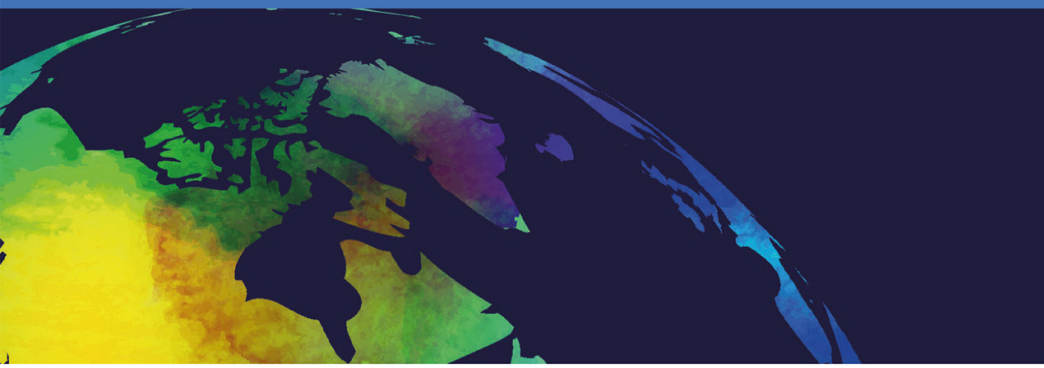
CRC FOCUS



Python for Islamic Astronomy

Modern Computational Approaches to Hijri Calendar, Qibla, and Prayer Times

MUHAMAD SYAZWAN FAID,
MUHAMMAD SYAOQI NAHWANDI,
SHAHRIN AHMAD, AND
MOHD SAIFUL ANWAR MOHD NAWAWI

CRC Press
Taylor & Francis Group

Python for Islamic Astronomy

Python for Islamic Astronomy: Modern Computational Approaches to Hijri Calendar, Qibla, and Prayer Times responds to the urgent need to improve calculation accuracy and data visualizations in the field of Islamic Astronomy. This field is becoming increasingly complex, leading to mistakes in determining the beginning of the Hijri month, Qibla directions, and prayer times. This book offers a more precise approach by showing how the Python environment can be tailored for astronomical computations and how the mathematical principles behind Qiblah determination can be implemented through elegant Python algorithms. The guide provides detailed methodologies for calculating prayer times with astronomical precision, allowing for accurate scheduling regardless of global location. The book also delves into the science of moonsighting, helping readers learn to compute and analyze observation data critical for Islamic calendar determinations. Advanced visualization chapters bring these calculations to life through practical applications: develop your own Qiblah compass, create visual representations of the sun's position during prayer times, and generate detailed lunar crescent visibility charts to aid in moon-sighting efforts. Perfect for programmers interested in Islamic Astronomy, religious scholars embracing technology, or anyone seeking to understand the mathematical foundations behind these traditional practices, this guide bridges ancient wisdom with modern computational techniques, making complex astronomical calculations accessible through the power of Python.

Key Features:

- The first book to provide practical guidance for using Python, supplemented by an interactive coding website, to solve real-world problems in the field of Islamic Astronomy.
- Uses the latest and most-trusted methods in Islamic Astronomy, ensuring all calculations are accurate and based on well-recognized references.
- Includes visualizations that help readers understand key topics like Qibla direction, prayer time zones, and lunar crescent visibility, making the content practical and user-friendly.

Muhamad Syazwan Faid is a senior lecturer at the Department of Islamic Studies, Centre for General Studies and Co-Curricular, Universiti Tun Hussein Onn Malaysia (UTHM). His expertise lies at the intersection of Islamic Astronomy and computational sciences, where he integrates traditional Islamic astronomical practices with modern technological advancements.

Muhammad Syaoqi Nahwandi is a lecturer at UIN Syekh Nurjati Cirebon, where he specializes in Islamic Astronomy and its integration with modern scientific techniques. His academic focus includes lunar crescent visibility, prayer time calculations, and Hijri calendar determination.

Shahrin Ahmad is a renowned Malaysian astronomer and the founder of Falak Online, a platform dedicated to promoting Islamic Astronomy and public astronomy education. With extensive experience in observational astronomy, Shahrin is widely regarded as a leading figure in advancing understanding and application of Islamic astronomical practices.

Mohd Saiful Anwar Mohd Nawawi is an associate professor at the Department of Fiqh-Usul and Applied Sciences, Academy of Islamic Studies, University of Malaya (UM). A renowned scholar in Islamic Astronomy, he specializes in the development of methodologies for Hijri calendar determination, Qibla direction, and prayer time calculations.

Python for Islamic Astronomy

Modern Computational Approaches to Hijri Calendar, Qibla, and Prayer Times

Muhamad Syazwan Faid,
Muhammad Syaoqi Nahwandi,
Shahrin Ahmad, and
Mohd Saiful Anwar Mohd Nawawi

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: Shutterstock

First edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-1-041-08261-3 (hbk)
ISBN: 978-1-041-09243-8 (pbk)
ISBN: 978-1-003-64912-0 (ebk)

DOI: 10.1201/ 9781003649120

Typeset in Times
by Deanta Global Publishing Services, Chennai, India

Contents

Acknowledgments

Alhamdulillah, praise be to Allah SWT for His Rahmah and with His permission we managed to complete this book successfully. With His permission, too, we gained the enthusiasm and spiritual strength needed in weathering the trials and tribulations on my journey. We would also like to extend our grateful thanks to the parties who have been immensely helpful and supportive throughout the time we worked on this book. Without their help, cooperation, and support, we may not have been able to come this far. Finally, to all parties directly or indirectly involved and who contributed to this study, we pray that you will be blessed by Allah SWT and bestowed with the best rewards. Thanks for everything.

This research was supported by Universiti Tun Hussein Onn Malaysia (UTHM) through Tier 1 vot (Q989).

1 Application of Islamic Astronomy for Muslim

This book is supplemented by an interactive coding website hosted on a Google Colab repository. Readers can explore the code used in the computation and calculation of Islamic Astronomy, observe how it is implemented within the Colab environment, and adjust the demonstrated code according to their own variables and real-life situations. Users do not need to rewrite the code manually from this book; instead, they can simply copy it directly from the Google Colab repository. Rewriting code often leads to mistakes, and the more mistakes made, the less enjoyable the exercise becomes; trust me, I speak from experience. The website contains 6 hyperlinks, which direct the user to the corresponding chapter. The website can be accessed on bit.ly/pythonforislamicastronomy, or using the QR Code given in Figure 1.1.

From this link, user can edit the code. The edited code will be saved on the user's own copy of Google Colab and won't affect the original coding.

Islamic Astronomy, also known as *'ilm al-hay'ah*, refers to the branch of science developed and practiced within the Islamic world that deals with the study of celestial bodies and their movements, primarily for religious, calendrical, and navigational purposes (Ilyas, 1997). This area of study combines observational astronomy with Islamic principles, making it a vital area of study for Muslims. Islamic Astronomy has historically contributed to various fields, from timekeeping and calendar formation to navigation and the understanding of celestial events (Yusuf, 2010). Its importance is seen in the precise ways it influences Muslims' religious obligations and broader scientific knowledge.

One of the most practical applications of Islamic Astronomy is determining the exact times for the five daily prayers (*Salah*), which rely on the sun's position in the sky. This ensures that Muslims perform their prayers at the correct times each day (Abas et al., 2022). In Malaysia, the Department of Islamic Development Malaysia (JAKIM) publishes detailed prayer timetables every

DOI: 10.1201/9781003649120-1

FIGURE 1.1 QR code for Google Colab website.

year, calculated using astronomical algorithms and verified through observation. Tools like the Almanak Falak JAKIM allow users to refer prayer times specific to the zone (Huda et al., 2014). The Umm al-Qura calendar in Makkah uses a combination of astronomical data and historical tradition to set prayer times and official schedules, especially critical for the millions of pilgrims during Hajj and Umrah (Rubin, 2017). The Muslim Council of Britain provides detailed prayer timetables for different cities, incorporating high-latitude adjustments for places like Edinburgh, where sunlight patterns complicate standard calculations (Ali, 2015).

The Hijri calendar is a lunar calendar in which each month begins with the sighting of the new crescent moon (*hilal*). This determination is critical for establishing the dates of key Islamic events such as Ramadan, Eid al-Fitr, and Hajj (Ilyas, 1984). Moon sighting activities are regularly held across Muslim-majority countries. For example, in Indonesia, the Ministry of Religious Affairs coordinates nationwide *rukyah* observations, such as those conducted at the Pelabuhan Ratu Observatory, to verify the start of Ramadan and other significant months (Wahidi et al., 2019). The State Mufti Department manages official crescent moon sightings from sites like Bukit Agok Observatory, with live national broadcasts of Ramadan and Shawwal announcements (Mohd Nawawi et al., 2024). Morocco is known for using *hisab* (astronomical calculations) alongside *rukyah* to determine the Islamic months, often leading to different Ramadan start dates compared to neighboring countries (Lairgi, 2025).

Muslims must face the Kaaba in Makkah when performing prayers. Islamic Astronomy provides methods for determining the Qibla direction accurately from any point on Earth. Historically, this was done using instruments like the astrolabe, and today, with the help of digital compasses and satellite technology (Faid, Nahwandi, et al., 2022). In recent years, researchers undertook a measurement project for mosque prayer spaces to ensure they were

accurately aligned with the Qibla based on updated astronomical and geospatial data (Yildirim et al., 2024). Al-Azhar Mosque, one of the oldest universities in the Islamic world, underwent a Qibla realignment in 1992 after new astronomical surveys corrected earlier orientation errors dating back centuries (Rabbat, 1996). Muslim communities often use smartphone apps (e.g., Muslim Pro) and Google Earth Qibla services, which apply astronomical algorithms to pinpoint the Kaaba's direction from anywhere in the country (Schumm, 2020).

Islamic teachings emphasize observing and responding to celestial events such as solar and lunar eclipses, which are occasions for special prayers. Islamic astronomers calculate and predict these events with precision, enabling communities to observe them properly (King, 1993). During the total lunar eclipse on July 27, 2018, Muslim communities worldwide, including in Saudi Arabia and Indonesia, performed *Salat al-Khusuf* (eclipse prayer) and organized observation sessions. Observatories like Bosscha Observatory in Indonesia broadcasted the event live, blending scientific outreach with religious observance (Izzuddin et al., 2022). During the 2019 annular solar eclipse, mosques across the kingdom held *Salat al-Kusuf*, and observatories provided live coverage with explanations linking religious practice to astronomical phenomena (Elmhamdi et al., 2024).

Before the advent of modern navigation tools, Islamic astronomers developed sophisticated navigation techniques based on celestial bodies. This was especially crucial for travelers, pilgrims, and maritime traders navigating long distances across deserts and seas. Muslim sailors and traders navigating the Indian Ocean between Africa, the Arabian Peninsula, and Southeast Asia used instruments like the *Kamal*, a simple device to measure the altitude of stars, to determine their position and direction. This knowledge was crucial in facilitating trade routes and pilgrimage journeys for centuries (Niri et al., 2023). North African caravans traveling across the Sahara to Makkah relied on astronomical observations, notably the Pole Star (Polaris), to maintain their route across the desert. During the Ottoman Empire, maritime explorers and military fleets used Islamic Astronomy-based charts and instruments like the astrolabe and quadrant to navigate the Mediterranean and Red Sea efficiently (Faid, Nawawi, et al., 2022).

Why Python Matters in Islamic Astronomy

2

Python is a widely used, interpreted, object-oriented, high-level programming language with dynamic semantics, designed for general-purpose programming. It is ubiquitous, and many people use Python-powered devices every day, whether they realize it or not. Python was created by Guido van Rossum and was first released on February 20, 1991. While you might know the python as a large snake, the name of the Python programming language actually comes from an old BBC television comedy sketch series called "Monty Python's Flying Circus" (Mehare et al., 2023). One of the great features of Python is that it is truly the work of one person. Typically, new programming languages are developed and published by large companies employing many professionals, and due to copyright regulations, it is very difficult to name any individuals involved in such projects. Python is an exception (Faid, Mohd Nawawi, et al., 2024).

