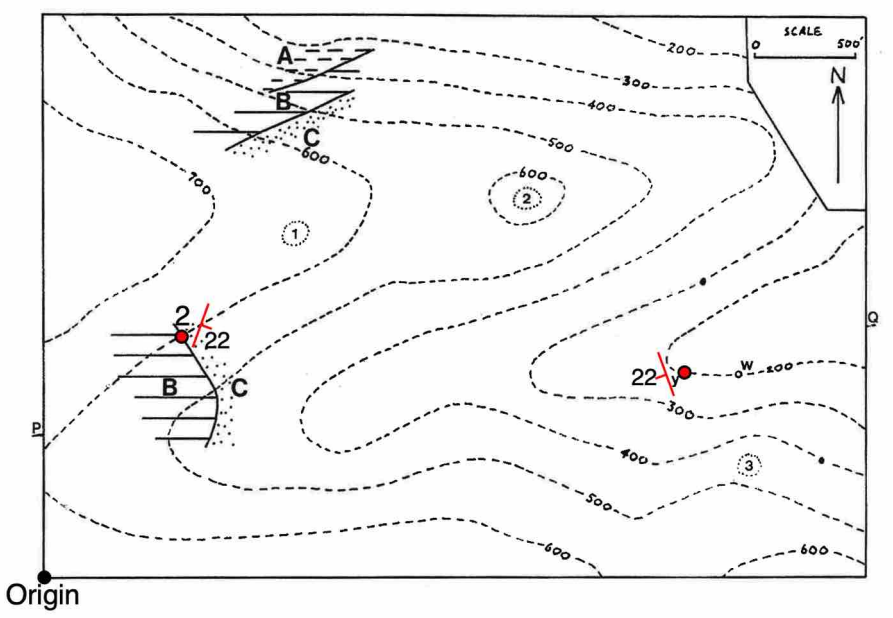


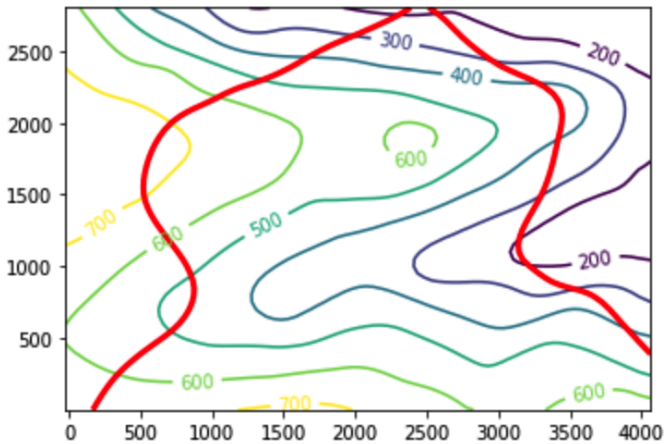
Figure 5.5: The contact between units B and C outcrops at point 2 with orientation 020/22 (RHR), and at point y with orientation 160/22 (RHR). From this information and a DEM of the terrain, we can determine the outcrop trace of the contact.

outcrops at point y and has an orientation 160/22 (RHR). The notebook ch5-3 draws the outcrop trace of the contact. Notice that the **ENU** coordinates of the points of the DEM grid are input from the text files *XG*, *YG*, and *ZG*.

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt

# Import function OutcropTrace
import sys, os
sys.path.append(os.path.abspath('../functions'))
from OutcropTrace import OutcropTrace as OutcropTrace

# Read the DEM grid
XG = np.loadtxt(os.path.abspath('../data/ch5-3/XG.txt'))
YG = np.loadtxt(os.path.abspath('../data/ch5-3/YG.txt'))
ZG = np.loadtxt(os.path.abspath('../data/ch5-3/ZG.txt'))

# Contour the terrain
cval = np.linspace(200,700,6)
cp = plt.contour(XG,YG,ZG,cval)
```

```
plt.clabel(cp, inline=True, fontsize=10, fmt="%d")

# Western contact
pi = np.pi
strike = 20*pi/180
dip = 22*pi/180
point2 = np.array([692, 1212, 600])
DG = OutcropTrace(strike,dip,point2,XG,YG,ZG)
cval = 0 # Contour only CG zero value
cp = plt.contour(XG,YG,DG,cval,colors='red',linewidths=3)

# Eastern contact
strike = 160*pi/180
dip = 22*pi/180
pointy = np.array([3203, 1031, 200])
DG = OutcropTrace(strike,dip,pointy,XG,YG,ZG)
cp = plt.contour(XG,YG,DG,cval,colors='red',linewidths=3)

# Make axes equal
plt.axis('equal');
```

Output:



5.2.3 Down-Plunge projection

Folded rock layers normally have a cylindrical symmetry; the layers are bent about a single axis or direction, called the fold axis. The fold axis is a line of minimum, zero curvature, and the lines of maximum, non-zero curvature are perpendicular to it (Suppe, 1985). The least distorted view of such structure

is in the plane perpendicular to the fold axis, which is called the profile plane (Fig. 5.6). Projecting the fold data to this profile plane is called a *Down-Plunge* projection.

Constructing a Down-Plunge projection by hand typically involves an orthographic projection, and this problem is complicated (and tedious), particularly if points on the fold are not at the same elevation. Fortunately, we can solve this problem as a coordinate transformation, where points on the fold referenced in an **ENU** coordinate system, are transformed to a new $\mathbf{X_1'X_2'X_3'}$ coordinate system, with $\mathbf{X_3'}$ parallel to the fold axis, and $\mathbf{X_1'X_2'}$ defining the profile plane (Fig. 5.6).

We will again derive the matrix **a** for this transformation using linear algebra. The direction cosines of the **ENU** coordinate system with respect to the **NED** coordinate system are given by Eq. 5.7. The direction cosines of the $\mathbf{X_1'X_2'X_3'}$ coordinate system with respect to the **NED** coordinate system can be found using Table 4.1. If the trend and plunge of the fold axis are T and P, these direction cosines are (Fig. 5.6):

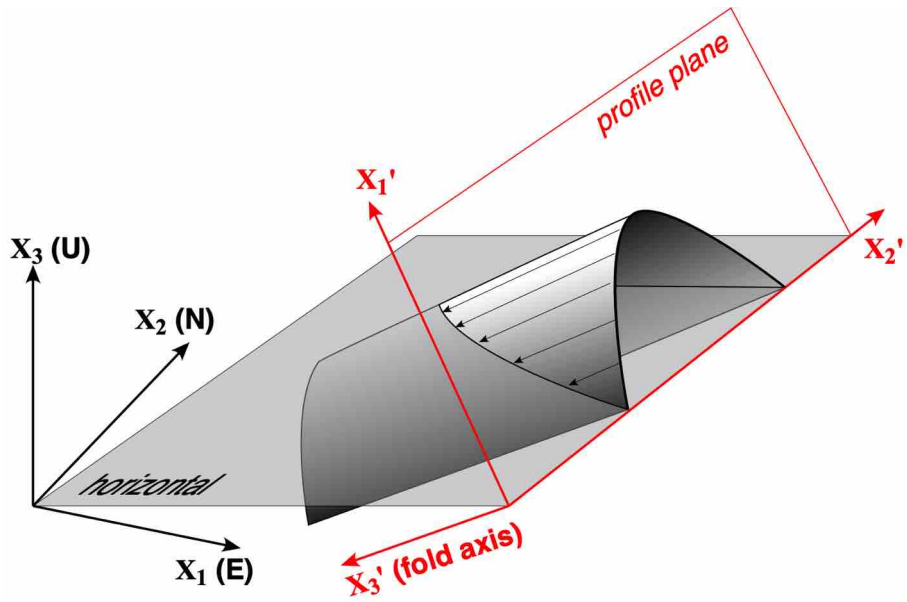


Figure 5.6: The two coordinate systems involved in a Down-Plunge projection of a fold. Modified from Allmendinger et al. (2012).

$$\begin{aligned}\mathbf{X_1}' &= [\cos(T)\cos(P-90), \sin(T)\cos(P-90), \sin(P-90)] \\ \mathbf{X_2}' &= [\cos(T+90), \sin(T+90), 0)] \\ \mathbf{X_3}' &= [\cos(T)\cos(P), \sin(T)\cos(P), \sin(P)]\end{aligned}$$

which simplifies to:

$$\begin{aligned}\mathbf{X_1}' &= [\cos(T)\sin(P), \sin(T)\sin(P), -\cos(P)] \\ \mathbf{X_2}' &= [-\sin(T), \cos(T), 0] \\ \mathbf{X_3}' &= [\cos(T)\cos(P), \sin(T)\cos(P), \sin(P)]\end{aligned} \tag{5.14}$$

The direction cosines of the new $\mathbf{X_1}'\mathbf{X_2}'\mathbf{X_3}'$ axes into the old **ENU** axes are the dot product between the new and old axes:

$$\mathbf{a} = \begin{pmatrix} \mathbf{X_1}' \cdot \mathbf{E} & \mathbf{X_1}' \cdot \mathbf{N} & \mathbf{X_1}' \cdot \mathbf{U} \\ \mathbf{X_2}' \cdot \mathbf{E} & \mathbf{X_2}' \cdot \mathbf{N} & \mathbf{X_2}' \cdot \mathbf{U} \\ \mathbf{X_3}' \cdot \mathbf{E} & \mathbf{X_3}' \cdot \mathbf{N} & \mathbf{X_3}' \cdot \mathbf{U} \end{pmatrix}$$

which is equal to:

$$\mathbf{a} = \begin{pmatrix} \sin(T)\sin(P) & \cos(T)\sin(P) & \cos(P) \\ \cos(T) & -\sin(T) & 0 \\ \sin(T)\cos(P) & \cos(T)\cos(P) & -\sin(P) \end{pmatrix} \tag{5.15}$$

This is the transformation matrix $\mathbf{a}$ for the Down-Plunge projection. To project i points on the fold with coordinates $[E_i, N_i, U_i]$, we just need to do the following:

$$\begin{aligned}X'_{1i} &= \sin(T)\sin(P)E_i + \cos(T)\sin(P)N_i + \cos(P)U_i \\ X'_{2i} &= \cos(T)E_i - \sin(T)N_i\end{aligned} \tag{5.16}$$

and then plot X'_{2i} against X'_{1i} (Fig. 5.6) to draw the Down-Plunge projection of the fold.

The function *DownPlunge* computes the Down-Plunge projection of a bed from the **ENU** coordinates of points on the bed, and the fold axis orientation:

```
import numpy as np

def DownPlunge(bedseg,trd,plg):
  '''
  DownPlunge constructs the down plunge projection of a bed

  bedseg is a npoints x 3 array, which holds npoints
  on the digitized bed, each point defined by
  3 coordinates: X1 = East, X2 = North, X3 = Up

  trd and plg are the trend and plunge of the fold axis

  NOTE: trd and plg should be entered in radians

  Python function translated from the Matlab function
  DownPlunge in Allmendinger et al. (2012)
  '''
  # Number of points in bed
  nvtex = bedseg.shape[0]

  # Allocate some arrays
  a=np.zeros((3,3))
  dpbedseg = np.zeros((np.shape(bedseg)))

  # Calculate the transformation matrix a(i,j) Eq. 5.15
  a[0,0] = np.sin(trd)*np.sin(plg)
  a[0,1] = np.cos(trd)*np.sin(plg)
  a[0,2] = np.cos(plg)
  a[1,0] = np.cos(trd)
  a[1,1] = -np.sin(trd)
  a[2,0] = np.sin(trd)*np.cos(plg)
  a[2,1] = np.cos(trd)*np.cos(plg)
  a[2,2] = -np.sin(plg)

  # Perform transformation
  for nv in range(0,nvtex,1):
    for i in range(0,3,1):
      dpbedseg[nv,i] = 0.0
      for j in range(0,3,1):
          dpbedseg[nv,i] = a[i,j]*bedseg[nv,j] + dpbedseg[nv,
   i]

  return dpbedseg
```

Let's use this function to draw the Down-Plunge projection of the Big Elk

anticline in southeastern Idaho, USA (Fig. 5.7; Albee and Cullins, 1975; Allmendinger et al., 2012). Text files contain the digitized contacts (**ENU**) of three tops along the fold: the Jurassic Twin Creek Limestone (*jtc.txt*), the Jurassic Stump Sandstone (*js.txt*), and the Cretaceous Peterson Limestone (*kp.txt*). The trend and plunge of the folds axis is 125/126 (in the next chapter we will see how to compute this axis). The notebook ch5-4 shows the solution to this problem:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt

# Import function DownPlunge
import sys, os
sys.path.append(os.path.abspath('../functions'))
from DownPlunge import DownPlunge as DownPlunge

# Trend and plunge of the fold axis in radians
trend = 125 * np.pi/180
plunge = 26 * np.pi/180

# Read the tops from the text files
jtc = np.loadtxt(os.path.abspath('../data/ch5-4/jtc.txt'))
js = np.loadtxt(os.path.abspath('../data/ch5-4/js.txt'))
kp = np.loadtxt(os.path.abspath('../data/ch5-4/kp.txt'))

# Transform the points
jtcdp = DownPlunge(jtc,trend,plunge)
jsdp = DownPlunge (js,trend,plunge)
kpdp = DownPlunge(kp,trend,plunge)

# Plot the down plunge section
plt.plot(jtcdp[:,1],jtcdp[:,0],'k-')
plt.plot(jsdp[:,1],jsdp[:,0],'r-')
plt.plot(kpdp[:,1],kpdp[:,0],'b-')

# Display the section's orientation
# Notice that the fold axis plunges SE
# Therefore the left side of the section is NE
# and the right side is SW
plt.text(-2.1e4,12e3,'NE')
plt.text(-0.6e4,12e3,'SW')

# Make axes equal
plt.axis('equal');
```

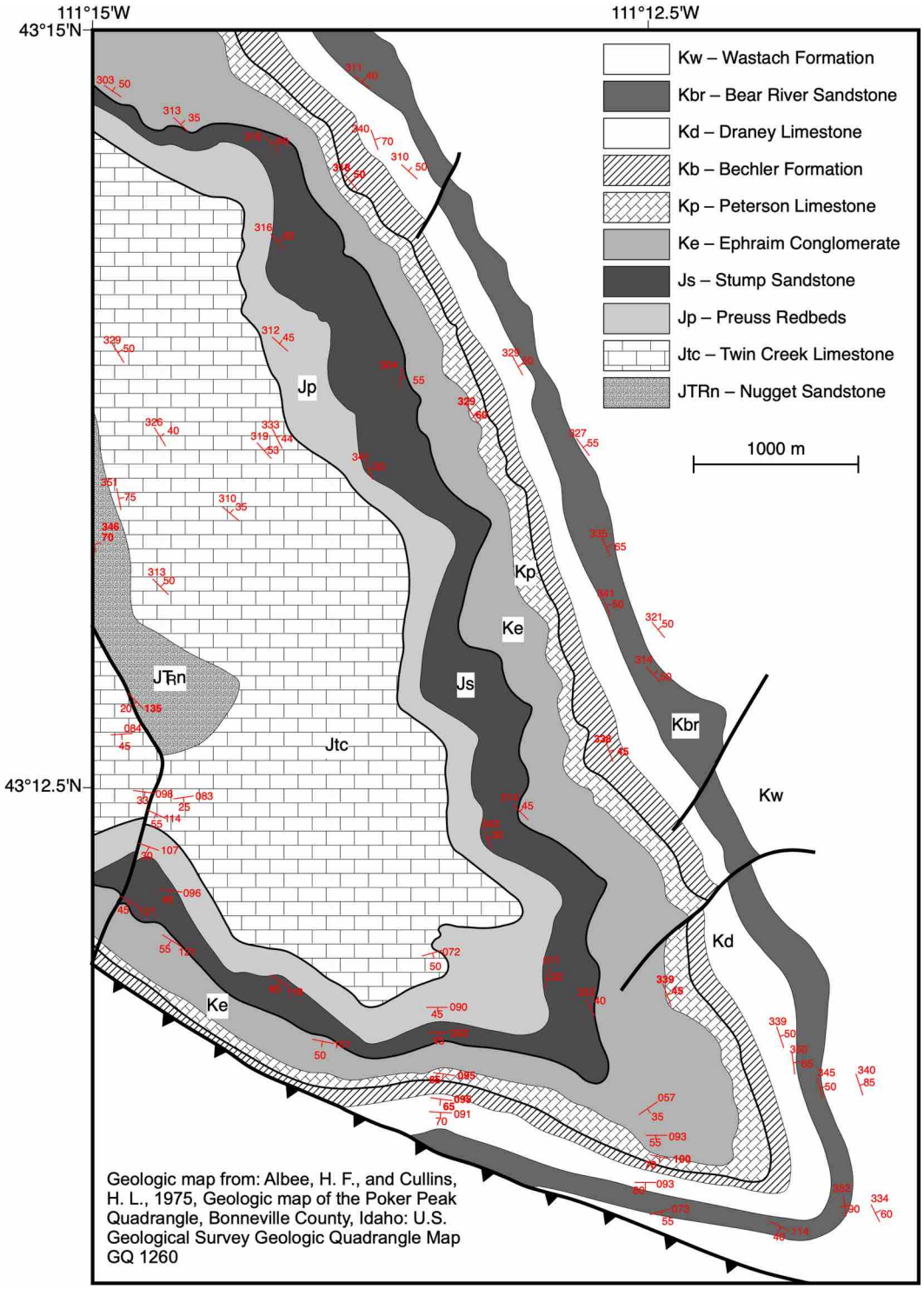
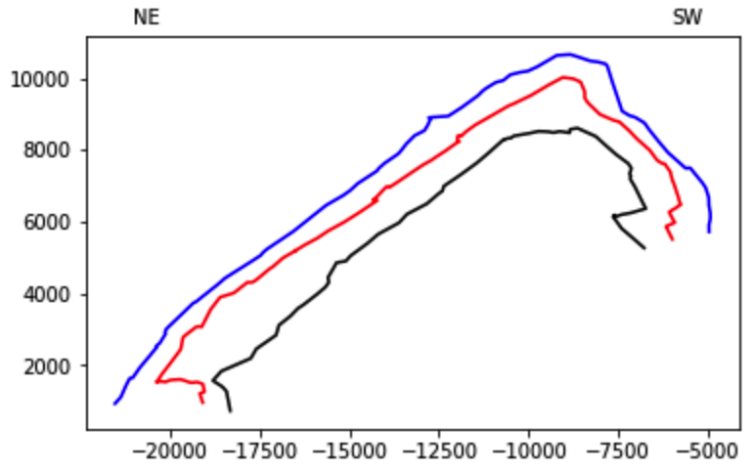


Figure 5.7: Simplified geologic map of the Big Elk anticline in southeastern Idaho. Modified from Allmendinger et al. (2012).

Output:



The thrust verges to the NE (the direction of transport is towards the NE), yet the fold verges to the SW in the opposite direction. Unit Jp, between Jtc and Js, contains evaporites (Fig. 5.7). Could this explain the observed fold geometry?

5.2.4 Rotations

Rotations are essential in geology. For example, when we measure the orientation of current lineations on tilted beds, and we want to estimate the orientation of the paleocurrent that deposited these beds, we need to rotate or unfold the beds (and the lineations) back to their pre-tilting orientation. The stereonet is a convenient device to rotate data around a horizontal axis or a vertical axis, but is rather difficult to rotate data around an axis of a different orientation (Marshak and Mitra, 1988).

A rotation is also a coordinate transformation from an old coordinate system $\mathbf{X_1X_2X_3}$ to a new coordinate system $\mathbf{X_1'X_2'X_3'}$ (Fig. 5.8). The rotation axis is specified by its trend and plunge, and the magnitude of rotation is given as an angle ω that is positive if the rotation is clockwise about the given axis and vice versa (Fig. 5.8). The difficult part is that the rotation axis may not coincide with the axes of either the old or the new coordinate system (unlike the Down-Plunge projection).

The components of the transformation matrix $\mathbf{a}$ defining the rotation, which

are the direction cosines of the new coordinate system $\mathbf{X_1'X_2'X_3'}$ with regards to the old coordinate system $\mathbf{X_1X_2X_3}$, can be found using spherical trigonometry (Allmendinger et al., 2012) or linear algebra as we did before. Here, we give them without proof. If $\cos\alpha$, $\cos\beta$ and $\cos\gamma$ are the direction cosines of the rotation axis in the **NED** coordinate system (Table 4.1), and ω is the amount of rotation, the elements of matrix **a** are:

$$\begin{aligned}
a_{11} &= \cos\omega + \cos^2\alpha(1-\cos\omega) \\
a_{12} &= -\cos\gamma\sin\omega + \cos\alpha\cos\beta(1-\cos\omega) \\
a_{13} &= \cos\beta\sin\omega + \cos\alpha\cos\gamma(1-\cos\omega) \\
a_{21} &= \cos\gamma\sin\omega + \cos\beta\cos\alpha(1-\cos\omega) \\
a_{22} &= \cos\omega + \cos^2\beta(1-\cos\omega) \\
a_{23} &= -\cos\alpha\sin\omega + \cos\beta\cos\gamma(1-\cos\omega) \\
a_{31} &= -\cos\beta\sin\omega + \cos\gamma\cos\alpha(1-\cos\omega) \\
a_{32} &= \cos\alpha\sin\omega + \cos\gamma\cos\beta(1-\cos\omega) \\
a_{33} &= \cos\omega + \cos^2\gamma(1-\cos\omega)
\end{aligned} \tag{5.17}$$

The function *Rotate* rotates a line (*trd* and *plg*) about a rotation axis (*rtrd* and *rplg*), an amount ω (*rot*):

```
import numpy as np
from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph

def Rotate(rtrd,rplg,rot,trd,plg,ans0):
  '''
  Rotate rotates a line by performing a coordinate
  transformation. The algorithm was originally written
  by Randall A. Marrett

  rtrd = trend of rotation axis
  rplg = plunge of rotation axis
  rot = magnitude of rotation
  trd = trend of the vector to be rotated
  plg = plunge of the vector to be rotated
  ans0 = A character indicating whether the line to be
    rotated is an axis (ans0 = 'a') or a vector
    (ans0 = 'v')
  trdr and plgr are the trend and plunge of the
    rotated line

```

```
  NOTE: All angles are in radians

  Rotate uses functions SphToCart and CartToSph

  Python function translated from the Matlab function
  Rotate in Allmendinger et al. (2012)
  '''
  # Allocate some arrays
  a = np.zeros((3,3)) #Transformation matrix
  pole = np.zeros(3) #Dir. cosines of rotation axis
  plotr = np.zeros(3) #Dir. cosines of rotated vector
  temp = np.zeros(3)  #Dir. cosines of unrotated vector

  # Convert rotation axis to direction cosines. The
  # convention here is X1 = North, X2 = East, X3 = Down
  pole[0] , pole[1], pole[2] = SphToCart(rtrd,rplg,0)

  # Calculate the transformation matrix a for the rotation
  # Eq. 5.17
  x = 1.0 - np.cos(rot)
  sinRot = np.sin(rot)
  cosRot = np.cos(rot)
  a[0,0] = cosRot + pole[0]*pole[0]*x
  a[0,1] = -pole[2]*sinRot + pole[0]*pole[1]*x
  a[0,2] = pole[1]*sinRot + pole[0]*pole[2]*x
  a[1,0] = pole[2]*sinRot + pole[1]*pole[0]*x
  a[1,1] = cosRot + pole[1]*pole[1]*x
  a[1,2] = -pole[0]*sinRot + pole[1]*pole[2]*x
  a[2,0] = -pole[1]*sinRot + pole[2]*pole[0]*x
  a[2,1] = pole[0]*sinRot + pole[2]*pole[1]*x
  a[2,2] = cosRot + pole[2]*pole[2]*x

  # Convert trend and plunge of vector to be rotated into
  # direction cosines
  temp[0] , temp[1], temp[2] = SphToCart(trd,plg,0)

  # Perform the coordinate transformation
  for i in range(0,3,1):
    plotr[i] = 0.0
    for j in range(0,3,1):
      plotr[i] = a[i,j]*temp[j] + plotr[i]

  # Convert to lower hemisphere projection if data are
  # axes (ans0 = 'a')
  if plotr[2] < 0.0 and ans0 == 'a' :
    plotr[0] = -plotr[0]
    plotr[1] = -plotr[1]
    plotr[2] = -plotr[2]

```

```
71    # Convert from direction cosines back to trend and plunge
72    trdr , plgr = CartToSph(plotr[0], plotr[1], plotr[2])
73
74    return trdr, plgr
```

The notebook ch5-5 illustrates the use of the function *Rotate* to solve the following problem from Leyshon and Lisle (1996): An overturned bed oriented 305/60 (RHR) has sedimentary lineations which indicate the palaeocurrent direction. These pitch at 60NW, with the current flowing up the plunge. Calculate the original trend of the palaeocurrents.

Besides rotating the lineations back to their pre-tilted orientation, there is an additional challenge in this problem. We need to figure out the orientation of the lineations from their pitch on the bed. We will do this as well using a rotation.

```
1  import math
2  pi = math.pi
3
4  # Import function Rotate and related functions
5  import sys, os
6  sys.path.append(os.path.abspath('../functions'))
7  from Pole import Pole as Pole
8  from Rotate import Rotate as Rotate
9  from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi
10
11 # Strike and dip of bed in radians
12 strike = 305*pi/180
13 dip = 60*pi/180
14
15 # Pole of bed
16 rtrd, rplg = Pole(strike, dip, 1)
17
18 # To find the orientation of the lineations
19 # rotate the strike line clockwise about the
20 # pole an amount equal to the pitch
21
22 # strike line
23 trd = strike
24 plg = 0
25
26 # rotation = pitch in radians
27 rot = 60 * pi/180
28
29 # orientation of lineations
```

```
trdr, plgr = Rotate(rtrd,rplg,rot,trd,plg,'a')

# Now we need to rotate the lineations about
# the strike line to their pre-tilted orientation

# The bed is overturned, so it has been rotated
# pass the vertical. The amount of rotation
# required to restore the bed to its pre-tilted
# orientation is 180- 60 = 120 deg, and it
# should be clockwise
rot = 120 * pi/180 # rotation in radians

# rotate lineations to their pre-tilted orientation
trdl, plgl = Rotate(trd,plg,rot,trdr,plgr,'a')

# The current flows up the plunge,
# so the trend of the paleocurrents is:
trdl = ZeroTwoPi(trdl + pi)
print('Original trend of the paleocurrents = {:.1f}'.format(
    trdl*180/pi))
```

Output:
Original trend of the paleocurrents = 65.0

5.2.5 Plotting great and small circles using rotations

The transformation matrix **a** that describes the rotation (Eq. 5.17), provides a simple and elegant way to draw great and small circles on a stereonet. As you may suspect from the previous notebook, to draw a great circle, we just need to rotate the strike line of the plane in fixed increments (e.g. 1°) around the pole to the plane until completing 180°. This is the same as drawing lines on the plane of incrementally larger rake from 0 to 180°. To draw a small circle, we just need to incrementally rotate (e.g. 1°increments) a line around the axis of the conic section until completing 360°. Any line making an angle less than 90°with the axis of rotation will trace out a cone, which plots on the stereonet as a small circle. The functions *GreatCircle* and *SmallCircle* return the path of great and small circles on an equal angle or equal area stereonet:

```
import numpy as np
from Pole import Pole as Pole
from Rotate import Rotate as Rotate
from StCoordLine import StCoordLine as StCoordLine

def GreatCircle(strike,dip,sttype):
  '''
  GreatCircle computes the great circle path of a plane
  in an equal angle or equal area stereonet of unit radius

  strike = strike of plane
  dip = dip of plane
  sttype = type of stereonet: 0 = equal angle, 1 = equal area
  path = x and y coordinates of points in great circle path

  NOTE: strike and dip should be entered in radians.

  GreatCircle uses functions StCoordLine, Pole and Rotate

  Python function translated from the Matlab function
  GreatCircle in Allmendinger et al. (2012)
  '''
  pi = np.pi
  # Compute the pole to the plane. This will be the axis of
  # rotation to make the great circle
  trda, plga = Pole(strike,dip,1)

  # Now pick the strike line at the intersection of the
  # great circle with the primitive of the stereonet
  trd = strike
  plg = 0.0

  # To make the great circle, rotate the line 180 degrees
  # in increments of 1 degree
  rot = np.arange(0,181,1)*pi/180
  path = np.zeros((rot.shape[0],2))

  for i in range(rot.shape[0]):
  # Avoid joining ends of path
    if rot[i] == pi:
      rot[i] = rot[i]*0.9999
    # Rotate line
    rtrd, rplg = Rotate(trda,plga,rot[i],trd,plg,'a')
    # Calculate stereonet coordinates of rotated line
    # and add to great circle path
    path[i,0], path[i,1] = StCoordLine(rtrd,rplg,sttype)

  return path
```

```
import numpy as np
from Rotate import Rotate as Rotate
from StCoordLine import StCoordLine as StCoordLine
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def SmallCircle(trda,plga,coneAngle,sttype):
  '''
  SmallCircle computes the paths of a small circle defined
  by its axis and cone angle, for an equal angle or equal
  area stereonet of unit radius

  trda = trend of axis
  plga = plunge of axis
  coneAngle = cone angle
  sttype = type of stereonet. 0 = equal angle, 1 = equal area
  path1 and path2 = vectors with the x and y coordinates of
        the points in the small circle paths
  np1 and np2 = Number of points in path1 and path2

  NOTE: All angles should be in radians

  SmallCircle uses functions ZeroTwoPi, StCoordLine and
  Rotate

  Python function translated from the Matlab function
  SmallCircle in Allmendinger et al. (2012)
  '''
  pi = np.pi
  # Find where to start the small circle
  if (plga - coneAngle) >= 0.0:
    trd = trda
    plg = plga - coneAngle
  else:
    if plga == pi/2.0:
      plga = plga*0.9999
    angle = np.arccos(np.cos(coneAngle)/np.cos(plga))
    trd = ZeroTwoPi(trda+angle)
    plg = 0.0

  # To make the small circle, rotate the starting line
  # 360 degrees in increments of 1 degree
  rot = np.arange(0,361,1)*pi/180
  path1 = np.zeros((rot.shape[0],2))
  path2 = np.zeros((rot.shape[0],2))
  np1 = 0
  np2 = 0
  for i in range(rot.shape[0]):
    # Rotate line: Notice that the line is considered as
```

```
    # a vector
    rtrd , rplg = Rotate(trda,plga,rot[i],trd,plg,'v')
    # Calculate stereonet coordinates and add to the
    # right path
    # If plunge of rotated line is positive add to
    # first path
    if rplg >= 0.0:
      path1[np1,0] , path1[np1,1] = StCoordLine(rtrd,rplg,
    sttype)
      np1 = np1 +1
    # Else add to the second path
    else:
      path2[np2,0] , path2[np2,1] = StCoordLine(rtrd,rplg,
    sttype)
      np2 = np2 +1

  return path1, path2, np1, np2
```

Normally, stereonets are displayed with the primitive equal to the horizontal (i.e. looking straight down). However, sometimes is convenient to look at the stereonet in another orientation. For example, one may want to plot data in the plane of a cross section (the view direction is perpendicular to the cross section), or in a down-plunge projection of a fold (the view direction is parallel to the fold axis). The function *GeogrToView* enables to plot great and small circles on a stereonet of any view direction, by transforming the poles of rotation from **NED** coordinates to the view direction coordinates:

```
import numpy as np
from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def GeogrToView(trd,plg,trdv,plgv):
  '''
  GeogrToView transforms a line from NED to View Direction
  coordinates
  trd = trend of line
  plg = plunge of line
  trdv = trend of view direction
  plgv = plunge of view direction
  rtrd and rplg are the new trend and plunge of the line
  in the view direction.

  NOTE: Input/Output angles are in radians

```

```
  GeogrToView uses functions ZeroTwoPi, SphToCart and
  CartToSph

  Python function translated from the Matlab function
  GeogrToView in Allmendinger et al. (2012)
  '''
  #some constants
  east = np.pi/2.0

  #Make transformation matrix between NED and View Direction
  a = np.zeros((3,3))
  a[2,0], a[2,1], a[2,2] = SphToCart(trdv,plgv,0)
  temp1 = trdv + east
  temp2 = 0.0
  a[1,0], a[1,1], a[1,2] = SphToCart(temp1,temp2,0)
  temp1 = trdv
  temp2 = plgv - east
  a[0,0], a[0,1], a[0,2] = SphToCart(temp1,temp2,0)

  #Direction cosines of line
  dirCos = np.zeros(3)
  dirCos[0], dirCos[1], dirCos[2] = SphToCart(trd,plg,0)

  # Transform line
  nDirCos = np.zeros(3)
  for i in range(0,3,1):
    nDirCos[i] = a[i,0]*dirCos[0] + a[i,1]*dirCos[1]+ a[i,2]*
   dirCos[2]

  # Compute line from new direction cosines
  rtrd, rplg = CartToSph(nDirCos[0],nDirCos[1],nDirCos[2])

  # Take care of negative plunges
  if rplg < 0.0 :
    rtrd = ZeroTwoPi(rtrd+np.pi)
    rplg = -rplg

  return rtrd, rplg
```

We put these three functions together in a function called *Stereonet*, that plots an equal angle or equal area stereonet in any view direction:

```
import numpy as np
import matplotlib.pyplot as plt
from Pole import Pole as Pole
from GeogrToView import GeogrToView as GeogrToView
```

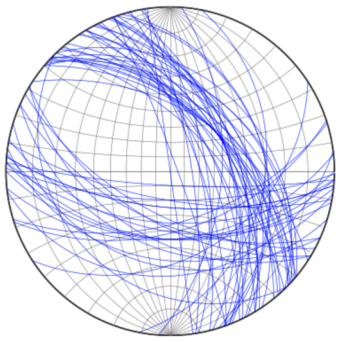
```
from SmallCircle import SmallCircle as SmallCircle
from GreatCircle import GreatCircle as GreatCircle

def Stereonet(trdv,plgv,intrad,sttype):
  '''
  Stereonet plots an equal angle or equal area stereonet
  of unit radius in any view direction

  trdv = trend of view direction
  plgv = plunge of view direction
  intrad = interval in radians between great or small circles
  sttype = type of stereonet. 0 = equal angle, 1 = equal area

  NOTE: All angles should be entered in radians

  Stereonet uses functions Pole, GeogrToView,
  SmallCircle and GreatCircle

  Python function translated from the Matlab function
  Stereonet in Allmendinger et al. (2012)
  '''
  pi = np.pi
  # some constants
  east = pi/2.0
  west = 3.0*east

  # Plot stereonet reference circle
  r = 1.0 # radius pf stereonet
  TH = np.arange(0,360,1)*pi/180
  X = r * np.cos(TH)
  Y = r * np.sin(TH)

  # Make a larger figure
  plt.rcParams['figure.figsize'] = [15, 7.5]
  plt.plot(X,Y, 'k')
  plt.axis ([-1, 1, -1, 1])
  plt.axis ('equal')
  plt.axis('off')

  # Number of small circles
  nCircles = int(pi/(intrad*2.0))

  # small circles
  # start at the North
  trd = 0.0
  plg = 0.0

  # If view direction is not the default (trdv=0,plgv=90)
  # transform line to view direction
```

```
54    if trdv != 0.0 and plgv != east:
55      trd, plg = GeogrToView(trd,plg,trdv,plgv)
56
57    # Plot small circles
58    for i in range(1,nCircles+1):
59      coneAngle = i*intrad
60      path1, path2, np1, np2 = SmallCircle(trd,plg,coneAngle,
       sttype)
61      plt.plot(path1[np.arange(0,np1),0], path1[np.arange(0,np1
       ),1], color='gray',linewidth=0.5)
62      if np2 > 0:
63        plt.plot(path2[np.arange(0,np2),0], path2[np.arange(0,
       np2),1], color='gray', linewidth=0.5)
64
65    # Great circles
66    for i in range(0,nCircles*2+1):
67      # Western half
68      if i <= nCircles:
69        # Pole of great circle
70        trd = west
71        plg = i*intrad
72        # Eastern half
73      else:
74        # Pole of great circle
75        trd = east
76        plg = (i-nCircles)*intrad
77      # If pole is vertical shift it a little bit
78      if plg == east:
79        plg = plg * 0.9999
80      # If view direction is not the default
81      # (trdv = 0,plgv = 90)
82      # transform line to view direction
83      if trdv != 0.0 and plgv != east:
84        trd, plg = GeogrToView(trd,plg,trdv,plgv)
85      # Compute plane from pole
86      strike, dip = Pole(trd,plg,0)
87      # Plot great circle
88      path = GreatCircle(strike,dip,sttype)
89      plt.plot(path[:,0],path[:,1],color='gray',linewidth=0.5)
```

Now, let's use all these functions to plot the bedding data of the Big Elk anticline (Fig. 5.7) in an equal angle stereonet, looking down and also along the fold axis. The notebook ch5-6 illustrates this. Notice that we read the strike and dips from the file *beasd.txt*:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
%matplotlib  inline
pi = np.pi

# Import function Stereonet and related functions
import sys, os
sys.path.append(os.path.abspath('../functions'))
from Pole import Pole as Pole
from GreatCircle import GreatCircle as GreatCircle
from GeogrToView import GeogrToView as GeogrToView
from Stereonet import Stereonet as Stereonet

# Draw a lower hemisphere equal angle stereonet
trdv = 0
plgv = 90 * pi/180
intrad = 10 * pi/180
Stereonet(trdv,plgv,intrad,0)

# Read the strike-dip data from the Big Elk anticline
beasd = np.loadtxt('beasd.txt')

# Plot the great circles
for i in range(beasd.shape[0]):
    path = GreatCircle(beasd[i,0]*pi/180,beasd[i,1]*pi/180,0)
    plt.plot(path[:,0], path[:,1], 'b', linewidth=0.5)
```

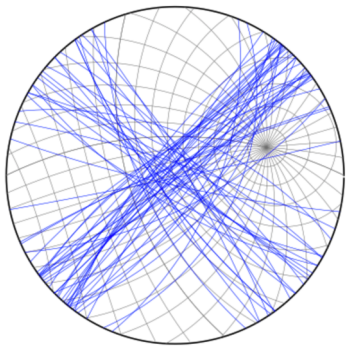
Output:



```
# Draw the same data in an equal angle stereonet,
# but make the view direction = fold axis
trdv = 125*pi/180
plgv = 26*pi/180
Stereonet(trdv,plgv,intrad,0)

```

```
# Plot the great circles
for i in range(beasd.shape[0]):
    # pole to bed
    trdp, plgp =  Pole(beasd[i,0]*pi/180,beasd[i,1]*pi/180,1)
    # transform pole
    trdpt, plgpt = GeogrToView(trdp,plgp,trdv,plgv)
    # bed from transformed pole
    striket, dipt = Pole(trdpt,plgpt,0)
    # plot great circle
    path = GreatCircle(striket,dipt,0)
    plt.plot(path[:,0], path[:,1], 'b', linewidth=0.5)
```

Output:



What do these plots tell you about the geometry of the fold?

5.3 Exercises

1. Figure 5.8 is a satellite image of the Sheep Mountain anticline, Wyoming, USA (Rioux, 1994). At 75 localities along the anticline in the Jurassic Sundance Formation (gray-green sandstone, siltstone and shale; Rioux, 1994), the **ENU** coordinates of points on the base and top of the unit (Fig. 5.8a, green and red points), and three points on a bed inside the unit (Fig. 5.8b, blue points), were recorded in Google Earth. You can visualize these points in Google Earth using the file *sdtp.kml*.

   The file *sdtp.txt* contains the **ENU** coordinates (UTM in meters) of the points. Each row is one locality, and it contains 15 columns corresponding to the **ENU** coordinates of points 1 to 5. Points 1 to 3 are on a bed inside the unit, and points 4 and 5 are on the base and top

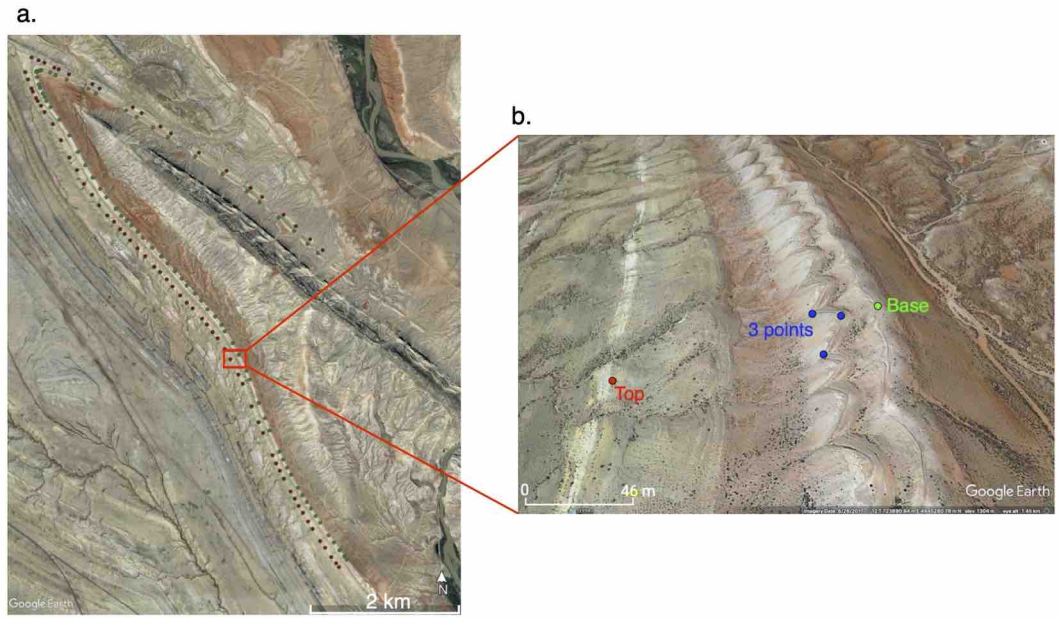


Figure 5.8: **a.** Base (green) and top (red) locations of the Jurassic Sundance Formation along the Sheep Mountain anticline, Wyoming, for exercise 1. **b.** Closeup of one of the localities (red rectangle in a) showing as well the three points (blue) used to determine strike and dip.

of the unit, respectively. Columns 1 to 3 are the **ENU** coordinates of point 1, columns 4 to 6 those of point 2, and so on.

(a) Compute the thickness of the Sundance Formation at the 75 localities. Notice that at each locality you will need to compute the strike and dip using points 1-3 and the *ThreePoint* function, and the thickness using points 4 and 5 and the *TrueThickness* function.

(b) Plot the bedding data along the anticline in and equal area, lower hemisphere stereonet. Plot these data as great circles.

(c) Suppose that the error in horizontal (**EN**) coordinates is ± 3 m and in the vertical is ± 1.5 m. These are conservative error estimates for Google Earth. What are the errors in strike and dip and thickness at the 75 localities? *Hint*: Use functions *ThreePointU* and *TrueThicknessU*.

(d) Make a plot of computed thickness versus computed dip, and another plot of computed thickness versus north (**N**) for the 75 lo-

calities. Include the thickness and dip error bars. Do you see any correlation between thickness and bedding dip? Is there a systematic variation of thickness along the anticline?

(e) The axis of the anticline is oriented 306/11. This is approximately the location where the bedding planes on the stereonet intersect. In the next chapter we will see how to compute this axis. Make a Down-Plunge projection of the base and top of the Jurassic Sundance Formation. *Hint*: Project the 75 base and top locations using the function *DownPlunge*.

2. Allmendinger and Judge (2013) discuss the uncertainty in thickness measurements and its implication for estimating the shortening of fold and thrust belts. Figure 5.9 is a geologic map of the Canmore east half area in Alberta, Canada (Price, 1970). The yellow dots are the points used by these authors to estimate the thickness of the Devonian Palliser (Dpa) and Mississippian Livingstone (Mlv) formations. These are in a homoclinal dip package in a thrust sheet.

   The files *dpa.txt* and *mlv.txt* contain the **ENU** coordinates (UTM in meters) of the points in the Palliser (Dpa) and Livingstone (Mlv) formations, respectively. Each row in the files is one locality, and it contains 12 columns corresponding to the **ENU** coordinates of points 1 to 4. Points 1-3 are either on the top or the base of the unit, and point 4 is on the other contact. Columns 1 to 3 are the **ENU** coordinates of point 1, columns 4 to 6 those of point 2, and so on.

   (a) Compute the thickness of the Palliser and Livingstone formations at the localities shown in Figure 5.9. What is the mean value of thickness of these two units? What is the standard deviation? *Hint*: Use the functions *ThreePoint* and *TrueThickness*. The NumPy functions *mean* and *std* compute the mean and standard deviation of an array.

   (b) Consider that the uncertainty in horizontal and vertical coordinates is ± 15.24 m. Compute again the mean value and standard deviation of the thickness of the units. This time these values will have uncertainties. *Hint*: Use the functions *ThreePointU* and *TrueThicknessU*. Notice that you can use the NumPy functions *mean* and *std* on arrays made of numbers with uncertainties (*uFloat*).

   (c) How do your results compare to those of Allmendinger and Judge (2013, their Table 1)?

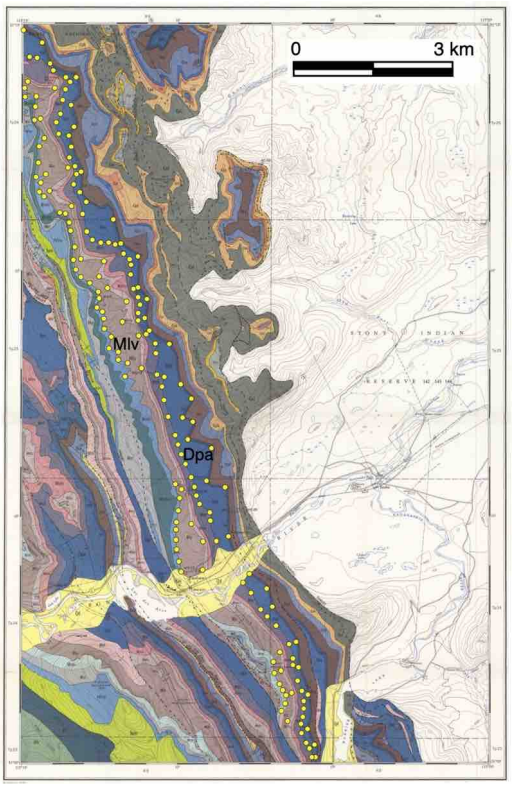


Figure 5.9: Geologic map of the Canmore east half area, Alberta, Canada (Price, 1970) for exercise 2. The yellow dots are points used to determine the thickness of the Devonian Palliser (Dpa) and Mississippian Livingstone (Mlv) formations.

3. The following exercise is from Ragan (2009): With the following information and the topographic map of Fig. 5.10, construct a geological map. The base of a 100 m thick sandstone unit of early Triassic age is exposed at point A; its attitude is 110/25 (RHR). Point B is on the east boundary of a 50 m thick, vertical diabase dike of Jurassic age; its trend is 020. At point C, the base of a horizontal Cretaceous sequence is exposed and at point D the base of a conformable sequence of Tertiary rocks is present.

   This exercise is normally solved graphically (Fig. 3.9). Here, you are going to use computation. The files *XGE.txt*, *YGE.txt*, and *ZGE.txt* are the **ENU** coordinates of the points of the DEM grid (these arrays follow the format of the NumPy *meshgrid* function). Points A, B, C and D have the following coordinates **ENU** coordinates:

   A = [232, 428, 370]

   B = [612, 322, 355]

   C = [281.5, 239, 395]

   D = [537, 183, 410]

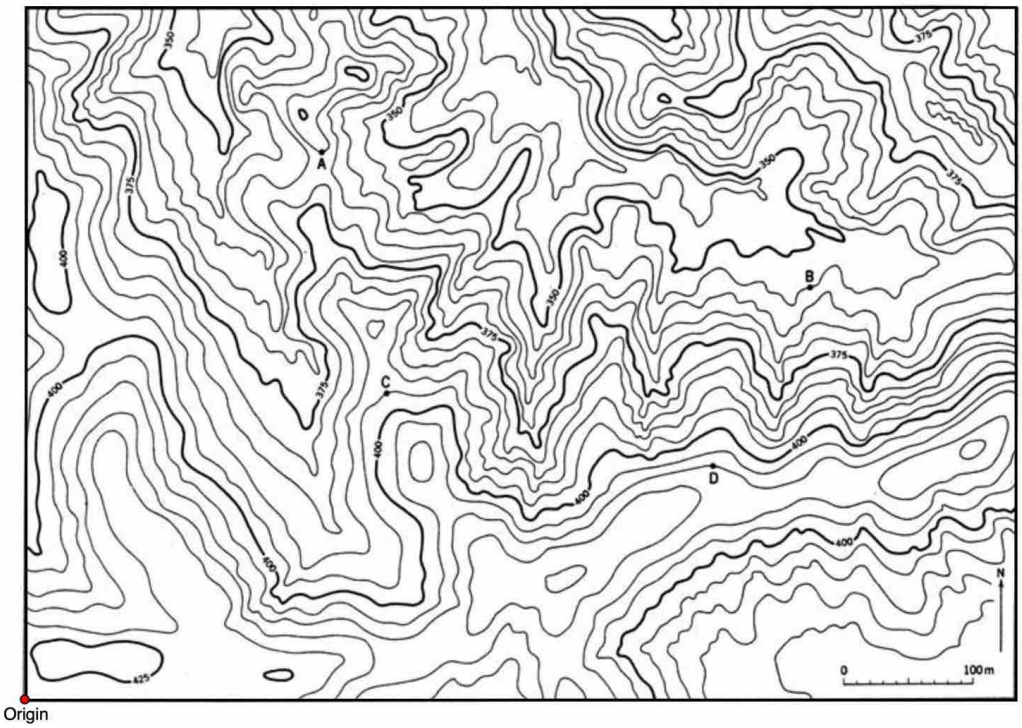


Figure 5.10: Topographic map for exercise 3. Notice that the origin is at the lower left corner of the map. This is exercise 2 in chapter 4 of Ragan (2009).

Hint: Use the function *OutcropTrace* to compute the contacts. Follow a procedure similar to notebook ch5-3 to draw the topographic contours and the contacts. In this case though, some contacts will intersect. At contacts intersections you will need to use the principle of superposition to determine which contact cuts the other one: e.g. the Jurassic dyke will cut the Triassic sequence, but it will be covered by the Cretaceous and Tertiary sequences.

4. This exercise is from Marshak and Mitra (1988). It is probably one of the hardest exercises to solve using rotations on a stereonet. Here, you will use the function *Rotate* to solve this problem:

   In eastern New York, there is an important unconformity called the Taconic unconformity. It separates Mid-Ordovician flysch from Devonian limestone. At a locality near the town of Catskill, the limestone (and the unconformity) is oriented 195/44 (RHR). An anticline occurs in the underlying flysch. One limb is presently oriented 240/73 (limb 1), and the other limb is presently oriented 020/41 (limb 2) (RHR). Flute

casts occur in the Ordovician strata on limb 1 (NW dipping limb) and they have an orientation of 037/52.

(a) What was the orientation of each of the two fold limbs before tilting of the unconformity? *Hint*: Rotate the pole of the unconformity about the strike line of the unconformity to make the pole vertical (and the unconformity horizontal). Apply the same rotation to the poles of the limbs and the flute casts. Find the orientation of the limbs from their rotated poles.

(b) What was the orientation of the fold axis prior to tilting? *Hint*: The fold axis is the intersection of the limbs before tilting of the unconformity. Use the function *Angles* to solve this.

(c) What was the trend of the current direction responsible for the formation of the flute casts in Ordovician time? *Hint*: Rotate the fold axis computed in (b) about a horizontal line perpendicular to it, to bring the axis to the horizontal. Apply the same rotation to the poles of the limbs and the flute casts. Then, rotate the poles of the limbs about the horizontal fold axis to make them vertical (and the limbs horizontal). Apply the same rotation to the flute casts. If the rotations are correct, the flute casts should be horizontal and their trend is the orientation of the paleocurrent in the Ordovician.

(d) What is the present orientation of the flute casts on limb 2 (SE dipping limb)? *Hint*: The trend of the flute casts on limb 2 in the Ordovician is the trend you got in (c) minus 180°. The plunge of the flute casts is of course zero. Apply to the flute casts on limb 2, the inverse of the rotations you applied to the pole of limb 2 to bring them back to their present orientation.

(e) Plot the results of (a), (b), (c) and (d) as poles and lines on a stereonet. Use colors to indicate the different stages, and symbols to indicate the different elements: unconformity, limbs, fold axis, and flute casts.

(f) Write a Python function to draw the arc of a rotation on a stereonet. Indicate the sense of rotation with an arrow at the end of the arc. Use this function to draw the arcs of the rotations in this problem.

References

Albee, H.F. and Cullins, H.L. 1975. Geologic Map of the Poker Peak Quadrangle, Bonneville County, Idaho. U.S. Geological Survey, Geologic Quadrangle Map GQ 1260.

Allmendinger, R.W., Cardozo, N. and Fisher, D.W. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Ammendinger, R.W. and Judge, P. 2013. Stratigraphic uncertainty and errors in shortening from balanced sections in the North American Cordillera. GSA Bulletin 125, 1569-1579.

Allmendinger, R.W. 2019. Modern Structural Practice: A structural geology laboratory manual for the 21st century. [Online]. [Accessed January, 2020].

Bennison, G.M., Olver, P.A. and Moseley, K.A. 2011. An Introduction to Geological Structures and Maps, 8th edition. Hodder Education.

Leyshon, P.R. and Lisle, R.J. 1996. Stereographic Projection Techniques in Structural Geology. Butterworth Heinemann.

Marshak, S. and Mitra, G. 1988. Basic Methods of Structural Geology. Prentice Hall.

Price, R.A. 1970. Geology, Canmore (east half), west of Fifth Meridian, Alberta: Geological Survey of Canada "A" Series Map 1265A, scale 1:50,000.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Rioux, R. L. 1994. Geologic Map of the Sheep Mountain-Little Sheep Mountain Area, Big Horn County, Wyoming, U.S. Geol. Surv. Open File Rep., 94-191.

Suppe, J. 1985. Principles of Structural Geology. Prentice-Hall.

Chapter 6

Tensors

A tensor is a physical entity that can be transformed from one coordinate system to another, changing its components in a predictable way, but without changing its fundamental nature, such that the tensor is independent of a particular coordinate system. Thus, scalars (e.g. mass, temperature and density), and vectors (e.g. velocity, force and poles to bedding) are tensors. More specifically, they are zero order (scalars) and first order (vectors) tensors. In this chapter, we will look at higher, second order tensors, which are commonly known as *tensors*. This is a short chapter, but it is fundamental to understand important concepts such as the Mohr Circle, the orientation tensor, and the mathematical background for important tensors in geology such as stress and strain.

6.1 Basic characteristics of a tensor

In three dimensions, a *tensor* is characterized by nine components:

$$\mathbf{T} = T_{ij} = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} \tag{6.1}$$

Notice that tensors are represented by capital bold letters and we use brackets to differentiate them from matrices such as the transformation matrix $\mathbf{a}$ in chapter 5. The nine components of the tensor T_{ij} give the values of the tensor

with reference to the three axes of the specific coordinate system $\mathbf{X_1X_2X_3}$. If we change the axes orientations, then the nine components will change but, similar to a vector, the tensor itself will not change.

Like matrices, tensors can be symmetric, asymmetric, or antisymmetric. If the nine components T_{ij} have different values, the tensor is asymmetric. If $T_{ij} = T_{ji}$, the tensor is symmetric. In this case the components above the principal diagonal are the same as those below the diagonal and only six components are required to define the tensor. Finally, if $T_{ij} = -T_{ji}$, the tensor is antisymmetric. In this case the components along the diagonal are zero and only three components are required to define the tensor.

Any asymmetric tensor $\mathbf{T}$ can be decomposed into a symmetric tensor $\mathbf{S}$ plus an antisymmetric tensor $\mathbf{A}$:

$$T_{ij} = S_{ij} + A_{ij} \quad where \quad S_{ij} = \frac{T_{ij} + T_{ji}}{2} \quad and \quad A_{ij} = \frac{T_{ij} - T_{ji}}{2} \tag{6.2}$$

We will use this property when dealing with infinitesimal strain (chapter 8).

For all symmetric tensors, there is one orientation of the coordinate axes for which all the components except those along the principal diagonal, are zero:

$$\mathbf{T} = T_{ij} = \begin{bmatrix} T_{11} & 0 & 0 \\ 0 & T_{22} & 0 \\ 0 & 0 & T_{33} \end{bmatrix} = \begin{bmatrix} T_2 & 0 & 0 \\ 0 & T_1 & 0 \\ 0 & 0 & T_3 \end{bmatrix} \tag{6.3}$$

Under this condition, the values along the diagonal are the principal axes of the tensor. Based on their magnitude, these axes are ranked as the maximum (T_1), intermediate (T_2), and minimum (T_3) principal axes. Notice that the indices of the principal axes do not have to coincide with the indices of the coordinate system, for example in Eq. 6.3 T_1 is parallel to the coordinate axis $\mathbf{X_2}$. The principal axes define the major, intermediate and minor axes of a three-dimensional surface known as the magnitude ellipsoid (Fig. 6.1). You have probably heard about this ellipsoid before in relation to stress or strain.

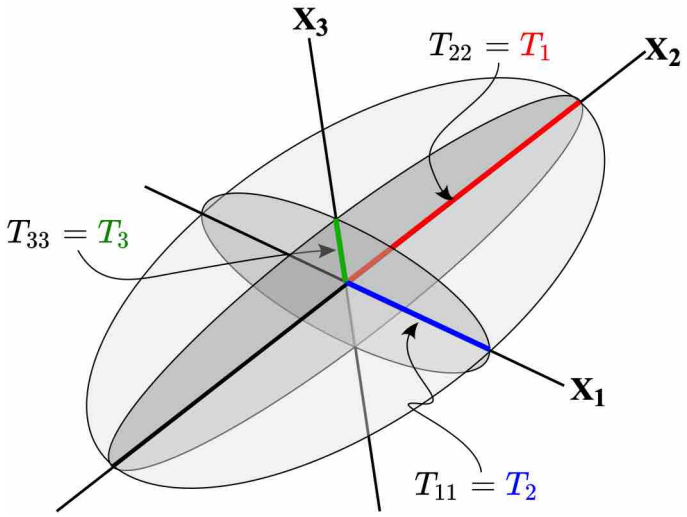


Figure 6.1: The magnitude ellipsoid and principal axes of a symmetric tensor **T** for the case described by Eq. 6.3. Notice that T_1, T_2 and T_3 define the major, intermediate and minor axes of the ellipsoid. Modified from Allmendinger et al. (2012).

6.2 Principal axes of a tensor

Determining the orientation of the coordinate system whose axes are parallel to the principal axes of a symmetric tensor involves solving the *eigenvalue* problem (Allmendinger et al., 2012). The mathematical solution to this problem gives a cubic polynomial:

$$\lambda^3 - I\lambda^2 - II\lambda - III = 0 \tag{6.4}$$

The three roots of λ are the three eigenvalues and they correspond to the magnitudes of the three principal axes. Once we know the eigenvalues, we can calculate the eigenvectors, which give the orientation of the three principal axes. Thus, we can find the principal axes of any symmetric tensor by finding its eigenvectors and eigenvalues. In Python, the NumPy *linalg.eigh* function computes the eigenvalues and eigenvectors of a symmetric array[1]. We will use this function later in the chapter when finding the principal axes of a tensor.

[1] *eigh* also sorts the eigenvalues in ascending order

The three coefficients I, II and III in Eq. 6.4 are known as the invariants of the tensor. They have the same values regardless of the coordinate system we choose. As we will see later (e.g. stress invariants in Chapter 7), these invariants are very important. Their values are given by:

$$\begin{aligned} I &= T_{11} + T_{22} + T_{33} = T_1 + T_2 + T_3 \\ II &= \frac{(T_{ij}T_{ij} - I^2)}{2} = -(T_1T_2 + T_2T_3 + T_3T_1) \\ III &= \det \mathbf{T} = |T_{ij}| = T_1T_2T_3 \end{aligned} \tag{6.5}$$

6.3 Tensors as vector operators

A tensor commonly relates two vectors, or more formally we can say that a tensor is a linear vector operator because the components of the tensor are the coefficients of a set of linear equations that relate two vectors:

$$\mathbf{u} = \mathbf{T}\mathbf{v} \quad or \quad u_i = T_{ij}v_j \tag{6.6}$$

A nice example of this relation is Cauchy's law (Chapter 7), which says that the traction on a plane (a vector) is equal to the stress (a tensor) times the pole to the plane (another vector). Since in three dimensions the indices i and j change from 1 to 3, Eq. 6.6 corresponds to three equations, one for each of the components of $\mathbf{u}$ in terms of the components of $\mathbf{v}$. In Python code this would look like:

```
# v (1 x 3 vector) and T (3 x 3 tensor) are declared before
u = np.zeros(3) # initialize u (1 x 3 vector)
for i in range(0,3,1): # free index
    for j in range(0,3,1): # dummy index
        u[i] = T[i,j]*v[j] + u[i]
```

An implication of Eq. 6.6 is that one can produce a tensor from a type of product of two vectors. This operation is known as the dyad product and it involves multiplying a column vector times a row vector, which gives a 3 x 3 matrix:

$$\mathbf{T} = \mathbf{u} \otimes \mathbf{v} \quad or \quad T_{ij} = u_i v_j \tag{6.7}$$

This gives nine equations (one for each component of the tensor) in terms of the components of $\mathbf{u}$ and $\mathbf{v}$. In Python code this would look like:

```
# u (1 x 3 vector) and v (1 x 3 vector) are declared before
T = np.zeros((3,3)) # initialize T (3 x 3 tensor)
for i in range(0,3,1): # free index
    for j in range(0,3,1): # free index
        T[i,j] = u[i]*v[j]
```

In section 6.5, we will use the dyad product to derive the orientation tensor.

6.4 Tensor transformations

If we know the components of a tensor in one coordinate system, we can determine what the components are in any other coordinate system, just as we did with vectors in section 5.1.2. All we need to know is the transformation matrix $\mathbf{a}$. The procedure, however, is more difficult because a second order tensor is more complicated than a vector. The new components of the tensor in terms of the old are given by (Allmendinger et al., 2012):

$$\mathbf{T}' = \mathbf{a}^{\mathbf{T}}\mathbf{T}\mathbf{a} \quad or \quad T'_{ij} = a_{ik}a_{jl}T_{kl} \tag{6.8}$$

This equation represent nine equations (since there are two fixed indices i and j) with nine terms each (since there are two dummy indices k and l). It is tedious to expand and solve Eq. 6.8 by hand. Fortunately, we can solve this equation using code:

```
# T_old (3 x 3 tensor) and a (3 x 3 transf. matrix) are
  declared before
T_new = np.zeros((3,3)) # initialize T_new (3 x 3 tensor)
for i in range(0,3,1): # free index
    for j in range(0,3,1): # free index
        for k in range(0,3,1): # dummy index
            for l in range(0,3,1): # dummy index
                T_new[i,j] = a[i,k]*a[j,l]*T_old[k,l] + T_new
  [i,j]
```

Likewise, we can compute the old coordinates of the tensor in terms of the new coordinates:

$$\mathbf{T} = \mathbf{a}\mathbf{T}'\mathbf{a}^{\mathbf{T}} \quad or \quad T_{ij} = a_{ki}a_{lj}T'_{kl} \tag{6.9}$$

In Python code this would look like:

```
# T_new (3 x 3 tensor) and a (3 x 3 transf. matrix) are
    declared before
T_old = np.zeros((3,3)) # initialize T_old (3 x 3 tensor)
for i in range(0,3,1): # free index
    for j in range(0,3,1): # free index
        for k in range(0,3,1): # dummy index
            for l in range(0,3,1): # dummy index
                T_old[i,j] = a[k,i]*a[l,j]*T_new[k,l] + T_old
    [i,j]
```

6.4.1 The Mohr Circle

Let's consider the case where the axes of the old coordinate system are parallel to the principal axes of the tensor $\mathbf{T}$. Now, let's change the coordinate system to a different orientation by rotating it an angle θ about one of the principal axes, for example the intermediate axis T_2 (Fig. 6.2). The transformation matrix $\mathbf{a}$ for this problem is:

$$\mathbf{a} = \begin{pmatrix} \cos\theta & \cos 90 & \cos(90-\theta) \\ \cos 90 & \cos 0 & \cos 90 \\ \cos(90+\theta) & \cos 90 & \cos\theta \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \tag{6.10}$$

The tensor $\mathbf{T}$ in the old coordinate system is:

$$\mathbf{T} = T_{ij} = \begin{bmatrix} T_1 & 0 & 0 \\ 0 & T_2 & 0 \\ 0 & 0 & T_3 \end{bmatrix} \tag{6.11}$$

Now, we can use Eq. 6.8 to calculate the components of the tensor in the new coordinate system. Substituting Eqs. 6.10 and 6.11 into Eq. 6.8 and carrying out the summation, we obtain:

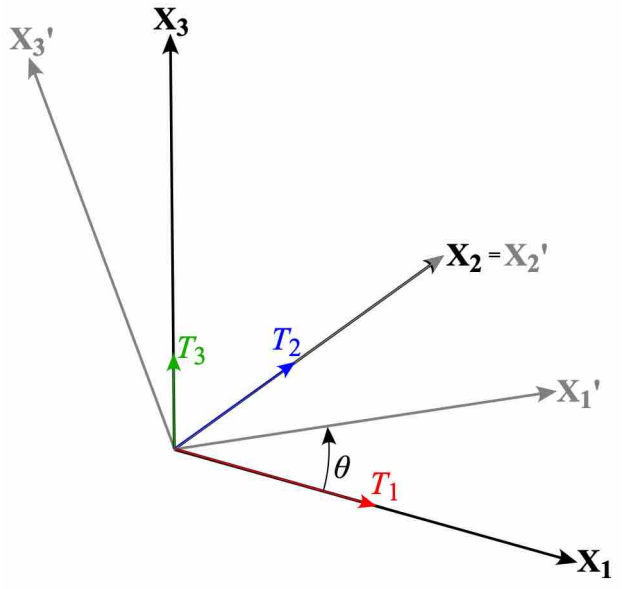


Figure 6.2: Rotation of principal coordinate system $\mathbf{X_1X_2X_3}$ about the principal axis T_2 an amount θ. Modified from Allmendinger (2019).

$$\mathbf{T}' = T'_{ij} = \begin{bmatrix} T_1\cos^2\theta + T_3\sin^2\theta & 0 & -(T_1 - T_3)\sin\theta\cos\theta \\ 0 & T_2 & 0 \\ -(T_1 - T_3)\sin\theta\cos\theta & 0 & T_1\sin^2\theta + T_3\cos^2\theta \end{bmatrix} \tag{6.12}$$

The components of $\mathbf{T}'$ can be expressed in a nicer way using the following trigonometric identities for double angles:

$$\sin 2\theta = 2\sin\theta\cos\theta \quad \sin^2\theta = \frac{1-\cos 2\theta}{2} \quad \cos^2\theta = \frac{1+\cos 2\theta}{2} \tag{6.13}$$

Substituting these equations into Eq. 6.12 and rearranging, we get the following equations for the components of the tensor in the new coordinate system:

$$\begin{aligned} T'_{11} &= \frac{T_1 + T_3}{2} + \frac{T_1 - T_3}{2}\cos 2\theta \\ T'_{33} &= \frac{T_1 + T_3}{2} - \frac{T_1 - T_3}{2}\cos 2\theta \\ T'_{13} = T'_{31} &= -\frac{T_1 - T_3}{2}\sin 2\theta \end{aligned} \tag{6.14}$$

If you recall, the equation of a circle of center $(c, 0)$ and radius r is:

$$\begin{aligned} x &= c - r\cos\alpha \\ y &= r\sin\alpha \end{aligned} \tag{6.15}$$

We can see that these equations are similar to Eq. 6.14. In fact, Eq. 6.14 describes a circle with center c and radius r (Fig. 6.3):

$$c = \left(\frac{T_1 + T_3}{2}, 0\right) \qquad r = \left(\frac{T_1 - T_3}{2}, 0\right) \tag{6.16}$$

This circle is known as the *Mohr Circle*, because it was devised by the German engineer Otto Mohr in the late 1800s. The Mohr Circle is basically a graphical device to rotate a symmetric tensor about one of its principal axes. It is commonly associated with stress, but it can be applied to any symmetric tensor (strain and permeability for example). We will see the application of the Mohr Circle in chapters 7 and 8.

6.5 The orientation tensor

The concepts discussed so far in this chapter are the mathematical basis for the following two chapters on stress and strain. We will now focus on a relatively simple yet important problem that relies on the concept of eigenvalues and eigenvectors: How do we find the best-fit fold axis to a group of bedding poles? And similarly, how do we find the best-fit plane to a group of lines?

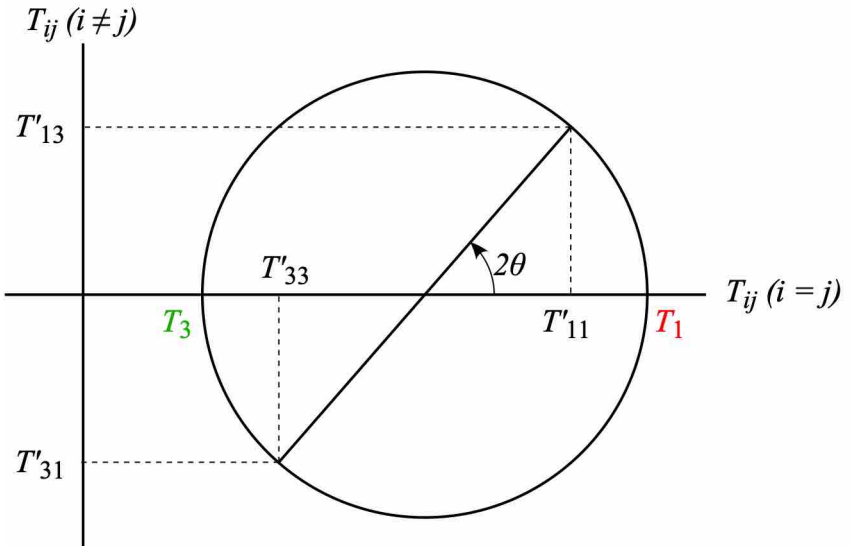


Figure 6.3: The Mohr Circle is the graphical representation of the rotation of a symmetric tensor about one of its principal axes. Modified from Allmendinger et al. (2012).

6.5.1 Best-fit fold axis

The solution to this problem is based on the least squares method (Charlesworth et al., 1976; Allmendinger et al., 2012). Suppose we are trying to calculate the axis $\mathbf{f}$ of a fold. If the fold is truly cylindrical, then all the bedding poles $\mathbf{p}_{[n]}$ should be perpendicular to $\mathbf{f}$, i.e. the angle θ_i between any pole $\mathbf{p}_{[i]}$ and $\mathbf{f}$ should be 90°, and $\cos\theta_i$ should be zero (Fig. 6.4). $\cos\theta_i$ is equal to the dot product between $\mathbf{f}$ and $\mathbf{p}_{[i]}$. Treating these lines as row vectors, the dot product can be written as:

$$\cos\theta_i = \mathbf{p}_{[i]}\mathbf{f}^{\mathbf{T}} \tag{6.17}$$

We can use the value of $\cos\theta_i$ to represent the deviation of a pole $\mathbf{p}_{[i]}$ from $\mathbf{f}$. The sum of the squares of the deviations of all the poles is:

$$S = \sum_{i=1}^{n} \cos^2\theta_i = \sum_{i=1}^{n} \left(\mathbf{p}_{[i]}\mathbf{f}^{\mathbf{T}}\right)^2 \tag{6.18}$$

Since the dot product is commutative ($\mathbf{p}_{[i]}\mathbf{f}^{\mathbf{T}} = \mathbf{f}\mathbf{p}_{[i]}^{\mathbf{T}}$), we can write Eq. 6.18

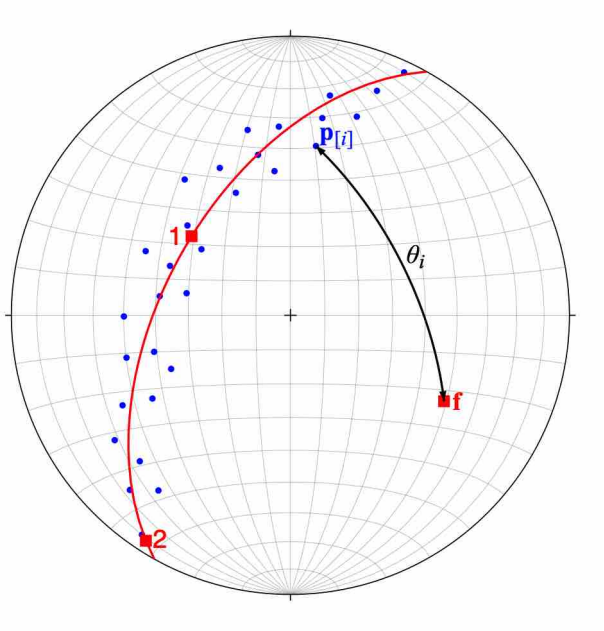


Figure 6.4: Best-fit fold axis to a group of bedding poles. A perfectly oriented pole $\mathbf{p}_{[i]}$ should be perpendicular to the fold axis $\mathbf{f}$, and θ_i should be $90°(\cos\theta_i = 0)$. Modified from Allmendinger (2019).

as:

$$S = \sum_{i=1}^{n} \mathbf{f}\mathbf{p}_{[i]}^{\mathbf{T}}\mathbf{p}_{[i]}\mathbf{f}^{\mathbf{T}} = \mathbf{f}\mathbf{T}\mathbf{f}^{\mathbf{T}} \tag{6.19}$$

$\mathbf{T}$ is a symmetric tensor known as the *orientation tensor* and it is composed of the sum of the dyad products of each pole $\mathbf{p}_{[i]}$ ($[\cos\alpha_i\ \cos\beta_i\ \cos\gamma_i]$) with itself (Eq. 6.7):

$$\begin{aligned}\mathbf{T} &= \sum_{i=1}^{n} \mathbf{p}_{[i]}^{\mathbf{T}}\mathbf{p}_{[i]} = \sum_{i=1}^{n} (p_i p_j)_{[i]} \\ &= \begin{bmatrix} \sum \cos^2\alpha_{[i]} & \sum \cos\alpha_{[i]}\cos\beta_{[i]} & \sum \cos\alpha_{[i]}\cos\gamma_{[i]} \\ \sum \cos\beta_{[i]}\cos\alpha_{[i]} & \sum \cos^2\beta_{[i]} & \sum \cos\beta_{[i]}\cos\gamma_{[i]} \\ \sum \cos\gamma_{[i]}\cos\alpha_{[i]} & \sum \cos\gamma_{[i]}\cos\beta_{[i]} & \sum \cos^2\gamma_{[i]} \end{bmatrix}\end{aligned} \tag{6.20}$$

You can think of $\mathbf{T}$ as describing an ellipsoid in three-dimensions. To find

the principal axes of this ellipsoid, we need to calculate the eigenvalues and eigenvectors of $\mathbf{T}$. The smallest eigenvalue of $\mathbf{T}$ is the minimization of the deviations S in Eqs. 6.18 and 6.19. If the fold were perfectly cylindrical, the lowest eigenvalue would be zero. Thus, the eigenvector corresponding to the lowest eigenvalue is the best-fit fold axis $\mathbf{f}$.

The function *Bingham* computes and plots a cylindrical best-fit to a distribution of bedding poles. It returns the eigenvalues and eigenvectors of $\mathbf{T}$, the uncertainty cones for the *Bingham* statistics, and the best-fit plane to the bedding poles. For more information on the Bingham statistics, please read Fisher et al. (1987):

```
import numpy as np
import matplotlib.pyplot as plt

from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi
from Stereonet import Stereonet as Stereonet
from GreatCircle import GreatCircle as GreatCircle
from StCoordLine import StCoordLine as StCoordLine


def Bingham(T,P):
  '''
  Bingham calculates and plots a cylindrical best fit to
  a distribution of poles to bedding. The statistical
  routine is based on algorithms in Fisher et al. (1988)

  USE: eigVec, confCone, bestFit = Bingham(T,P)

  T and P = Vectors of lines trends and plunges
  respectively

  eigVec = 3 x 3 matrix with eigenvalues (column 1), trends
  (column 2) and plunges (column 3) of the eigenvectors.
  Maximum eigenvalue and corresponding eigenvector are
  in row 1, intermediate in row 2, and minimum in row 3.

  confCone = 2 x 2 matrix with the maximum (column 1) and
  minimum (column 2) radius of the 95% elliptical
  confidence cone around the eigenvector corresponding
  to the largest (row 1), and lowest (row 2) eigenvalue

  besFit = 1 x 2 vector containing the strike and dip
  (right hand rule) of the best fit great circle to
  the distribution of lines
```

```
36
37    NOTE: Input/Output trends and plunges, as well as
38    confidence cones are in radians. Bingham plots the
39    input poles, eigenvectors and best fit great circle
40    in an equal area stereonet.
41
42    Bingham uses functions ZeroTwoPi, Pole, StCoordLine
43    SphToCart, CartToSph, Rotate, GreatCircle, SmallCircle,
44    GeogrToView, and Stereonet
45
46    Python function translated from the Matlab function
47    Bingham in Allmendinger et al. (2012)
48    '''
49    # Some constants
50    pi = np.pi
51    east = pi/2
52    twopi = pi*2
53
54    # Number of lines
55    nlines = len(T)
56
57    # Initialize the orientation matrix
58    a = np.zeros((3,3))
59
60    # Fill the orientation matrix with the sums of the
61    # squares (for the principal diagonal) and the products
62    # of the direction cosines of each line. cn, ce and cd
63    # are the north, east and down direction cosines
64    for i in range(nlines):
65      cn,ce,cd = SphToCart(T[i],P[i],0)
66      a[0,0] = a[0,0] + cn*cn
67      a[0,1] = a[0,1] + cn*ce
68      a[0,2] = a[0,2] + cn*cd
69      a[1,1] = a[1,1] + ce*ce
70      a[1,2] = a[1,2] + ce*cd
71      a[2,2] = a[2,2] + cd*cd
72
73    # The orientation matrix is symmetric so the off-diagonal
74    # components can be equated
75    a[1,0] = a[0,1]
76    a[2,0] = a[0,2]
77    a[2,1] = a[1,2]
78
79    # Calculate the eigenvalues and eigenvectors of the
80    # orientation matrix using function eigh.
81    # D is a vector of eigenvalues and V is a full matrix
82    # whose columns are the corresponding eigenvectors
83    D, V = np.linalg.eigh(a)
84
```

```
85    # Normalize the eigenvalues by the number of lines and
86    # convert the corresponding eigenvectors to the lower
87    # hemisphere
88    for i in range(3):
89      D[i] = D[i]/nlines
90      if V[2,i] < 0:
91        V[0,i] = -V[0,i]
92        V[1,i] = -V[1,i]
93        V[2,i] = -V[2,i]
94
95    # Initialize eigVec
96    eigVec = np.zeros((3,3))
97    #Fill eigVec
98    eigVec[0,0] = D[2]     # Maximum eigenvalue
99    eigVec[1,0] = D[1]     # Intermediate eigenvalue
100   eigVec[2,0] = D[0]     # Minimum eigenvalue
101   # Trend and plunge of largest eigenvalue: column 3 of V
102   eigVec[0,1], eigVec[0,2] = CartToSph(V[0,2], V[1,2], V
      [2,2])
103   # Trend and plunge of interm. eigenvalue: column 2 of V
104   eigVec[1,1], eigVec[1,2] = CartToSph(V[0,1], V[1,1], V
      [2,1])
105   # Trend and plunge of minimum eigenvalue: column 1 of V
106   eigVec[2,1], eigVec[2,2] = CartToSph(V[0,0], V[1,0], V
      [2,0])
107
108   # Initialize confCone
109   confCone = np.zeros((2,2))
110   # If there are more than 25 lines, calculate confidence
111   # cones at the 95% confidence level. The algorithm is
112   # explained in Fisher et al. (1987)
113   if nlines >= 25:
114     e11 = 0
115     e22 = 0
116     e12 = 0
117     d11 = 0
118     d22 = 0
119     d12 = 0
120     en11 = 1/(nlines*(eigVec[2,0] - eigVec[0,0])**2)
121     en22 = 1/(nlines*(eigVec[1,0] - eigVec[0,0])**2)
122     en12 = 1/(nlines*(eigVec[2,0] - eigVec[0,0])*(eigVec[1,0]
        - eigVec[0,0]))
123     dn11 = en11
124     dn22 = 1/(nlines*(eigVec[2,0] - eigVec[1,0])**2)
125     dn12 = 1/(nlines*(eigVec[2,0] - eigVec[1,0])*(eigVec[2,0]
        - eigVec[0,0]))
126     vec = np.zeros((3,3))
127     for i in range(3):
128       vec[i,0] = np.sin(eigVec[i,2] + east)*np.cos(twopi -
```

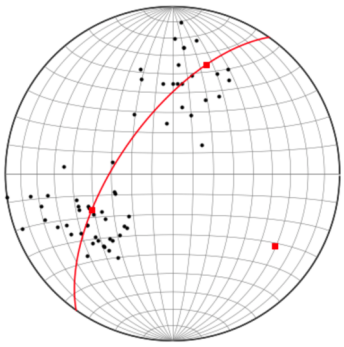
```
    eigVec[i,1])
129    vec[i,1] = np.sin(eigVec[i,2] + east)*np.sin(twopi -
    eigVec[i,1])
130    vec[i,2] = np.cos(eigVec[i,2] + east)
131   for i in range(nlines):
132    c1 = np.sin(P[i]+east)*np.cos(twopi-T[i])
133    c2 = np.sin(P[i]+east)*np.sin(twopi-T[i])
134    c3 = np.cos(P[i]+east)
135    u1x = vec[2,0]*c1 + vec[2,1]*c2 + vec[2,2]*c3
136    u2x = vec[1,0]*c1 + vec[1,1]*c2 + vec[1,2]*c3
137    u3x = vec[0,0]*c1 + vec[0,1]*c2 + vec[0,2]*c3
138    e11 = u1x*u1x * u3x*u3x + e11
139    e22 = u2x*u2x * u3x*u3x + e22
140    e12 = u1x*u2x * u3x*u3x + e12
141    d11 = e11
142    d22 = u1x*u1x * u2x*u2x + d22
143    d12 = u2x*u3x * u1x*u1x + d12
144   e22 = en22*e22
145   e11 = en11*e11
146   e12 = en12*e12
147   d22 = dn22*d22
148   d11 = dn11*d11
149   d12 = dn12*d12
150   d = -2*np.log(.05)/nlines
151   # initialize f
152   f = np.zeros((2,2))
153   if abs(e11*e22-e12*e12) >= 0.000001:
154    f[0,0] = (1/(e11*e22-e12*e12)) * e22
155    f[1,1] = (1/(e11*e22-e12*e12)) * e11
156    f[0,1] = -(1/(e11*e22-e12*e12)) * e12
157    f[1,0] = f[0,1]
158    # Calculate the eigenvalues and eigenvectors
159    # of the matrix f using function eigh
160    # The next lines follow steps 1-4 outlined
161    # on pp. 34-35 of Fisher et al. (1987)
162    DD, _ = np.linalg.eigh(f)
163    if DD[0] > 0 and DD[1] > 0:
164     if d/DD[0] <= 1 and d/DD[1] <= 1:
165      confCone[0,1] = np.arcsin(np.sqrt(d/DD[1]))
166      confCone[0,0] = np.arcsin(np.sqrt(d/DD[0]))
167   # Repeat the process for the eigenvector
168   # corresponding to the smallest eigenvalue
169   if abs(d11*d22-d12*d12) >= 0.000001:
170    f[0,0] = (1/(d11*d22-d12*d12)) * d22
171    f[1,1] = (1/(d11*d22-d12*d12)) * d11
172    f[0,1] = -(1/(d11*d22-d12*d12)) * d12
173    f[1,0] = f[0,1]
174    DD, _ = np.linalg.eigh(f)
175    if DD[0] > 0.0 and DD[1] > 0.0:
```

```
176         if d/DD[0] <= 1 and d/DD[1] <= 1:
177           confCone[1,1] = np.arcsin(np.sqrt(d/DD[1]))
178           confCone[1,0] = np.arcsin(np.sqrt(d/DD[0]))
179
180   # Calculate the best fit great circle
181   # to the distribution of points
182   bestFit = np.zeros(2)
183   bestFit[0] = ZeroTwoPi(eigVec[2,1] + east)
184   bestFit[1] = east - eigVec[2,2]
185
186   # Plot stereonet
187   Stereonet(0, 90*pi/180, 10*pi/180, 1)
188
189   # Plot lines
190   for i in range(nlines):
191     xp,yp = StCoordLine(T[i],P[i],1)
192     plt.plot(xp,yp,'k.')
193
194   # Plot eigenvectors
195   for i in range(3):
196     xp,yp = StCoordLine(eigVec[i,1],eigVec[i,2],1)
197     plt.plot(xp,yp,'rs')
198
199   # Plot best fit great circle
200   path = GreatCircle(bestFit[0],bestFit[1],1)
201   plt.plot(path[:,0],path[:,1],'r')
202
203   # Show plot
204   plt.show()
205
206   return eigVec, confCone, bestFit
```

Let's use this function to compute the best-fit fold axis for the Big Elk anticline (southeastern Idaho), using the bedding data in Fig. 5.7. The notebook ch6-1 illustrates this. Notice that the bedding data (strike and dips) are read from the file *beasd.txt*:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
pi = np.pi

# Import functions Pole and Bingham
import sys, os
sys.path.append(os.path.abspath('../functions'))
from Pole import Pole as Pole
from Bingham import Bingham as Bingham

# Read the bedding data from the Big Elk anticline
beasd =np.loadtxt(os.path.abspath('../data/ch6-1/beasd.txt'))

# Initialize poles
T = np.zeros(len(beasd))
P = np.zeros(len(beasd))

# Compute poles to bedding
for i in range(len(T)):
     T[i],P[i] = Pole(beasd[i,0]*pi/180,beasd[i,1]*pi/180,1)

# Compute cylindrical best fit
eigVec,confCone,bestFit = Bingham(T,P)

# Print best-fit fold axis
print('Best-fit fold axis: trend = {:.1f}, plunge {:.1f}'.
    format(eigVec[2,1]*180/pi,eigVec[2,2]*180/pi))
# Print best-fit plane
print('Best-fit plane: strike = {:.1f}, dip {:.1f}\n'.format(
    bestFit[0]*180/pi,bestFit[1]*180/pi))
# Print confidence cone
print('95% elliptical confidence cones:')
print('Axis 1 (largest eigenvalue): Max = {:.1f}, Min {:.1f}'
    .format(confCone[0,0]*180/pi,confCone[0,1]*180/pi))
print('Axis 3 (lowest eigenvalue): Max = {:.1f}, Min {:.1f}'.
    format(confCone[1,0]*180/pi,confCone[1,1]*180/pi))
```

Output:



Best-fit fold axis: trend = 125.3, plunge 26.1
Best-fit plane: strike = 215.3, dip 63.9

95% elliptical confidence cones:
Axis 1 (largest eigenvalue): Max = 16.9, Min 5.9
Axis 3 (lowest eigenvalue): Max = 8.2, Min 5.8

6.5.2 Line distributions

The orientation tensor **T** (Eq. 6.20) is also useful for characterizing line distributions. Figure 6.5 shows three end-members of line distributions. The bipolar distribution (Fig. 6.5a) consists of a group of lines that are parallel or sub-parallel to each other and plunge in opposite directions, with some limited scatter. In this case, the lines define an elongate ellipsoid (a cigar), with a large (near 1.0) eigenvalue (long axis of the ellipsoid) and two small (near zero) eigenvalues (short axes of the ellipsoid). Also, the eigenvector of the largest eigenvalue describes well the preferred orientation of the lines (Fig. 6.5a). Notice that in this case, the mean vector calculation (section 4.4.1) will fail since the lines plunging in opposite directions will cancel each other out.

In the girdle distribution (Fig. 6.5b), all the lines are close to being coplanar, and they define a flattened ellipsoid (or pancake) with one small (near zero) eigenvalue corresponding to the pole of the best-fit plane through the lines, and two large relatively equal eigenvalues (near 0.5) whose eigenvectors lie within the best-fit plane. Finally, in the random distribution (Fig. 6.5c), the lines define a sphere and the eigenvalues are relatively equal. Thus, in general, the eigenvalues of **T** are a good way to characterize line distributions.

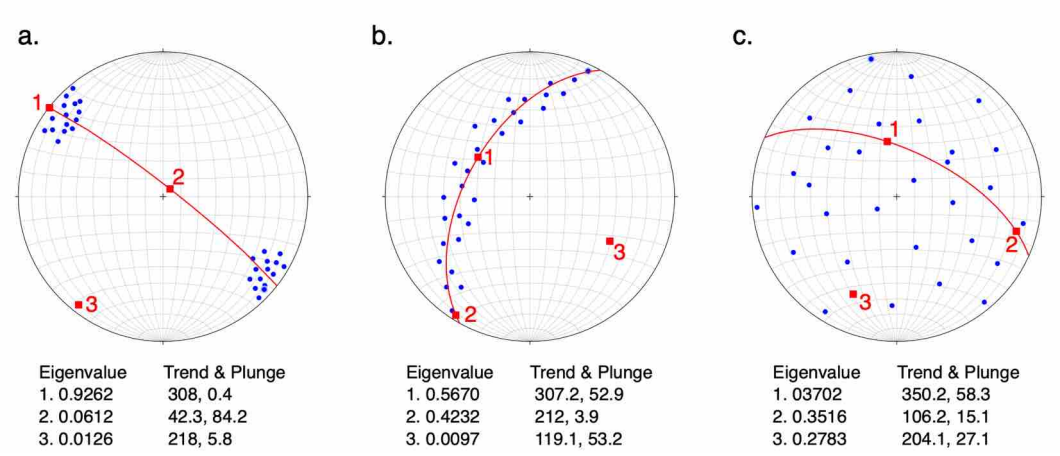


Figure 6.5: Three end members of line distributions. **a.** Bipolar, **b.** Girdle, and **c.** random. Each diagram has 30 lines. Modified from Allmendinger (2019).

6.5.3 Best-fit plane

In section 4.4.3 we look at the problem of calculating the orientation of a plane from three points on the plane. But what if we have more than three points on the plane? We can use the least squares approach above to solve this problem (Allmendinger, in press). This will also allow us to evaluate the goodness of fit of the plane to the data points. Let's assume the points are defined by position vectors $\mathbf{p}_{[i]}$ in an east-north-up (**ENU**) coordinate system. First, the centroid $\mathbf{c}$ of the position vectors $\mathbf{p}_{[i]}$ is calculated:

$$\mathbf{c} = \frac{\sum_{i=1}^{n} \mathbf{p}_{[i]}}{n} \tag{6.21}$$

Then, the centroid $\mathbf{c}$ is subtracted from the position vectors $\mathbf{p}_{[i]}$:

$$\mathbf{s}_{[i]} = \mathbf{p}_{[i]} - \mathbf{c} \tag{6.22}$$

And we use the vectors $\mathbf{s}_{[i]}$ to construct a covariance matrix $\mathbf{C}$ (which looks very similar to the orientation matrix $\mathbf{T}$):

$$\mathbf{C} = \begin{bmatrix} \sum(s_E)^2_{[i]} & \sum(s_E)_{[i]}(s_N)_{[i]} & \sum(s_E)_{[i]}(s_U)_{[i]} \\ \sum(s_N)_{[i]}(s_E)_{[i]} & \sum(s_N)^2_{[i]} & \sum(s_N)_{[i]}(s_U)_{[i]} \\ \sum(s_U)_{[i]}(s_E)_{[i]} & \sum(s_U)_{[i]}(s_N)_{[i]} & \sum(s_U)^2_{[i]} \end{bmatrix} \tag{6.23}$$

Finally, the eigenvalues and eigenvectors of the covariance matrix **C** are determined. The eigenvector corresponding to the smallest eigenvalue is the pole to the best-fit plane, and the square root of the smallest eigenvalue is the standard deviation of the distance of each point from the best-fit plane.

The function *FitPlane* calculates the best-fit plane to a group of lines:

```
import numpy as np

from Pole import Pole as Pole
from CartToSph import CartToSph as CartToSph

def FitPlane(pts):
  '''
  Fitplane computes the best-fit plane for a group of
  points (position vectors) on the plane

  USE: strike, dip, stdev = FitPlane(pts)

  pts is a n x 3 matrix containing the East (column 1),
  North (column 2), and Up (column 3) coordinates
  of n points on the plane

  strike and dip are returned in radians

  stdev is the standard deviation of the distance of
  each point from the best-fit plane

  FitPlane uses functions Pole and CartToSph
  '''
  # Compute the centroid of the selected points
  avge = np.mean(pts[:,0])
  avgn = np.mean(pts[:,1])
  avgu = np.mean(pts[:,2])

  # Compute the points vectors minus the centroid
  pts[:,0] = pts[:,0] - avge
  pts[:,1] = pts[:,1] - avgn
  pts[:,2] = pts[:,2] - avgu

  # Compute the covariance/orientation matrix
  a = np.zeros((3,3))
```

```
  for i in range(pts.shape[0]):
    ce = pts[i,0]
    cn = pts[i,1]
    cu = pts[i,2]
    # compute orientation matrix
    a[0,0] = a[0,0] + ce*ce
    a[0,1] = a[0,1] + ce*cn
    a[0,2] = a[0,2] + ce*cu
    a[1,1] = a[1,1] + cn*cn
    a[1,2] = a[1,2] + cn*cu
    a[2,2] = a[2,2] + cu*cu
  # The orientation matrix is symmetric so the off-diagonal
  # components can be equated
  a[1,0] = a[0,1]
  a[2,0] = a[0,2]
  a[2,1] = a[1,2]

  # calculate the eigenvalues and eigenvectors of the
  # orientation matrix: use function eigh
  D, V = np.linalg.eigh(a)

  # Calculate pole to best-fit plane = lowest eigenvalue
  # vector in N, E, D coordinates
  cn = V[1,0]
  ce = V[0,0]
  cd = -V[2,0]

  # Find trend and plunge of pole to best fit plane
  trd, plg = CartToSph(cn,ce,cd)

  # Find Best fit plane
  strike, dip = Pole(trd,plg,0)

  # Calculate standard deviation = square root of
  # minimum eigenvalue
  stdev = np.sqrt(D[0])

  return strike, dip, stdev
```

Let's use this function to compute the orientation of the contact between the Jurassic Js and Cretaceous Ke in the northeastern part of the Poker Peak Quadrangle, Idaho (Albee et al., 1975) (yellow contact in Fig. 6.6). The notebook ch6-2 shows the solution to this problem. Notice that the UTM **ENU** coordinates of the points on the Js-Ke contact are read from the file *jske.txt*:

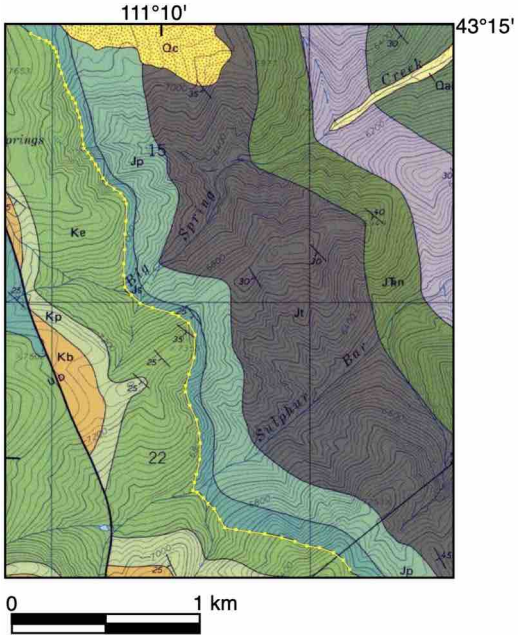


Figure 6.6: Northeastern portion of the Poker Peak Quadrangle, Idaho (Albee et al., 1975). The yellow dots are points on the Js-Ke contact.

```
# Import libraries
import numpy as np
pi = np.pi

# Import function FitPlane
import sys, os
sys.path.append(os.path.abspath('../functions'))
from FitPlane import FitPlane as FitPlane

# Read the points on the contact
# Coordinates are UTM (ENU) in meters
jske = np.loadtxt(os.path.abspath('../data/ch6-2/jske.txt'))

# Compute best-fit plane
strike, dip, stdev = FitPlane(jske)

# Print strike and dip of plane
print('Strike = {:.1f}, Dip = {:.1f}'.format(strike*180/pi,
    dip*180/pi))

# Print standard deviation of the distance of each point
# from the best-fit plane
print('Standard deviation = {:.1f} m'.format(stdev))
```

```
Output:
Strike = 153.3, Dip = 29.9
Standard deviation = 246.8 m
```

This orientation is not far from the strike and dip symbols shown close to the Js-Ke contact in Fig. 6.6. To learn more about fitting a plane to a distribution of lines in three-dimensions, you can read Fernandez (2005).

6.6 Exercises

1. The file *csdtp.txt* contains the strikes and dips (RHR) of the Jurassic Sundance Formation around the Sheep Mountain Anticline, Wyoming (Fig. 5.8). You can visualize these bedding orientations in Google Earth using the file *csdtp.kml*. Compute the anticline's best-fit fold axis using the function *Bingham*.

2. The file *biaxial.txt*, *girdle.txt*, and *random.txt* contain the lines (trend and plunge) of the distributions shown in Fig. 6.5. Compute the eigenvalues and eigenvectors, and the 95% confidence cones of these line distributions using the function *Bingham*.

3. Trede et al. (2019) published an interesting article about the appropriate sample size for strike and dip measurements. Figure 6.7 shows the Lidar scan of one of their test surfaces, a two meter-sized foliation in Cambro-Ordovician mica schists of the Svarthola Cave, SW Norway. The file *fol.txt* contains the **ENU** coordinates of the scanned points (grey points, Fig. 6.7), and the file *kfol.txt* contains the **ENU** coordinates of regularly spaced points on the foliation (red circles, Fig. 6.7).

   (a) On each of the points of the regularly spaced grid (red circles, Fig. 6.7), compute the strike and dip of the foliation using the Lidar scanned points (grey points, Fig. 6.7) within a radius r of 0.1 m from the grid point. Do this only if there are more than 3 points within the search radius r.

   (b) Repeat the procedure in a for larger search radii r of 0.3, 0.5, 0.7 and 0.9 m.

   (c) Plot a histogram of the strike, and another histogram of the dip distribution, for the different search radii. Use different colors to

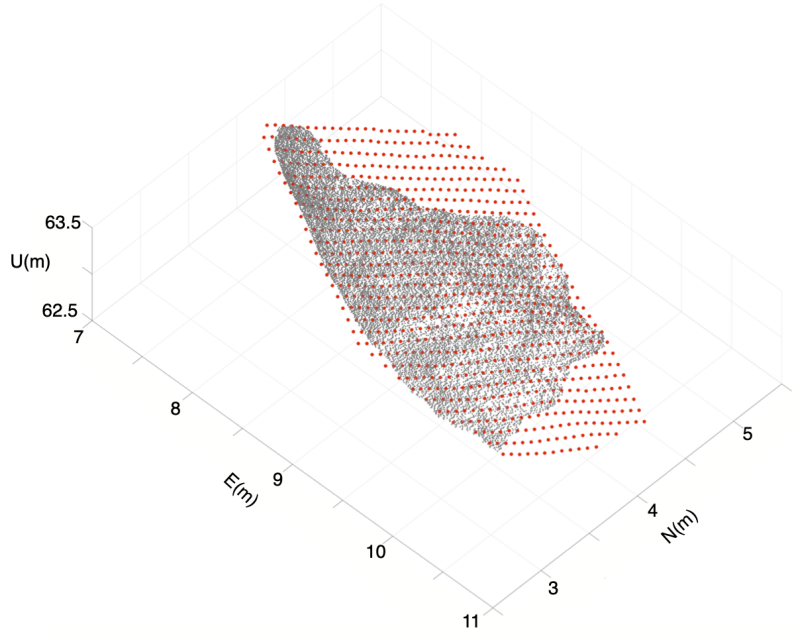


Figure 6.7: Two-metre size foliation plane in Cambro-Ordovician mica schists of the Svarthole Cave, SW Norway. The grey points are Lidar scanned points on the foliation. The red circles are points on a regular grid covering the foliation. The Lidar scan frequency is 300 kHz, and vertical and horizontal point spacing are 0.013 and 0.022, respectively. From Trede et al. (2019).

denote the different radii (see Fig. 8 of Trede et al., 2019 for an example).

(d) At what search radius r, does the orientation of the foliation stabilize? What is the best sample size to measure the foliation?

Hint: Use function *FitPlane*.

References

Albee, H.F. and Cullins, H.L. 1975. Geologic Map of the Poker Peak Quadrangle, Bonneville County, Idaho. U.S. Geological Survey, Geologic Quadrangle Map GQ 1260.

Allmendinger, R.W., Cardozo, N. and Fisher, D.M. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Allmendinger, R.W. 2019. Modern Structural Practice: A structural geology laboratory manual for the 21st century. [Online]. [Accessed January, 2020].

Allmendinger, R.W. In press. GMDE: Extracting Quantitative Information from Geologic Maps. Geosphere.

Charlesworth, H.A.K., Langenberg, C.W. and Ramsden, J. 1976. Determining axes, axial planes, and sections of macroscopic folds using computer-based methods. Canadian Journal of Earth Science 13, 54-65.

Fernandez, O. 2005. Obtaining a best fitting plane through 3D georeferenced data. Journal of Structural Geology 27, 855–858.

Fisher, N.I., Lewis, T. and Embleton, B.J.J. 1987. Statistical analysis of spherical data. Cambridge University Press.

Trede, C., Cardozo, N. and Watson, L. 2019. What is the appropriate sample size for strike and dip measurements? An evaluation from compass, smartphone and LIDAR measurements. Norwegian Journal of Geology 99, 1-14.