Of course, Guido van Rossum did not develop and expand all the components of Python by himself. The rapid spread of Python across the globe is the result of the continuous work of thousands of (very often anonymous) programmers, testers, users (most of them not IT experts), and enthusiasts, but it must be said that the initial idea (the seed from which Python sprouted) came from one person – Guido (Ozgur et al., 2021). Python is maintained by the Python Software Foundation, an organization and community dedicated to developing, improving, expanding, and popularizing the Python language and its environment. There are billions of lines of code written in Python, which means nearly unlimited opportunities for code reuse and learning from well-designed examples. Moreover, there is a large and highly active Python community, always happy to help (Faid, Nawawi, Saadon, et al., 2023).

DOI: 10.1201/9781003649120-2

Several factors make Python great for learning:

- It's easy to learn – The time required to learn Python is shorter than many other languages, meaning you can start real programming more quickly.
- It's easy to use for writing new software – Often, it's possible to write code faster when using Python.
- It's easy to obtain, install, and use – Python is free, open-source, and cross-platform; not all languages can boast that.

Programming skills prepare you for a career in almost any industry and are essential if you want to pursue more advanced and higher-paying software development and engineering roles. Python is a programming language that opens more doors than others. With solid Python knowledge, you can work in various jobs and industries. And the more you understand Python, the more you can do in the 21st century. Even if you don't need it for work, you'll find it useful to know (Blank & Deb, 2020).

Python is a great language for science, particularly in astronomy. Various packages like NumPy, SciPy, Scikit-Image, and Astropy (to name just a few) are all excellent examples of Python's suitability for astronomy, and there are many use cases. [NumPy, Astropy, and SciPy are fiscally sponsored projects by NumFOCUS; Scikit-Image is an affiliated project.] These tools make it easier to use Python in various astronomical projects.

For example, the European Southern Observatory (ESO), which operates the Very Large Telescope (VLT), offers data for download on their site. You can visit www.eso.org/UserPortal and create a username for their portal. If you're looking for data from the SPHERE instrument, you can download full datasets for any nearby stars with exoplanet or protoplanetary disks. It's a fantastic and engaging project for any Pythonista to reduce that data and make the hidden planets or disks visible amid the noise (Rhodes, 2011).

By using the tools offered by NumPy, SciPy, Astropy, Scikit-Image, and many more in combination, with a little patience and persistence, it's possible to analyze vast amounts of available astronomical data to produce some stunning results. Python plays a significant role in Islamic Astronomy, particularly in determining prayer times, the Qibla direction, and the beginning of the Hijri months. These aspects are central discussions in Islamic Astronomy.

PRAYER TIMES CALCULATION

The determination of prayer times involves using the position of the sun as a reference to measure the start and end times of each prayer. The position of the sun after its zenith is used to determine Zuhr, the sun's shadow for Asr, sunrise and sunset for Maghrib and Syuruk, while the sun's light below the horizon is used for Fajr and Isha (Faid et al., 2019). Python, with its libraries such as NumPy, SciPy, and Astropy, can be used to perform precise calculations of the sun's position, which is crucial for determining accurate prayer times (Faid et al., 2021).

QIBLA DIRECTION DETERMINATION

The determination of the Qibla direction requires accurate readings of latitude and longitude. Historically, Islamic scholars used the position of the sun and solar eclipses to determine these coordinates (Faid, Husien, et al., 2016). Today, GPS provides latitude and longitude, but surveyors and astronomers still require sun position data to obtain the azimuth direction. Python can be used to calculate the azimuth angle accurately, incorporating latitude, longitude, and the sun's position, which is vital for ensuring that the Qibla direction is correct (Amin, 2018).

DETERMINING THE BEGINNING OF THE HIJRI MONTHS

The beginning of an Islamic month is determined by the visibility of the crescent moon (Shariff et al., 2016). The moon's visibility depends on its position relative to the sun, and the calculations involved are more complex than those for the sun because of the moon's faster and more dynamic movement influenced by the gravitational forces of the sun and the Earth. Python, through the use of libraries like Astropy, allows for precise calculations of the moon's position, which is essential for determining the new moon and thus the start of the Hijri month (Muztaba et al., 2023).

Importance of Accuracy of Calculation

3

The precision in calculating the positions of the sun and moon directly impacts the determination of prayer times, the Qibla direction, and the visibility of the crescent moon. If these calculations are inaccurate, it could lead to incorrect prayer times, Qibla direction, or the start of the Islamic month. Therefore, it is crucial for those in authoritative positions to use accurate calculations. Public institutions like JAKIM, JUPEM, and the State Mufti Departments in Malaysia are responsible for producing accurate data, which the public relies on (Shariff et al., 2017). For example, the Qibla direction of mosques and suraus must be certified by official surveyors and the Mufti Department to ensure that the direction used by the local Muslim community is accurate. Similarly, the release of Prayer Time Tables and the annual Hijri Calendar involves precise calculations and is usually vetted by the State Falak Council before being published (Faid, Shariff, et al., 2016).

As the public, they are not burdened with performing these precise calculations ourselves. Instead, we are encouraged to rely on authoritative sources for accurate information. However, increasing public awareness and understanding of Islamic astronomical calculations can enhance appreciation for this knowledge and reduce confusion and the spread of misinformation (Faid et al., 2018). By using Python in Islamic Astronomy, we can harness its power to perform accurate and efficient calculations, ensuring that the Islamic practices related to time and direction are observed correctly and consistently (Faid et al., 2025). Accuracy in Islamic Astronomy is a highly critical aspect, particularly in determining the new Hijri month (Syazwan Faid et al., 2025). This is due to several factors that have a direct impact on the religious practices of Muslims, as well as broader social and economic implications (Wahyuni et al., 2022).

DOI: 10.1201/9781003649120-3

ACCURATE DETERMINATION OF RELIGIOUS PRACTICES

The determination of the new Hijri month is crucial as it sets the dates for key events in the Islamic calendar, such as the beginning of Ramadan, Eid al-Fitr, and Eid al-Adha. Accuracy in determining the Hijri month ensures that practices like fasting and the Eid prayer are performed on the correct days. Even a deviation as small as 0.5 degrees in the moon's position can lead to differences in date determination, potentially causing confusion within the community (Adegoke, 2013, 2017).

SOCIAL AND ECONOMIC IMPACT

Errors in determining the Hijri month can lead to misunderstandings among Muslims, especially in societies where religious dates play a significant role in social and economic planning. For instance, a change in the date of Eid can affect various aspects such as holiday planning, festival preparations, and related economic activities (Mohd Nawawi et al., 2024).

IMPORTANCE OF MODERN TECHNOLOGY AND CALCULATIONS

The use of modern technology, such as programming in Python, and advanced astronomical tools, allows for more accurate and consistent calculations in determining the Hijri month. These calculations involve various astronomical factors, such as the positions of the moon, sun, and Earth, as well as other factors like weather conditions and topography that can affect the visibility of the new moon (Junaidi, 2022). By employing advanced algorithms and precise mathematical calculations, the likelihood of errors in determining the Hijri month is significantly reduced. This accuracy is essential to ensure that Muslims worldwide can practice their faith with full confidence that the dates they observe are correct and valid (Gharaybeh, 2025).

COMPLEXITY OF CALCULATIONS

The process of calculating the new Hijri month is highly complex, involving hundreds of mathematical formulas and thousands of variables. Every aspect must be meticulously calculated to ensure accurate results. For example, factors such as the moon's altitude angle, the distance between the moon and the sun, and the times of sunset and moonset all play crucial roles in these calculations. Any error in these calculations can lead to incorrect date determination, which not only affects religious practices but can also cause confusion and division among Muslims (Faid, Shariff, et al., 2024; Meeus, 1991). Therefore, accuracy in Islamic Astronomy is not only necessary to maintain the correctness of religious practices and rituals but also to ensure social harmony and stability within the community (Faid, Nawawi, et al., 2024). By leveraging technology and precise scientific approaches, Muslims can observe their religious practices with greater confidence and alignment with the teachings of their faith. With the advancement of technology, the use of astronomical calculation software and programming algorithms has greatly facilitated this process, ensuring that each calculation is done accurately and efficiently. However, even with the aid of technology, expertise and in-depth knowledge of Islamic Astronomy are still required to ensure accuracy in determining the new Hijri month (Rasyid et al., 2024).

Python can be utilized to model and visualize astronomical data in a highly efficient and accurate manner. Below are some specific applications of Python in the field of Islamic Astronomy (Falak).

CALCULATION AND VISUALIZATION OF CRESCENT MOON POSITION

Python can be used to accurately calculate the position of the moon each night. By applying precise astronomical formulas, it is possible to determine the time and location where the crescent moon will be visible. Python also allows us to plot the moon's movement in the form of graphs or maps, making it easier for astronomers to predict and confirm the visibility of the crescent moon. This is crucial for determining the start of the Hijri month and Islamic celebrations like Ramadan and Shawal (Rasyid et al., 2023).

CALCULATION AND VISUALIZATION OF PRAYER TIMES

Python can be employed to calculate prayer times based on the sun's position. By inputting geographical location data, Python can calculate sunrise, solar noon, sunset, and other critical times needed to determine prayer times. Additionally, Python can plot the sun's position and the brightness of the sky throughout the day. This helps astronomers and the general public understand the changes in prayer times throughout the year, including variations due to seasonal changes and location (Razzak, 2024).

CALCULATION OF QIBLA DIRECTION

Python can be used to calculate the sun's shadow trajectory to determine the Qibla direction. By calculating the sun's position at a specific time, we can determine the direction of the shadow that will indicate the accurate Qibla direction. This is particularly useful for ensuring that the prayer direction is correct, especially in places where visual guidance might not be available. Python can generate maps or graphs showing the Qibla direction for various locations around the world (Asrin et al., 2018). By using Python, we can simplify and expedite calculation processes that would otherwise require a significant amount of time and effort if done manually. Python not only simplifies the calculation process but also ensures that the results are accurate and reliable. This makes Python an invaluable tool in the field of Islamic Astronomy (Falak) and Islamic Astronomy at large (Al-Rajab et al., 2023; Loucif et al., 2024).

Setting up Python for Astronomical Calculations

4

GETTING STARTED WITH PYTHON

Google Colaboratory, commonly known as Google Colab, is a free cloud-based platform provided by Google that allows users to write and execute Python code through the browser. It is especially popular among data scientists, educators, and researchers for its simplicity, power, and seamless integration with Google Drive. No installation is required, just a Google account.

With Google Colab, you can:

- Write and run Python code instantly.
- Use pre-installed scientific libraries like NumPy, Pandas, Matplotlib, and more.
- Collaborate with others in real time.
- Save and share notebooks easily through Google Drive.

This makes it a convenient tool for projects such as calculating Islamic Astronomy matter using Python.

DOI: 10.1201/9781003649120-4

USING PYTHON FOR PRAYER TIME CALCULATIONS IN GOOGLE COLAB

You can use Google Colab to run Python scripts that calculate prayer times based on astronomical data and geographical coordinates. Here's how to get started:

Step 1: Access Google Colab
Go to https://colab.research.google.com. You'll need to sign in with your Google account.
Step 2: Create a New Notebook
- Click on **"File" > "New Notebook"**.
- A new notebook with a Python environment will open in your browser.

Step 3: Install dependencies or libraries and start coding!

BASIS OF PYTHON OPERATION

Python can handle basic arithmetic just like a scientific calculator. Below are the key operations:

Addition

```
a = 10
b = 5
result = a + b
print(result) # Output: 15
```

Subtraction

```
a = 10
b = 5
result = a - b
print(result) # Output: 5
```

Multiplication

```
a = 10
b = 5
result = a * b
print(result) # Output: 50
```

Division

```
a = 10
b = 5
result = a / b
print(result) # Output: 2.0
```

Integer Division

```
a = 10
b = 3
result = a // b
print(result) # Output: 3
```

Modulus

```
a = 10
b = 3
result = a % b
print(result) # Output: 1
```

Exponentiation

```
a = 2
b = 3
result = a ** b
print(result) # Output: 8
```

To use trigonometric functions, we import the math module. Python's math functions work in **radians**, not degrees.