Chapter 7

Stress

Geoscientists and engineers are familiar with the concept of stress. Stress describes the forces per unit area, i.e. the tractions, acting on imaginary planes of varied orientations inside a body. These tractions and the stress occur at a particular instant of time.

7.1 The stress tensor

A traction is defined as a force, **f**, divided by the area of the plane, A, on which it acts:

$$\mathbf{t} = \frac{\mathbf{f}}{A} \tag{7.1}$$

A force of one Newton acting on one square meter is one pascal (1 N/m^2 = 1 Pa). A pascal is very small, the atmospheric pressure is about 10^5 Pa. Therefore, we use a more convenient unit called megapascal (1 MPa = 10^6 Pa).

In the subsurface, the vertical traction due to the weight of the overburden (σ_v) is equal to:

$$\sigma_v = \rho g h \tag{7.2}$$

where ρ is the rock density, g is gravity, and h is depth. The rock density increases with depth, but assuming an average rock density of 2400 kg/m^3, σ_v is about 24 MPa at 1 km depth. 24-25 MPa/km is a reasonable value for the gradient of σ_v with depth.

If we now consider a point in space and a Cartesian coordinate system $\mathbf{X_1X_2X_3}$ centered at this point (Fig. 7.1), the components of the stress tensor, $\boldsymbol{\sigma}$, are just the tractions on planes that are perpendicular to the axes of the coordinate system:

$$\sigma_{ij} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} \tag{7.3}$$

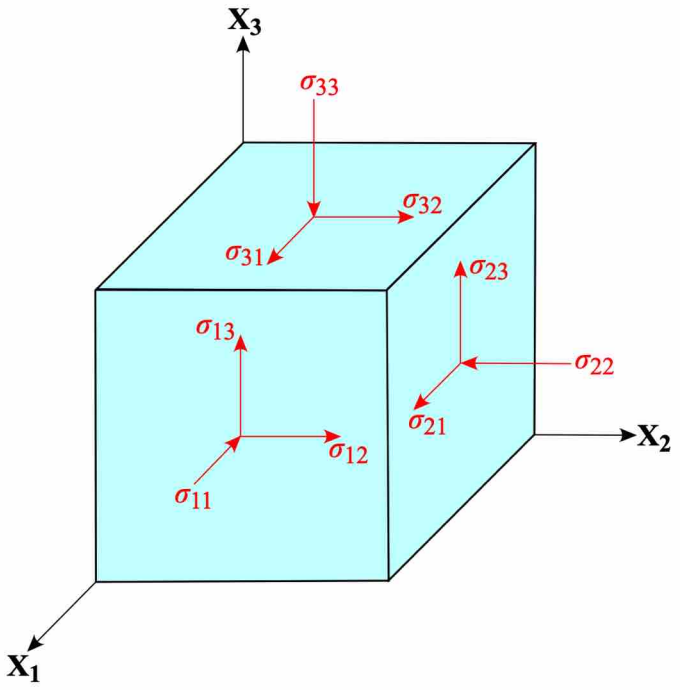


Figure 7.1: The components of the stress tensor are the normal and shear tractions on planes perpendicular to the coordinate axes.

For each component, the first index indicates the axis perpendicular to the plane, and the second index the axis to which the traction is parallel (Fig. 7.1). Tractions with equal indices (e.g. σ_{11}) act perpendicular to the plane and are called normal tractions, while tractions with different indices (e.g.

σ_{12}) act parallel to the plane and are called shear tractions. Because normal tractions are predominantly compressive within the earth (Eq. 7.2), in geology we consider compressive normal tractions as positive and tensional normal tractions as negative. This is opposite to material science, where tensional normal tractions are considered positive.

From Fig. 7.1 it is also clear that if the cube is in equilibrium (it does not rotate about one of the coordinates axis), the stress tensor must be symmetric ($\sigma_{ij} = \sigma_{ji}$).

7.1.1 Cauchy's law

The stress tensor relates two vectors, the traction on a plane, **p**, and the pole to the plane, **n** (Fig. 7.2). This is nicely expressed by Cauchy's law (for a proof of this law see Allmendinger et al., 2012):

$$p_i = \sigma_{ij} n_j \tag{7.4}$$

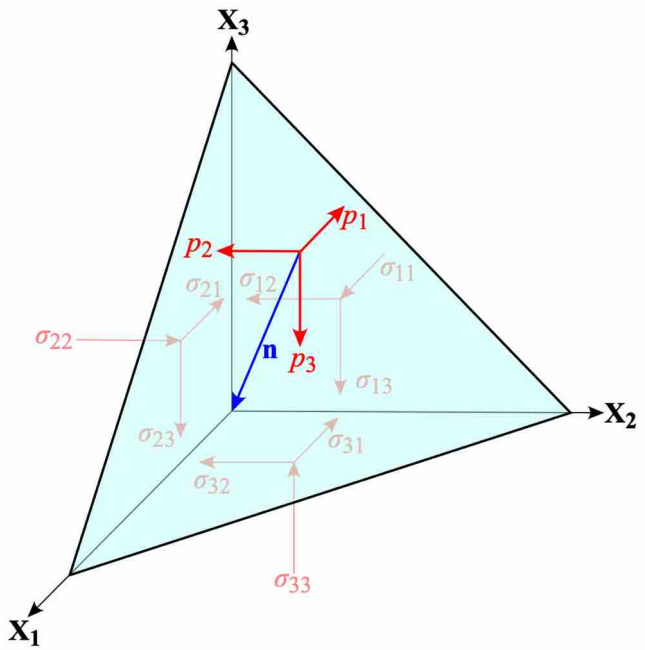


Figure 7.2: Tractions p_i on a plane oblique to the coordinate axes. **n** is the pole to the plane.

Notice that p_i are tractions parallel to the axes of the coordinate system and $\mathbf{n}$ is defined by its direction cosines with respect to these axes (Fig. 7.2).

The function *Cauchy* calculates the tractions p_i on a plane of any orientation in any coordinate system. *Cauchy* uses function *DirCosAxes* which calculates the direction cosines of the $\mathbf{X_1X_2X_3}$ axes. To define the orientation of these axes, it is just necessary to give the trend and plunge of $\mathbf{X_1}$ and the trend of $\mathbf{X_3}$:

```
import numpy as np

from DirCosAxes import DirCosAxes as DirCosAxes
from SphToCart import SphToCart as SphToCart

def Cauchy(stress,tX1,pX1,tX3,strike,dip):
  '''
  Given the stress tensor in a X1X2X3 coordinate system,
  Cauchy computes the X1X2X3 tractions on an arbitrarily
  oriented plane

  USE: T,pT = Cauchy(stress,tX1,pX1,tX3,strike,dip)

  stress = 3 x 3 stress tensor
  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  strike = strike of plane
  dip = dip of plane
  T = 1 x 3 vector with X1, X2 and X3 tractions
  pT = 1 x 3 vector with direction cosines of pole
    to plane with respect to X1X2X3

  NOTE = Plane orientation follows the right hand rule
  Input/Output angles are in radians

  Cauchy uses functions DirCosAxes and SphToCart

  Python function translated from the Matlab function
  Cauchy in Allmendinger et al. (2012)
  '''
  # Compute direction cosines of X1X2X3 with respect
  # to NED
  dC = DirCosAxes(tX1,pX1,tX3)

  # Calculate direction cosines of pole to plane
  p = np.zeros(3)
  p[0],p[1],p[2] = SphToCart(strike,dip,1)

```

```
  # Transform pole to plane to stress coordinates X1X2X3
  # The transformation matrix is the direction cosines of
  # X1X2X3
  pT = np.zeros(3)
  for i in range(3):
    for j in range(3):
      pT[i] = dC[i,j]*p[j] + pT[i]

  # Calculate the tractions in stress coordinates X1X2X3
  T = np.zeros(3)
  # Compute tractions using Cauchy's law: Eq. 7.4
  for i in range(3):
    for j in range(3):
      T[i] = stress[i][j]*pT[j] + T[i]

  return T, pT
```

```
import numpy as np

from SphToCart import SphToCart as SphToCart

def DirCosAxes(tX1,pX1,tX3):
  '''
  DirCosAxes calculates the direction cosines of a right
  handed, orthogonal X1X2X3 cartesian coordinate system
  of any orientation with respect to NED

  USE: dC = DirCosAxes(tX1,pX1,tX3)

  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  dC = 3 x 3 matrix containing the direction cosines
    of X1 (row 1), X2 (row 2), and X3 (row 3)

  Note: Input angles should be in radians

  DirCosAxes uses function SphToCart

  Python function translated from the Matlab function
  DirCosAxes in Allmendinger et al. (2012)
  '''
  # Some constants
  east = np.pi/2.0
  west = 3.0*east
  # tolerance for near zero values
  tol = 1e-6

```

```
  # Initialize matrix of direction cosines
  dC = np.zeros((3,3))

  # Direction cosines of X1
  dC[0,0],dC[0,1],dC[0,2] = SphToCart(tX1,pX1,0)

  # Calculate plunge of axis 3
  # If axis 1 is horizontal
  if abs(pX1) < tol:
    dt = abs(tX1-tX3)
    if abs(dt - east) < tol or abs(dt - west) < tol:
      pX3 = 0.0
    else:
      pX3 = east
  # Else
  else:
    # From dot product, theta = 90, cos(theta) = 0
    pX3 = np.arctan(-(dC[0,0]*np.cos(tX3)+dC[0,1]*np.sin(tX3)
   )/dC[0,2])

  # Direction cosines of X3
  dC[2,0],dC[2,1],dC[2,2] = SphToCart(tX3,pX3,0)

  # X2 is the cross product of X3 and X1
  dC[1,:] = np.cross(dC[2,:],dC[0,:])
  # Make it a unit vector
  dC[1,:] = dC[1,:]/np.linalg.norm(dC[1,:])

  return dC
```

7.1.2 Stress transformation

If we know the components of the stress tensor in one coordinate system $\mathbf{X_1X_2X_3}$, we can calculate the components of the tensor in another coordinate system $\mathbf{X'_1X'_2X'_3}$ (section 6.4):

$$\boldsymbol{\sigma'} = \mathbf{a}^{\mathbf{T}}\boldsymbol{\sigma}\mathbf{a} \quad \text{or} \quad \sigma'_{ij} = a_{ik}a_{jl}\sigma_{kl} \tag{7.5}$$

where $\mathbf{a}$ is the transformation matrix between the new and the old coordinate system. The function *TransformStress* tranforms the stress tensor from one coordinate system to another:

```
import numpy as np

from DirCosAxes import DirCosAxes as DirCosAxes

def TransformStress(stress,tX1,pX1,tX3,ntX1,npX1,ntX3):
  '''
  TransformStress transforms a stress tensor from
  old X1X2X3 to new X1'X2'X3' coordinates

  USE: nstress = TransformStress(stress,tX1,pX1,tX3,ntX1,npX1
   ,ntX3)

  stress = 3 x 3 stress tensor
  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  ntX1 = trend of X1'
  npX1 = plunge of X1'
  ntX3 = trend of X3'
  nstress = 3 x 3 stress tensor in new coordinate system

  NOTE: All input angles should be in radians

  TransformStress uses function DirCosAxes

  Python function translated from the Matlab function
  TransformStress in Allmendinger et al. (2012)
  '''
  # Direction cosines of axes of old coordinate system
  odC = DirCosAxes(tX1,pX1,tX3)

  # Direction cosines of axes of new coordinate system
  ndC = DirCosAxes(ntX1,npX1,ntX3)

  # Transformation matrix between old and new
  # coordinate systems
  a = np.zeros((3,3))
  for i in range(3):
    for j in range(3):
      # Use dot product
      a[i,j] = np.dot(ndC[i,:],odC[j,:])

  # Transform stress: Eq. 7.5
  nstress = np.zeros((3,3))
  for i in range(3):
    for j in range(3):
      for k in range(3):
        for l in range(3):
```

```
48            nstress[i,j] = a[i,k] * a[j,l] * stress[k,l] +
    nstress[i,j]
49
50    return nstress
```

7.1.3 Principal axes of stress

Since stress is a symmetric tensor, there is one orientation of the coordinate system for which the non-diagonal components of the tensor are zero, and only normal tractions act on the planes perpendicular to the coordinate axes (Fig. 7.3). These normal tractions are the principal stresses: σ_1 is the maximum, σ_2 is the intermediate, and σ_3 is the minimum principal stress.

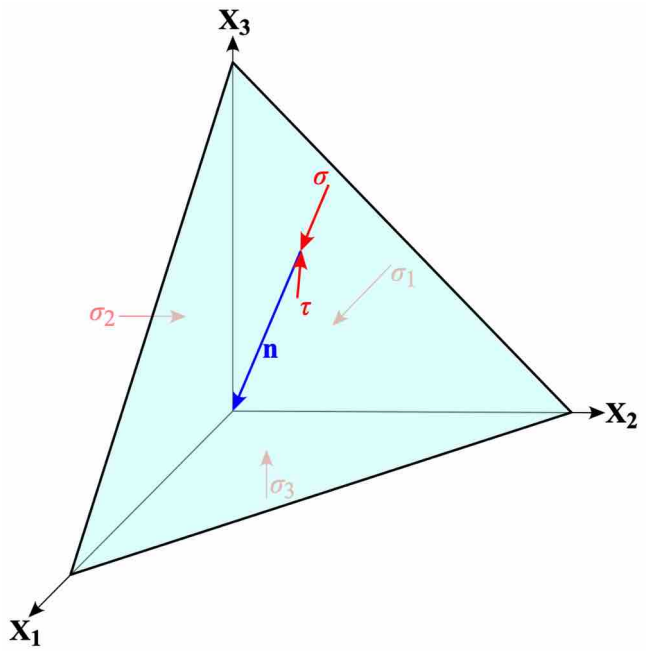


Figure 7.3: Plane oblique to the principal stress coordinate system $\mathbf{X_1X_2X_3}$. σ_1, σ_2 and σ_3 are the principal stresses, and σ and τ are the normal and maximum shear tractions on the plane.

As we saw in section 6.2, finding the principal stresses is an eigenvalue problem. The function *PrincipalStress* calculates the principal stresses and their orientations for a given stress tensor in a Cartesian coordinate system of any orientation:

```
import numpy as np

from DirCosAxes import DirCosAxes as DirCosAxes
from CartToSph import CartToSph as CartToSph

def PrincipalStress(stress,tX1,pX1,tX3):
  '''
  Given the stress tensor in a X1X2X3 coordinate system
  PrincipalStress calculates the principal stresses and
  their orientations (trend and plunge)

  USE: pstress,dCp = PrincipalStress(stress,tX1,pX1,tX3)

  stress = Symmetric 3 x 3 stress tensor
  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  pstress = 3 x 3 matrix containing the magnitude
    (column 1), trend (column 2), and plunge
    (column 3) of the  maximum (row 1),
    intermediate (row 2), and minimum (row 3)
    principal stresses
  dCp = 3 x 3 matrix with direction cosines of the
    principal stress directions: Max. (row 1),
    Int. (row 2), and Min. (row 3) with respect
    to North-East-Down

  NOTE: Input/Output angles are in radians

  PrincipalStress uses functions DirCosAxes and CartToSph

  Python function translated from the Matlab function
  PrincipalStress in Allmendinger et al. (2012)
  '''
  # Compute direction cosines of X1X2X3
  dC = DirCosAxes(tX1,pX1,tX3)

  # Initialize pstress
  pstress = np.zeros((3,3))

  # Calculate the eigenvalues and eigenvectors
  # of the stress tensor
  D, V = np.linalg.eigh(stress)

  # Fill principal stress magnitudes
  pstress[0,0] = D[2] # Maximum principal stress
  pstress[1,0] = D[1] # Interm. principal stress
  pstress[2,0] = D[0] # Minimum principal stress
```

```
49
50
51    # The direction cosines of the principal stresses are
52    # with respect to the X1X2X3 stress coordinate system,
53    # so they need to be transformed to the NED coordinate
54    # system
55    tV = np.zeros((3,3))
56    for i in range(3):
57      for j in range(3):
58        for k in range(3):
59          tV[j,i] = dC[k,j]*V[k,i] + tV[j,i]
60
61    # Initialize dCp
62    dCp = np.zeros((3,3))
63
64    # Direction cosines of principal stresses
65    for i in range(3):
66      for j in range(3):
67        dCp[i,j] = tV[j,2-i]
68      # Avoid precision issues
69      # Make sure the principal axes are unit vectors
70      dCp[i,:] = dCp[i,:]/np.linalg.norm(dCp[i,:])
71
72    # Trend and plunge of principal stresses
73    for i in range(3):
74      pstress[i,1],pstress[i,2] = CartToSph(dCp[i,0],dCp[i,1],
     dCp[i,2])
75
76    return pstress,dCp
```

If we know the principal stresses, and the direction cosines of the pole to the plane **n** with respect to them (Fig. 7.3), it is possible to compute the normal, σ, and maximum shear, τ, tractions acting on the plane (Ramsay, 1967):

$$\begin{aligned} \sigma &= \sigma_1 l^2 + \sigma_2 m^2 + \sigma_3 n^2 \\ \tau^2 &= (\sigma_1 - \sigma_2)^2 l^2 m^2 + (\sigma_2 - \sigma_3)^2 m^2 n^2 + (\sigma_3 - \sigma_1)^2 n^2 l^2 \end{aligned} \tag{7.6}$$

where l, m and n are the direction cosines of **n** with respect to σ_1, σ_2 and σ_3, respectively. Notice that this equation just tells us about the magnitude of the maximum shear traction. It does not say anything about the direction and sense of it. In section 7.4.1 we will see a more comprehensive way to solve this problem.

Let's use the functions introduced so far to solve the following problem: The maximum, intermediate and minimum principal stresses are 40, 30, and 20 MPa, respectively. σ_1 is vertical and σ_3 is horizontal and trends N-S:

1. Compute the tractions parallel to the principal stress directions, acting on a plane with orientation (RHR) 040/65.
2. Compute the normal and maximum shear tractions acting on the same plane
3. Compute the stress tensor on a new coordinate system with $\mathbf{X_1}$ 030/45 and $\mathbf{X_3}$ trending 210.
4. Demonstrate that these new components represent still the same tensor, by computing the principal stresses from the new components.

The notebook ch7-1 shows the solution to this problem:

```
# Import libraries
import numpy as np
pi = np.pi

# Import Cauchy, TransformStress and PrincipalStress
import sys, os
sys.path.append(os.path.abspath('../functions'))
from Cauchy import Cauchy as Cauchy
from TransformStress import TransformStress as
    TransformStress
from PrincipalStress import PrincipalStress as
    PrincipalStress

# Stress tensor in principal stress coordinate system
stress = np.array([[40, 0, 0],[ 0, 30, 0],[ 0, 0, 20]])

# trend and plunge of X1, and trend of X3
tX1 = 0*pi/180
pX1 = 90*pi/180
tX3 = 0*pi/180

# plane orientation
strike = 40*pi/180
dip = 65*pi/180

# X1, X2 and X3 tractions on the plane
T,pT = Cauchy(stress,tX1,pX1,tX3,strike,dip)
```

```
print('X1, X2 and X3 tractions = ', T.round(3),'\n')

# Compute the normal and maximum shear tractions
# on the plane: Eq. 7.6
l2 = pT[0]*pT[0]
m2 = pT[1]*pT[1]
n2 = pT[2]*pT[2]
s1 = stress[0,0]
s2 = stress[1,1]
s3 = stress[2,2]
s12 = s1 - s2
s23 = s2 - s3
s31 = s3 - s1
sigma = s1*l2 + s2*m2 + s3*n2
tau = np.sqrt(s12*s12*l2*m2 + s23*s23*m2*n2 + s31*s31*n2*l2)
print('Sigma = {:.3f}, Tau = {:.3f}\n'.format(sigma,tau))

# New coordinate system
# trend and plunge of X'1,and trend of X'3
ntX1 = 30*np.pi/180
npX1 = 45*np.pi/180
ntX3 = 210*np.pi/180

# Transform stress to new coordinate system
nstress = TransformStress(stress,tX1,pX1,tX3,ntX1,npX1,ntX3)
print('Stress in new coordinate system = \n', nstress.round
    (3),'\n')

# Principal stresses from new components
pstress, dCp = PrincipalStress(nstress,ntX1,npX1,ntX3)
pstress[:,1:3] = pstress[:,1:3]*180/pi
print('Sigma1 = {:.3f}, Trend = {:.1f}, Plunge = {:.1f}'.
    format(pstress[0,0],pstress[0,1],pstress[0,2]))
print('Sigma2 = {:.3f}, Trend = {:.1f}, Plunge = {:.1f}'.
    format(pstress[1,0],pstress[1,1],pstress[1,2]))
print('Sigma3 = {:.3f}, Trend = {:.1f}, Plunge = {:.1f}'.
    format(pstress[2,0],pstress[2,1],pstress[2,2]))
```

```
Output:
X1, X2 and X3 tractions = [16.905 20.828 11.651]
Sigma = 28.392, Tau = 7.015
Stress in new coordinate system =
[[31.25 3.062 8.75]
[3.062 27.5 -3.062]
[8.75 -3.062 31.25]]
Sigma1 = 40.000, Trend = 149.0, Plunge = -90.0
Sigma2 = 30.000, Trend = 270.0, Plunge = -0.0
Sigma3 = 20.000, Trend = 180.0, Plunge = 0.0
```

The principal stresses computed from the new components of the stress tensor are the same as the input principal stresses. Try changing the orientations of the principal stresses in the notebook (e.g. $tX1 = 45$, $pX1 = 0$). Check how the results change. This is a great way to understand the concepts introduced so far.

7.2 Mohr Circle for stress

As discussed in section 6.4.1, the Mohr Circle is derived by making a rotation about one of the principal axes of a symmetric tensor. In the case of stress, the rotation is about one of the principal stresses. In Figure 7.4a, the old axes are parallel to the principal axes of the stress tensor, and the rotation is about the σ_2 axis. If the amount of rotation θ is such that the new $\mathbf{X_1'}$ axis is parallel to the pole to the plane (Fig. 4.7b), the components σ'_{11} and σ'_{13} are the normal (σ) and shear (τ) tractions on the plane, respectively. These tractions are given by:

$$\begin{aligned} \sigma &= \frac{\sigma_1 + \sigma_3}{2} + \frac{\sigma_1 - \sigma_3}{2} \cos 2\theta \\ \tau &= \frac{\sigma_1 - \sigma_3}{2} \sin 2\theta \end{aligned} \tag{7.7}$$

which are similar to Eq. 6.14[1] and define the Mohr Circle for stress.

[1]The sign change in the equation for τ is due to our convention of considering compressional tractions as positive.

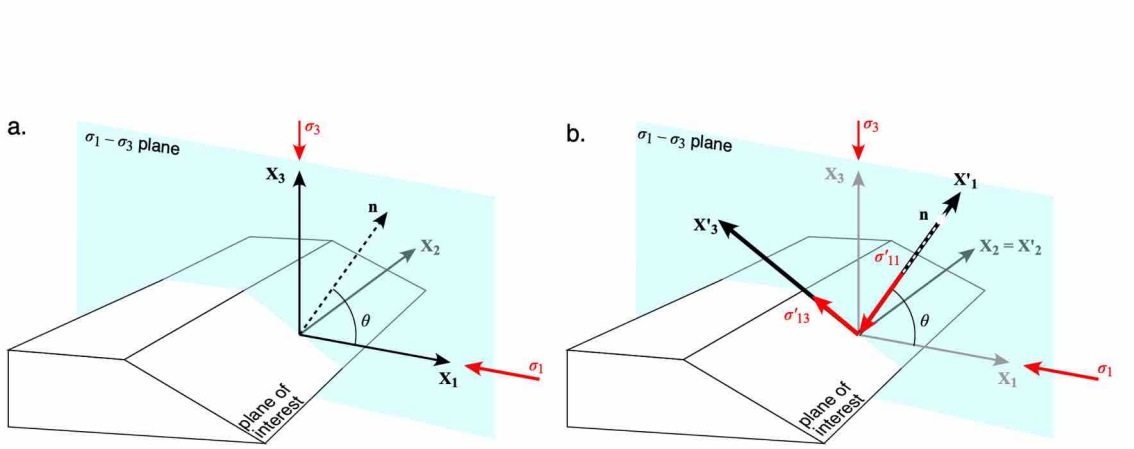


Figure 7.4: Rotation of the stress tensor about σ_2. **a.** The old coordinate system is parallel to the principal axes of the tensor. **b.** In the new coordinate system $\mathbf{X'_1}$ is parallel to the pole to the plane of interest. Modified from Allmendinger et al. (2012).

The horizontal axis of the Mohr Circle for stress is σ and the vertical axis is τ. σ is positive if compressional, and τ is positive if the sense of shear is anticlockwise (e.g. Fig. 7.4b). The center, C, of the Mohr Circle is $(\sigma_1 + \sigma_3)/2$, and the radius, r, is $(\sigma_1 - \sigma_3)/2$ (Fig. 7.5).

If the pole to the plane makes an angle θ with respect to σ_1 (Fig. 7.4), the normal and shear tractions on the plane can be found by drawing in the Mohr Circle a radius CP at an angle 2θ with respect to σ_1 (Fig. 7.5). The coordinates of point P are the normal and shear tractions on the plane.

If we draw lines from σ_1 and σ_3 parallel to the planes on which these tractions act, the lines will intersect at a point Op (Fig. 7.5). This point is known as the pole for planes (Ragan, 2009) and it has a very useful property: A line drawn from Op to any point P on the Mohr Circle circumference is parallel to the plane on which the tractions given by P act (Fig. 7.5). Thus, from the pole for planes, we can rapidly find out the tractions on any plane parallel to σ_2.

Notice that we can rotate the stress tensor about anyone of the three principal axes. Thus in 3D, the Mohr Circle for stress consists of three circles, each one describing the rotation about one of the principal stresses. In section 7.4.2 we will see how to plot the Mohr Circle for stress in 3D.

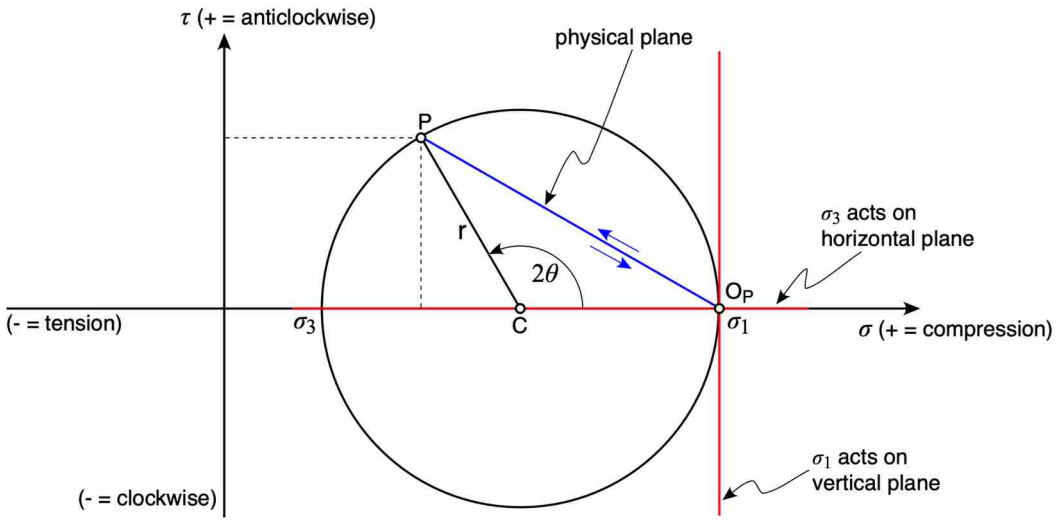


Figure 7.5: Mohr Circle for stress for the situation shown in Fig. 7.4. The normal and shear tractions on the plane (point P) can be found by either measuring 2θ from σ_1 or by tracing the plane from Op.

7.2.1 Special states of stress

There are some special states of stress that can be nicely illustrated with the Mohr Circle:

- Biaxial stress: Two of the principal stresses are non-zero and are different (Fig. 7.6a).
- Triaxial stress: This is the most general type of stress. It has three non-zero, different principal stresses (Fig. 7.6b).
- Cylindrical stress: When two of the principal stresses are equal and the third is different (Fig. 7.6c). If the two equal principal stresses are zero, the stress is known as uniaxial.
- Hydrostatic stress: All three principal stresses have the same value (Fig. 7.6d). In this case, any direction is a principal axis (the stress tensor is a sphere), and there are no shear tractions on the body. This condition is typical of static fluids.
- Pure shear stress: Two of the principal stresses are equal in magnitude

but opposite in sign (one is compressional and the other is tensional), and the third is zero (Fig. 7.6e).

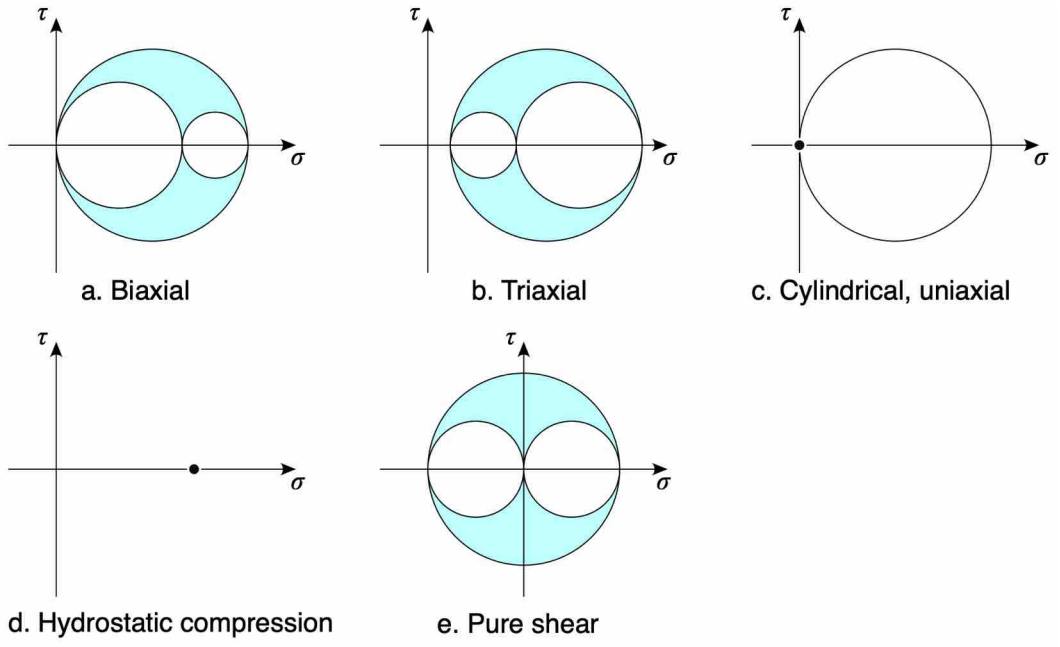


Figure 7.6: Mohr Circle for several special states of stress. **a.** Biaxial: $\sigma_3 = 0$, σ_1 and $\sigma_2 \neq 0$; **b.** Triaxial: σ_1, σ_2 and $\sigma_3 \neq 0$; **c.** Cylindrical uniaxial: σ_2 and $\sigma_3 = 0$, $\sigma_1 \neq 0$; **d.** Hydrostatic: $\sigma_1 = \sigma_2 = \sigma_3$; **e.** Pure shear: $\sigma_3 = -\sigma_1$, $\sigma_2 = 0$. Modified from Allmendinger et al. (2012).

7.3 Mean and deviatoric stress

The special states of stress above allow us to introduce two fundamental types of stress tensors, the mean and the deviatoric stress. The mean traction is the average of the three normal tractions (the elements in the diagonal of the stress tensor):

$$\sigma_m = \frac{\sigma_{11} + \sigma_{22} + \sigma_{33}}{3} \tag{7.8}$$

Notice that the mean traction is the same regardless of the coordinate system because the sum of the normal tractions is the first invariant of the stress

tensor (section 6.2). Knowing the mean traction, we can express the stress tensor as the sum of two tensors:

$$\sigma_{ij} = \begin{bmatrix} \sigma_m & 0 & 0 \\ 0 & \sigma_m & 0 \\ 0 & 0 & \sigma_m \end{bmatrix} + \begin{bmatrix} \sigma_{11} - \sigma_m & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} - \sigma_m & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} - \sigma_m \end{bmatrix} \tag{7.9}$$

The tensor on the left is the mean stress tensor. It is a hydrostatic state of stress that may cause a body to shrink or expand but, on first glance, it is difficult to see how it would change the shape of the body, unless the body is heterogeneous in strength. The tensor on the right is the deviatoric stress tensor. It is the part of the stress tensor that is viewed as causing distortion of the body. For a cylindrical state of stress (Fig. 7.6c), it is easy to visualize these tensors. The mean stress is represented by the center of the Mohr Circle, while the deviatoric stress is represented by the same circle but centered at the origin.

7.4 Applications

In this section we will look at a problem involving stress, namely calculating the magnitude and orientation of the normal and maximum shear tractions on a plane. This problem is crucial to any question involving faulting and fracturing in the upper crust. We will also use the proposed solution to draw the Mohr Circle for stress in 3D.

7.4.1 Normal and shear tractions on a plane

Figure 7.7 illustrates the problem. We want to determine the normal and maximum shear tractions on a plane of any orientation, under a stress with principal axes of any orientation. There are three coordinate systems involved: 1. The geographic coordinate system **NED** where the data are input and the results are output, 2. The principal stress coordinate system which is defined by the orientation of the principal stresses σ_1, σ_2 and σ_3, and 3. A coordinate system defined by the fault plane, with the pole to the plane, **n**, as the first axis, the line of zero shear traction on the fault plane,

b, as the second axis, and the line of maximum shear traction on the fault plane, **s**, as the third axis.