Importing Math Module

```
import math
```

Convert Degrees to Radians

```
degrees = 90
radians = math.radians(degrees)
print(radians) # Output: 1.5707963267948966
```

Trigonometric Functions

Sine (sin)

```
angle = 30
result = math.sin(math.radians(angle))
print(result) # Output: 0.49999999999999994
```

Cosine (cos)

```
angle = 60
result = math.cos(math.radians(angle))
print(result) # Output: 0.5000000000000001
```

Tangent (tan)

```
angle = 45
result = math.tan(math.radians(angle))
print(result) # Output: 0.9999999999999999
```

Inverse Tangent (Arctangent/atan)

The atan() function returns the angle (in **radians**) whose tangent is the given number. To convert it into degrees, use math.degrees():

```
x = 1 # tan(angle) = 1
angle_rad = math.atan(x)
angle_deg = math.degrees(angle_rad)
print(angle_rad) # Output: 0.7853981633974483
(radians)
print(angle_deg) # Output: 45.0 (degrees)
```

This is especially useful in triangle calculations, astronomy, and physics when determining angles based on known side ratios.

BASIC CONDITIONAL STATEMENTS IN PYTHON

Conditional statements allow your program to make decisions based on certain conditions. This is useful in nearly all practical applications.

The if Statement

The if statement executes a block of code only if a specified condition is true.

```
temperature = 30
if temperature > 25:
   print("It's a hot day.")
```

The else Statement

The else block runs when the if condition is false.

```
temperature = 20
if temperature > 25:
   print("It's a hot day.")
else:
   print("It's a nice day.")
```

Using or in Conditions

The or keyword checks if **at least one** of multiple conditions is true.

```
day = "Sunday"
is_holiday = True

if day == "Sunday" or is_holiday:
   print("You can rest today.")
else:
   print("It's a working day.")
```

Combined Example

Here is a real-world style example:

```
score = 80
if score >= 90:
   print("Excellent")
elif score >= 70:
   print("Good job")
else:
   print("Keep trying")
```

Explanation

- If the score is 90 or more → print **"Excellent"**
- If the score is 70 or more (but less than 90) → print **"Good job"**
- If the score is less than 70 → print **"Keep trying"**

Output

```
Good job
```

Another Example Using or

```
score = 60
if score < 70 or score > 100:
   print("Score needs review")
else:
   print("Score is acceptable")
```

Output

```
Score needs review
```

These basic control structures allow your Python programs to make decisions and react to different inputs. This shows how Python checks conditions in order. It stops at the first condition that is True. These if, else, and or statements give your Python program logic, allowing it to behave differently based on input or conditions. This is essential in automation, data filtering, simulations, and more.

print() in Python

The print() function is used to display information or output on the screen. It is one of the most basic and commonly used functions in Python programming.

```
print("Welcome to the world of Python!")
```

Output

```
Welcome to the world of Python!
```

You can also print numbers or the result of calculations:

```
print(5 + 3)
```

Output

```
8
```

The print() function is essential for checking values, debugging, and interacting with users. Sometimes, you want to print a sentence that includes variable values, such as a person's name or age. In Python, this is made easier and cleaner with formatted strings, also called f-strings.

```
name = "Aminah"
age = 18
print(f"My name is {name} and I am {age} years old.")
```

Output

```
My name is Aminah and I am 18 years old.
```

In this example, the f before the quotation mark tells Python to treat the string as a **formatted string**. Inside the curly braces {}, Python will insert the value of the variable. You can even include calculations:

```
length = 5
width = 3
print(f"The area is {length * width} square units.")
```

Output

```
The area is 15 square units.
```

Formatted strings make your code shorter, cleaner, and easier to understand, especially when combining text and values. Sometimes you want to control the number of decimal places shown. You can use format specifiers like :.2f to round to 2 decimal places.

```
azimuth_kiblat = 292.347182
print(f"Azimuth Kiblat: {azimuth_kiblat:.2f}")
```

Output

```
Azimuth Kiblat: 292.35
```

In {azimuth_kiblat:.2f}:

- : starts the format
- .2f means **2 decimal places** in **floating-point format**

This is very useful in math, geography, and science applications.

CONVERTING DECIMAL DEGREES TO DEGREES, MINUTES, AND SECONDS

In many fields such as geography, surveying, and navigation, angles are often represented not just in decimal degrees (e.g., 292.35°) but in a more traditional format: Degrees, Minutes, and Seconds (DMS). While decimal degrees are easier for calculations, DMS format is more readable and commonly used in GPS coordinates, navigation systems, and official land records.

Given a decimal degree value:

1. **Degrees**: The whole number part of the decimal degree
2. **Minutes**: Multiply the decimal part by 60, then take the whole number
3. **Seconds**: Multiply the remaining decimal by 60 again

Example: Convert 292.35° to DMS

1. **Degrees** = 292
2. **Decimal part** = 0.35
3. **Minutes** = 0.35 × 60 = 21.0 → **21 minutes**
4. **Seconds** = 0.0 × 60 = 0 → **0 seconds**

Final Answer

292.35∘=292∘ 21′ 0″292.35^\circ = 292^\circ\ 21′\ 0″292.35∘=292∘ 21′ 0″

Here is a Python script that performs this conversion:

decimal_degree = 292.35

```
# Step 1: Get whole degrees
degrees = int(decimal_degree)
# Step 2: Get the decimal part and convert to minutes
decimal_part = decimal_degree - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

# Step 3: Convert the decimal part of minutes to seconds
seconds = round((minutes_total - minutes) * 60)

print(f"{decimal_degree}° = {degrees}° {minutes}′
{seconds}″")
```

Output

```
292.35° = 292° 21′ 0″
```

This method is useful for displaying locations and angles in a standardized, human-friendly format.

INSTALLING PYTHON PACKAGES WITH PIP

As you begin writing Python programs, you will often need to use additional tools, libraries, or frameworks that are not included in Python by default.

These tools are called packages. Python provides a powerful command-line tool called pip to help you install and manage these packages easily.

What Is pip?

pip stands for "Pip Installs Packages". It is the official package installer for Python.

With pip, you can:

- Install new Python libraries (like numpy, matplotlib, flask)
- Upgrade packages
- Uninstall packages
- Check what packages are installed

Installing a Package Using pip

You can install a Python package from the internet using this simple command:

```
pip install package_name
```

Example

To install requests, a popular package for HTTP:

```
pip install requests
```

Listing Installed Packages

To view all packages currently installed:

```
pip list
```

This will show the name and version of each package. pip is a tool for installing and managing Python packages. You can install, upgrade, and remove packages easily using simple commands. pip connects to the Python Package Index (PyPI) to download packages from the internet. Learning to use pip is essential for working with real-world Python projects.

WORKING WITH DATES AND TIMES IN PYTHON, THE DATETIME MODULE

In many real-world programs, especially those involving calendars, logs, astronomical events, or scheduling, you need to work with dates and times. Python makes this easy through a built-in module called datetime.

The datetime module allows you to:

- Create and manipulate dates and times
- Extract day, month, year, hour, minute, etc.
- Perform date arithmetic (e.g., add days, subtract dates)
- Format dates into readable strings

Importing the Datetime Module

To use the datetime features, you first need **to import it:**

```
from datetime import datetime
```

This line imports the datetime class from the datetime module.

Creating a Specific Date

You can define a date and time using the datetime() constructor:

```
date = datetime(2024, 1, 14)
```

This creates a datetime object representing:

- **Year**: 2024
- **Month**: January (1)
- **Day**: 14

By default, the time is set to midnight (00:00:00) if not specified.

You can also include the time:

```
date = datetime(2024, 1, 14, 15, 30)
```

This means 14 January 2024, at 3:30 PM.

Accessing Date Components

Once a datetime object is created, you can access parts of it:

```
print(date.year) # 2024
print(date.month) # 1
print(date.day) # 14
```

You can also get the **day of the week** or **day of the year**:

```
print(date.weekday()) # 6 (Sunday; Monday is 0)
print(date.timetuple().tm_yday) # 14 (14th day of the
year)
```

Why Is This Useful?

Handling dates and times accurately is essential in:

- Astronomical calculations (like Qibla direction)
- Logging timestamps in applications
- Scheduling tasks or sending reminders
- Managing calendar events

Loop

A **loop** in programming is used to repeat a set of instructions **if a certain condition is true**. Loops are useful when we want to:

- Repeat a task many times
- Search for a value
- Simulate continuous or time-based processes

Python provides two main types of loops:

- for loop – used when you know how many times you want to repeat
- while loop – used when you want to repeat **until** a condition is met

A while loop continues **executing** the code inside it **as long as** the condition is true.

```
while condition:
# do something
If the condition becomes false, the loop stops.
```

Example 1: Simple Counter

```
count = 0
while count < 5:
print("Counting:", count)
count += 1
Counting: 0
Counting: 1
Counting: 2
Counting: 3
Counting: 4
```

Why Use while?

Use while when:

- You don't know how many times the loop should run.
- You're waiting for a condition to become true.

Example

Write a loop that prints numbers until the square of the number is greater than 100.

python

```
n = 1
while True:
print(n)
if n * n > 100:
break
n += 1
```

Summary

- while loops are perfect for conditions with **unknown repetitions**.
- Always ensure there's a way to **exit the loop**.
- They're useful for simulations, astronomy calculations, and search-based problems.

USING SKYFIELD

Skyfield is a Python library designed to calculate the positions of stars, planets, and satellites orbiting the Earth. Its results are expected to align with the positions generated by the United States Naval Observatory and their Astronomical Almanac within 0.0005 arcseconds (half a milli-arcsecond, or mas). Skyfield is written entirely in pure Python and can be installed without any compilation, making it accessible for various Python environments. It supports Python versions 2.6, 2.7, and 3, with NumPy as its only binary dependency. NumPy is a fundamental package for scientific computing with Python, and its vector operations make Skyfield efficient. Before Skyfield was developed, an older version known as PyEphem was used. Skyfield builds on the capabilities of PyEphem, offering more accurate and modern astronomical calculations (Faid, Mohd Nawawi, et al., 2023; Faid, Nawawi, & Saadon, 2023; Faid, Nawawi, Wahab, et al., 2023).

PyEphem uses popular astronomical calculation techniques derived from Jean Meeus's *Astronomical Algorithms*, such as the IAU 1980 model of Earth's nutation and the VSOP87 planetary theory. These techniques are still employed by various authoritative bodies around the world (Holwerda et al., 2016). PyEphem offers accuracy up to 1 arcsecond, which is sufficient for calculating lunar and solar data. However, for new projects, the Skyfield library should be prioritized over PyEphem. Its modern design encourages better Python coding practices and utilizes NumPy to accelerate calculations. PyEphem's reliance on C libraries often results in frustrating installation issues. If the Python Package Index (PyPI) doesn't have a wheel for your system, you would need a C compiler and a Python development environment to install PyEphem.

Another drawback of PyEphem is its handling of angular units, which can be confusing. The library tries to be clever by interpreting string inputs like '1.23' as degrees of declination (or hours, when setting the right ascension), but floating-point inputs like 1.23 are assumed to be in radians. The angles returned by PyEphem add to the confusion: when printed, they display in degrees, but performing arithmetic on them reveals that they are in radians. This leads to significant confusion and makes the code harder to read, but fixing it would break existing programs that rely on PyEphem. Moreover, PyEphem's compute () method modifies its objects in place rather than returning a result. While this reflects how the underlying C library operates, it makes using compute () in list comprehensions difficult, you end up with a list of None objects. Therefore, in this context, Skyfield will be used for calculating lunar and solar data. Skyfield's advantages, such as modern design, better

integration with Python, and ease of use, make it a superior choice for astronomical calculations in contemporary projects. Skyfield is a modern Python library that makes it easy to compute positions of planets and other celestial objects using high-precision data from NASA's Jet Propulsion Laboratory (JPL). This guide explains how to install and use the **DE440s ephemeris**, which contains positional data for the solar system.