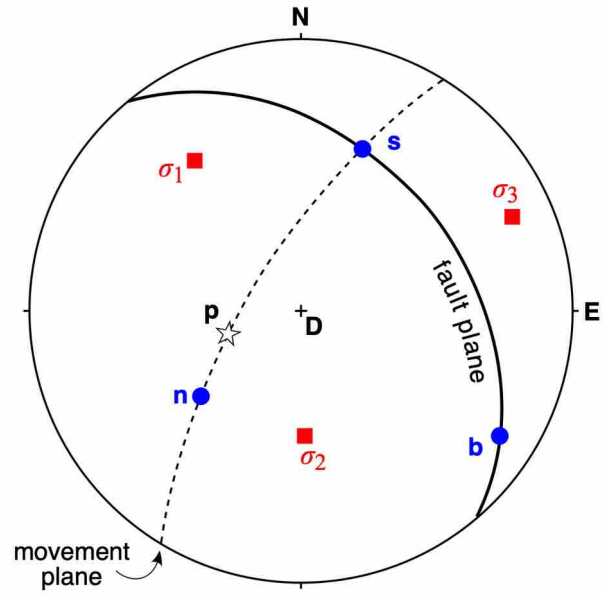


Figure 7.7: Lower hemisphere, equal area stereonet illustrating the three coordinate systems involved in determining the maximum shear traction on a fault plane with pole **n**. These coordinate systems are: **NED**, $\sigma_1\sigma_2\sigma_3$, and **nbs**. **b** and **s** are the directions of zero and maximum shear traction, respectively. Modified from Allmendinger et al. (2012).

The solution to this problem consists of the following steps:

1. Compute the principal stresses and their directions. Here we can use function *PrincipalStress*.

2. Transform the pole to the plane **n** to principal stress coordinates.

3. Compute the traction vector **p** (Fig. 7.7) in principal stress coordinates using Cauchy's law (Eq. 7.4).

4. **n**, **p** and the maximum shear traction **s** fall along the same plane (Fig. 7.7). This plane is known as the *movement plane* (Marshak and Mitra, 1988). The pole to the movement plane is the line of zero shear traction **b**. Therefore, we can calculate **b** by the cross product of **p** and **n**, and **s** by the cross product of **n** and **b**. Obviously after doing this, we need to make **b** and **s** unit vectors by dividing them by their magnitude.

5. Make a transformation matrix **a** between the principal stress coordinates and the fault plane coordinates. This matrix has as the first row **n**, the second row **b**, and the third row **s**.

6. Transform the stress tensor from principal stress coordinates to fault plane coordinates. For the transformed tensor, $\mathbf{X'_1}$ is the pole to the plane, and $\mathbf{X'_3}$ is the direction of the maximum shear traction **s**. Therefore, σ'_{11} is the normal traction on the plane, and σ'_{13} is the maximum shear traction on the plane.

7. Transform **n**, **b** and **s** from principal stress coordinates to **NED** coordinates and find their orientations (trend and plunge).

The function *ShearOnPlane* follows these steps and computes the normal and maximum shear tractions on a plane of any orientation, and under a stress with principal axes of any orientation:

```
import numpy as np
from PrincipalStress import PrincipalStress as
   PrincipalStress
from SphToCart import SphToCart as SphToCart
from CartToSph import CartToSph as CartToSph

def ShearOnPlane(stress,tX1,pX1,tX3,strike,dip):
  '''
  ShearOnPlane calculates the direction and magnitudes of
  the normal and maximum shear tractions on an arbitrarily
  oriented plane

  USE: TT,dCTT,R = ShearOnPlane(stress,tX1,pX1,tX3,strike,dip
   )

  stress = 3 x 3 stress tensor
  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  strike = strike of plane
  dip = dip of plane
  TT = 3 x 3 matrix with the magnitude (column 1),
    trend (column 2) and plunge (column 3) of the:
    normal traction on the plane (row 1), zero shear
    traction (row 2), and max. shear traction (row 3)
  dCTT = 3 x 3 matrix with the direction cosines of unit
    vectors parallel to: normal traction on the plane
    (row 1), zero shear traction (row 2), and maximum
```

```
    shear traction (row 3) with respect to NED
    R = Stress ratio

  NOTE = Input stress tensor does not need to be along
    principal stress directions
    Plane orientation follows the right hand rule
    Input/Output angles are in radians

  ShearOnPlane uses functions PrincipalStress, Cauchy and
  CartToSph

  Python function translated from the Matlab function
  ShearOnPlane in Allmendinger et al. (2012)
  '''
  # Compute principal stresses and their orientations
  pstress, dCp = PrincipalStress(stress,tX1,pX1,tX3)

  # Update stress tensor to principal stress directions
  stress = np.zeros((3,3))
  for i in range(3):
    stress[i,i] = pstress[i,0]

  # Calculate stress ratio
  R = (stress[1,1] - stress[0,0])/(stress[2,2]-stress[0,0])

  # Calculate direction cosines of pole to plane
  p = np.zeros(3)
  p[0], p[1], p[2] = SphToCart(strike,dip,1);

  # Transform pole to plane to  principal stress coordinates
  pT = np.zeros(3);
  for i in range(3):
    for j in range(3):
      pT[i] = dCp[i,j]*p[j] + pT[i]

  # Calculate the tractions in principal stress coordinates
  T = np.zeros(3)
  # Compute tractions using Cauchy's law
  for i in range(3):
    for j in range(3):
      T[i] = stress[i,j]*pT[j] + T[i]

  # Find the B axis by the cross product of T and pT
  B = np.cross(T,pT)

  # Find the max. shear traction orientation
  # by the cross product of pT and B
  S = np.cross(pT,B)

```

```
76    # Convert B and S to unit vectors
77    B = B/np.linalg.norm(B)
78    S = S/np.linalg.norm(S)
79
80    # Now we can write the transformation matrix from
81    # principal stress coordinates to plane coordinates
82    a = np.zeros((3,3))
83    a[0,:] = pT
84    a[1,:] = B
85    a[2,:] = S
86
87    # Transform stress from principal to plane coordinates
88    # Do it only for the first row since we are just
89    # interested in the plane: sigma'11 = normal traction,
90    # sigma'12 = zero, and sigma'13 = max. shear traction
91    TT = np.zeros((3,3))
92    for i in range(3):
93      TT[i,0] = stress[0,0]*a[0,0]*a[i,0] + stress[1,1]*a[0,1]*
       a[i,1] +stress[2,2]*a[0,2]*a[i,2]
94
95    # Transform normal traction, zero shear
96    # and max. shear traction to NED coords
97    dCTT = np.zeros((3,3))
98    for i in range(3):
99      for j in range(3):
100       dCTT[0,i] = dCp[j,i]*pT[j] + dCTT[0,i]
101       dCTT[1,i] = dCp[j,i]*B[j] + dCTT[1,i]
102       dCTT[2,i] = dCp[j,i]*S[j] + dCTT[2,i]
103
104   # Compute trend and plunge of normal traction,
105   # zero shear, and max. shear traction on plane
106   for i in range(3):
107     TT[i,1],TT[i,2] = CartToSph(dCTT[i,0],dCTT[i,1],dCTT[i
      ,2])
108
109   return TT, dCTT, R
```

Function *ShearOnPlane* also computes an important parameter called the *principal stress ratio*, R:

$$R = \frac{(\sigma_2 - \sigma_1)}{(\sigma_3 - \sigma_1)} \tag{7.10}$$

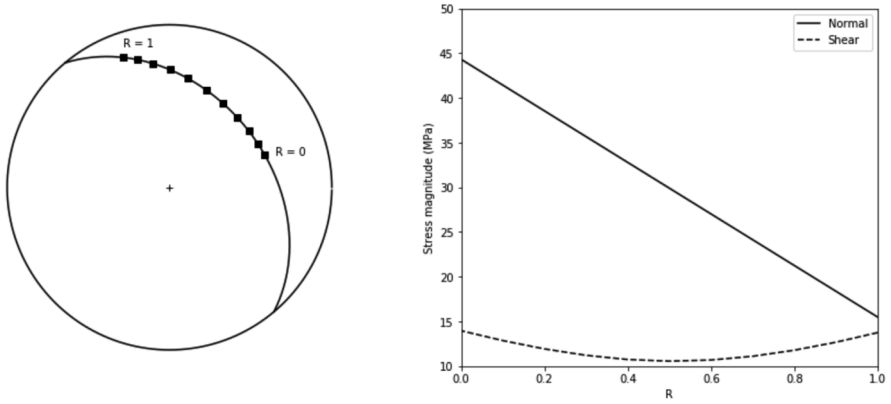
when R is 1, σ_2 is equal to σ_3; when R is 0, σ_2 is equal to σ_1. This ratio is of key importance for predicting faulting and fracturing in the crust (Gephart,

1990; Morris and Ferrill, 2009). Let's look at the influence of this parameter for the case illustrated in Figure 7.7. Let's assume $\sigma_1 = 50$ MPa and $\sigma_3 = 10$ MPa, and let's vary σ_2 between σ_3 ($R = 1$) and σ_1 ($R = 0$). The notebook ch7-2 shows the solution to this problem:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
pi = np.pi

# Import ShearOnPlane, GreatCircle and StCoordLine
import sys, os
sys.path.append(os.path.abspath('../functions'))
from ShearOnPlane import ShearOnPlane as ShearOnPlane
from GreatCircle import GreatCircle as GreatCircle
from StCoordLine import StCoordLine as StCoordLine

# Stress tensor in principal stress coordinate system
# start with R = 1, sigma2 = sigma3
stress = np.array([[50, 0, 0],[ 0, 10, 0],[ 0, 0, 10]])

# Trend and plunge of sigma1, and trend of sigma3
tX1 = 325*pi/180
pX1 = 33*pi/180
tX3 = 66*pi/180

# Plane orientation
strike = 320*pi/180
dip = 40*pi/180

# Number of R increments
rinc = 11

# sigma2 increment
sstep = (stress[0,0] - stress[2,2])/(rinc-1)

# Initialize array
nort = np.zeros(rinc) # normal tractions
sht = np.zeros(rinc) # max. shear traction
tsht = np.zeros(rinc) # trend max. shear traction
psht = np.zeros(rinc) # plunge max. shear traction
rval = np.zeros(rinc) # R value

# Compute normal and shear tractions for all Rs
for i in range(rinc):
    stress[1,1] = stress[2,2] + sstep*i
    # Compute normal and maximum shear tractions on plane
    TT,dCTT,R = ShearOnPlane(stress,tX1,pX1,tX3,strike,dip)
```

```
    # Extract values
    nort[i] = TT[0,0]
    sht[i] = TT[2,0]
    tsht[i] = TT[2,1]
    psht[i] = TT[2,2]
    rval[i] = R

# Make a larger figure
plt.rcParams['figure.figsize'] = [15, 6]

# Plot fault plane and max. shear tractions
# orientations in a lower hemisphere, equal
# area stereonet
plt.subplot(1,2,1)
# Plot the primitive of the stereonet
r = 1; # unit radius
TH = np.arange(0,360,1)*pi/180
x = r * np.cos(TH)
y = r * np.sin(TH)
plt.plot(x,y,'k')
# Plot center of circle
plt.plot(0,0,'k+')
# Make axes equal and remove them
plt.axis('equal')
plt.axis('off')
# Plot fault plane
path = GreatCircle(strike,dip,1)
plt.plot(path[:,0], path[:,1], 'k')
# Plot max. shear tractions orientations
for i in range(rinc):
    x, y = StCoordLine(tsht[i],psht[i],1)
    plt.plot(x,y,'ks')
    if i == 0:
        plt.annotate('R = 1',(x,y),textcoords='offset points'
   ,xytext=(0,10))
    if i == rinc-1:
        plt.annotate('R = 0',(x,y),textcoords='offset points'
   ,xytext=(10,0))

# Plot normal and shear tractions versus R
plt.subplot(1,2,2)
plt.plot(rval,nort,'k')
plt.plot(rval,sht,'k--')
plt.axis([0, 1, 10, 50])
plt.xlabel('R')
plt.ylabel('Stress magnitude (MPa)')
plt.legend(['Normal', 'Shear']);
```

Output:



This is essentially Figure 6.9 of Allmendinger et al. (2012). In one of the exercises of section 7.5, you will explore more the significance of the principal stress ratio, R.

7.4.2 The Mohr Circle for stress in 3D

We can also use the solution above to draw a Mohr Circle for stress in 3D. Here it is also important to determine the sense of the maximum shear traction. According to our convention, anticlockwise shear is positive, and clockwise shear is negative. The function *StressMohrCircle* draws the Mohr Circle in 3D for a give stress tensor and stress coordinate system. It also plots and delivers the normal and maximum shear tractions on a group of input planes.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from PrincipalStress import PrincipalStress as
       PrincipalStress
4  from SphToCart import SphToCart as SphToCart
5  from CartToSph import CartToSph as CartToSph
6
7  def StressMohrCircle(stress,tX1,pX1,tX3,planes):
8    '''
9    Given the stress tensor in a X1X2X3 coordinate system,
10   and a group of n planes, StressMohrCircle draws the Mohr
11   Circle for stress (including the planes). It
12   also returns the normal and max. shear tractions on the
13   planes and their orientations
14
```

```
  USE: ns, ons = StressMohrCircle(stress,tX1,pX1,tX3,planes)

  stress = 3 x 3 stress tensor
  tX1 = trend of X1
  pX1 = plunge of X1
  tX3 = trend of X3
  planes =  n x 2 vector with strike and dip of planes
  ns = n x 2 vector with the normal and max. shear
    tractions on the planes
  ons = n x 4 vector with the trend and plunge of the
    normal traction (columns 1 and 2), and the
    max. shear traction (columns 3 and 4)

  NOTE = Planes orientation follows the right hand rule
    Input and output angles are in radians

  StressMohrCircle uses functions PrincipalStress
  '''
  # tolerance for near zero values
  tol = 1e-6

  # Compute principal stresses and their orientations
  pstress, dCp = PrincipalStress(stress,tX1,pX1,tX3)

  # Update stress tensor to principal stresses
  stress = np.zeros((3,3))
  for i in range(3):
    stress[i,i] = pstress[i,0]

  # Draw sigma1-sigma3 circle
  c = (stress[0,0] + stress[2,2])/2.0
  r = (stress[0,0] - stress[2,2])/2.0
  th =np.arange(0,2*np.pi,np.pi/50)
  x = r * np.cos(th)+c
  y = r*np.sin(th)
  fig, ax = plt.subplots()
  plt.plot(x,y,'k-')

  # Draw sigma1-sigma2 circle
  c = (stress[0,0] + stress[1,1])/2.0
  r = (stress[0,0] - stress[1,1])/2.0
  th =np.arange(0,2*np.pi,np.pi/50)
  x = r * np.cos(th)+c
  y = r*np.sin(th)
  plt.plot(x,y,'k-')

  # Draw sigma2-sigma3 circle
  c = (stress[1,1] + stress[2,2])/2.0
  r = (stress[1,1] - stress[2,2])/2.0
```

```
64    th =np.arange(0,2*np.pi,np.pi/50)
65    x = r * np.cos(th)+c
66    y = r*np.sin(th)
67    plt.plot(x,y,'k-')
68
69
70    # Initialize pole to plane
71    p = np.zeros(3)
72
73    # Initialize vectors with normal and
74    # max. shear tractions
75    ns = np.zeros((np.size(planes,0),2))
76    ons = np.zeros((np.size(planes,0),4))
77
78    # Compute normal and max. shear tractions
79    for i in range(np.size(planes,0)):
80
81      # Calculate direction cosines of pole to plane
82      p[0],p[1],p[2] = SphToCart(planes[i,0],planes[i,1],1)
83
84      # Trend and plunge of pole = dir. normal traction
85      ons[i,0],ons[i,1] = CartToSph(p[0],p[1],p[2])
86
87      # Transform pole to  principal stress coordinates
88      pT = np.zeros(3)
89      for j in range(3):
90        for k in range(3):
91          pT[j] = dCp[j,k]*p[k] + pT[j]
92
93      # Calculate the tractions in principal stress
94      # coordinates
95      T = np.zeros(3)
96      for j in range(3):
97        for k in range(3):
98          T[j] = stress[j,k]*pT[k] + T[j]
99
100     # Find the B and S axes
101     B = np.cross(T,pT);
102     S = np.cross(pT,B);
103     B = B/np.linalg.norm(B);
104     S = S/np.linalg.norm(S);
105
106     # Transformation matrix from principal
107     # stress coordinates to plane coordinates
108     a = np.zeros((3,3))
109     a[0,:] = pT
110     a[1,:] = B
111     a[2,:] = S
112
```

```
113     # normal and max. shear tractions
114     ns[i,0] = stress[0,0]*a[0,0]*a[0,0] + stress[1,1]*a[0,1]*
        a[0,1]+ stress[2,2]*a[0,2]*a[0,2]
115     ns[i,1] = stress[0,0]*a[0,0]*a[2,0] + stress[1,1]*a[0,1]*
        a[2,1]+ stress[2,2]*a[0,2]*a[2,2]
116
117     # Calculate direction cosines of max.
118     # shear traction with respect to NED
119     ds = np.zeros(3)
120     for j in range(3):
121       for k in range(3):
122         ds[j] = dCp[k,j]*S[k] + ds[j]
123
124     # Trend and plunge of max. shear traction
125     ons[i,2],ons[i,3] = CartToSph(ds[0],ds[1],ds[2])
126
127     # Cross product of pole and max. shear traction
128     ps = np.cross(p,ds)
129
130     # Make clockwise shear traction negative
131     if np.abs(ps[2]) < tol: # Dip slip
132       if ds[2] > 0.0: # Normal slip
133         if pT[0]*pT[2] < 0.0:
134           ns[i,1] *= -1.0
135       else: # Reverse slip
136         if pT[0]*pT[2] >= 0.0:
137           ns[i,1] *= -1.0
138     else: # Oblique slip
139       if ps[2] < 0.0:
140         ns[i,1] *= -1.0
141
142   # Plot planes
143   plt.plot(ns[:,0],ns[:,1],'ks')
144
145   # Make axes equal and plot grid
146   plt.axis ('equal')
147   plt.grid()
148
149   # Move x-axis to center and y-axis to origin
150   ax.spines['bottom'].set_position('center')
151   ax.spines['left'].set_position('zero')
152
153   # Eliminate upper and right axes
154   ax.spines['right'].set_color('none')
155   ax.spines['top'].set_color('none')
156
157   # Show ticks in the left and lower axes only
158   ax.xaxis.set_ticks_position('bottom')
159   ax.yaxis.set_ticks_position('left')
```

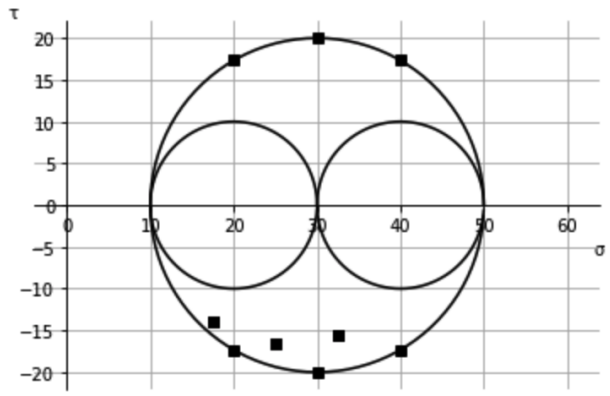
```
160
161   # Add labels at end of axes
162   ax.set_xlabel('σ',x=1)
163     ax.set_ylabel('τ',y=1,rotation=0)
164
165   # Show plot
166   plt.show()
167
168   return ns, ons
```

Let's use this function to plot the Mohr Circle for the following condition: The maximum, intermediate and minimum principal stresses are 50, 30, and 10 MPa, respectively. σ_1 is horizontal and trends E-W, while σ_3 is vertical. Plot on the Mohr Circle the normal and maximum shear tractions on planes with the following orientation (RHR): 000/30, 000/45, 000/60, 180/30, 180/45, 180/60, 045/30, 045/45, 045/60, 225/30, 225/45 and 225/60. The notebook ch7-3 shows the solution to this problem:

```
1  # Import libraries
2  import numpy as np
3  pi = np.pi
4
5  # Import StressMohrCircle
6  import sys, os
7  sys.path.append(os.path.abspath('../functions'))
8  from StressMohrCircle import StressMohrCircle as
       StressMohrCircle
9
10 # Stress tensor in principal stress coordinate system
11 stress = np.array([[50, 0, 0],[ 0, 30, 0],[ 0, 0, 10]])
12
13 # Trend and plunge of sigma1, and trend of sigma3
14 tX1 = 90*pi/180
15 pX1 = 0*pi/180
16 tX3 = 90*pi/180
17
18 # Planes orientations
19 planes = np.array
       ([[0,30],[0,45],[0,60],[180,30],[180,45],[180,60],[45,30],
        [45,45],[45,60],[225,30],[225,45],[225,60]])
20 planes = planes * pi/180 # radians
21
22 # Plot Mohr Circle
23 ns, ons = StressMohrCircle(stress,tX1,pX1,tX3,planes)
24
```

```
25 # Print normal and shear tractions
26 print('Strike\tDip\tNormal\tTrend\tPlunge\tShear\tTrend\
       tPlunge\t')
27 # return to degrees
28 planes = planes*180/pi
29 ons = ons*180/pi
30 # print
31 for i in range(np.size(planes,0)):
32     print('{:.1f}\t{:.1f}\t{:.3f}\t{:.1f}\t{:.1f}\t{:.3f}\t
       {:.1f}\t{:.1f}'.format(planes[i,0],planes[i,1],ns[i,0],ons
       [i,0],ons[i,1],ns[i,1],ons[i,2],ons[i,3]))
```

Output:



Strike	Dip	Normal	Trend	Plunge	Shear	Trend	Plunge
0.0	30.0	20.000	270.0	60.0	17.321	270.0	−30.0
0.0	45.0	30.000	270.0	45.0	20.000	270.0	−45.0
0.0	60.0	40.000	270.0	30.0	17.321	270.0	−60.0
180.0	30.0	20.000	90.0	60.0	−17.321	90.0	−30.0
180.0	45.0	30.000	90.0	45.0	−20.000	90.0	−45.0
180.0	60.0	40.000	90.0	30.0	−17.321	90.0	−60.0
45.0	30.0	17.500	315.0	60.0	−13.919	291.0	−27.8
45.0	45.0	25.000	315.0	45.0	−16.583	281.3	−39.8
45.0	60.0	32.500	315.0	30.0	−15.612	261.9	−46.1
225.0	30.0	17.500	135.0	60.0	−13.919	111.0	−27.8
225.0	45.0	25.000	135.0	45.0	−16.583	101.3	−39.8
225.0	60.0	32.500	135.0	30.0	−15.612	81.9	−46.1

N-S striking planes parallel to σ_2 plot on the σ_1-σ_3 circle. Planes dipping east show positive, anticlockwise shear, while those dipping west show negative, clockwise shear. Another way to visualize the N-S planes is by using the pole to planes, Op. With σ_1 horizontal and σ_3 vertical, Op is at σ_1 (Fig. 7.5). From Op you can trace N-S planes of any dip to the E or W, and quickly verify that E dipping planes have anticlockwise shear, while W dipping planes

have clockwise shear. Planes non-parallel to σ_2 are more difficult to visualize. On the NE-SW striking planes dipping both to the SE and NW, the shear tractions are clockwise and they plot inside the σ_1-σ_3 circle but outside the σ_2-σ_3 and σ_1-σ_2 circles (in fact no plane orientation will plot inside these two internal circles).

7.5 Exercises

1. This is exercise 6 in chapter 6 of Allmendinger et al. (2012): In the Oseberg field, North Sea, the principal stresses are oriented $\sigma_1 = 080/00$, $\sigma_2 = 000/90$, and $\sigma_3 = 170/00$. If at 2 km depth, $\sigma_1 = 50$ MPa, $\sigma_2 = 40$ MPa, and $\sigma_3 = 30$ MPa, what are the normal and shear tractions on a plane oriented (RHR) 040/65? *Hint*: Use function *ShearOnPlane*.

2. Morris and Ferrill (2009) discuss the importance of the intermediate principal stress, σ_2, for faulting. Let's look at their first example: The maximum and minimum principal stresses are 95 and 25 MPa, respectively. σ_1 is vertical and σ_3 is horizontal and trends E-W. Compute the normal and maximum shear tractions acting on three planes with orientation (RHR) 180/45, 090/45 and 135/45, and for principal stress ratios, R, between 1 and 0. Display your results graphically in a lower hemisphere equal area stereonet showing the maximum shear directions on the planes, and a plot of R versus normal and maximum shear tractions. How do the normal and maximum shear tractions vary on the N-S and E-W planes? How do they vary in the SE-NW plane? *Hint*: This problem is similar to the notebook ch7-2. Modify the notebook to include the new stress and the three planes. Use colors to indicate the different planes.

3. The slip tendency is the ratio between the maximum shear and normal tractions on a plane ($T_s = \tau/\sigma$). It is a proxy for the tendency of a surface to undergo slip under a given stress field (Morris et al., 1996). Write a Python function that for a given stress with coordinate axes of any orientation, plots a lower hemisphere equal area stereonet and a Mohr Circle colored by slip tendency. The function should work as follows:

SlipTendency(stress,tX1,pX1,tX3)

where *stress* is the stress tensor, *tX1* and *pX1* are the trend and plunge of $\mathbf{X_1}$, and *tX3* is the trend of $\mathbf{X_3}$. Figure 7.8 shows how the output of the function would look like for the stress tensor in the notebook ch7-3.

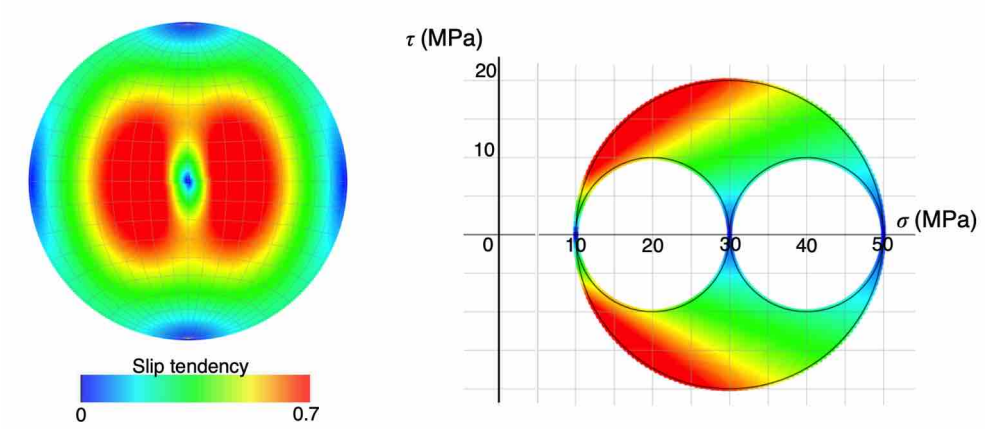


Figure 7.8: Slip tendency for the state of stress in the notebook ch7-3. Left is lower hemisphere equal area stereonet, and right is Mohr Circle. Figure made with the program GeoKalk by Nestor Cardozo.

Hint: Use the function *StressMohrCircle* as the base of your new function. In the new function, calculate the normal and shear tractions on a set of planes varying in strike from 0 to 360° and dip from 0 to 90°. A 1° increment in strike and dip would be sufficient. Plot each pole (not the plane itself) colored by slip tendency on a stereonet and a Mohr Circle. Use a color scale bar similar to Fig. 7.8. Since you will be plotting about 35,000 poles and since in this case the direction of the maximum shear traction is not relevant, you may consider calculating the normal and maximum shear tractions using Eq. 7.6.

Based on your new function, at which dip angle would you expect the N-S oriented planes in the notebook ch7-3 to slip? Will the planes slip at the possible maximum shear traction $(\sigma_1 - \sigma_3)$?

References

Allmendinger, R.W., Cardozo, N. and Fisher, D.M. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Gephart, J.W. 1990. Stress and the direction of slip on fault planes. Tectonics 9, 845-858.

Marshak, S. and Mitra, G. 1988. Basic Methods of Structural Geology. Prentice Hall.

Morris, A.P., Ferrill, D.A. and Henderson, D.B. 1996. Slip-tendency analysis and fault reactivation. Geology 24, 275-278.

Morris, A.P. and Ferrill, D.A. 2009. The importance of the effective intermediate principal stress (σ_2') to fault slip patterns. Journal of Structural Geology 31, 950-959.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Ramsay, J.G. 1967. Folding and Fracturing of Rocks. McGraw-Hill, New York.

Chapter 8

Strain

Stresses within the Earth change through time, and these changes can lead to deformation. Deformation is more complicated than stress because it involves the comparison of states of the rock material at two different points in time. Therefore, one needs to establish both temporal and spatial reference frames. This chapter covers the basics of deformation and strain, including infinitesimal, finite, and progressive strain. This is a relatively long and complicated material, it comprises several chapters in classical books such as Ramsay (1967) and Means (1976). However, our purpose is not to discuss in detail the theory of strain, but rather to introduce some interesting strain problems in geosciences, and their computation.

8.1 Deformation and strain

Strictly speaking, deformation involves rigid body deformation (translation and rotation), and non-rigid body deformation (changes in shape and volume) or strain. In geology, rigid-body deformation is rather difficult to determine, with a few exceptions (e.g. paleolatitude of a continent from paleomagnetic pole). We often just focus on strain.

Consider the deformation shown in Fig. 8.1. The square and inscribed circle (Fig. 8.1a) are first translated, then rotated, and then sheared. The translation vector $\mathbf{t}$ (Fig. 8.1b) and rotation ω (Fig. 8.1c) describe the first stage of rigid body-deformation. Shear of the objects defines the second stage

of non-rigid body deformation or strain (Fig. 8.1d).

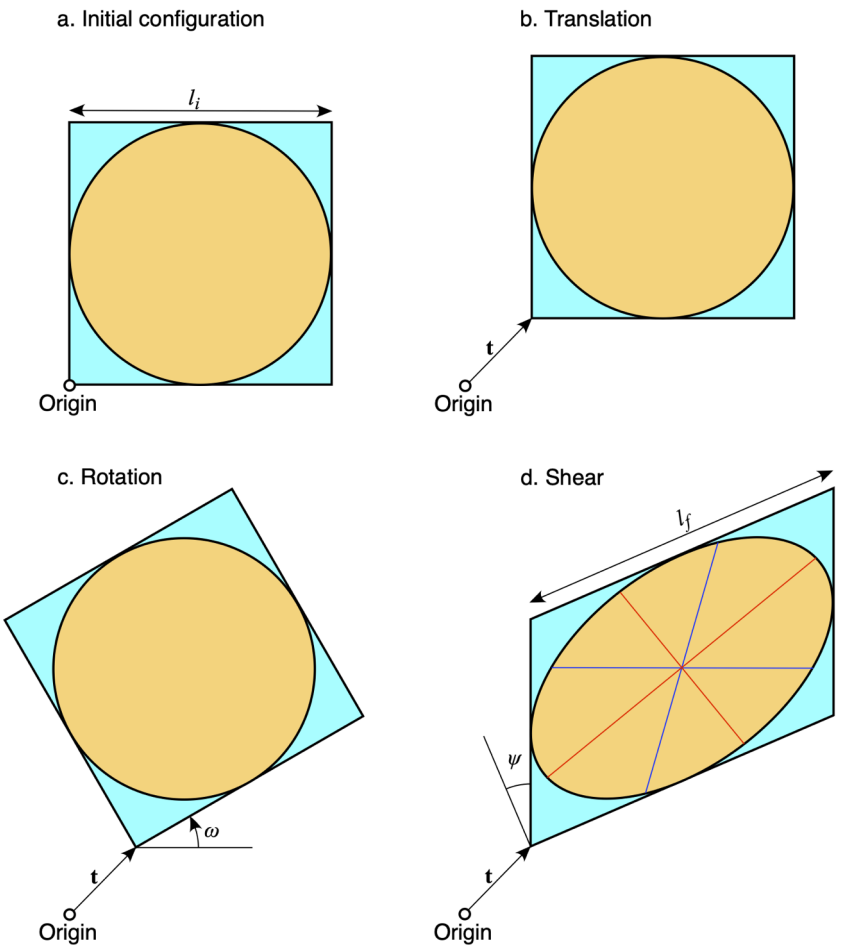


Figure 8.1: Deformation of a square of side l_i and inscribed circle. **a.** Initial configuration. **b.** Translation **t**. **c.** Rotation ω. **d.** Shear. l_f and ψ are the final length and angular shear of the long side of the parallelogram. Red and blue lines in ellipse are principal axes and lines of no finite elongation (LNFE), respectively.

The changes in shape (distortion) of the bodies can be described by the changes in length and angle of lines in the bodies. If the initial length of a line is l_i and the deformed length of the line is l_f, the change in length of the line can be defined by either one of the following parameters:

$$\begin{aligned} e &= \frac{l_f - l_i}{l_i} \\ S &= \frac{l_f}{l_i} = 1 + e \\ \lambda &= S^2 = (1 + e)^2 \end{aligned} \tag{8.1}$$

where e is elongation, S is stretch, and λ is quadratic elongation. For the long side of the parallelogram in Fig. 8.1d, e is positive, and S and λ are larger than 1. For the short side, e is negative, and S and λ are lower than 1. Lines that have their original length have $e = 0$, and S and $\lambda = 1$ (blue lines, Fig. 8.1d). Notice than e cannot be lower than -1, and S and λ cannot be lower than 0 (a line can't be shortened more than its original length).

Changes in angle are measured by the angular shear ψ, which is the change in angle between two originally perpendicular lines (Fig. 8.1d). From the angular shear, one can calculate the shear strain γ:

$$\gamma = \tan \psi \tag{8.2}$$

Changes in area or volume (dilation) can be described using the areal or volumetric stretch:

$$\begin{aligned} S_A &= \frac{A_f}{A_i} \\ S_V &= \frac{V_f}{V_i} \end{aligned} \tag{8.3}$$

where A_i and A_f are initial and final area, and V_i and V_f are initial and final volume. We can also determine areal or volumetric elongation.

In this chapter, we make two major assumptions about strain: Strain is continuous (it is distributed uniformly across the body), and strain is homogeneous (it is identical across the body). Clearly these assumptions are incorrect: Rocks are full of discontinuities, and geological structures (e.g. folds) exhibit heterogeneous strain. However at the appropriate scale, rocks can be described as continuum materials, and the heterogeneous strain of

geological structures can be represented by domains of homogeneous (yet compatible) strain. This makes possible applying homogeneous strain to geologic deformation.

For homogeneous strain, straight lines remain straight, parallel lines remain parallel, and circles become ellipses in 2D, or spheres become ellipsoids in 3D (Fig. 8.1d). The resultant ellipse (or ellipsoid) is called the strain ellipse (or ellipsoid). The stretches along the axes of this ellipse (or ellipsoid, red lines in Fig. 8.1d) are called the principal stretches, and they are denoted by the symbols S_1, S_2, S_3, for the maximum, intermediate, and mininmum principal stretch, respectively. The volumetric stretch can also be defined as:

$$S_V = S_1 S_2 S_3 \tag{8.4}$$

The deformation in Fig. 8.1 is volume and area constant (S_V and $S_A = 1$). Under this condition, there are two lines in the strain ellipse that have their original length (blue lines, Fig. 8.1d). These lines are known as the lines of no finite elongation (LNFE).

8.2 Deformation and displacement gradients

Consider the deformation shown in Fig. 8.2. The cube is transformed into a brick-shaped body. It is shortened along the $\mathbf{X_2}$ axis by half ($e = -0.5$, $S = 0.5$), and it is stretched along the $\mathbf{X_3}$ axis twice ($e = 1$, $S = 2$).

We can express the deformed coordinates $\mathbf{x}$ of any point in the cuboid in terms of the undeformed coordinates $\mathbf{X}$ of the same point in the cube[1]:

$$\begin{aligned} x_1 &= 1X_1 + 0X_2 + 0X_3 \\ x_2 &= 0X_1 + 0.5X_2 + 0X_3 \\ x_3 &= 0X_1 + 0X_2 + 2X_3 \end{aligned} \tag{8.5}$$

Likewise, we can express the undeformed coordinates $\mathbf{X}$ of any point in the cube in terms of the deformed coordinates $\mathbf{x}$ of the same point in the cuboid:

[1] By convention, we use $\mathbf{X}$ to refer to the undeformed coordinates, and $\mathbf{x}$ to denote the deformed coordinates

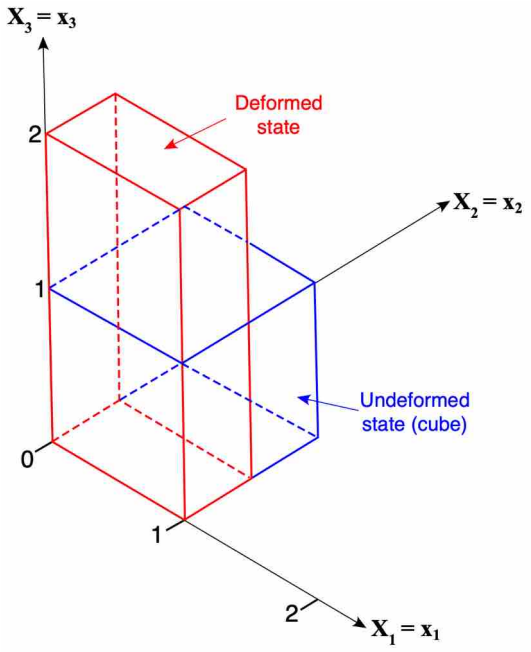


Figure 8.2: Deformation of a cube in to a brick-shaped body. The principal axes of strain are parallel to the coordinate axes before **X** and after **x** deformation. Modified from Means (1976).

$$\begin{aligned} X_1 &= 1x_1 + 0x_2 + 0x_3 \\ X_2 &= 0x_1 + 2x_2 + 0x_3 \\ X_3 &= 0x_1 + 0x_2 + 0.5x_3 \end{aligned} \tag{8.6}$$

Equations 8.5 and 8.6 are called *coordinate transformations*, but they are fundamentally different from the transformation of coordinate axes in Chapter 5. The coordinate transformations here are between two different states in time. Eq. 8.5 is called a *Green* transformation (new in terms of old), while Eq. 8.6 is a *Cauchy* transformation (old in terms of new).

We can take the partial derivatives of these equations, which are just the coefficients of the equations:

$$\frac{\partial x_i}{\partial X_j} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 2 \end{bmatrix} \tag{8.7}$$

and:

$$\frac{\partial X_i}{\partial x_j} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \tag{8.8}$$

These partial derivatives are known as the deformation gradients and unlike the coordinate transformations, they are homogeneous and independent of the coordinate axes (i.e. they are tensors). Eq. 8.7 is called the *Green* deformation gradient, while Eq. 8.8 is the *Cauchy* deformation gradient. Since in Fig. 8.2 the principal axes of strain are parallel to the coordinate axes, we can easily see one thing: The components of the Green deformation gradient are the stretches along the coordinate axes, and the components of the Cauchy deformation gradient are the inverse of the stretches along the coordinate axes. For other more complex situations this will not be the case, but in general the Green and Cauchy deformation gradient tensors are related to the stretch (Allmendinger et al., 2012).

Another way to study the deformation is to look at the displacement **u** between the undeformed and deformed states (Fig. 8.2). The displacement can either be expressed in the undeformed configuration:

$$\begin{aligned} u_1 &= 0X_1 + 0X_2 + 0X_3 \\ u_2 &= 0X_1 - 0.5X_2 + 0X_3 \\ u_3 &= 0X_1 + 0X_2 + 1X_3 \end{aligned} \tag{8.9}$$

or in the deformed configuration:

$$\begin{aligned} u_1 &= 0x_1 + 0x_2 + 0x_3 \\ u_2 &= 0x_1 - 1x_2 + 0x_3 \\ u_3 &= 0x_1 + 0x_2 + 0.5x_3 \end{aligned} \tag{8.10}$$

Eq. 8.9 is known as the *Lagrangian* displacement (in terms of old coordinates), while Eq. 8.10 is the *Eulerian* displacement (in terms of new coordinates). We can take the partial derivatives of these equations, which are just the coefficients of the equations:

$$\frac{\partial u_i}{\partial X_j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.11}$$

and:

$$\frac{\partial u_i}{\partial x_j} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \tag{8.12}$$

These partial derivatives are known as the displacement gradients and unlike the displacement, they are homogeneous and independent of the coordinate axes (i.e. they are tensors). Eq. 8.11 is called the *Lagrangian* displacement gradient, while Eq. 8.12 is the *Eulerian* displacement gradient. For the case of Fig. 8.2, we can see that the components of the Lagrangian displacement gradient are the elongations along the coordinate axes, while the components of the Eulerian displacement gradient are the elongations along the coordinate axes with respect to the deformed reference frame ($\tilde{e} = (l_f - l_i)/l_f$). This will not be the case for more complicated situations, but in general the Lagrangian and Eulerian displacement gradient tensors are related to the elongation (Allmendinger et al., 2012).

Table 8.1 summarizes this section.

	Old coordinates	**New coordinates**
Coordinate transformations	Green $x_i = \frac{\partial x_i}{\partial X_j} X_j$	Cauchy $X_i = \frac{\partial X_i}{\partial x_j} x_j$
Displacement	Lagrangian $u_i = \frac{\partial u_i}{\partial X_j} X_j$	Eulerian $u_i = \frac{\partial u_i}{\partial x_j} x_j$

Table 8.1: Coordinate transformations and displacement via deformation and displacement gradient tensors. Rigid body deformation is not included.

8.3 Infinitesimal strain

Suppose that a line parallel to the $\mathbf{X_1}$ axis is elongated 1% of its initial length. In this case the displacement gradient is:

$$\frac{\partial u_1}{\partial X_1} = \frac{0.01}{1.0} = 0.01 \qquad \text{and} \qquad \frac{\partial u_1}{\partial x_1} = \frac{0.01}{1.01} = 0.0099$$

Thus, when strains are small ($e < 1\%$), $\frac{\partial u_i}{\partial X_i} \approx \frac{\partial u_i}{\partial x_i}$, and the difference between the displacement gradient in the initial or final state is insignificant. Small strains are called *infinitesimal strains*, and they are important in a number of fields in geosciences, particularly in geophysics.

For infinitesimal strain, the distinction between old and new coordinates is irrelevant, and therefore we can just use one column in Table 8.1. The displacement gradient $\mathbf{e}$[2] is an asymmetric tensor, and it can be decomposed into a symmetric and an antisymmetric tensor (Eq. 6.2):

$$e_{ij} = \varepsilon_{ij} + \omega_{ij} \tag{8.13}$$

where:

$$\varepsilon_{ij} = \frac{1}{2}(e_{ij} + e_{ji}) = \begin{bmatrix} e_{11} & \frac{e_{12}+e_{21}}{2} & \frac{e_{13}+e_{31}}{2} \\ \frac{e_{21}+e_{12}}{2} & e_{22} & \frac{e_{23}+e_{32}}{2} \\ \frac{e_{31}+e_{13}}{2} & \frac{e_{32}+e_{23}}{2} & e_{33} \end{bmatrix} \tag{8.14}$$

and:

$$\omega_{ij} = \frac{1}{2}(e_{ij} - e_{ji}) = \begin{bmatrix} 0 & \frac{e_{12}-e_{21}}{2} & \frac{e_{13}-e_{31}}{2} \\ \frac{e_{21}-e_{12}}{2} & 0 & \frac{e_{23}-e_{32}}{2} \\ \frac{e_{31}-e_{13}}{2} & \frac{e_{32}-e_{23}}{2} & 0 \end{bmatrix} \tag{8.15}$$

The symmetric tensor $\boldsymbol{\varepsilon}$ is called the strain tensor. The diagonal, ε_{ii}, terms of this tensor correspond to the elongations along the axes of the reference system, and the off-diagonal terms, ε_{ij}, to half the shear strain ($\varepsilon_{ij} = \frac{\gamma_{ij}}{2}$). Like

[2]We use $\mathbf{e}$ to denote the displacement gradient. This is different than the non-bold italic letter e which is used for elongation.

any symmetric tensor, the strain tensor has three principal axes which can be determined by computing the eigenvalues and eigenvectors of the tensor (section 6.2). These principal axes define the infinitesimal strain ellipsoid.

The antisymmetric tensor $\boldsymbol{\omega}$ is also known as an axial vector. The Cartesian coordinates, r_i, of this vector give the orientation of the rotation axis:

$$r_1 = \frac{-(\omega_{23} - \omega_{32})}{2} \quad r_2 = \frac{-(-\omega_{13} + \omega_{31})}{2} \quad \text{and} \quad r_3 = \frac{-(\omega_{12} - \omega_{21})}{2} \tag{8.16}$$

The amount of rotation in radians is just the length of the vector $\mathbf{r}$:

$$\omega = |\mathbf{r}| = \sqrt{r_1^2 + r_2^2 + r_3^2} \tag{8.17}$$

The function *InfStrain* computes the infinitesimal strain and rotation tensors from a displacement gradient, as well as the principal strains, and the components and amount of rotation:

```
import numpy as np

from CartToSph import CartToSph as CartToSph
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi

def InfStrain(e):
  '''
  InfStrain computes infinitesimal strain from an input
  displacement gradient tensor

  USE: eps,ome,pstrain,rotc,rot = InfStrain(e)

  e = 3 x 3 displacement gradient tensor
  eps = 3 x 3 strain tensor
  ome = 3 x 3 rotation tensor
  pstrain = 3 x 3 matrix with magnitude (column 1), trend
      (column 2) and plunge (column 3) of maximum
      (row 1), intermediate (row 2),and minimum
      (row 3) principal strains
  rotc = 1 x 3 vector with rotation components
  rot = 1 x 3 vector with rotation magnitude and trend
    and plunge of rotation axis

  NOTE: Output trends and plunges of principal strains
```

```
    and rotation axes are in radians

  InfStrain uses function CartToSph and ZeroTwoPi

  Python function translated from the Matlab function
  InfStrain in Allmendinger et al. (2012)
  '''
  # Initialize variables
  eps = np.zeros((3,3))
  ome = np.zeros((3,3))
  pstrain = np.zeros((3,3))
  rotc = np.zeros(3)
  rot = np.zeros(3)

  # Compute strain and rotation tensors
  # Eqs. 8.14 and 8.15
  for i in range(3):
    for j in range(3):
      eps[i,j]=0.5*(e[i,j]+e[j,i])
      ome[i,j]=0.5*(e[i,j]-e[j,i])

  # Compute principal strains and orientations.
  # Here we use the function eigh. D is a vector
  # of eigenvalues (i.e. principal strains), and V is a
  # full matrix whose columns are the corresponding
  # eigenvectors (i.e. principal strain directions)
  D,V = np.linalg.eigh(eps)

  # Maximum principal strain
  pstrain[0,0] = D[2]
  pstrain[0,1],pstrain[0,2] = CartToSph(V[0,2],V[1,2],V[2,2])
  # Intermediate principal strain
  pstrain[1,0] = D[1]
  pstrain[1,1],pstrain[1,2] = CartToSph(V[0,1],V[1,1],V[2,1])
  # Minimum principal strain
  pstrain[2,0] = D[0]
  pstrain[2,1],pstrain[2,2] = CartToSph(V[0,0],V[1,0],V[2,0])

  # Calculate rotation components, Eq. 8.16
  rotc[0]=(ome[1,2]-ome[2,1])*-0.5
  rotc[1]=(-ome[0,2]+ome[2,0])*-0.5
  rotc[2]=(ome[0,1]-ome[1,0])*-0.5

  # Compute rotation magnitude, Eq. 8.17
  rot[0] = np.sqrt(rotc[0]**2+rotc[1]**2+rotc[2]**2)
  # Compute trend and plunge of rotation axis
  rot[1],rot[2] = CartToSph(rotc[0]/rot[0],rotc[1]/rot[0],
   rotc[2]/rot[0])
  # If plunge is negative
```

```
73    if rot[2] < 0:
74      rot[1] = ZeroTwoPi(rot[1]+np.pi)
75      rot[2] = -rot[2]
76      rot[0] = -rot[0]
77
78    return eps,ome,pstrain,rotc,rot
```

8.3.1 Mohr Circle for infinitesimal strain

As discussed in section 6.4.1, the rotation of the infinitesimal strain tensor about one principal axis can be represented by a Mohr Circle. We start with the infinitesimal strain tensor in principal coordinates:

$$\varepsilon_{ij} = \begin{bmatrix} \varepsilon_1 & 0 & 0 \\ 0 & \varepsilon_2 & 0 \\ 0 & 0 & \varepsilon_3 \end{bmatrix}$$

and perform a rotation θ about $\mathbf{X_2}$ (Fig. 8.3a), which is described by the transformation matrix:

$$a_{ij} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

The tensor transformation equation is:

$$\varepsilon'_{ij} = a_{ik} a_{jl} \varepsilon_{kl} \tag{8.18}$$

which results in the new form of the strain tensor:

$$\varepsilon'_{ij} = \begin{bmatrix} \varepsilon_1 \cos^2\theta + \varepsilon_3 \sin^2\theta & 0 & -(\varepsilon_1 - \varepsilon_3)\sin\theta\cos\theta \\ 0 & \varepsilon_2 & 0 \\ -(\varepsilon_1 - \varepsilon_3)\sin\theta\cos\theta & 0 & \varepsilon_1 \sin^2\theta + \varepsilon_3 \cos^2\theta \end{bmatrix} \tag{8.19}$$

Upon rearranging, we get:

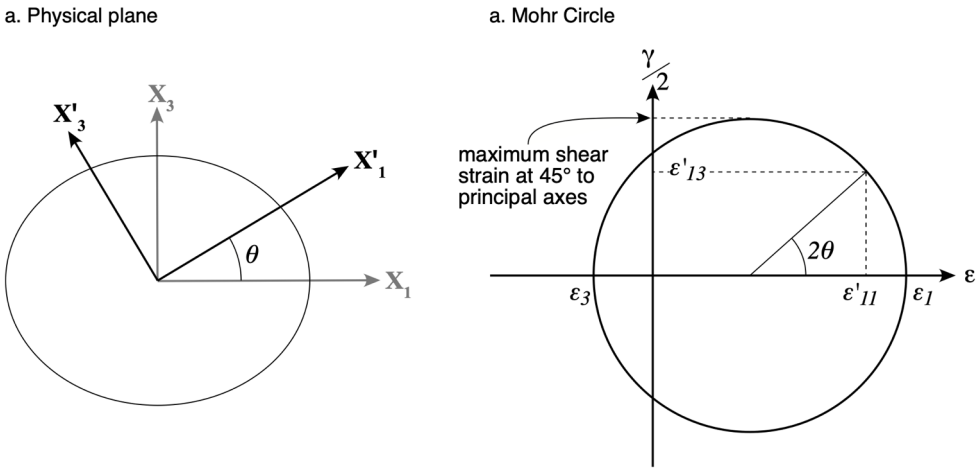


Figure 8.3: Rotation of the infinitesimal strain tensor about principal axis $\mathbf{X_2}$. **a.** Physical plane representation, **b.** Mohr Circle representation. Modified from Allmendinger et al. (2012).

$$\begin{aligned} \varepsilon'_{11} &= \frac{\varepsilon_1 + \varepsilon_3}{2} + \frac{\varepsilon_1 - \varepsilon_3}{2} \cos 2\theta \\ \varepsilon'_{33} &= \frac{\varepsilon_1 + \varepsilon_3}{2} - \frac{\varepsilon_1 - \varepsilon_3}{2} \cos 2\theta \\ \varepsilon'_{13} = \varepsilon'_{31} &= \frac{\gamma}{2} = -\frac{\varepsilon_1 - \varepsilon_3}{2} \sin 2\theta \end{aligned} \tag{8.20}$$

which are the equations of the Mohr Circle for infinitesimal strain (Fig. 8.3b). This Mohr Circle is not very useful, but perhaps the most important thing illustrated by it is that the two directions of maximum shear strain are at $\pm 45°$to the principal axes, ε_1 and ε_3 (Fig. 8.3b). Turning this around for the infinitesimal strain in a shear zone, the infinitesimal shortening and extension directions are always at 45°to the shear zone boundary. For example, gash fractures, foliation at the edge of a shear zone, and P and T axes of earthquakes (section 8.3.2) are all oriented 45°to the shear (fault) zone (Fig. 8.4).

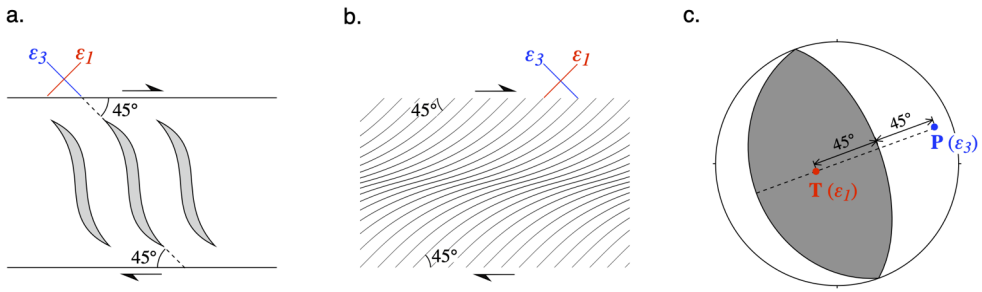


Figure 8.4: The infinitesimal extension and shortening directions are always at 45°to the shear zone boundaries. **a.** Sigmoidal veins, **b.** Foliation in shear zone, and **c.** P and T axes of earthquakes. Modified from Allmendinger et al. (2012).

8.3.2 Applications of Infinitesimal Strain

P and T axes

Our first application is based on the observation we just made regarding the orientation of the principal axes of infinitesimal strain with respect to a shear zone (Fig. 8.4). In a fault, the infinitesimal shortening **P** and extension **T** axes are at 45°to the slip vector (e.g. fault striae), and they lie on the plane defined by the fault striae and the fault pole. This plane is known as the movement plane M (Fig. 8.5).

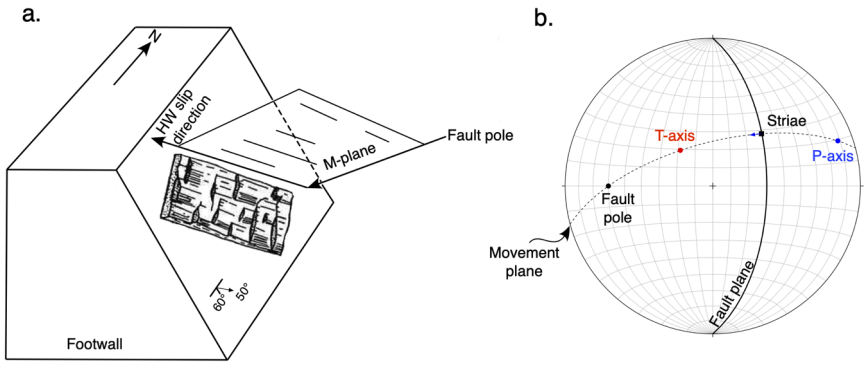


Figure 8.5: **a.** Reverse fault, striae and movement plane M. **b.** Representation of **a** on a lower hemisphere equal area stereonet. Modified from Marshak and Mitra (1988).

The displacement gradient tensor, e_{ij} of a population of n small faults with poles vectors, n_i, and slip vectors, u_i, is given by (Allmendinger et al., 2012):

$$e_{ij} = \frac{\sum_{i=1}^{n} (M_g u_i n_j)}{V} \tag{8.21}$$

where M_g is the geometric moment and V is the volume of the region being deformed. As discussed in section 8.3, we can decompose this equation to yield the infinitesimal strain and rotation tensors:

$$e_{ij} = \varepsilon_{ij} + \omega_{ij} = \frac{M_g (u_i n_j + u_j n_i)}{2V} + \frac{M_g (u_i n_j - u_j n_i)}{2V} \tag{8.22}$$

Because $M_g/2V$ is a scalar, the orientations of the principal axes of infinitesimal strain are identical to the principal axes of the symmetric tensor $(u_i n_j + u_j n_i)$, which are just a function of the fault planes and striae orientations. The **P** and **T** axes are therefore a simple, direct representation of fault geometry and the orientation and sense of slip (Allmendinger et al., 1989).

The function *PTAxes* computes and plots the **P** and **T** axes from the orientation of several faults and their slip vectors:

```
import numpy as np
import matplotlib.pyplot as plt

from CartToSph import CartToSph as CartToSph
from ZeroTwoPi import ZeroTwoPi as ZeroTwoPi
from SphToCart import SphToCart as SphToCart
from Stereonet import Stereonet as Stereonet
from GreatCircle import GreatCircle as GreatCircle
from StCoordLine import StCoordLine as StCoordLine

def PTAxes(fault,slip,sense, fpsv):
  '''
  PTAxes computes the P and T axes from the orientation
  of several fault planes and their slip vectors. Results
  are plotted in an equal area stereonet

  USE: P,T,senseC = PTAxes(fault,slip,sense)

  fault = nfaults x 2 vector with strikes and dips of
    faults
  slip = nfaults x 2 vector with trend and plunge of
```

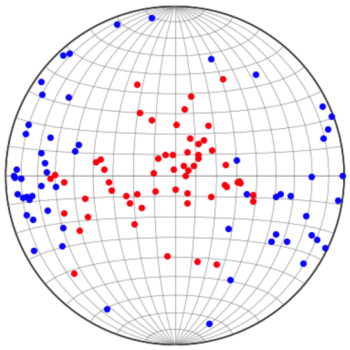
```
    slip vectors
  sense = nfaults x 1 vector with sense of faults
  fpsv = A flag to tell wether the fault plane and
    slip vector are plotted (1) or not
  P = nfaults x 2 vector with trend and plunge of P axes
  T = nfaults x 2 vector with trend and plunge of T axes
  senseC = nfaults x 1 vector with corrected sense of slip

  NOTE: Input/Output angles are in radians

  PTAxes uses functions SphToCart, ZeroTwoPi, CartToSph,
  Stereonet, GreatCircle and StCoordLine

  Python function based on the Matlab function
  PTAxes in Allmendinger et al. (2012)
  '''
  pi = np.pi
  # Initialize some vectors
  n = np.zeros(3)
  u = np.zeros(3)
  eps = np.zeros((3,3))
  P = np.zeros((np.size(fault,0),2))
  T = np.zeros((np.size(fault,0),2))
  senseC = sense

  # For all faults
  for i in range(np.size(fault,0)):
    # Direction cosines of pole to fault and slip vector
    n[0],n[1],n[2] = SphToCart(fault[i,0],fault[i,1],1)
    u[0],u[1],u[2] = SphToCart(slip[i,0],slip[i,1],0)
    # Compute u(i)*n(j) + u(j)*n(i)
    for j in range(3):
      for k in range(3):
        eps[j,k]=u[j]*n[k]+u[k]*n[j]
    # Compute orientations of principal axes of strain
    # Here we use the function eigh
    _,V = np.linalg.eigh(eps)
    # P orientation
    P[i,0],P[i,1] = CartToSph(V[0,2],V[1,2],V[2,2])
    if P[i,1] < 0:
      P[i,0] = ZeroTwoPi(P[i,0]+pi)
      P[i,1] = P[i,1]*-1.0
    # T orientation
    T[i,0],T[i,1] = CartToSph(V[0,0],V[1,0],V[2,0])
    if T[i,1] < 0.0:
      T[i,0] = ZeroTwoPi(T[i,0]+pi)
      T[i,1] = T[i,1]*-1.0
    # Determine 3rd component of pole cross product slip
    cross = n[0] * u[1] - n[1] * u[0]
```

```
    # Use cross and first character in sense to
    # determine if kinematic axes should be switched
    s2 = 'N'
    if sense[i][0] == 'T' or sense[i][0] == 't':
      s2 = 'Y'
    if (sense[i][0]=='R' or sense[i][0]=='r') and cross>0.0:
      s2 = 'Y'
    if (sense[i][0]=='L' or sense[i][0]=='l') and cross<0.0:
      s2 = 'Y'
    if s2 == 'Y':
      temp1 = P[i,0]
      temp2 = P[i,1]
      P[i,0] = T[i,0]
      P[i,1] = T[i,1]
      T[i,0] = temp1
      T[i,1] = temp2
      if cross < 0.0:
        senseC[i] = 'TL'
      if cross > 0.0:
        senseC[i] = 'TR'
    else:
      if cross < 0.0:
        senseC[i] = 'NR'
      if cross > 0.0:
        senseC[i] = 'NL'

  # Plot in equal area stereonet
  Stereonet(0,90*pi/180,10*pi/180,1)
  # Plot P and T axes
  for i in range(np.size(fault,0)):
    if fpsv == 1:
      # Plot fault
      path = GreatCircle(fault[i,0],fault[i,1],1)
      plt.plot(path[:,0],path[:,1],'k')
      # Plot slip vector (black circle)
      xp,yp = StCoordLine(slip[i,0],slip[i,1],1)
      plt.plot(xp,yp,'ko','MarkerFaceColor','k')
    # Plot P axis (blue circle)
    xp,yp = StCoordLine(P[i,0],P[i,1],1)
    plt.plot(xp,yp,'bo','MarkerFaceColor','b')
    # Plot T axis (red circle)
    xp,yp = StCoordLine(T[i,0],T[i,1],1)
    plt.plot(xp,yp,'ro','MarkerFaceColor','r')

  # Show plot
  plt.show()

  return P,T,senseC
```

Let's use this function to plot the **P** and **T** axes for a group of faults from the Central Andes in Northern Argentina. The file *jujuy.txt* contains the orientation (strike and dip) of the faults, and the orientation (trend and plunge) and sense of movement of the slip vectors (striae)[3]. The notebook ch8-1 shows how to do this:

```
# Import libraries
import numpy as np
pi = np.pi

# Import function PTAxes
import sys, os
sys.path.append(os.path.abspath('../functions'))
from PTAxes import PTAxes as PTAxes

# Read the faults
jujuy = np.loadtxt(os.path.abspath('../data/ch8-1/jujuy.txt')
    , usecols = [0,1,2,3])
fault = np.zeros((np.size(jujuy,0),2))
fault[:,0] = jujuy[:,0] * pi/180
fault[:,1] = jujuy[:,1] * pi/180
slip = np.zeros((np.size(jujuy,0),2))
slip[:,0] = jujuy[:,2] * pi/180
slip[:,1] = jujuy[:,3] * pi/180
sense = np.loadtxt(os.path.abspath('../data/ch8-1/jujuy.txt')
    , usecols = 4, dtype = 'str')

# Compute P and T axes and plot them
# Don't plot the faults and slip vectors
P,T,senseC= PTAxes(fault,slip,sense,0)
```

Output:



[3]This is the same file included in the program FaultKin by Richard Allmendinger

Here, the overall shortening direction given by the **P** axes (blue dots) is about E-W. From the **P** and **T** axes, it is possible to calculate a moment tensor summation and the kinematic axes, which define two nodal planes separating the regions of extension (**T** axes) and shortening (**P** axes). The regions of extension are typically shaded, and so the diagram looks like a beach ball (Fig. 8.4c). These "beach ball" diagrams are also used to display the focal mechanisms of earthquakes, where P-waves first arrivals pushing the ground up are marked as **T** axes, and those pushing the ground down are marked as **P** axes. We leave the moment tensor summation for the Exercises section.

2D strain from GPS data

The global positioning system (GPS) has revolutionized earth sciences in the last 30 years by providing geoscientists with real time monitoring of active deformation. Continuous GPS measurements provide mm resolution of the displacement of stations. Because the changes in distance between stations is very small (10s of mm) relative to the distance between stations (10s of km), the strains measured by GPS networks are infinitesimal. Figure 8.6 shows this problem in two dimensions:

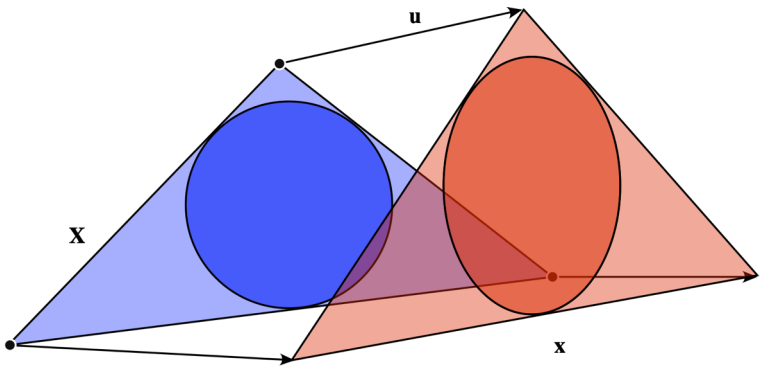


Figure 8.6: Three points in an initial configuration **X** move along non-parallel vectors **u** to a final configuration **x**, resulting in strain. Modified from Allmendinger et al. (2012).

If the strain is homogeneous, the displacement of the stations is expressed by the following equation:

$$u_i = t_i + e_{ij} X_j \tag{8.23}$$

where t_i is the translation vector and e_{ij} is the displacement gradient tensor. In matrix form, this equation can be written as:

$$\begin{bmatrix} {}^1u_1 \\ {}^1u_2 \\ {}^2u_1 \\ {}^2u_2 \\ \cdots \\ \cdots \\ {}^nu_1 \\ {}^nu_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & {}^1X_1 & {}^1X_2 & 0 & 0 \\ 0 & 1 & 0 & 0 & {}^1X_1 & {}^1X_2 \\ 1 & 0 & {}^2X_1 & {}^2X_2 & 0 & 0 \\ 0 & 1 & 0 & 0 & {}^2X_1 & {}^2X_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 0 & {}^nX_1 & {}^nX_2 & 0 & 0 \\ 0 & 1 & 0 & 0 & {}^nX_1 & {}^nX_2 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ e_{11} \\ e_{12} \\ e_{21} \\ e_{22} \end{bmatrix} \tag{8.24}$$

where the superscripts 1 to n refer to the stations. The column vector to the right of Eq. 8.24 contains the unknowns, which are the two components of the translation vector (t_1 and t_2) and the four components of the displacement gradient tensor (e_{11}, e_{12}, e_{21} and e_{22}). Therefore, in 2D there are 6 unknowns, and since each station delivers 2 equations, we need a minimum of 3 non-colinear stations to determine the strain ellipse[4].

Notice that Eq. 8.24 is written not just for 3 stations but for n stations. If there are more than 3 stations in 2D (or 4 stations in 3D), the system is overdetermined (more equations than unknowns), and we can use the extra information to assess the uncertainties of the model parameters.

The solution to Eq. 8.24 is a classical application of inverse theory, specifically the solution of the linear least-squares problem (Press et al., 1986). This problem has the form:

$$\mathbf{y} = \mathbf{M}\mathbf{x} \tag{8.25}$$

where $\mathbf{y}$ is the vector with known displacements, $\mathbf{M}$ is the matrix with the location of the stations (this matrix is known as the design matrix), and $\mathbf{x}$ is the vector with the unknowns. To solve for $\mathbf{x}$, $\mathbf{y}$ is multiplied by the inverse of $\mathbf{M}$:

$$\mathbf{x} = \mathbf{M}^{-1}\mathbf{y} \tag{8.26}$$

[4]In 3D there are 12 unknowns and each station delivers 3 equations. Therefore, we need a minimum of 4 non-colinear stations to determine the strain ellipsoid.

In Matlab or Python, there are routines specifically designed to solve this problem. We use the function *lscov* by Paul Müller to solve this problem.

There are three main strategies to compute the strain from a network of stations: i. To calculate the strain on triangular cells whose vertices are defined by the stations (Delaunay triangulation), ii. To calculate the strain on a regular grid of cells, using the stations within a radius, r, from the center of each cell (nearest neighbor method), or iii. To calculate the strain on a regular grid of cells, using all the stations weighted by their distance to the center of each cell (distance weighted method; Cardozo and Allmendinger; 2009). In this last case, the weighting factor is given by:

$$W = \exp\left[\frac{-d^2}{2\alpha^2}\right] \tag{8.27}$$

where d is the distance from the station to the center of the cell, and α is a constant that specifies how the impact of a station decays with distance. A larger value of α smooths out local variations.

The function *GridStrain* computes and plots the infinitesimal strain of a network of GPS stations[5]. Notice that after computing the displacement gradient for each cell, we use the function *InfStrain* to compute the strain:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from scipy.spatial import Delaunay

from lscov import lscov as lscov
from InfStrain import InfStrain as InfStrain

def GridStrain(pos,disp,k,par,plotpar,plotst):
  '''
  GridStrain computes the infinitesimal strain of a network
  of stations with displacements in x (east) and y (north).
  Strain in z is assumed to be zero (plane strain)

  USE: cent,eps,ome,pstrain,rotc = GridStrain(pos,disp,k,par,
   plotpar,plotst)
```

[5]This is a long but handy function. In the function we use the character \ to break long lines.