Step 1: Install the Skyfield Library

To begin, install the Skyfield library in Google Colab or Jupyter Notebook. In a new cell, type the following and press **Shift + Enter**:

```
!pip install skyfield
```

This will install the core Skyfield library, which is needed to perform astronomical computations.

Step 2: Understand What an Ephemeris Is

An **ephemeris** is a table or dataset that provides the positions of celestial objects at regular intervals. Skyfield uses ephemerides published by NASA JPL to calculate accurate positions of planets and the moon. For this example, we will use **DE440s**, a compact but high-accuracy ephemeris suitable for many applications.

Step 3: Load the Ephemeris DE440s

Next, we will load the DE440s ephemeris using Skyfield. Type the following code into a new cell:

```
from skyfield.api import load
# Load the time scale
ts = load.timescale()
# Load the ephemeris data (DE440s) from the internet
planets = load('de440s.bsp')
```

load.timescale() prepares the time system used in calculations. load('de440s.bsp') downloads and loads the DE440s ephemeris file. This file will be downloaded from the internet the first time you run the code. Make sure your internet connection is stable.

Step 4: Verify Successful Loading

Once the ephemeris file is downloaded, Skyfield will store it locally (or in the Colab session temporarily). If successful, no error will be shown, and you can now access planetary positions like this:

```
earth = planets['earth']
mars = planets['mars']
```

Qiblah Calculation Using Python

5

The calculation of the Qibla direction is based on three points of location. The first is the location of the Kaaba, which is located at 21.4225 North, 39.8262 East. The second location is the calculated position of the observer. The third point is the north pole, where it acts as the axis for the angle between Kaaba and the calculated position. Therefore, the formula for the Qibla direction toward the Kaaba from the calculated position is,

$$\Delta\lambda = \left(\lambda_{Location} - \lambda_{Kaabah}\right) \quad \text{Equation 5.1}$$

$\Delta\lambda$ is the formula for calculating the distance from the longitude of the Kaaba to the longitude of the place for a location located in the east longitude, which can be obtained by calculating the difference in the values of the two longitudes. Then,

$$A = \sin\Delta\lambda \quad \text{Equation 5.2}$$

$$B = \cos\left(\varphi_{Location}\right) \times \tan(\varphi_{Kaabah}) \quad \text{Equation 5.3}$$

$$C = \sin\varphi_{Location} \times \cos\Delta\lambda \quad \text{Equation 5.4}$$

$$D = {}^{A}\!\!\diagup\!\!{}_{\left(B-C\right)} \quad \text{Equation 5.5}$$

Then the Qibla Direction (θ) is calculated using the formula,

$$\theta = Tan^{-1}\left(D\right) \quad \text{Equation 5.6}$$

DOI: 10.1201/9781003649120-5

The formula for Qibla Calculation above can only be used for locations in Southeast Asia. This is because the calculation formula for calculating (Δλ) only considers conditions when a location is in the east longitude which has a value greater than the longitude of the Kaaba. To make the formula for calculating the direction of the Qibla can be used in all locations in the east and west longitudes, the distance from the longitude of the Kaaba to the longitude of the place (Δλ) is calculated using the following formula:

$$dev = abs\left(\lambda_{Location} - \lambda_{Kaabah}\right) \qquad \text{Equation 5.7}$$

$$\text{Distance between Longitude}, \Delta\lambda = \begin{cases} 360 - dev, if\ dev > 180 \\ dev, if\ dev \le 180 \end{cases} \qquad \text{Equation 5.8}$$

The value of location longitude in the formula is positive if the location is in east longitude, and it is negative if the location is in west longitude.

In the Universal Qibla Direction Calculation, the reference value of the Qibla direction (θ) is not always from North to West. However, it can vary depending on the Qibla direction value (θ) and the Longitude of the Place ($\lambda_{location}$). To find out the reference of Qibla direction, use the following provisions:

$$\text{Arah Mata Angin Kiblat} = \begin{cases} UB & if\ \theta > 0\ and\ \lambda_{location} > \lambda_{Kaabah} \\ UT & if\ \theta > 0\ and\ \lambda_{location} \le \lambda_{Kaabah} \\ UB & if\ \theta > 0\ and\ \lambda_{location} < 0\ and\ c \ge 180 \\ UT & if\ \theta > 0\ and\ \lambda_{location} < 0\ and\ c < 180 \\ SB & if\ \theta < 0\ and\ \lambda_{location} > \lambda_{Kaabah} \\ ST & if\ \theta < 0\ and\ \lambda_{location} \le \lambda_{Kaabah} \\ SB & if\ \theta < 0\ and\ \lambda_{location} < 0\ and\ c \ge 180 \\ ST & if\ \theta < 0\ and\ \lambda_{location} < 0\ and\ c < 180 \end{cases}$$

"UB" is a statement that the Qibla direction uses a reference from the north to the west.

"SB" is a statement that the Qibla direction uses a reference from the south to the west.

"UT" is a statement that the Qibla direction uses a reference from the north to the east.

"ST" is a statement that the Qibla direction uses a reference from the south to the east.

The Azimuth of Qibla formula also varies depending on the reference of Qibla Direction value (θ). To find out the Azimuth of Qibla value, use the following provisions:

$$\text{Azimuth Kiblat} = \begin{cases} 360-\theta & if\ UB \\ 180-\theta & if\ SB \\ \theta & if\ UT \\ 180+\theta & if\ ST \end{cases}$$

QIBLA CALCULATION

Exercise 1

Determine the Qibla direction of Penang, which has geographical latitude of 5.2632 North and 100.4846 East. The Qaabah geographical latitude is 21.4225 North, longitude is 39.8262 East. First, determine the variables, which are written in Python as,

```
φ_Location = 5.2632
λ_Location = 100.4846
φ_Kaabah = 21.4225
λ_Kaabah = 39.8262
Difference_Longitude = (λ_Location-λ_Kaabah )
```

Second, perform the calculation. Using the above formula, the calculation can be written in Python as

In Python, this is written as

```
import math
A = math.sin(math.radians(abs(Difference_Longitude)))
```

To confirm the result of A, write in Python

```
print(A)
0.8717137230643722
```

Next, to calculate B to D, in Python is written as,

```
B = math.cos(math.radians(φ_Location))*math.tan(math.r
adians(φ_Kaabah))
```

```
print(B)
0.3906945822201198
C = math.sin(math.radians(φ_Location))*math.cos(math.r
adians(Difference_Longitude))
print(C)
0.0449496276198435
D = A/((B-C))
print(D)
2.5212623104570837
```

The direction of the Qibla θ is calculated in Python as

```
θ = math.degrees(math.atan(D))
print(θ)
68.36540021170762
```

The direction of the Qibla does not necessarily translate to the azimuth of the Qibla based on the magnetic compass. Our phone determines the azimuth based on the reference of the magnetic compass. Therefore, a minor computation needs to be added to determine the azimuth of the Qibla. First, correction of the difference in longitude based on Equation 5.8, written in Python as,

```
#Determine the Azimuth of the Qibla
if Difference_Longitude > 180:
  delta_λ = 360 - Difference_Longitude
else:
  delta_λ = Difference_Longitude
print(delta_λ )
60.6584
```

Then, the quadrant of the calculated user geographical location, with Kaaba as the center of the quadrant, in Python,

```
if θ > 0:
       if λ_Location > λ_Kaabah:
                quadrant = "UB" # Utara Barat
       elif λ_Location <= λ_Kaabah:
                 quadrant = "UT" # Utara Timur
   elif λ_Location < 0:
                if c >= 180:
                                quadrant = "UB"
                else:
                                quadrant = "UT"
```

```
elif θ < 0:
   if λ_Location > λ_Kaabah:
      quadrant = "SB" # Selatan Barat
   elif λ_Location <= λ_Kaabah:
            quadrant = "ST" # Selatan Timur
elif λ_Location < 0:
    if c >= 180:
            quadrant = "SB"
            else:
                    quadrant = "ST"
print(quadrant)
UB
```

If the quadrant is UB, meaning Utara-Barat in Malay or Northwest in English. This means that the Azimuth of the Qibla is located at the Northwest from the computed location. To convert from Qibla direction to Qibla azimuth, written in Python

```
if quadrant == "UB":
  azimuth_kiblat = 360 – θ
elif quadrant == "SB":
  azimuth_kiblat = 180 – θ
elif quadrant == "UT":
  azimuth_kiblat = θ
elif quadrant == "ST":
  azimuth_kiblat = 180 + θ
print(azimuth_kiblat)
291.63459978829235
```

Therefore, the azimuth of the Qibla is 291.63 degree. To convert the result in degree,

```
# To Convert in Degree Form
degrees = int(azimuth_kiblat)
decimal_part = azimuth_kiblat – degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
# Step 3: Convert the decimal part of minutes to
seconds
seconds = round((minutes_total – minutes) * 60)
print(f"{azimuth_kiblat}° = {degrees}° {minutes}'
{seconds}"")
291.63459978829235° = 291° 38' 5"
```

Finally, to print the result in a text

```
print(f'The azimuth of the Qibla for Location with
coordinates {φ_Location} N Latitude, {λ_Location}
Longitude, is {azimuth_kiblat:0.2f}')
The azimuth of the Qibla for Location with coordinates
5.2632 Latitude, 100.4846 Longitude, is 291.63
```

Exercise 2

Determine the Qibla direction of a location that has geographical latitude of 1.21 South and 108.411 East. Use the coding from the Google Colab website to generate the azimuth.

Exercise 3

Determine the Qibla direction for the city of Edinburgh, which has geographical latitude of 55.9533° North and 3.1883° West. This exercise is not available on the Google Colab website, meaning you need to make your own, based on the example provided.

Exercise 4

Calculate the azimuth of the Qibla for the city of Apia in Samoa with geographical latitude of 13.833 South and geographical longitude of 171.75 West.

Exercise 5

Calculate the azimuth of the Qibla for the city of Washington, DC, in the USA with geographical latitude of 38.904722 North and geographical longitude of 77.016389 West.

Istiwa' A'zam and Rashdul Qibla

6

Istiwa' A'zam refers to the astronomical event when the sun is directly above the Kaaba in Makkah. During this moment, the shadow of any vertical object anywhere on Earth points directly away from the Kaaba, making it one of the most accurate methods for determining the Qibla (prayer direction). This phenomenon is not affected by Earth's magnetic field, unlike compass-based Qibla methods, making it especially reliable.

Istiwa' A'zam happens twice a year, when the apparent path of the sun crosses the exact latitude of the Kaaba, which occurs on 28th May and 15th July approximately. On these dates, the sun passes directly overhead the Kaaba, around 12:16 p.m. (Saudi time).

At the time of Istiwa' A'zam: The sun is directly over the Kaaba. A vertical stick or object will cast a shadow that points exactly in the opposite direction of the Qibla. This method is free from magnetic disturbances, unlike a compass. Thus, shadows on these dates provide a natural, accurate Qibla direction, especially for mosques or homes without modern tools.

However, since Istiwa' A'zam only occurs twice a year, it is not always practical for everyday or long-term Qibla alignment. For this reason, Muslims use Rashdul Qibla, which refers to the general calculation of the Qibla direction using sun shadow. Rashdul Qibla refers to the determination of the Qibla direction using the sun's position and the direction of its shadow. When the sun's azimuth matches the azimuth of the Qibla, the shadow of any upright object will point directly opposite the Kaaba, effectively giving an accurate Qibla direction. This moment can be calculated for any location in the world, not just during the special events of Istiwa' A'zam (when the sun is directly above the Kaaba). Thus, Rashdul Qibla can be used on many dates throughout the year.

Modern tools like mobile compasses often suffer from magnetic interference, leading to significant inaccuracies, up to 20° in some cases. This could result in the prayer direction being off by over 1000 km from the Kaaba. In contrast, the sun's position can be precisely calculated using astronomical algorithms and ephemeris data. This makes Rashdul Qibla:

DOI: 10.1201/9781003649120-6

- Independent of magnetic errors.
- Applicable worldwide.
- Simple to verify using natural observation.