```
18
19    pos = nstations x 2 matrix with x (east) and y (north)
20      positions of stations in meters
21    disp = nstations x 2 matrix with x (east) and y (north)
22      displacements of stations in meters
23    k = Type of computation: Delaunay (k = 0), nearest
24      neighbor (k = 1), or distance weighted (k = 2)
25    par = Parameters for nearest neighbor or distance
26      weighted computation. If Delaunay (k = 0), enter
27      a scalar corresponding to the minimum internal
28      angle of a triangle valid for computation.
29      If nearest neighbor (k = 1), input a 1 x 3 vector
30      with grid spacing, number of nearest neighbors,
31      and maximum distance to neighbors. If distance
32      weighted (k = 2), input a 1 x 2 vector with grid
33      spacing and distance weighting factor alpha
34    plotpar = Parameter to color the cells: Max elongation
35      (plotpar = 0), minimum elongation
36      (plotpar = 1), rotation (plotpar = 2),
37      or dilatation (plotpar = 3)
38    plotst = A flag to plot the stations (1) or not (0)
39    cent = ncells x 2 matrix with x and y positions of cells
40      centroids
41    eps = 3 x 3 x ncells array  with strain tensors of
42      the cells
43    ome = 3 x 3 x ncells array with rotation tensors of
44      the cells
45    pstrain = 3 x 3 x ncells array with magnitude and
46      orientation of principal strains of
47      the cells
48    rotc = ncells x 3 matrix with rotation components
49      of cells
50
51    NOTE: Input/Output angles are in radians. Output
52      azimuths are given with respect to North
53      pos, disp, grid spacing, max. distance to
54      neighbors, and alpha should be in meters
55
56    GridStrain uses functions lscov and InfStrain
57
58    Python function translated from the Matlab function
59    GridStrain in Allmendinger et al. (2012)
60    '''
61    pi = np.pi
62    # If Delaunay
63    if k == 0:
64      # Indexes of triangles vertices
65      # Use function Delaunay
66      tri = Delaunay(pos)
```

```
67      inds = tri.simplices
68      # Number of cells
69      ncells = np.size(inds,0)
70      # Number of stations per cell = 3
71      nstat = 3
72      # Centers of cells
73      cent = np.zeros((ncells,2))
74      for i in range(ncells):
75        # Triangle vertices
76        v1x=pos[inds[i,0],0]
77        v2x=pos[inds[i,1],0]
78        v3x=pos[inds[i,2],0]
79        v1y=pos[inds[i,0],1]
80        v2y=pos[inds[i,1],1]
81        v3y=pos[inds[i,2],1]
82        # Center of cell
83        cent[i,0]=(v1x + v2x + v3x)/3.0
84        cent[i,1]=(v1y + v2y + v3y)/3.0
85        # Triangle internal angles
86        s1 = np.sqrt((v3x-v2x)**2 + (v3y-v2y)**2)
87        s2 = np.sqrt((v1x-v3x)**2 + (v1y-v3y)**2)
88        s3 = np.sqrt((v2x-v1x)**2 + (v2y-v1y)**2)
89        a1 = np.arccos((v2x-v1x)*(v3x-v1x)/(s3*s2)+\
90              (v2y-v1y)*(v3y-v1y)/(s3*s2))
91        a2 = np.arccos((v3x-v2x)*(v1x-v2x)/(s1*s3)+\
92              (v3y-v2y)*(v1y-v2y)/(s1*s3))
93        a3 = np.arccos((v2x-v3x)*(v1x-v3x)/(s1*s2)+\
94              (v2y-v3y)*(v1y-v3y)/(s1*s2))
95        # If any of the internal angles is less than
96        # specified minimum, invalidate triangle
97        if a1 < par or a2 < par or a3 < par:
98          inds[i,:] = np.zeros(3)
99    # If nearest neighbor or distance weighted
100   else:
101     # Construct grid
102     xmin = min(pos[:,0]); xmax = max(pos[:,0])
103     ymin = min(pos[:,1]); ymax = max(pos[:,1])
104     cellsx = int(np.ceil((xmax-xmin)/par[0]))
105     cellsy = int(np.ceil((ymax-ymin)/par[0]))
106     xgrid = np.arange(xmin,(xmin+(cellsx+1)*par[0]),par[0])
107     ygrid = np.arange(ymin,(ymin+(cellsy+1)*par[0]),par[0])
108     XX,YY = np.meshgrid(xgrid,ygrid)
109     # Number of cells
110     ncells = cellsx * cellsy
111     # Number of stations per cell (nstat) and
112     # other parameters
113     # If nearest neighbor
114     if k == 1:
115       nstat = par[1] # max neighbors
```

```
116       sqmd = par[2]**2 # max squared distance
117     # If distance weighted
118     elif k == 2:
119       nstat = np.size(pos,0) # all stations
120       dalpha = 2.0*par[1]*par[1] # 2*alpha*alpha
121     # Cells' centers
122     cent = np.zeros((ncells,2))
123     count = 0
124     for i in range(cellsy):
125       for j in range(cellsx):
126         cent[count,0] = (XX[i,j]+XX[i,j+1])/2.0
127         cent[count,1] = (YY[i,j]+YY[i+1,j])/2.0
128         count += 1
129     # Initialize stations indexes for cells to -1
130     inds = np.ones((ncells,nstat), dtype=int)*-1
131     # Initialize weight matrix for distance weighted
132     wv = np.zeros((ncells,nstat*2))
133     # For all cells set stations indexes
134     for i in range(ncells):
135       # Initialize sq distances to -1.0
136       sds = np.ones(nstat)*-1.0
137       # For all stations
138       for j in range(np.size(pos,0)):
139         # Sq distance from cell center to station
140         dx = cent[i,0] - pos[j,0]
141         dy = cent[i,1] - pos[j,1]
142         sd = dx**2 + dy**2
143         # If nearest neighbor
144         if k == 1:
145           # If within the max sq distance
146           if sd <= sqmd:
147             minsd = min(sds)
148             mini = np.argmin(sds)
149             # If less than max neighbors
150             if minsd == -1.0:
151               sds[mini] = sd
152               inds[i,mini] = j
153             # If max neighbors
154             else:
155               # If sq distance is less
156               # than neighbors max sq distance
157               maxsd = max(sds)
158               maxi = np.argmax(sds)
159               if sd < maxsd:
160                 sds[maxi] = sd
161                 inds[i,maxi] = j
162         # If distance weighted
163         elif k == 2:
164           # All stations indexes
```

```
            inds[i,:] = np.arange(nstat)
            # Eq. 8.27: Weight factor
            weight = np.exp(-sd/dalpha)
            wv[i,j*2] = weight
            wv[i,j*2+1] = weight

    # Initialize arrays
    y = np.zeros(nstat*2)
    M = np.zeros((nstat*2,6))
    e = np.zeros((3,3))
    eps = np.zeros((3,3,ncells))
    ome = np.zeros((3,3,ncells))
    pstrain = np.zeros((3,3,ncells))
    rotc = np.zeros((ncells,3))

    # For each cell
    for i in range(ncells):
      # If required minimum number of stations
      if min(inds[i,:]) >= 0:
        # Eq. 8.24: Displacements column vector y
        # and design matrix M. X1 = North, X2 = East
        for j in range(nstat):
          ic = inds[i,j]
          y[j*2] = disp[ic,1]
          y[j*2+1] = disp[ic,0]
          M[j*2,:] = [1.,0.,pos[ic,1],pos[ic,0],0.,0.]
          M[j*2+1,:] = [0.,1.,0.,0.,pos[ic,1],pos[ic,0]]
        # Eqs. 8.25-8.26: Find x using function lscov
        # If Delaunay or nearest neighbor
        if k == 0 or k == 1:
          x = lscov(M,y)
        # If distance weighted
        elif k == 2:
          x = lscov(M,y,wv[i,:])
        # Displacement gradient tensor
        for j in range(2):
          e[j,0] = x[j*2+2]
          e[j,1] = x[j*2+3]
        # Compute strain
        eps[:,:,i],ome[:,:,i],pstrain[:,:,i],\
          rotc[i,:],_ = InfStrain(e)

    # Variable to plot
    # If maximum principal strain
    if plotpar == 0:
      vp = pstrain[0,0,:]
      lcb = "emax"
    # If minimum principal strain
    elif plotpar == 1:
```

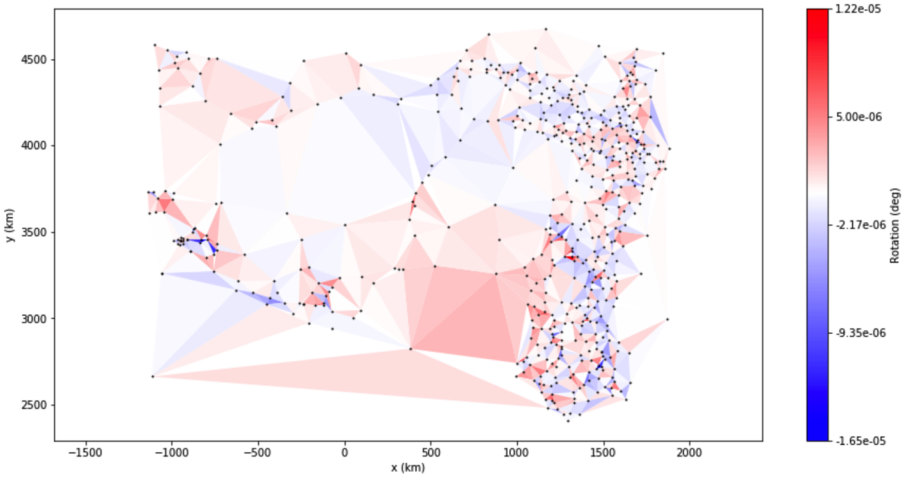
```
214     vp = pstrain[2,0,:]
215     lcb = "emin"
216   # If rotation:
217   # For plane strain, rotation = rotc(3)
218   elif plotpar == 2:
219     vp = rotc[:,2]*180/pi
220     lcb = "Rotation (deg)"
221   # If dilatation
222   elif plotpar == 3:
223     vp = pstrain[0,0,:]+pstrain[1,0,:]+pstrain[2,0,:]
224     lcb = "dilatation"
225 
226   # Make a figure
227   fig, ax = plt.subplots()
228   fig.set_size_inches(15.0, 7.5)
229 
230   # Patches and colors for cells
231   patches = []
232   colors = []
233 
234   # Fill cells patches and colors
235   # If Delaunay
236   if k == 0:
237     for i in range(ncells):
238       # If minimum number of stations
239         if min(inds[i,:]) >= 0:
240           xpyp = [[pos[inds[i,0],0],pos[inds[i,0],1]],\
241               [pos[inds[i,1],0],pos[inds[i,1],1]],\
242               [pos[inds[i,2],0],pos[inds[i,2],1]]]
243           # length in km
244           xpyp = np.divide(xpyp,1e3)
245           polygon = Polygon(xpyp, True)
246           patches.append(polygon)
247           colors.append(vp[i])
248   # If nearest neighbor or distance weighted
249   if k == 1 or k == 2:
250     count = 0
251     for i in range(cellsy):
252       for j in range(cellsx):
253         # If minimum number of stations
254         if min(inds[count,:]) >= 0:
255           xpyp = [[XX[i,j],YY[i,j]],[XX[i,j+1],YY[i,j+1]],\
256             [XX[i+1,j+1],YY[i+1,j+1]],[XX[i+1,j],YY[i+1,j]]]
257           # length in km
258           xpyp = np.divide(xpyp,1e3)
259           polygon = Polygon(xpyp, True)
260           patches.append(polygon)
261           colors.append(vp[count])
262         count += 1
```

```
263
264   # Collect cells patches
265   pcoll = PatchCollection(patches)
266   # Cells colors
267   pcoll.set_array(np.array(colors))
268   # Color map is blue to red
269   pcoll.set_cmap('bwr')
270   # Positive values are red, negative are
271   # blue and zero is white
272   vmin = min(vp)
273   vmax = max(vp)
274   norm=mcolors.TwoSlopeNorm(vmin=vmin,vcenter=0.0,vmax=vmax)
275   pcoll.set_norm(norm)
276
277   # Draw cells
278   ax.add_collection(pcoll)
279
280   # Plot stations
281   if plotst == 1:
282     plt.plot(pos[:,0]*1e-3,pos[:,1]*1e-3,'k.',markersize=2)
283
284   # Axes
285   plt.axis('equal')
286   plt.xlabel('x (km)')
287   plt.ylabel('y (km)')
288
289   # Color bar with nice ticks
290   intv = (vmax-vmin)*0.25
291   ticks=[vmin,vmin+intv,vmin+2*intv,vmin+3*intv,vmax]
292   lticks = ['{:.2e}'.format(ticks[0]),\
293       '{:.2e}'.format(ticks[1]),'{:.2e}'.format(ticks[2]),\
294       '{:.2e}'.format(ticks[3]),'{:.2e}'.format(ticks[4])]
295   cbar = fig.colorbar(pcoll, label=lcb, ticks=ticks)
296   cbar.ax.set_yticklabels(lticks)
297
298   # Show plot
299   plt.show()
300
301   return cent,eps,ome,pstrain,rotc
```

Let's use this function to compute the infinitesimal rotation in Tibet and the Himalaya. The file *tibet.txt* contains the UTM, east and west, coordinates of GPS stations in the Tibetan Plateau region and their displacements in meters (from Zhang et al., 2004). The notebook ch8-2 shows the solution to this problem using the Delaunay, nearest neighbor, and distance weighted methods.

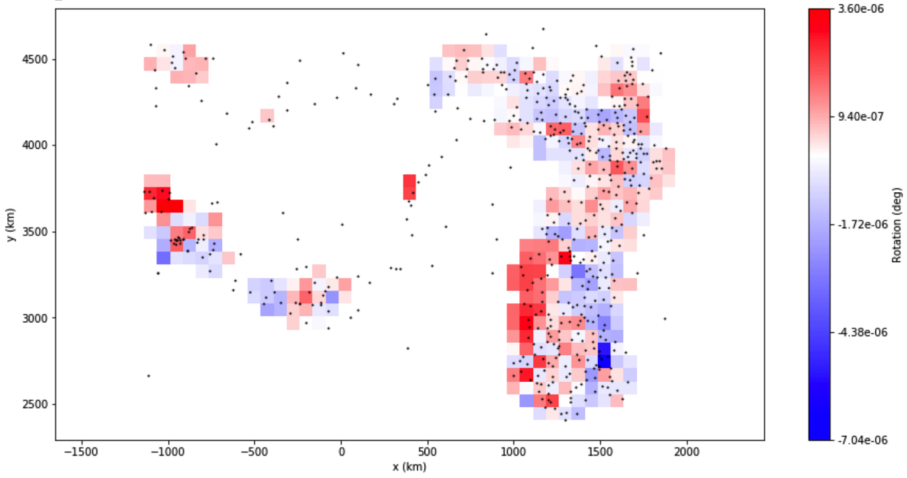
```
import numpy as np
pi = np.pi

# Import function GridStrain
import sys, os
sys.path.append(os.path.abspath('../functions'))
from GridStrain import GridStrain as GridStrain

# Load Zhang et al. GPS data from the Tibetan plateau
# load x, y coordinates and displacements
tibet =np.loadtxt(os.path.abspath('../data/ch8-2/tibet.txt'))
pos = np.zeros((np.size(tibet,0),2))
pos[:,0] = tibet[:,0]
pos[:,1] = tibet[:,1]
disp = np.zeros((np.size(tibet,0),2))
disp[:,0] = tibet[:,2]
disp[:,1] = tibet[:,3]

# Rotation from Delaunay triangulation, plot stations
par = 10 * pi/180 #Minimum internal angle of triangles
cent,eps,ome,pstrain,rotc = GridStrain(pos,disp,0,par,2,1)
```

Output:



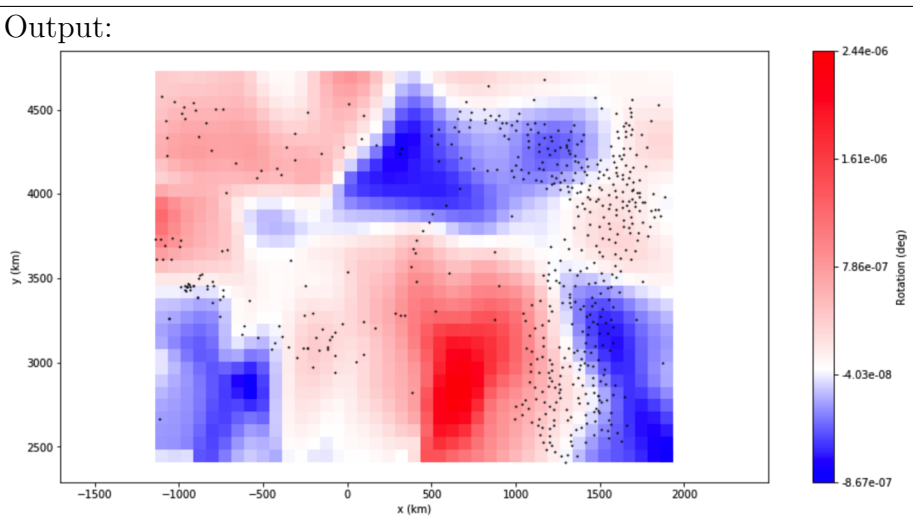

```
# Rotation from nearest neighbor
# Grid spacing = 75 km, neighbors 6,
# max. distance = 150 km, plot stations
par=[75e3,6,150e3]
cent,eps,ome,pstrain,rotc = GridStrain(pos,disp,1,par,2,1)
```

Output:




```
# Rotation from distance Weighted
# Grid spacing = 75 km, alpha = 150 km, plot stations
par=[75e3,150e3]
cent,eps,ome,pstrain,rotc = GridStrain(pos,disp,2,par,2,1)
```

Output:



Given the uneven distribution of the stations, the distance weighted method is perhaps the best representation of the infinitesimal strain. The rotation is about a downward vertical axis. The GPS-based rotation shows large coherent domains of clockwise (positive) and counterclockwise (negative) rotation. Interestingly enough, these domains are consistent with the permanent, long-term deformation of this region (Allmendinger et al., 2007).

8.4 Finite strain

When deformations are large, the initial and final states are not anymore identical:

$$dX_i \neq dx_i \quad \text{and} \quad \frac{\partial u_i}{\partial X_i} \neq \frac{\partial u_i}{\partial x_i} \tag{8.28}$$

and therefore, we need to use all the tensors in Table 8.1.

The mathematics of finite strain is quite lengthy (e.g. Means, 1976; Allmendinger et al., 2012) and we will not cover it in detail here. Rather, we will focus on the main tensors required to compute finite strain. We start with the finite strain tensor in the undeformed configuration:

$$E_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i} + \frac{\partial u_k}{\partial X_i}\frac{\partial u_k}{\partial X_j}\right] = \frac{1}{2}\left[e_{ij} + e_{ji} + e_{ki}e_{kj}\right] \tag{8.29}$$

where E_{ij} is known as the *Lagrangian finite strain tensor*. Similarly, we can compute the finite strain tensor in the deformed configuration:

$$\bar{E}_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}\right] = \frac{1}{2}\left[\bar{e}_{ij} + \bar{e}_{ji} - \bar{e}_{ki}\bar{e}_{kj}\right] \tag{8.30}$$

where $\bar{E}_{ij}$ is known as the *Eulerian finite strain tensor*[6]. Notice that Eqs. 8.29 and 8.30 are similar to Eq. 8.14 for the infinitesimal strain tensor, but with an extra term ($e_{ki}e_{kj}$ or $\bar{e}_{ki}\bar{e}_{kj}$).

From the deformation gradient $\mathbf{F}$, we can also compute deformation tensors. In the undeformed configuration:

$$C_{ij} = \frac{\partial x_k}{\partial X_i}\frac{\partial x_k}{\partial X_j} = F_{ki}F_{kj} \tag{8.31}$$

where C_{ij} is known as the *Green deformation tensor*, and in the deformed configuration:

[6] We use an upper bar to denote tensors in the deformed configuration, e.g. $\bar{e}_{ij}$ and $\bar{E}_{ij}$

$$\bar{C}_{ij} = \frac{\partial X_k}{\partial x_i}\frac{\partial X_k}{\partial x_j} = \bar{F}_{ki}\bar{F}_{kj} \tag{8.32}$$

where $\bar{C}_{ij}$ is known as the *Cauchy deformation tensor*.

The finite strain and deformation tensors are related. In the undeformed configuration:

$$E_{ij} = \frac{1}{2}\left(C_{ij} - \delta_{ij}\right) \quad \text{and} \quad C_{ij} = \delta_{ij} + 2E_{ij} \tag{8.33}$$

and in the deformed configuration:

$$\bar{E}_{ij} = \frac{1}{2}\left(\delta_{ij} - \bar{C}_{ij}\right) \quad \text{and} \quad \bar{C}_{ij} = \delta_{ij} - 2\bar{E}_{ij} \tag{8.34}$$

where δ_{ij} is the Kronecker delta, a function that returns 1 if $i = j$, or 0 otherwise.

Thus, the finite strain tensors do not contain any more information than the deformation tensors, and viceversa. They are all symmetric tensors that have principal axes and can be represented by Mohr Circles (see next section). From Eqs. 8.33 and 8.34, it is clear that E_{ij} and C_{ij}, and $\bar{E}_{ij}$ and $\bar{C}_{ij}$, have the same principal axes orientations. Also, these tensors are easily related to the quadratic elongation, λ_i, along the coordinate axes:

$$\lambda_i = C_{ii} = 1 + 2E_{ii} \tag{8.35}$$

and:

$$\frac{1}{\lambda_i} = \bar{C}_{ii} = 1 - 2\bar{E}_{ii} \tag{8.36}$$

Table 8.2 shows the finite strain and deformation tensors for the deformation in Fig. 8.2.

You can see that the diagonal components of the Green deformation tensor, C_{ij}, are the quadratic elongations, λ_i, along the coordinate axes, and the

Undeformed	$E_{ij} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -0.375 & 0 \\ 0 & 0 & 1.5 \end{bmatrix}$	$C_{ij} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0 & 4 \end{bmatrix}$
Deformed	$\bar{E}_{ij} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1.5 & 0 \\ 0 & 0 & 0.375 \end{bmatrix}$	$\bar{C}_{ij} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}$

Table 8.2: Finite strain and deformation tensors for the deformation in Fig. 8.2

diagonal components of the Cauchy deformation tensor, $\bar{C}_{ij}$, are the inverse of the quadratic elongations, $1/\lambda_i$. The diagonal components of the Lagrangian and Eulerian finite strain tensors, E_{ij} and $\bar{E}_{ij}$, can be found from Eqs. 8.35 and 8.36.

The function *FinStrain* computes the finite strain from the displacement gradient tensor, in the undeformed ($frame = 0$) or deformed ($frame = 1$) configuration. Notice that the function computes the magnitude and orientation of the maximum shear strain for the case of plane strain (Ramsay, 1967; his Eqs. 3.45 and 3.46).

```
import numpy as np

from CartToSph import CartToSph as CartToSph

def FinStrain(e,frame):
  '''
  FinStrain computes finite strain from an input
  displacement gradient tensor

  USE: eps,pstrain,dilat,maxsh = FinStrain(e,frame)

  e = 3 x 3 Lagrangian or Eulerian displacement gradient
    tensor
  frame = Reference frame. 0 = undeformed (Lagrangian)
    state, 1 = deformed (Eulerian) state
  eps = 3 x 3 Lagrangian or Eulerian strain tensor
  pstrain = 3 x 3 matrix with magnitude (column 1), trend
    (column 2) and plunge (column 3) of maximum
    (row 1), intermediate (row 2), and minimum
    (row 3) elongations
  dilat = dilatation
```

```
  maxsh = 1 x 2 vector with max. shear strain and
    orientation with respect to maximum principal
    strain direction. Only valid in 2D

  NOTE: Output angles are in radians

  FinStrain uses function CartToSph

  Python function translated from the Matlab function
  FinStrain in Allmendinger et al. (2012)
  '''
  # Initialize variables
  eps = np.zeros((3,3))
  pstrain = np.zeros((3,3))
  maxsh = np.zeros(2)

  # Compute strain tensor
  for i in range(3):
    for j in range(3):
      eps[i,j] = 0.5*(e[i][j]+e[j][i])
      for k in range(3):
        # If undeformed reference frame:
        # Lagrangian strain tensor, Eq. 8.29
        if frame == 0:
          eps[i,j] = eps[i,j] + 0.5*(e[k][i]*e[k][j])
        # If deformed reference frame:
        # Eulerian strain tensor, Eq. 8.30
        elif frame == 1:
          eps[i,j] = eps[i,j] - 0.5*(e[k][i]*e[k][j])

  # Compute principal elongations and orientations
  D, V = np.linalg.eigh(eps)

  # Principal elongations
  for i in range(3):
    ind = 2-i
    # Magnitude
    # If undeformed reference frame:
    # Lagrangian strain tensor, Eq. 8.33
    if frame == 0:
      pstrain[i,0] = np.sqrt(1.0+2.0*D[ind])-1.0
    # If deformed reference frame:
    # Eulerian strain tensor, Eq. 8.34
    elif frame == 1:
      pstrain[i,0] = np.sqrt(1.0/(1.0-2.0*D[ind]))-1.0
    # Orientations
    pstrain[i,1],pstrain[i,2] = CartToSph(V[0,ind],V[1,ind],V
    [2,ind])

```

```
70    # Dilatation
71    dilat = (1.0+pstrain[0,0])*(1.0+pstrain[1,0])*(1.0+pstrain
       [2,0]) - 1.0
72
73    # Maximum shear strain: This only works if plane strain
74    lmax = (1.0+pstrain[0,0])**2 # Maximum quad. elongation
75    lmin = (1.0+pstrain[2,0])**2 # Minimum quad. elongation
76    # Maximum shear strain: Ramsay (1967) Eq. 3.46
77    maxsh[0] = (lmax-lmin)/(2.0*np.sqrt(lmax*lmin))
78    # Angle of maximum shear strain with respect to maximum
79    # principal strain. Ramsay (1967) Eq. 3.45
80    # If undeformed reference frame
81    if frame == 0:
82      maxsh[1] = np.pi/4.0
83    # If deformed reference frame
84    elif frame == 1:
85      maxsh[1] = np.arctan(np.sqrt(lmin/lmax))
86
87    return eps, pstrain, dilat, maxsh
```

8.4.1 Mohr Circle for finite strain

As we said before, the finite strain and deformation tensors are symmetric tensors. Therefore, a rotation about one of the principal axis of the tensor can be represented by a Mohr Circle. As you may suspect, there are Mohr Circles for finite strain in the undeformed and deformed configuration (Ramsay, 1967). For geoscientists, however, the Mohr Circle for finite strain in the deformed configuration is much more important, since in nature we do observe deformed rocks.

To derive this Mohr Circle, we can start with the Cauchy deformation tensor in a principal axes coordinate system:

$$\bar{C}_{ij} = \begin{bmatrix} \bar{C}_1 & 0 & 0 \\ 0 & \bar{C}_2 & 0 \\ 0 & 0 & \bar{C}_3 \end{bmatrix}$$

and perform a rotation θ about $\mathbf{X_2}$, which is described by the transformation matrix:

$$a_{ij} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

The tensor transformation equation is:

$$\bar{C}'_{ij} = a_{ik} a_{jl} \bar{C}_{kl} \tag{8.37}$$

which results in the new form of the tensor:

$$\bar{C}'_{ij} = \begin{bmatrix} \bar{C}_1 \cos^2\theta + \bar{C}_3 \sin^2\theta & 0 & -(\bar{C}_1 - \bar{C}_3)\sin\theta\cos\theta \\ 0 & \bar{C}_2 & 0 \\ -(\bar{C}_1 - \bar{C}_3)\sin\theta\cos\theta & 0 & \bar{C}_1 \sin^2\theta + \bar{C}_3 \cos^2\theta \end{bmatrix} \tag{8.38}$$

Upon rearranging, we get:

$$\begin{aligned} \bar{C}'_{11} &= \frac{\bar{C}_1 + \bar{C}_3}{2} + \frac{\bar{C}_1 - \bar{C}_3}{2}\cos 2\theta \\ \bar{C}'_{33} &= \frac{\bar{C}_1 + \bar{C}_3}{2} - \frac{\bar{C}_1 - \bar{C}_3}{2}\cos 2\theta \\ \bar{C}'_{13} &= \bar{C}'_{31} = -\frac{\bar{C}_1 - \bar{C}_3}{2}\sin 2\theta \end{aligned} \tag{8.39}$$

As stated in Eq. 8.36, $\bar{C}_{ii} = 1/\lambda_i$. Also $\bar{C}_{ij}$ for $i \neq j$ is equal to γ/λ. Using $\lambda' = 1/\lambda$ and $\gamma' = \gamma/\lambda$, we get the equations for the Mohr Circle for finite strain in the deformed configuration:

$$\begin{aligned} \lambda' &= \frac{(\lambda'_1 + \lambda'_3)}{2} + \frac{(\lambda'_1 - \lambda'_3)}{2}\cos 2\theta \\ \gamma' &= -\frac{(\lambda'_1 - \lambda'_3)}{2}\sin 2\theta \end{aligned} \tag{8.40}$$

As an example, Figure 8.7a shows a deformed circle and an inscribed triangle after 30°clockwise shear. Figure 8.7b shows the Mohr Circle for this

deformation. In the Mohr Circle, the horizontal axis is λ', and the vertical axis is γ'. We follow the convention of Ragan (2009): Positive angular shear, ψ, corresponds to anticlockwise rotation of the original line's normal (Fig. 8.7a), and the γ' axis in the Mohr Circle increases downwards (Fig. 8.7b). Notice that the angle between the horizontal axis and a line from the origin to any point in the Mohr circle is equal to ψ. Therefore, a line from the origin and tangent to the Mohr Circle indicates $\psi_{\max}$ (Fig. 8.7b).

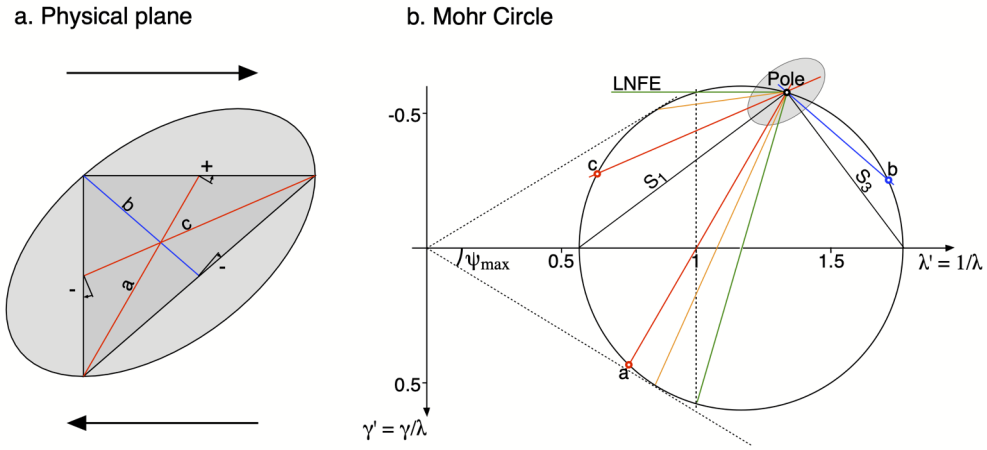


Figure 8.7: **a.** Physical plane, and **b.** Mohr Circle for 30°clockwise shear of a circle and a triangle. The pole, the triangle bisectors a, b and c (red), the principal strain axes S_1 and S_3, the LNFE (green), and the lines of maximum shear strain (orange) are drawn in the Mohr Circle.

From points in the Mohr Circle, one can trace the corresponding lines with the same orientation than in the physical plane (Fig. 8.7b, triangle bisectors a, b and c). These lines will intersect at a point called the *pole* to the Mohr Circle (Fig. 8.7b). From the pole, one can trace lines of any orientation; they will intersect the circle at points that represent the strain of lines with the same orientation in the physical plane. Thus, you can imagine the pole to be the center of the strain ellipse, and from it, it is easy to trace the principal axes of strain (S_1 and S_3), the lines of no finite elongation (LNFE, $\lambda' = 1$), and the lines of maximum shear strain (Fig. 8.7b).

8.4.2 2D finite strain from displacement data

If we have a group of points or stations with displacement data, we can determine the finite strain following a strategy similar to the one we used for infinitesimal strain. The function *GridFinStrain* computes and plots the finite strain for a group of points with displacement data. This function is very similar to our previous function *GridStrain*, and therefore we are only including its header here. Notice that after computing the displacement gradient in the undeformed ($frame = 0$) or deformed ($frame = 1$) configuration, we use our function *FinStrain* to compute the finite strain in each cell.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from scipy.spatial import Delaunay

from lscov import lscov as lscov
from FinStrain import FinStrain as FinStrain

def GridFinStrain(pos,disp,frame,k,par,plotpar,plotst):
  '''
  GridFinStrain computes the finite strain of a group
  of points with displacements in x and y.
  Strain in z is assumed to be zero (plane strain)

  USE: cent,eps,pstrain,dilat,maxsh = GridFinStrain(pos,disp,
   frame,k,par,plotpar,plotst)

  pos = npoints x 2 matrix with x and y position
    of points
  disp = nstations x 2 matrix with x and y
    displacements of points
  frame = Reference frame. 0 = undeformed (Lagrangian)
    state, 1 = deformed (Eulerian) state
  k = Type of computation: Delaunay (k = 0), nearest
    neighbor (k = 1), or distance weighted (k = 2)
  par = Parameters for nearest neighbor or distance
    weighted computation. If Delaunay (k = 0), enter
    a scalar corresponding to the minimum internal
    angle of a triangle valid for computation.
    If nearest neighbor (k = 1), input a 1 x 3 vector
    with grid spacing, number of nearest neighbors,
    and maximum distance to neighbors. If distance
    weighted (k = 2), input a 1 x 2 vector with grid
```

```
    spacing and distance weighting factor alpha
  plotpar = Parameter to color the cells: Max elongation
    (plotpar = 0), minimum elongation
    (plotpar = 1), dilatation (plotpar = 2),
    or max. shear strain (plotpar = 3)
  plotst = A flag to plot the stations (1) or not (0)
  cent = ncells x 2 matrix with x and y positions of the
    cells centroids
  eps = 3 x 3 x ncells array  with strain tensors of
    the cells
  pstrain = 3 x 3 x ncells array with magnitude and
    orientation of principal strains of the cells
  dilat = ncells x 1 vector with dilatation of the cells
  maxsh = ncells x 2 matrix with max. shear strain and
    orientation with respect to maximum principal
    strain direction, of the cells.
    Only valid for plane strain

  NOTE: Input/Output angles are in radians. Output
    azimuths are given with respect to y
    pos, disp, grid spacing, max. distance to
    neighbors, and alpha should be in the same
    length units

  GridFinStrain uses functions lscov and FinStrain
  '''
  # Please check the rest of the function in our git
  # repository
```

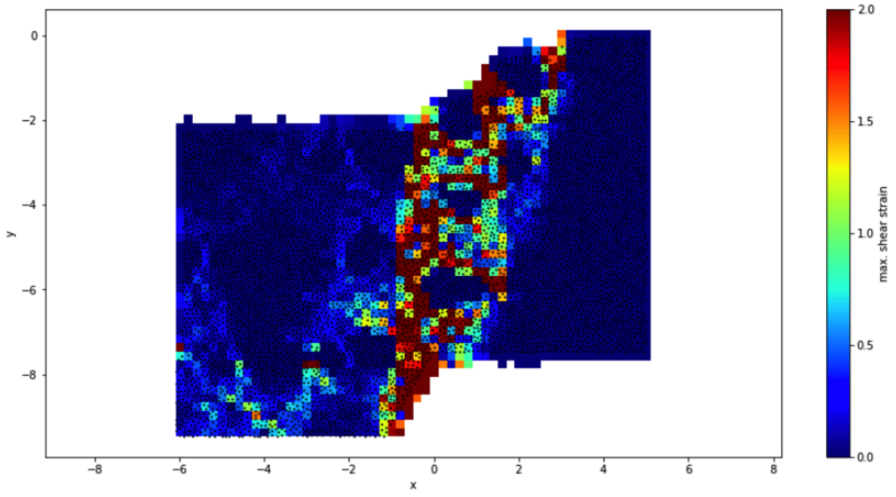
Let's use this function to compute the maximum shear strain of a discrete element model[7] of a normal fault. The file *demfault.txt* contains the deformed x and y coordinates of the elements and their displacements in meters. The notebook ch8-3 shows the solution to this problem using the nearest neighbor method:

```
import numpy as np

# Import function GridFinStrain
import sys, os
sys.path.append(os.path.abspath('../functions'))
from GridFinStrain import GridFinStrain as GridFinStrain

# load x, y DEFORMED coordinates and displacements
```

[7]The discrete element method is a mechanical method that simulates the rocks as an assembly of elements. For an example, check the program cdem by Cardozo and Hardy.

```
demfault = np.loadtxt(os.path.abspath('../data/ch8-3/demfault
    .txt'))
pos = np.zeros((np.size(demfault,0),2))
pos[:,0] = demfault[:,0]
pos[:,1] = demfault[:,1]
disp = np.zeros((np.size(demfault,0),2))
disp[:,0] = demfault[:,2]
disp[:,1] = demfault[:,3]

# Max. shear strain from nearest neighbor, Def. config.
# Grid spacing = 0.2 m, neighbors 6,
# max. distance = 0.4 m, plot stations
par = [0.2,6,0.4]
cent,eps,pstrain,dilat,maxsh = GridFinStrain(pos,disp,1,1,par
    ,3,1)
```

Output:



The maximum shear strain delineates the internal structure of the fault. You can find more information about computing strain from displacement or velocity data in Cardozo and Allmendinger (2009).