While Istiwa' A'zam only happens twice a year (around 28 May and 15 July), Rashdul Qibla can be calculated for any day using the method of finding when the sun's azimuth equals the Qibla azimuth for a given location. The position of the sun can be accurately calculated due to constant observation by the astronomer and improvement in ephemeris calculation by the astrometric researcher. The calculation of the time when the sun shadow is facing the Qibla is as follows. First step, calculate the first auxiliary angle to get the value of the sun's hour angle with the following formula:

$$\mathrm{U} = \tan^{-1}\left(\frac{1}{\tan - \theta \mathrm{x} \sin\varphi_{Location}}\right) \qquad \text{Equation 6.1}$$

then calculate the second auxiliary angle with the following formula

$$\mathrm{T} - \mathrm{U} = \cos^{-1}\left(\frac{\tan\delta x \cos U}{\tan\varphi_{Location}}\right) \qquad \text{Equation 6.2}$$

After the two auxiliary angles are obtained, then calculate the sun's hour angle using the following provisions:

$$t\_HMS) = \left(\begin{cases} -(T-U)+U, if\ U > 0 \\ |T-U|+U, if\ U \le 0 \end{cases}\right)/15 \qquad \text{Equation 6.3}$$

add the sun's hour angle (in hours) by 12 o'clock to get the solar time of Rashdul Qibla:

$$R = 12 + t\_HMS \qquad \text{Equation 6.4}$$

Finally, the Rashdul Qibla time according to local mean time can be obtained using the following formula:

$$Rashd\ alQibla\ Mean\ time = \begin{cases} R - EoT + \dfrac{TZ \times 15 - \lambda_{location}}{15}, if\ TZ > 0 \\ R - EoT - \dfrac{|TZ \times 15| - |\lambda_{location}|}{15}, if\ TZ \le 0 \end{cases}$$

Equation 6.5

Where θ is the Qibla direction, as calculated from the previous chapter. TZ is Time Zone of the location. δ is sun declination. EoT is Equation of Time. The formula of Equation of Time is complex and time consuming.

EQUATION OF TIME

The Equation of Time explains the difference between:

- Solar Time – time based on the sun's actual position in the sky (what a sundial shows)
- Clock Time – time shown by regular clocks (mean time)

This difference can be up to ±16 minutes. It changes every day of the year.

- Solar noon (when the sun is at its highest point) is not always exactly at 12:00 PM.
- The Equation of Time helps explain why the sun sometimes appears early or late.

TWO CAUSES OF THE DIFFERENCE

1. Earth's orbit is elliptical
   - Earth moves faster when close to the sun (January)
   - Earth moves slower when farther (July)
2. Earth's axis is tilted
   - The tilt (23.44°) changes the sun's path across the sky throughout the year

WHY DOES IT MATTER?

- Astronomy – correcting solar positions

- Sundials – to adjust for real sun time
- Islamic Prayer Times – calculating accurate Zuhur and other solar-based times
- Navigation – aligning maps and sun angles

SIMPLE DEFINITION

Equation of Time is equal to Solar time minus clock time. If EoT is positive, the sun is behind the clock. If EoT is negative, the sun is ahead of the clock

CALCULATION EQUATION OF TIME

There are several Python libraries that are able to calculate the Equation of Time. An example is pvlib. Pvlib requires installation from pip.

```
pip install pvlib
```

Once installed, pvlib only requires information of the day of the year. For example, to calculate the Equation of Time on January 14, 2024.

```
import pvlib
from datetime import datetime
import math
# define the date
date = datetime(2024, 1, 14)
# calculate the day of the year
day_of_year = date.timetuple().tm_yday
```

Then, to determine the result of the Equation of Time,

```
EoT = pvlib.solarposition.equation_of_time_pvcdrom(
day_of_year) / 60
print(f"Equation of Time on date {date} is {EoT}")
Equation of Time on date 2024-01-14 00:00:00 is
-0.14995874998914036
```

This Equation of Time is in decimal format, to convert as hour, minute, second.

```
# To Convert in Degree Form
```

```
degrees = int(EoT)
decimal_part = EoT - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
# Step 3: Convert the decimal part of minutes to
seconds
seconds = round((minutes_total - minutes) * 60)
print(f"{EoT}° = {degrees}° {minutes}' {seconds}"")
-0.14995874998914036° = 0° -8' -60"
```

The result is negative 8 minutes and 60 seconds. This means that the sun position is located ahead of the clock.

Exercise 1

Determine the Equation of Time for the date 26 May 2025.

Install the required libraries.

```
pip install pvlib
```

Input the date into Python variables.

```
Year = 2025
Month = 5
Day = 26
```

Input the date into the pvlib library

```
import pvlib
from datetime import datetime
import math
# define the date
date = datetime(Year, Month, Day)
# calculate the day of the year
day_of_year = date.timetuple().tm_yday
```

Input compute the EoT.

```
          EoT = pvlib.solarposition.equation_of_tim
          e_pvcdrom(day_of_year) / 60
print(f"Equation of Time on date {date} is {EoT}")
```

```
Equation of Time on date 2025-05-26 00:00:00 is
0.05194889980865643
```

Convert to hour.

```
# To Convert in Degree Form
degrees = int(EoT)
decimal_part = EoT - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
# Step 3: Convert the decimal part of minutes to
seconds
seconds = round((minutes_total - minutes) * 60)
print(f"{EoT}° = {degrees}° {minutes}′ {seconds}″")
0.05194889980865643° = 0° 3′ 7″
```

The Equation of Time during 26 May 2025 is 3 minutes 7 second positive.

SUN DECLINATION

The sun declination can be calculated using the Skyfield Library. First, the Skyfield library needs to be installed.

```
pip install skyfield
```

Skyfield requires some information to operate; it requires general import of which Skyfield function the user want to use; the ephemeris, observation object, location of the user. First, the general import of the Skyfield function. For this case, we want to use the *load* function, to load ephemeris data and *N, S, W, E* to load geographical direction data, and *wgs84* to load earth geographical location data. In Python this is written as,

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
```

Next, from the *load* function, determine which ephemeris and target that is going to be used. In this case, ephemeris 440s is used due to its small size and accuracy. Observation target is earth and sun, since this calculation only involves these two objects. In Python this is written as

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Then, Skyfield requires location input to calculate the sun position. Let's say we calculate from the position of Kuala Lumpur, Malaysia, with coordinates of 3.1319° N, 101.6841° E, during the date of 14 October 2025.

```
name_city = "Kuala Lumpur"
lat_location = 3.1319
long_location = 101.6841
year = 2025
month = 10
day = 14
```

Next is to give the information to Skyfield user location. In this case, the user is from earth and calculation is computed on lat_location and long_location coordinates.

```
kuala_lumpur = earth + wgs84.latlon(lat_location,
long_location, elevation_m=0)
```

Then, input the date of the calculated observation, from the computed position to the targeted position. The computed position is kuala_lumpur, while the targeted position is sun. The code, written in Python as

```
astro = kuala_lumpur.at(ts.utc(year, month, day)).
observe(sun)
```

astro means that the input is in astrometric position of the sun. To convert into apparent position, as observed by the observer at kuala_lumpur, is;

```
app = astro.apparent()
```

From this app data, we can calculate the right ascension, declination, and distance of the sun, as observed from earth. To determine the declination of the sun using Skyfield.

```
app = astro.apparent()
ra_dec,dec_app,d_app = app.radec()
print(dec_app)
-08deg 01' 18.7"
```

Declination of the sun at Kuala Lumpur on 14 October 2025 is −08° 01' 18.7".

Exercise 2

Determine sun declination at Cape Town South Africa, during 26 May 2025, where it has a time zone of UTC/GMT +2 hours.

Install skyfield library

```
pip install skyfield
```

Import Necessary Function

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
```

Load Ephemeris Data and Planet Objects

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Insert the variable

```
#insert the variable
name_city = "Cape Town"
lat_location = -33.9221
long_location = 18.4231
year = 2025
month = 5
day = 26
tz = 2
```

Determine the sun declination

```
kuala_lumpur = earth + wgs84.latlon(lat_location,
long_location, elevation_m=0)
astro = kuala_lumpur.at(ts.utc(year, month, day)).
observe(sun)
app = astro.apparent()
app = astro.apparent()
ra_dec,dec_app,d_app = app.radec()
print(dec_app)
+21deg 04' 00.3"
```

Declination of the sun at Cape Town South Africa, on 26 May 2025 is +21° 04' 00.3''.

CALCULATION OF RASHDUL KIBLAT

The calculation of Rashdul Kiblat involve calculating the Equation of Time and Sun Declination and the formula from 9 to 13. Using the previous example, on 26 May 2025, at Cape Town, where the Equation of Time is 0° 3′ 7″ and the sun declination is +21° 04' 00.3", we can determine the time when the sun shadow facing the Qibla direction.

First, determine the Qibla direction from the previous chapter coding.

Qibla Direction Coding

```
# Qibla Direction Calculation
φ_Location = -33.9221
λ_Location = 18.4231
φ_Kaabah = 21.4225
λ_Kaabah = 39.8262
Difference_Longitude = abs(λ_Location-λ_Kaabah )

#Calculation of Qibla Direction
import math
A = math.sin(math.radians(abs(Difference_Longitude)))
B = math.cos(math.radians(φ_Location))*math.tan(math.r
    adians(φ_Kaabah))
C = math.sin(math.radians(φ_Location)) * math.cos(m
    ath.radians(Difference_Longitude))
D = A/(B-C)
θ = math.degrees(math.atan(D))

#Determine the Azimuth of the Qibla
if Difference_Longitude > 180:
    delta_λ = 360 - Difference_Longitude
else:
    delta_λ = Difference_Longitude

if θ > 0:
   if λ_Location > λ_Kaabah:
       quadrant = "UB" # Utara Barat
   elif λ_Location <= λ_Kaabah:
       quadrant = "UT" # Utara Timur
   elif λ_Location < 0:
       if c >= 180:
           quadrant = "UB"
       else:
           quadrant = "UT"
```

```
elif θ < 0:
   if λ_Location > λ_Kaabah:
       quadrant = "SB" # Selatan Barat
   elif λ_Location <= λ_Kaabah:
        quadrant = "ST" # Selatan Timur
   elif λ_Location < 0:
       if c >= 180:
           quadrant = "SB"
       else:
           quadrant = "ST"

if quadrant == "UB":
   azimuth_kiblat = 360 - θ
elif quadrant == "SB":
   azimuth_kiblat = 180 - θ
elif quadrant == "UT":
   azimuth_kiblat = θ
elif quadrant == "ST":
   azimuth_kiblat = 180 + θ
print(azimuth_kiblat)
   23.354225930229862
```

The Qibla azimuth in Cape Town is 23.354225930229862.

To calculate Rashdul Qibla, from formula 9, determining U using Python code is written as

```
U = math.degrees(math.atan(1 / (math.tan(math.
radians(-azimuth_kiblat)) * math.sin(math.
radians(φ_Location)))))
print(U)
76.45187800936165
```

Sun declination, from previous coding is extracted from the variable dec_app. This is in degree format and cannot be included in computation. To express the sun declination in decimal format and make it computable, the code is

```
dec_app.degrees
```

Next, the value of T_U can be calculated using Equation 6.2 from sun declination and U variable. In Python this is written as,

```
T_U = math.degrees(math.acos(math.tan(math.radians(
dec_app.degrees)) * math.cos(math.radians(U)) / math.t
an(math.radians(lat_location))))
print(T_U)
97.71100015734693
```

After the T_U value is obtained, calculate the sun's hour angle (t_{HMS}) using Equation 6.3 using Python;

```
if U > 0:
   t_HMS = (-abs(T_U) + U) / 15
else:
   t_HMS = (abs(T_U) + U) / 15
print(t_HMS)
-1.4172748098656853
```

The Rashdul Qibla time, R in solar time, is obtained using Equation 6.4, and is expressed in Python as

```
R = 12 + t_HMS
print(R)
10.582725190134315
```

The value of R is in solar time. To change the direction of Qibla in solar time to local time, use Equation 6.5. Equation 6.5 requires data from time zone, I, and Equation of Time, EoT. In Python this is written as,

```
if tz > 0:
    Rashdul_Kiblat = R - EoT + (tz * 15 - long_
    location) / 15
else:
   Rashdul_Kiblat = R - EoT + (abs(tz * 15) -
   abs(long_location)) / 15
   print(Rashdul_Kiblat)
   11.302569623658991
```

The answer is in decimal. To convert to hour minute format,

```
# To Convert in Degree Form
degrees = int(Rashdul_Kiblat)
decimal_part = Rashdul_Kiblat - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
# Step 3: Convert the decimal part of minutes to
seconds
seconds = round((minutes_total - minutes) * 60)
print(f"{degrees}° {minutes}' {seconds}"")
11° 18' 9"
```

Rashdul Kiblat will occur, or The shadow of the sun, will point toward Qibla direction at 11:18 AM on 25 May 2025. At Cape Town South Africa.