8.5 Progressive strain

In our discussion of finite strain, we focused on the undeformed (initial) and deformed (final) states. However, finite strain is actually the cumulative result of a series of strain increments:

$$
\begin{aligned}
{}^1x_i &= {}^1F_{ij}\,{}^1X_j \\
{}^2X_i &= {}^1x_i \\
{}^2x_i &= {}^2F_{ij}\,{}^2X_j \\
{}^3X_i &= {}^2x_i \\
{}^3x_i &= {}^3F_{ij}\,{}^3X_j \\
&\cdots \\
{}^nx_i &= {}^nF_{ij}\,{}^nX_j
\end{aligned}
\tag{8.41}
$$

where F_{ij} is the Green deformation gradient, and 1 to n are the strain increments. This can also be written as:

$$x_i = {}^nF_{ij} \ldots {}^3F_{ij}\,{}^2F_{ij}\,{}^1F_{ij}\,{}^1X_j \tag{8.42}$$

Notice that since ${}^2\mathbf{F}\,{}^1\mathbf{F} \neq {}^1\mathbf{F}\,{}^2\mathbf{F}$, for finite strain we must know the order of deformation. For example, if we want to determine the finite strain produced by a group of faults in a region, we must know the order at which these faults formed. This is often very difficult, if not impossible, to determine.

Let's look at some simple (yet complicated enough) deformations, which are characterized by the same incremental deformation gradient through time. For simplicity, we will assume that there is no strain along the $\mathbf{X_2}$ axis, and all strain is in the $\mathbf{X_1X_3}$ plane. Thus, we will be dealing with plane strain.

8.5.1 Pure shear

For pure shear, the principal stretches are along the coordinate axes (e.g. Fig. 8.2). Let's assume the maximum principal stretch, S_1 is along $\mathbf{X_1}$, and the minimum principal strecth, S_3 is along $\mathbf{X_3}$ (Fig. 8.8a). $S_2 = 1$ (plane strain) and is parallel to $\mathbf{X_2}$. The Green deformation gradient for this case is:

$${}^{PS}F_{ij} = \begin{bmatrix} S_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \tag{8.43}$$

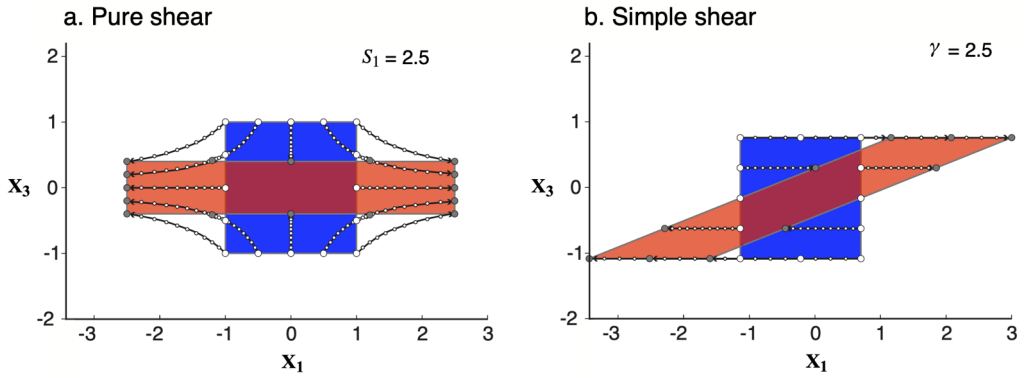


Figure 8.8: **a.** Pure shear, and **b.** Simple shear. Blue is undeformed, and red is deformed geometry. White and gray large circles are initial and final positions, respectively. The displacement paths of the points are divided in 10 increments. Modified from Allmendinger et al. (2012).

The function *PureShear* deforms a collection of points using pure shear, a value of S_1 (*st1*), and assuming plane strain and area conservation ($S_1 S_3 = 1$). The function displays the points' displacement paths for a number of increments (*ninc*), and the progressive strain history as the value of S_1 versus the angle S_1 makes with the $\mathbf{X_1}$ axis (Θ). Notice that we compute these last two parameters from the eigenvalues and eigenvectors of the Green deformation tensor:

```
import numpy as np
import matplotlib.pyplot as plt

def PureShear(pts,st1,ninc):
  '''
  PureShear computes and plots displacement paths and
  progressive finite strain history for pure shear with
  maximum stretching parallel to the X1 axis

  USE: paths,pfs = PureShear(pts,st1,ninc)

  pts: npoints x 2 matrix with X1 and X3 coord. of points
  st1 = Maximum principal stretch
  ninc = number of strain increments
  paths = displacement paths of points
  pfs = progressive finite strain history. column 1 =
    orientation of maximum stretch with respect to X1
    in degrees, column 2 = maximum stretch magnitude
```

```
  NOTE: Intermediate principal stretch is 1.0 (Plane
    strain). Output orientations are in radians

  Python function based on the Matlab function
  PureShear in Allmendinger et al. (2012)
  '''
  # Compute minimum principal stretch and incr. stretches
  st1inc=st1**(1.0/ninc)
  st3=1.0/st1
  st3inc=st3**(1.0/ninc)

  # Initialize displacement paths
  npts = np.size(pts,0) # Number of points
  paths = np.zeros((ninc+1,npts,2))
  paths[0,:,:] = pts # Initial points of paths

  # Calculate incremental deformation gradient tensor
  F = np.array([[st1inc, 0.0], [0.0, st3inc]])

  # Initialize figure
  fig = plt.figure(figsize=(15,5)) # Define the size
  ax1 = fig.add_subplot(1, 2, 1)
  ax2 = fig.add_subplot(1, 2, 2)

  # Compute displacement paths
  for i in range(npts): # for all points
    for j in range(ninc+1): # for all strain increments
      for k in range(2):
        for L in range(2):
          paths[j,i,k] = F[k,L]*paths[j-1,i,L] + paths[j,i,k]
    # Plot displacement path of point
    xx = paths[:,i,0]
    yy = paths[:,i,1]
    ax1.plot(xx,yy,'k.-')

  # Plot initial and final polygons
  inpol = np.zeros((npts+1,2))
  inpol[0:npts,]=paths[0,0:npts,:]
  inpol[npts,] = inpol[0,]
  ax1.plot(inpol[:,0],inpol[:,1],'b-')
  finpol = np.zeros((npts+1,2))
  finpol[0:npts,]=paths[ninc,0:npts,:]
  finpol[npts,] = finpol[0,]
  ax1.plot(finpol[:,0],finpol[:,1],'r-')

  # Release plot and set axes
  ax1.set_xlabel('X1')
  ax1.set_ylabel('X3')
  ax1.grid()
```

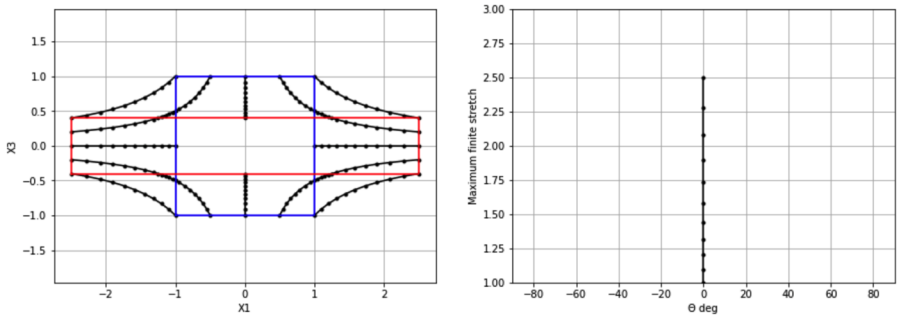
```
  ax1.axis('equal')

  # Initalize progressive finite strain history
  pfs = np.zeros((ninc+1,2))
  pfs[0,:] = [0, 1] #Initial state

  # Calculate progressive finite strain history
  for i in range(1,ninc+1):
    # Determine the finite deformation gradient tensor
    finF = np.linalg.matrix_power(F, i)
    # Determine Green deformation tensor
    G = np.dot(finF,finF.conj().transpose())
    # Stretch magnitude and orientation: Maximum
    # eigenvalue and their corresponding eigenvectors
    # of Green deformation tensor
    D, V = np.linalg.eigh(G)
    pfs[i,0] = np.arctan(V[1,1]/V[0,1])
    pfs[i,1] = np.sqrt(D[1])

  # Plot progressive finite strain history
  ax2.plot(pfs[:,0]*180/np.pi,pfs[:,1],'k.-')
  ax2.set_xlabel('Θ deg')
  ax2.set_ylabel('Maximum finite stretch')
  ax2.set_xlim(-90,90)
  ax2.set_ylim(1,max(pfs[:,1])+0.5)
  ax2.grid()

  # show plot
  plt.show()

  return paths,pfs
```

Let's use this function to reproduce the deformation shown in Fig. 8.8a. The notebook ch8-4 shows how to do this:

```
# Import libraries
import numpy as np

# Import function PureShear
import sys, os
sys.path.append(os.path.abspath('../functions'))
from PureShear import PureShear as PureShear

# Initial points coordinates
pts = np.zeros((16,2))
pts[:,0]=[-1,-1,-1,-1,-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5]
```

```
pts[:,1]=[-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5,-1,-1,-1,-1]
st1 = 2.5
ninc = 10

paths,psf = PureShear(pts,st1,ninc)
```

Output:



As you can see the finite strain axes, (S_1 and S_3), do not rotate throughout the deformation (Θ is 0 throughout the deformation). Pure shear is an *irrotational* deformation.

8.5.2 Simple shear

For plane strain and simple shear along the $\mathbf{X_1}$ axis (Fig. 8.8b), the Green deformation gradient is given by:

$$ {}^{SS}F_{ij} = \begin{bmatrix} 1 & 0 & \gamma \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.44} $$

where γ is the shear strain.

The function *SimpleShear* deforms a collection of points using simple shear and a value of shear strain (*gamma*). The function displays the points' displacement paths for a number of increments (*ninc*), and the progressive strain history as the value of S_1 versus Θ:

```
import numpy as np
import matplotlib.pyplot as plt
```

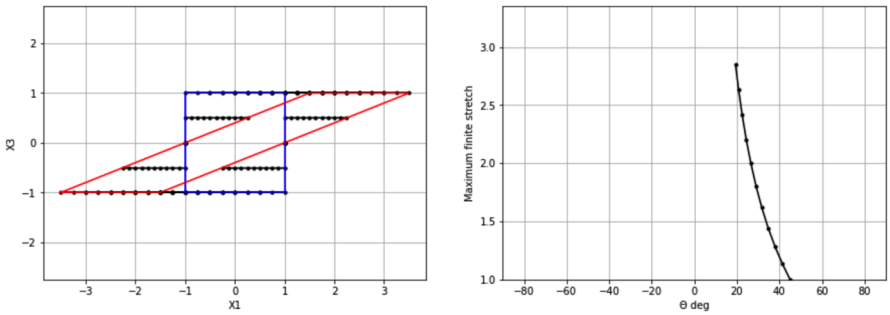
```
3
4  def SimpleShear(pts,gamma,ninc):
5    '''
6    SimpleShear computes and plots displacement paths and
7    progressive finite strain history for simple shear
8    parallel to the X1 axis
9
10   USE: paths,pfs = SimpleShear(pts,gamma,ninc)
11
12   pts: npoints x 2 matrix with X1 and X3 coord. of points
13   gamma = Engineering shear strain
14   ninc = number of strain increments
15   paths = displacement paths of points
16   pfs = progressive finite strain history. column 1 =
17     orientation of maximum stretch with respect to X1
18     in degrees, column 2 = maximum stretch magnitude
19
20   NOTE: Intermediate principal stretch is 1.0 (Plane
21     strain). Output orientations are in radians
22
23   Python function based on the Matlab function
24   SimpleShear in Allmendinger et al. (2012)
25   '''
26   # Incremental engineering shear strain
27   gammainc = gamma/ninc
28
29   # Initialize displacement paths
30   npts = np.size(pts,0) # Number of points
31   paths = np.zeros((ninc+1,npts,2))
32   paths[0,:,:] = pts # Initial points of paths
33
34   # Calculate incremental deformation gradient tensor
35   F = np.array([[1.0, gammainc],[0.0, 1.0]])
36
37   # Initialize figure
38   fig = plt.figure(figsize=(15,5)) # Define the size
39   ax1 = fig.add_subplot(1, 2, 1)
40   ax2 = fig.add_subplot(1, 2, 2)
41
42   # Compute displacement paths
43   for i in range(npts): # for all points
44     for j in range(ninc+1): # for all strain increments
45       for k in range(2):
46         for L in range(2):
47           paths[j,i,k] = F[k,L]*paths[j-1,i,L] + paths[j,i,k]
48     # Plot displacement path of point.
49     xx = paths[:,i,0]
50     yy = paths[:,i,1]
51     plt. figure(1)
```

```
    ax1.plot(xx,yy,'k.-')

  # Plot initial and final polygons
  inpol = np.zeros((npts+1,2))
  inpol[0:npts,]=paths[0,0:npts,:]
  inpol[npts,] = inpol[0,]
  ax1.plot(inpol[:,0],inpol[:,1],'b-')
  finpol = np.zeros((npts+1,2))
  finpol[0:npts,]=paths[ninc,0:npts,:]
  finpol[npts,] = finpol[0,]
  ax1.plot(finpol[:,0],finpol[:,1],'r-')

  # Release plot and set axes
  ax1.set_xlabel('X1')
  ax1.set_ylabel('X3')
  ax1.grid()
  ax1.axis('equal')

  # Initalize progressive finite strain history
  pfs = np.zeros((ninc+1,2))
  # In. state: Max. extension is at 45 deg from shear zone
  pfs[0,:] = [np.pi/4.0, 1.0]

  # Calculate progressive finite strain history
  for i in range(1,ninc+1):
    # Determine the finite deformation gradient tensor
    finF = np.linalg.matrix_power(F, i)
    # Determine Green deformation tensor
    G = np.dot(finF,finF.conj().transpose())
    # Stretch magnitude and orientation: Maximum
    # eigenvalue and their corresponding eigenvectors
    # of Green deformation tensor
    D, V = np.linalg.eigh(G)
    pfs[i,0] = np.arctan(V[1,1]/V[0,1])
    pfs[i,1] = np.sqrt(D[1])

  # Plot progressive finite strain history
  ax2.plot(pfs[:,0]*180/np.pi,pfs[:,1],'k.-')
  ax2.set_xlabel('Θ deg')
  ax2.set_ylabel('Maximum finite stretch')
  ax2.set_xlim(-90,90)
  ax2.set_ylim(1,max(pfs[:,1])+0.5)
  ax2.grid()

  # Show plot
  plt.show()

  return paths,pfs
```

Let's use this function to reproduce the deformation shown in Fig. 8.8b. The notebook ch8-5 shows how to do this:

```
# Import libraries
import numpy as np

# Import function SimpleShear
import sys, os
sys.path.append(os.path.abspath('../functions'))
from SimpleShear import SimpleShear as SimpleShear

# Initial points coordinates
pts = np.zeros((16,2))
pts[:,0]=[-1,-1,-1,-1,-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5]
pts[:,1]=[-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5,-1,-1,-1,-1]
gamma = 2.5
ninc = 10
paths,psf = SimpleShear(pts,gamma,ninc)
```

Output:



In this case, the finite strain axes, (S_1 and S_3), rotate throughout the deformation. Θ is initially 45°(as predicted by the theory of infinitesimal strain, Fig. 8.4), and it progressively decreases to a value of 20°by the end of the deformation. Simple shear is a *rotational* deformation.

8.5.3 General shear

Sub-simple shear (De Paor, 1983), or general shear, is a combination of pure shear and simple shear. If the shear direction and S_1 are parallel to $\mathbf{X_1}$, $S_2 = 1$ (plane strain) and area is constant, the Green deformation gradient for general shear is (Allmendinger et al., 2012):

$$ {}^{GS}F_{ij} = \begin{bmatrix} S_1 & 0 & \frac{\gamma(S_1-S_3)}{2\ln S_1} \\ 0 & 1 & 0 \\ 0 & 0 & S_3 \end{bmatrix} \tag{8.45} $$

If on the other hand the shear direction is parallel to $\mathbf{X_1}$, but S_1 is parallel to $\mathbf{X_3}$, the Green deformation gradient is (Allmendinger et al., 2012):

$$ {}^{GS}F_{ij} = \begin{bmatrix} S_3 & 0 & \frac{\gamma(S_3-S_1)}{2\ln S_3} \\ 0 & 1 & 0 \\ 0 & 0 & S_1 \end{bmatrix} \tag{8.46} $$

A simple dimensionless measure of the ratio of simple to pure shear is the cosine of the acute angle between the eigenvectors of the Green deformation gradient $\mathbf{F}$. This measure is known as the *kinematic vorticity number*, W_k (Truesdell, 1953). General shear is characterized by a W_k between 0 (pure shear) and 1 (simple shear).

The function *GeneralShear* deforms a collection of points using general shear and values of S_1 (*st1*) and γ (*gamma*), for a shear direction parallel to $\mathbf{X_1}$, and S_1 parallel ($kk = 0$) or perpendicular to the shear direction ($kk = 1$). Similar to the two previous functions, *GeneralShear* displays the points' displacement paths for a number of increments (*ninc*), and S_1 versus Θ:

```
import numpy as np
import matplotlib.pyplot as plt

def GeneralShear(pts,st1,gamma,kk,ninc):
  '''
  GeneralShear computes displacement paths, kinematic
  vorticity numbers and progressive finite strain
  history, for general shear with a pure shear stretch,
  no area change, and a single shear strain

  USE: paths,wk,pfs = GeneralShear(pts,st1,gamma,kk,ninc)

  pts = npoints x 2 matrix with X1 and X3 coord. of points
  st1 = Pure shear stretch parallel to shear zone
  gamma = Engineering shear strain
  kk = An integer that indicates whether the maximum
    finite stretch is parallel (kk = 0), or
    perpendicular (kk=1) to the shear direction
  ninc = number of strain increments
```

```
20    paths = displacement paths of points
21    wk = Kinematic vorticity number
22    pfs = progressive finite strain history. column 1 =
23      orientation of maximum stretch with respect to
24      X1, column 2 = maximum stretch magnitude
25
26    NOTE: Intermediate principal stretch is 1.0 (Plane
27      strain). Output orientations are in radians
28
29    Python function translated from the Matlab function
30    GeneralShear in Allmendinger et al. (2012)
31    '''
32    # Compute minimum principal stretch and incr. stretches
33    st1inc =st1**(1.0/ninc)
34    st3 =1.0/st1
35    st3inc =st3**(1.0/ninc)
36
37    # Incremental engineering shear strain
38    gammainc = gamma/ninc
39
40    # Initialize displacement paths
41    npts = np.size(pts,0) # Number of points
42    paths = np.zeros((ninc+1,npts,2))
43    paths[0,:,:] = pts # Initial points of paths
44
45    # Initialize figure
46    fig = plt.figure(figsize=(15,5)) # Define the size
47    ax1 = fig.add_subplot(1, 2, 1)
48    ax2 = fig.add_subplot(1, 2, 2)
49
50    # Calculate incremental deformation gradient tensor
51    # If max. finite stretch parallel to shear direction
52    if kk == 0:
53      F=np.zeros((2,2))
54      F[0,]=[st1inc, (gammainc*(st1inc-st3inc))/(2.0*np.log(
       st1inc))]
55      F[1,]=[0.0, st3inc]
56    # If max. finite stretch perpendicular to shear direction
57    elif kk == 1:
58      F=np.zeros((2,2))
59      F[0,]= [st3inc, (gammainc*(st3inc-st1inc))/(2.0*np.log(
       st3inc))]
60      F[1,]= [0.0, st1inc]
61
62    # Compute displacement paths
63    for i in range(npts): # for all points
64      for j in range(ninc+1): # for all strain increments
65        for k in range(2):
66          for L in range(2):
```

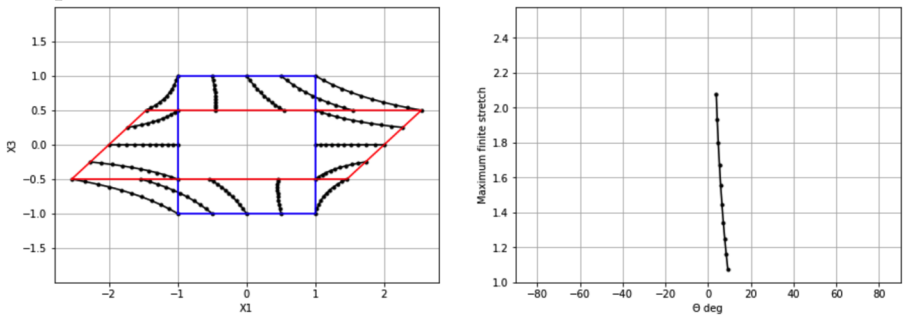
```
67            paths[j,i,k] = F[k,L]*paths[j-1,i,L] + paths[j,i,k]
68      # Plot displacement path of point
69      xx = paths[:,i,0]
70      yy = paths[:,i,1]
71      ax1.plot(xx,yy,'k.-')
72
73    # Plot initial and final polygons
74    inpol = np.zeros((npts+1,2))
75    inpol[0:npts,]=paths[0,0:npts,:]
76    inpol[npts,] = inpol[0,]
77    ax1.plot(inpol[:,0],inpol[:,1],'b-')
78    finpol = np.zeros((npts+1,2))
79    finpol[0:npts,]=paths[ninc,0:npts,:]
80    finpol[npts,] = finpol[0,]
81    ax1.plot(finpol[:,0],finpol[:,1],'r-')
82
83    # Set axes
84    ax1.set_xlabel('X1')
85    ax1.set_ylabel('X3')
86    ax1.grid()
87    ax1.axis('equal')
88
89    # Determine the eigenvectors of the flow (apophyses)
90    # Since F is not symmetrical, use function eig
91    _,V = np.linalg.eig(F)
92    theta2 = np.arctan(V[1,1]/V[0,1])
93    wk = np.cos(theta2)
94
95    # Initalize progressive finite strain history.
96    # We are not including the initial state
97    pfs = np.zeros((ninc,ninc))
98
99    # Calculate progressive finite strain history
100   for i in range(1,ninc+1):
101     # Determine the finite deformation gradient tensor
102     finF = np.linalg.matrix_power(F, i)
103     # Determine Green deformation tensor
104     G = np.dot(finF,finF.conj().transpose())
105     # Stretch magnitude and orientation: Maximum
106     # eigenvalue and their corresponding eigenvectors
107     # of Green deformation tensor
108     D, V = np.linalg.eigh(G)
109     pfs[i-1,0] = np.arctan(V[1,1]/V[0,1])
110     pfs[i-1,1] = np.sqrt(D[1])
111
112   # Plot progressive finite strain history
113   ax2.plot(pfs[:,0]*180/np.pi,pfs[:,1],'k.-')
114   ax2.set_xlabel('Θ deg')
115   ax2.set_ylabel('Maximum finite stretch')
```

```
  ax2.set_xlim(-90,90)
  ax2.set_ylim(1,max(pfs[:,1])+0.5)
  ax2.grid()

  # Show plot
  plt.show()

  return paths,wk,pfs
```

Let's use this function to simulate the deformation produced by $S_1 = 2.0$ and $\gamma = 0.5$, for a shear direction parallel to $\mathbf{X_1}$, and S_1 parallel or perpendicular to it. The notebook ch8-6 shows the solution to this problem:

```
# Import libraries
import numpy as np

# Import function GeneralShear
import sys, os
sys.path.append(os.path.abspath('../functions'))
from GeneralShear import GeneralShear as GeneralShear

# Initial points coordinates
pts = np.zeros((16,2))
pts[:,0]=[-1,-1,-1,-1,-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5]
pts[:,1]=[-1,-0.5,0,0.5,1,1,1,1,1,0.5,0,-0.5,-1,-1,-1,-1]
st1 = 2.0
gamma = 0.5
ninc = 10

# Max. finite stretch parallel to shear direction
kk = 0
paths1,wk1,psf1= GeneralShear(pts,st1,gamma,kk,ninc)
print('Wk = {:.4f}'.format(wk1))
```

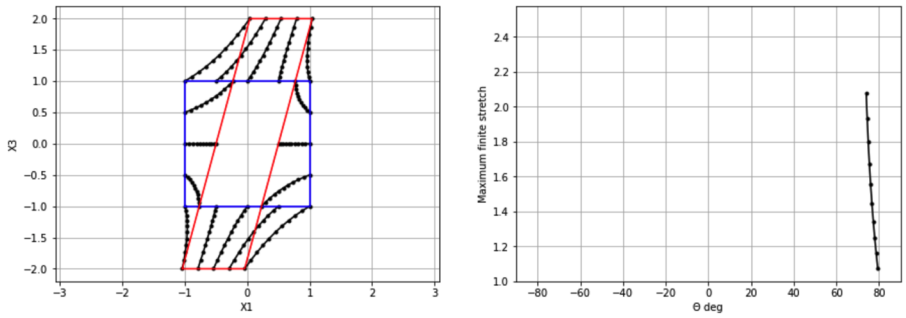
Output:



Wk = 0.3393

```
# Max. finite stretch perpendicular to shear direction
kk = 0
paths2,wk2,pfs2= GeneralShear(pts,st1,gamma,kk,ninc)
print('Wk = {:.4f}'.format(wk2))
```

Output:



Wk = 0.3393

In this case, the contribution of simple shear is about one third that of pure shear ($W_k = 0.34$). Try running the notebook with different values of S_1 and γ, and check W_k and Θ throughout the deformation.

8.6 Exercises

1. This exercise is from Rowland and Duebendorfer (1994). The file *basinAndRange.txt* contains the strike and dip (RHR), slip direction, and slip sense of faults within the Basin and Range province, specifically southern Nevada and the Lake Mead area. Plot the **P** and **T** axes for these data using the function *PTAxes*. Are these data consistent with the regional E-W extension direction in the Basin and Range province?

2. From the **P** and **T** axes, one can compute an unweighted moment tensor summation as follows (Allmendinger et al., 1989):

$$\mathbf{K}=\begin{bmatrix} \sum(P_N)^2-(T_N)^2 & \sum(P_N)(P_E)-(T_N)(T_E) & \sum(P_N)(P_D)-(T_N)(T_D) \\ \sum(P_E)(P_N)-(T_E)(T_N) & \sum(P_E)^2-(T_E)^2 & \sum(P_E)(P_D)-(T_E)(T_D) \\ \sum(P_D)(P_N)-(T_D)(T_N) & \sum(P_D)(P_E)-(T_D)(T_E) & \sum(P_D)^2-(T_D)^2 \end{bmatrix}$$

where the subscripts N, E, and D refer to the east, north and down direction cosines of the **P** and **T** axes. The eigenvalues and eigenvectors

of **K** give the relative magnitudes and orientations of the kinematic axes. These kinematic axes define the orientation of two nodal planes, or possible faults, separating the areas of infinitesimal extension (**T** axes) from those of infinitesimal shortening (**P** axes). The faults intersect at the intermediate eigenvector, the minimum and maximum eigenvectors define the movement plane, and the fault slip vectors are on the movement plane at 45°from the minimum and maximum eigenvectors. Modify function *PTAxes* to plot the nodal planes. This is not a simple problem, don't give up.

3. The file *andes.txt* contains GPS data (stations UTM coordinates and displacements in meters) from the Central Andes (Kendrick et al., 2001; Brooks et al., 2003). Compute the infinitesimal rotation in this region using the function *GridStrain*. What does the rotation tell you about the tectonics of this region? Check Allmendinger et al. (2005) to support your answer.

4. The file *antithetic.txt* contains data from a discrete element model of a low angle normal fault (deformed elements' coordinates and displacements). Use the function *GridFinStrain* to compute the shear strain of this model.

5. The file *trilobite.txt* contains the undeformed x and y coordinates of points delineating a trilobite specimen. Deform the trilobite using **a.** pure shear with $S_1 = 2.0$, **b.** simple shear with $\gamma = 1.0$, and **c.** general shear with $S_1 = 1.5$, $\gamma = 1.0$, and shear direction and S_1 parallel to x. *Hint:* Use functions *PureShear*, *SimpleShear*, and *GeneralShear*.

References

Allmendinger, R.W., Gephart, J.W. and Marrett, R.A. 1989. Notes on fault slip analysis. GSA short course.

Allmendinger, R.W., Smalley, R.J., Bevis, M. et al. 2005. Bending the Bolivian orocline in real time. Geology 33, 905-908.

Allmendinger, R.W., Reilinger, R. and Loveless, J. 2007. Strain and rotation rate from GPS in Tibet, Anatolia, and the Altiplano. Tectonics 26, TC3013.

Allmendinger, R.W., Cardozo, N. and Fisher, D.M. 2012. Structural Geology Algorithms: Vectors and Tensors. Cambridge University Press.

Brooks, B. A., Bevis, M., R. Smalley, J. et al. 2003. Crustal Motion in the Southern Andes (26-36°S): do the Andes behave like a microplate? Geochemistry, Geophysics, Geosystems - G3 4, GC000505.

Cardozo, N. and Allmendinger, R.W. 2009. SSPX: A program to compute strain from displacement/velocity data. Computers and Geosciences 35, 1343-1357.

DePaor, D.G. 1983. Ortographic analysis of geological structures - I. Deformation theory. Journal of Structural Geology 5, 255-277.

Kendrick, E., Bevis, M., Smalley, R. and Brooks, B. 2001. An integrated crustal velocity field for the Central Andes. Geochemistry, Geophysics, Geosystems - G3 2, GC000191.

Marshak, S. and Mitra, G. 1988. Basic Methods of Structural Geology. Prentice Hall.

Means, W.D. 1976. Stress and Strain: Basic Concepts of Continuum Mechanics for Geologists. New York: Springer-Verlag.

Press, W.H., Flannery, B.P., Teoukolsky, S.A. and Vetterling, W.T. 1986. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press.

Ragan, D.M. 2009. Structural Geology: An Introduction to Geometrical Techniques. Cambridge University Press.

Ramsay, J.G. 1967. Folding and Fracturing of Rocks. McGraw-Hill, New York.

Rowland, S.M. and Duebendorfer, E.M. 1994. Structural Analysis and Synthesis: A Laboratory Course in Structural Geology. Wiley.

Truesdell, C. 1953. Two measures of vorticity. Journal of Rational Mechanics and Analysis 2, 173-217.

Zhang, P., Zhengkang, S., Min, W., et al. 2004, Continuous deformation of the Tibetan Plateau from Global Positioning System data: Geology 32, 809-812.