Exercise 3

Determine the time of Rashdul Kiblat of Oslo in Norway, which has geographical latitude of 59.913333 North and 10.738889 East on July 30, 2025, that has Time zone UTC+1. The Qaabah geographical latitude is 21.4225 Noth, longitude is 39.8262 East.

Determine the Qibla direction

```
# Qibla Direction Calculation
φ_Location = 59.913333
λ_Location = 10.738889
φ_Kaabah = 21.4225
λ_Kaabah = 39.8262
Difference_Longitude = abs(λ_Location-λ_Kaabah )

#Calculation of Qibla Direction
import math

A = math.sin(math.radians(abs(Difference_Longitude)))
B = math.cos(math.radians(φ_Location))*math.tan(math.r
    adians(φ_Kaabah))
C = math.sin(math.radians(φ_Location)) * math.cos(m
    ath.radians(Difference_Longitude))
D = A/(B-C)
θ = math.degrees(math.atan(D))

#Determine the Azimuth of the Qibla
if Difference_Longitude > 180:
    delta_λ = 360 - Difference_Longitude
else:
    delta_λ = Difference_Longitude

if θ > 0:
   if λ_Location > λ_Kaabah:
       quadrant = "UB" # Utara Barat
   elif λ_Location <= λ_Kaabah:
       quadrant = "UT" # Utara Timur
   elif λ_Location < 0:
       if c >= 180:
           quadrant = "UB"
       else:
           quadrant = "UT"
elif θ < 0:
   if λ_Location > λ_Kaabah:
       quadrant = "SB" # Selatan Barat
```

```
    elif λ_Location <= λ_Kaabah:
        quadrant = "ST" # Selatan Timur
    elif λ_Location < 0:
        if c >= 180:
            quadrant = "SB"
        else:
            quadrant = "ST"

if quadrant == "UB":
    azimuth_kiblat = 360 - θ
elif quadrant == "SB":
    azimuth_kiblat = 180 - θ
elif quadrant == "UT":
    azimuth_kiblat = θ
elif quadrant == "ST":
    azimuth_kiblat = 180 + θ
print(azimuth_kiblat)
139.01065227776655
```

The qibla direction is 139.01. Then, determine the Equation of Time. Install pvlib first.

```
pip install pvlib
```

Then run the code,

```
import pvlib
from datetime import datetime
import math

Year = 2025
Month = 7
Day = 30

# define the date
date = datetime(Year, Month, Day)

# calculate the day of the year
day_of_year = date.timetuple().tm_yday
EoT = pvlib.solarposition.equation_of_time_pvcdrom(
day_of_year) / 60

print(f"Equation of Time on date {date} is {EoT}")
Equation of Time on date 2025-07-30 00:00:00 is
-0.10191142861300423
```

The Equation of Tme is −0.1019, then determine the solar declination. Install skyfield first.

```
pip install skyfield
```

then run the code to calculate Sun Declination

```
#Determine the Sun Declination
#Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
#Load Ephemeris Data and Planet Objects
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

#insert the variable
name_city = "Oslo "
lat_location = 59.913333
long_location = 10.738889
year = 2025
month = 7
day = 30
tz = 1
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
astro = location.at(ts.utc(year, month, day)).
observe(sun)
app = astro.apparent()
app = astro.apparent()
ra_dec,dec_app,d_app = app.radec()
print(dec_app)
+18deg 34' 49.6"
```

Determine the time of Rashdul Kiblat

```
# Rashdul Qiblat Computation
U = math.degrees(math.atan(1 / (math.tan(math.
    radians(-azimuth_kiblat)) * math.sin(math.
    radians(φ_Location)))))
T_U = math.degrees(math.acos(math.tan(math.radians(
      dec_app.degrees)) * math.cos(math.radians(U)) /
      math.tan(math.radians(lat_location))))
```

```
if U > 0:
   t_HMS = (-abs(T_U) + U) / 15
else:
   t_HMS = (abs(T_U) + U) / 15
R=12+t_HMS
if tz > 0:
    Rashdul_Kiblat = R - EoT + (tz * 15 - long_
    location) / 15
else:
   Rashdul_Kiblat = R - EoT + (abs(tz * 15) -
   abs(long_location)) / 15
# To Convert in Degree Form
degrees = int(Rashdul_Kiblat)
decimal_part = Rashdul_Kiblat - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
# Step 3: Convert the decimal part of minutes to
seconds
seconds = round((minutes_total - minutes) * 60)
print(f"{degrees}° {minutes}′ {seconds}″")
10° 22′ 17″
```

Rashdul Qiblat occurs at 10:22:17 in Osla on July 30, 2025.

Exercise 4

Use the Python program you've made to calculate the Rashdul Qibla for the city of Washington, DC, in the USA that has geographical latitude of 38.904722 North and geographical longitude of 77.016389 West on March 25, 2025, that has Time zone UTC-5.

Exercise 5

Use the Python program you've made to calculate the Rashdul Qibla for the city of Johannesburg in South Africa that has geographical latitude of 26.204444 South and geographical longitude of 28.045556 East on November 3, 2025, that has Time zone UTC+2.

Prayer Times Calculation

7

Prayer time calculation is based on the position of the sun. Each country has different methods of prayer time calculation. In Malaysia, the prayer time is calculated based on tabular extrapolation on the position of the solar, its hour angle, and Equation of Time.

The tabular extrapolation means that the calculation requires manual computation and cannot be automated for yearly determinations. Therefore, the calculation of prayer times in this does not use tabular extrapolation; instead, it is extracted directly from the sun's position, using the Skyfield library.

ZUHUR

The earliest time for Zuhr begins after solar noon, when the sun passes the local meridian and reaches its highest point in the sky. It is based on Prophet saying:

> *The time for Zuhr is when the sun has passed its zenith until a person's shadow is equal in length to his height.*
>
> Sahih Muslim (612)

The earliest time for Zuhr (Dhuhr) prayer begins just after the sun passes its zenith, known as the solar transit. To ensure the sun has fully crossed the meridian and begun its descent, an offset equal to half of the sun's apparent diameter is applied. This corresponds to approximately 1 minute and 4 seconds.

Using the Skyfield library in Python, the solar daily transit time can be accurately calculated based on the actual position of the sun. This method does not require the Equation of Time, as Skyfield already accounts for the sun's apparent motion and Earth's orbital variations. To calculate the Zuhr prayer

DOI: 10.1201/9781003649120-7

time. First, we need the corresponding location and date. For this case, we use London coordinates, which are 51.5072 North, 0.1276° West, time zone of GMT+1, during the date of 28 May 2025. Then, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
```

The import almanac is added since the function to find solar transit is located under the function almanac.

Put input into the variable

```
latitude = 51.5072
longitude = -0.1276
timezone = 1
day = 28
month = 5
year = 2025
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
```

Skyfield require range of calculate date. If we want to locate the time of the transit between 29 May 2025 to 30 May 2025, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, to find the time of solar transit

```
t = almanac.find_transits(location, sun, t0, t1)
print(t)
     <Time tt=[2460823.99928731]>
```

The result is in Julian Day Number. The extract hour, minute and seconds of solar transit from

```
hour_solar_transit = t.utc.hour
minutes_solar_transit = t.utc.minute
second_solar_transit = t.utc.second

print(hour_solar_transit)
print(minutes_solar_transit )
print(second_solar_transit)
[11.]
[57.]
[49.23954726]
```

This means that solar transit will occur at 11:57:49. However, this is not time-zone corrected. In addition, the Zuhur will occur at 1 minute 4 seconds after solar transit, which is 0.017778 in hour. Therefore, to make the correction based on actual Zuhur time.

```
zuhur_time = hour_solar_transit + (minutes_solar_
transit / 60) + (second_solar_transit / 3600 ) +
timezone + 0.017778
print(zuhur_time)
[12.98145565]
```

This means that after timezone and 1 minute 4 seconds correction, the Zuhr time is [12.98145565]. This is not in time format; to convert into time format:

```
# To Convert in Degree Form
degrees = int(zuhur_time )
decimal_part = float(zuhur_time) - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

seconds = round((minutes_total - minutes) * 60)

sun_astro = location.at(ts.utc(year, month, day,
hour_solar_transit, minutes_solar_transit, second_
solar_transit)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()

# Check if the sun is above the horizon at Zuhr time
if sun_alt.degrees <= 0:
    zuhur = "Zuhur Does Not Occur"
else:
```

```
        zuhur = f"Zuhur Occurs at {degrees}° {minutes}'
        {seconds}""

print(zuhur)
Zuhur Occurs at 12° 58′ 53″
```

This means that the zuhur prayer time at 51.5072 North, 0.1276° West, time zone of GMT+1, during the date of 28 May 2025 occurs at 12:58:53. The above code include counter measure should the Zuhur be calculated at a location near the North or South pole.

Exercise 1

Determine the Zuhr prayer time for Abuja, Nigeria, which is located at coordinates 9.0563° North, 7.4985° East, with a time zone of GMT+1, on the date of 1 January 2014.

Install skyfield

```
pip install skyfield
```

Import Necessary Function

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
```

Load Ephemeris Data and Planet Objects

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Input the Variable

```
lat_location = 9.0563
long_location = 7.4985
timezone = 1
day = 1
month = 1
year = 2014
```

Input info to location variable

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
```

Determine the Range of Data

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Find the time of Transit

```
t = almanac.find_transits(location, sun, t0, t1)
```

Extract Hour, Minutes, Seconds

```
hour_solar_transit = t.utc.hour
minutes_solar_transit = t.utc.minute
second_solar_transit = t.utc.second

print(hour_solar_transit)
print(minutes_solar_transit )
print(second_solar_transit)
```

Zuhur Time in decimal

```
zuhur_time = hour_solar_transit + (minutes_solar_
transit / 60) + (second_solar_transit / 3600 ) +
timezone + 0.017778
print(zuhur_time)
```

Zuhur in Time Format

```
degrees = int(zuhur_time )
decimal_part = zuhur_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

seconds = round((minutes_total - minutes) * 60)

sun_astro = location.at(ts.utc(year, month, day,
hour_solar_transit, minutes_solar_transit, second_
solar_transit)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()
# Check if the sun is above the horizon at zuhur time
```

```
if sun_alt.degrees <= 0:
    zuhur = "Zuhur Does Not Occur"
else:
   zuhur = f"Zuhur Occurs at {degrees}° {minutes}′
   {seconds}″"

print(zuhur)
Zuhur occurs at 12° 34′ 37″
```

This means that Zuhur prayer time at Abuja, Kenya coordinate, which are 9.0563 North, 7.4985 East, time zone of GMT+1, during the date of 1 January 2014 will occur at 13:05:08.

Exercise 2

Determine the Zuhr prayer time for Buenos Aires, Argentina, which is located at coordinates 34.6037° South, 58.3816° West, with a time zone of GMT–3, on the date of 5 November 2023.

Exercise 3

Determine the Zuhr prayer time for Vancouver, Canada, which is located at coordinates 49.2827° North, 123.1207° West, with a time zone of GMT–8, on the date of 10 September 2020.

ASAR

The beginning of Asar prayer time is based on the length of the sun's shadow. This is based on the prophetic saying;

> *...Then he prayed Asr when everything was similar (to the length of) its shadow...*
> *...Then he prayed Asr when the shadow of everything was about twice as long as it...*

There are several interpretations for the actual length of shadow for the start of the Asar prayer time. First, as adopted by the Asy-Syafie School of Islamic Thought, the length of the sun shadow for Asar prayer times is

$$\textbf{\textit{Asar Shadow Syafie}} = 1 \times \textbf{\textit{height of a stick}} \qquad \text{Equation 7.1}$$

Or written in Python as

```
sun_shadow_asar = 1
```

because we assume the height of the stick is 1, regardless of the unit. For Hanafi School of Islamic Thought, the length of the sun shadow for Asar prayer times is

$$Asar\ Shadow\ Hanafi = 2 \times height\ of\ a\ stick \qquad \text{Equation 7.2}$$

Or written in Python as

```
sun_shadow_asar = 2
```

Other opinion, as adopted by Malaysia, the length of the sun shadow for Asar prayer time is

$$Asar\ Shadow\ Others = Sun\ Shadow\ during\ solar\ transit + 1\ height\ of\ a\ stick \qquad \text{Equation 7.3}$$

Or written in Python as

```
sun_shadow_asar = 1 + sun_shadow_transit
```

To calculate the prayer time of Asar, we need to calculate the length of the sun shadow in a time loop from solar transit until the stipulated length of the sun shadow for Asar prayer time. For example, for the London coordinate, which are 51.5072 North, 0.1276° West, time zone of GMT+1, during the date of 28 May 2025, where the Asar sun shadow is Equation 7.3. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = 51.5072
long_location = -0.1276
```

```
timezone = 1
day = 28
month = 5
year = 2025
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
```

Skyfield require range of calculate date. If we want to locate the time of the transit between 29 May 2025 to 30 May 2025, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, to find the time of solar transit

```
t = almanac.find_transits(location, sun, t0, t1)
```

After finding the solar transit, determine the position of sun altitude at the time of the solar transit

```
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
sun_astro = location.at(ts.utc(year, month, day,
h, m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
```

In the above code, sun_astro is the astrometric position of the sun. The astrometric position needs to be converted to apparent position to determine the altitude of the sun. Therefore, the code sun_app = sun_astro.apparent() makes the conversion from astrometric to apparent position. Then, to determine the altitude of the sun, sun_alt, sun_az, distance = sun_app.altaz(), where sun_alt is the sun altitude, sun_az is the sun azimuth. distance is the apparent distance between the sun and earth.

```
print(sun_alt)
60deg 02' 14.0"
```

From the above code, we know that the altitude of the sun is 60° 02' 14.0". Then, determine the length of the sun shadow during transit, which is written as

```
sun_shadow_transit = 1/ (math.tan(math.radians(sun_alt
.degrees)))
print(sun_shadow_transit)
0.576484553268423
```

The result is 0.5764, indicating that at the time of solar transit, the length of the sun's shadow is approximately 57.64% of the stick's height, assuming the stick height is 1. Then to determine the sun shadow has reach the length from Equation 7.3.

```
sun_shadow_asar = 1 + sun_shadow_transit
print(sun_shadow_asar)
1.5764845532684229
```

At the time of solar transit, the length of the sun's shadow is approximately 0.5764, meaning it is only 57.64% of the stick's height. However, for the Asar prayer, the required shadow length is 1.5764, or 157.64% of the stick's height. This indicates that the sun must descend further in the sky before the Asar time begins. To accurately determine the time for Asar prayer, a time-based loop is implemented starting from the moment of solar transit. This loop checks the length of the shadow at each point in time until it reaches the required Asar length. To make the process more efficient, the loop is divided into three stages: hour loop, minute loop, and second loop, allowing the program to incrementally and precisely identify the exact time when the condition is met. To develop with hour time loop, with the rule of the loop the sun shadow does not pass the length of the Asar sun shadow,

```
# Start with hour
test = 1
while True:

  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day, h,
  m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow= 1 / math.tan(math.radians(sun_alt.
  degrees))
```

```
    if sun_alt.degrees <= 0:
      break
    if test > 24:
      break
    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    h += 1
    test +=1
    print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
    {sun_shadow} at hour {h}')

Sun Altitude [58.06080941], Sun_Shadow
0.6233945983146184 at hour [12.]
Sun Altitude [60.03162483], Sun_Shadow
0.5766145603035456 at hour [13.]
Sun Altitude [57.62354596], Sun_Shadow
0.6340429848026038 at hour [14.]
Sun Altitude [51.70606881], Sun_Shadow
0.7895804424162487 at hour [15.]
Sun Altitude [43.7299877], Sun_Shadow
1.0453443161558726 at hour [16.]
Sun Altitude [34.76344347], Sun_Shadow
1.4407721100751296 at hour [17.]
```

From the printed output above, we can see that the Sun's shadow exceeds the required Asar shadow length at hour 17. Therefore, we step back one hour and proceed with a minute-level time loop to more precisely determine the exact minute when the required shadow length is first reached.

```
# Once the condition is met for hours, move to minutes
h_asar = h - 1
test=1
while True:
  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day,
  h_asar, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow = 1 / math.tan(math.radians(sun_alt.
  degrees))

  if sun_alt.degrees <= 0:
    break
  if test > 1440:
    break
```

```
    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    m += 1
    test +=1
    print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
    {sun_shadow} at minute {m}')

    # Increment time in minutes
m_asar = m - 1

Sun Altitude [34.76344347], Sun_Shadow
1.4407721100751296 at minute 2
Sun Altitude [34.60973771], Sun_Shadow
1.449055572925344 at minute 3
Sun Altitude [34.45594556], Sun_Shadow
1.4574083986033666 at minute 4
Sun Altitude [34.30206933], Sun_Shadow
1.4658314538010262 at minute 5
Sun Altitude [34.14811128], Sun_Shadow
1.4743256209720872 at minute 6
Sun Altitude [33.99407369], Sun_Shadow
1.4828917986639925 at minute 7
Sun Altitude [33.8399588], Sun_Shadow
1.4915309018584026 at minute 8
Sun Altitude [33.68576885], Sun_Shadow
1.50024386232121 at minute 9
Sun Altitude [33.53150605], Sun_Shadow
1.5090316289614694 at minute 10
Sun Altitude [33.37717262], Sun_Shadow
1.517895168199992 at minute 11
Sun Altitude [33.22277074], Sun_Shadow
1.5268354643482442 at minute 12
Sun Altitude [33.06830259], Sun_Shadow
1.5358535199970826 at minute 13
Sun Altitude [32.91377034], Sun_Shadow
1.544950356416044 at minute 14
Sun Altitude [32.75917614], Sun_Shadow
1.554127013963914 at minute 15
Sun Altitude [32.60452212], Sun_Shadow
1.5633845525101653 at minute 16
Sun Altitude [32.44981041], Sun_Shadow
1.5727240518677097 at minute 17
```

From the printed output above, we can see that the Sun's shadow exceeds the required Asar shadow length at minute 23. Therefore, we step back one minute

and proceed with a second-level time loop to more precisely determine the exact minute when the required shadow length is first reached.

```
# Once the condition is met for minutes, move to
seconds
test=1
while True:
  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day,
  h_asar, m_asar, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow = 1 / math.tan(math.radians(sun_alt.
  degrees))

  if sun_alt.degrees <= 0:
    break
  if test > 86400:
    break
  if sun_shadow >= sun_shadow_asar:
    break # Exit the loop if the shadow length matches
    or exceeds the desired length
  s += 1
  test +=1
  print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
  {sun_shadow} at second {s}')

  # Increment time in seconds

s_asar = s
```

```
Sun Altitude [32.44981041], Sun_Shadow
1.5727240518677097 at second 2
Sun Altitude [32.44723141], Sun_Shadow
1.5728804108174916 at second 3
Sun Altitude [32.44465239], Sun_Shadow
1.573036792844485 at second 4
Sun Altitude [32.44207335], Sun_Shadow
1.5731931979536897 at second 5
Sun Altitude [32.4394943], Sun_Shadow
1.573349626150443 at second 6
Sun Altitude [32.43691523], Sun_Shadow
1.5735060774396996 at second 7
Sun Altitude [32.43433615], Sun_Shadow
1.573662551826798 at second 8
```

```
Sun Altitude [32.43175705], Sun_Shadow
1.5738190493167965 at second 9
Sun Altitude [32.42917794], Sun_Shadow
1.573975569914748 at second 10
Sun Altitude [32.42659881], Sun_Shadow
1.5741321136259998 at second 11
Sun Altitude [32.42401967], Sun_Shadow
1.5742886804555098 at second 12
Sun Altitude [32.42144051], Sun_Shadow
1.5744452704086311 at second 13
Sun Altitude [32.41886133], Sun_Shadow
1.5746018834903772 at second 14
Sun Altitude [32.41628214], Sun_Shadow
1.5747585197059564 at second 15
Sun Altitude [32.41370294], Sun_Shadow
1.5749151790605282 at second 16
Sun Altitude [32.41112372], Sun_Shadow
1.5750718615593058 at second 17
Sun Altitude [32.40854448], Sun_Shadow
1.5752285672074033 at second 18
Sun Altitude [32.40596523], Sun_Shadow
1.5753852960098962 at second 19
Sun Altitude [32.40338597], Sun_Shadow
1.5755420479721345 at second 20
Sun Altitude [32.40080668], Sun_Shadow
1.5756988230991047 at second 21
Sun Altitude [32.39822739], Sun_Shadow
1.5758556213961656 at second 22
Sun Altitude [32.39564808], Sun_Shadow
1.5760124428683018 at second 23
Sun Altitude [32.39306875], Sun_Shadow
1.5761692875208748 at second 24
Sun Altitude [32.39048941], Sun_Shadow
1.5763261553589674 at second 25
Sun Altitude [32.38791005], Sun_Shadow
1.5764830463876598 at second 26
```

From the printed output above, we can see that the sun's shadow exceeds the required Asar shadow length at seconds 26. Therefore, we step back one second. The Asar time, after the time loop, with timezone correction is

```
asar_time = (h_asar + (m_asar) / 60 + s_asar / 3600) +
timezone
print(asar_time)
[17.27388889]
```

This is observed in decimal. To convert to time format.

```
asar_time = float(asar_time)
degrees = int(asar_time)
decimal_part = asar_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees <= 0 or test >86400:
    asar = "Asar Does Not Occur"
else:
    asar = f"Asar Occurs at {degrees}° {minutes}'
    {seconds}""

print(asar)
Asar Occurs at 17° 16′ 26″
```

Therefore, asar prayer time at London, during 28 May 2025, based on Equation 7.3 occurs at 17:16:26. The second part is a counter measure should the calculated position is near the North or South Pole. Now, use the same London coordinate, which are 51.5072 North, 0.1276° West, time zone of GMT+1, during the date of 28 May 2025. Determine the prayer time of asar, where the asar sun shadow is Equation 7.1. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = 51.5072
long_location = -0.1276
timezone = 1
day = 28
month = 5
year = 2025
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
```

```
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, to find the time of solar transit

```
t = almanac.find_transits(location, sun, t0, t1)
```

After finding the solar transit, determine the position of sun altitude at the time of the solar transit

```
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
sun_astro = observer.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_alt)
60deg 02' 14.0"
```

Then, determine the length of the sun shadow during transit, which is written as

```
sun_shadow_transit = 1/ (math.tan(math.radians(sun_alt
.degrees)))
print(sun_shadow_transit)
0.576484553268423
```

Then to determine the sun shadow has reach the length from Equation 7.1.

```
sun_shadow_asar = 1
```

At the time of solar transit, the length of the sun's shadow is approximately 0.57, meaning it is only 57% of the stick's height. However, for the Asar prayer, the required shadow length is 1 or 100% of the stick's height. This indicates that the sun must descend further in the sky before the Asar time begins. Write

the hour time loop, with the rule of the loop the sun shadow does not pass the length of the afar sun shadow

```
# Start with hour
test=1
while True:

  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day, h,
  m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow= 1 / math.tan(math.radians(sun_alt.
  degrees))

  if sun_alt.degrees <= 0:
    break
  if test > 24:
    break

  if sun_shadow >= sun_shadow_asar:
    break # Exit the loop if the shadow length matches
    or exceeds the desired length
  h += 1
  print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
  {sun_shadow} at hour {h}')
```

Proceed with a minute-level time loop to more precisely determine the exact minute when the required shadow length is first reached.

```
# Once the condition is met for hours, move to minutes
h_asar = h - 1
test=1

while True:
  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day,
  h_asar, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow = 1 / math.tan(math.radians(sun_alt.
  degrees))
```

```
    if sun_alt.degrees <= 0:
      break
    if test > 1440:
      break

    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    m += 1
    print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
    {sun_shadow} at minute {m}')
    # Increment time in minutes
m_asar = m - 1
```

Proceed with a second-level time loop to more precisely determine the exact minute when the required shadow length is first reached.

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day,
  h_asar, m_asar, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  sun_shadow = 1 / math.tan(math.radians(sun_alt.
  degrees))
  if sun_alt.degrees <= 0:
    break
  if test > 86400:
    break

  if sun_shadow >= sun_shadow_asar:
  break # Exit the loop if the shadow length matches
  or exceeds the desired length
  s += 1
  print(f'Sun Altitude {sun_alt.degrees}, Sun_Shadow
  {sun_shadow} at second {s}')

  # Increment time in seconds

s_asar = s
```

The asar time, after the time loop, with timezone correction is

```
asar_time = (h_asar + (m_asar) / 60 + s_asar / 3600) +
timezone
print(asar_time)
```

This is observed in decimal. To convert to time format.

```
asar_time = float(asar_time)
degrees = int(asar_time)
decimal_part = asar_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

seconds = round((minutes_total - minutes) * 60)
if sun_alt.degrees <= 0 or test >86400:
    asar = "Asar Does Not Occur"
else:
    asar = f"Asar Occurs at {degrees}° {minutes}'
    {seconds}""

print(asar)
Asar Occurs at 15° 52' 6"
```

Therefore, asar prayer time at London, during 28 May 2025, based on Equation 7.1 occurs at 15:52:06. Finally, use the same London coordinate, which are 51.5072 North, 0.1276° West, time zone of GMT+1, during the date of 28 May 2025. Determine the prayer time of asar, where the asar sun shadow is Equation 7.2. Following from the above code, the change is Equation 7.2. Where,

```
sun_shadow_asar = 2
```

The hour time loop, with the rule of the loop the sun shadow does not pass the length of the afar sun shadow

```
# Start with hour
test = 1
while True:

  # Calculate the shadow length
  sun_astro = location.at(ts.utc(year, month, day, h,
  m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
```

```
    sun_shadow= 1 / math.tan(math.radians(sun_alt.
    degrees))

    if sun_alt.degrees <= 0:
      break
    if test > 24:
      break

    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    h += 1

  # Once the condition is met for hours, move to minutes
  h_asar = h - 1
  test=1
  while True:
    # Calculate the shadow length
    sun_astro = location.at(ts.utc(year, month, day,
    h_asar, m, s)).observe(sun)
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    sun_shadow = 1 / math.tan(math.radians(sun_alt.
    degrees))

    if sun_alt.degrees <= 0:
      break
    if test > 1440:
      break

    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    m += 1

    # Increment time in minutes
  m_asar = m - 1
  test = 1

  # Once the condition is met for minutes, move to
  seconds
  while True:
    # Calculate the shadow length
    sun_astro = location.at(ts.utc(year, month, day,
    h_asar, m_asar, s)).observe(sun)
```

```
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    sun_shadow = 1 / math.tan(math.radians(sun_alt.
    degrees))

    if sun_alt.degrees <= 0:
      break
    if test > 86400:
      break

    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length matches
      or exceeds the desired length
    s += 1

    # Increment time in seconds

  s_asar = s
```

The asar time, after the time loop, with timezone correction is

```
asar_time = (h_asar + (m_asar) / 60 + s_asar / 3600) +
timezone
print(asar_time)
```

This is observed in decimal. To convert to time format.

```
asar_time = float(asar_time)
degrees = int(asar_time)
decimal_part = asar_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
print(f"{degrees}° {minutes}′ {seconds}″")
17° 53′ 55″
```

Therefore, asar prayer time at London, during 28 May 2025, based on Equation 7.2 occurs at 17:53:55.

Exercise 1

Using the coordinates of Cairo, which are 30.0444° North, 31.2357° East, and a time zone of GMT+2, determine the Asar prayer time for the date 10 March 2025.

Use Equation 14 to compute the shadow length required at Asar.

Exercise 2

Using the coordinates of Kuala Lumpur, which are 3.1390° North, 101.6869° East, and a time zone of GMT+8, determine the Asar prayer time for the date 21 September 2025.

Apply Equation 15 to evaluate the required shadow length for Asar.

Exercise 3

Using the coordinates of New York City, which are 40.7128° North, 74.0060° West, and a time zone of GMT−4, determine the Asar prayer time for the date August 5, 2025.

Use Equation 16 to find when the shadow length meets the Asar requirement.

MAGHRIB

The beginning of Maghrib prayer time is based on the position of the sunset. This is based on the prophetic saying;

> *...Then he prayed Maghrib when the sun had set and the fasting person breaks fast....*

The position of the sun must be completely below the horizon to mark the beginning of Maghrib prayer. This means that the upper limb of the sun must have fully disappeared. However, factors such as atmospheric refraction and the observer's elevation above sea level can affect the apparent position of the sun at sunset. These variables must be carefully considered when calculating the precise time for Maghrib prayer. The determination of Maghrib prayer time is based on the find_settings function. Skyfield uses the official definition of sunrise and sunset from the United States Naval Observatory, which defines them as the moment when the center of the sun is 50 arcminutes below the horizon, to account for both the average solar radius of 16 arcminutes and for roughly 34 arcminutes of atmospheric refraction at the horizon. To determine the time of the maghrib prayer time, assuming the user is located at Sydney Tower Eye, with Latitude 33.8688° South, Longitude 151.2093° East, on 23

March 2025, where the timezone in effect is GMT + 11, with elevation of 350 meter. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = -33.8688
long_location = 151.2093
timezone = 11
day = 23
month = 3
year = 2025
ele = 350
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
```

```
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunset

```
t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
print(h,m,s)
[8.] [5.] [28.44139129]
```

The time of the sunset is 8:05:28.44, at UTC time. To convert into local time with GMT + 11.

```
maghrib_time = float(h + m / 60 + s / 3600 + timezone)
print(maghrib_time)
19.09123372
```

When calculating prayer times near sunset, there may be instances where the computed time exceeds 24 hours. This typically occurs due to the addition of the local time zone offset to the base time (often in UTC). To correct this and ensure the resulting prayer time remains within a 24-hour format, the following code is used to normalize the time

```
maghrib_time %= 24 # Ensure 24-hour clock format
print(maghrib_time)
```

From this moment, we can convert into time format,

```
maghrib_time = float(maghrib_time)
degrees = int(maghrib_time)
decimal_part = maghrib_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

sun_astro = location.at(ts.utc(year, month, day, h, m,
s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()
if sun_alt.degrees >= 0:
    maghrib = "Maghrib Does Not Occur"
```

```
else:
    maghrib = f"Maghrib Occurs at {degrees}° {minutes}'
    {seconds}""
print(maghrib)
Maghrib Occurs at 19° 5′ 28″
```

This means that Maghrib at Sydney Tower Eye, with Latitude 33.8688° South, Longitude 151.2093° East, on 23 March 2025, where the timezone in effect is GMT + 11, with elevation of 350 meter occur at 19:5:28.

Exercise 1

Determine the Maghrib prayer time on 15 July 2025 for a user located at the KL Tower, Malaysia, with the following details:

- Latitude: 3.1579° North
- Longitude: 101.7123° East
- Elevation: 300 meters
- Time Zone: GMT +8

Exercise 2

Calculate the Maghrib prayer time for Cairo, Egypt on 10 October 2025, with the following coordinates:

- Latitude: 30.0444° North
- Longitude: 31.2357° East
- Elevation: 75 meters
- Time Zone: GMT +2

Exercise 3

On 1 January 2025, determine the time of Maghrib prayer for a user located at the Empire State Building, New York, USA:

- Latitude: 40.7484° North
- Longitude: 73.9857° West
- Elevation: 381 meters
- Time Zone: GMT −5 (Standard Time, no DST)

Exercise 4

Compute the Maghrib prayer time for a location in Makkah, Saudi Arabia, on 28 May 2025:

- Latitude: 21.4225° North
- Longitude: 39.8262° East
- Elevation: 277 meters
- Time Zone: GMT +3

ISYA'

The beginning of Isya' prayer time is based on the position of the sky condition after sunset. This is based on the prophetic saying;

> *...Then he prayed Isha when the twilight had vanished....*

The disappearance of twilight from the sky is caused by the sun's continued descent below the horizon after sunset. At a certain angle, known as the solar depression angle, the sun's rays are no longer refracted by the atmosphere in a way that illuminates the night sky. When this critical angle is reached, the sky becomes completely dark, marking the beginning of the 'Isha (Isya') prayer time. However, Islamic scholars differ in their opinions regarding the exact degree of solar depression that signifies the start of 'Isha. While many adopt angles between 15° and 18° below the horizon, there is no unanimous agreement, and various regions apply different standards based on jurisprudential reasoning and observational studies. To determine the Isya' prayer time, at Bangkok where Latitude 13.7563° North, Longitude 100.5018° East and Time Zone GMT+7, with elevation of 150 meter on 16 June 2025 for 18 degrees of solar depression during Isya'. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = 13.7563
long_location = 100.5018
```

```
timezone = 7
day = 16
month = 6
year = 2025
ele = 150
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunset

```
t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
print(h,m,s)
[11.] [48.] [36.56323494]
```

Since the 'Isha (Isya') prayer occurs after sunset, it is necessary to perform a time-based loop starting from the moment of sunset and continuing until the solar altitude reaches 18 degrees of solar depression. The time loop operation is similar with the previous asar prayer time loop operation. First determine the solar altitude,

```
sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_alt)
-01deg 06' 01.2"
```

To develop with hour time loop, with the rule of the loop the solar degree does not pass the altitude of – 18.

```
# Start with hour
test=1
while True:

  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day, h,
  m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  isha_angle = 18
  elevation_correction = 0.0293 * math.sqrt(ele)
  isha_angle_actual = -isha_angle - elevation_correction

  if sun_alt.degrees >= 0:
    break
  if test > 24:
    break

  if sun_alt.degrees <= isha_angle_actual:
    break # Exit the loop if the solar altitude
    located below -18 degree
  print(f'Sun Altitude {sun_alt.degrees}, at hour
  {h}')
  h += 1
  test += 1
Sun Altitude [-1.23504713], at hour [11.]
Sun Altitude [-14.26400679], at hour [12.]
```

In the above code, **if sun_alt.degrees <= isha_angle_actual:** is used to determine whether the solar altitude has reach below 18 degree of depression. In

this example, the elevation factor is considered for high-altitude locations. When calculating prayer times, particularly for Isha and Fajr, the position of the sun below the horizon is critical. Typically, the angle used for Isha is when the sun is 18 degrees below the horizon. However, this standard angle is based on observations made at sea level.

At higher elevations, the situation changes due to the geometry of the Earth and the observer's horizon. When a person is located on a mountain or at a high-altitude location, their view of the sky is broader, and they are physically above a portion of the atmosphere. As a result, the sun appears to set faster, and darkness arrives sooner compared to someone at sea level. To accurately reflect this earlier onset of night, the Isha angle must be adjusted slightly downward, meaning a larger depression angle. This is why we include an *elevation correction factor* in the calculation. The correction is often calculated using the formula:

$$\textbf{elevation correction} = 0.0293 \times \sqrt{\left(\textbf{elevation in meters}\right)} \qquad \text{Equation 7.4}$$

This value is subtracted from the standard Isha angle (e.g., −18°), giving a slightly steeper actual angle for high-altitude locations. For instance, at 4000 meters elevation, the correction might be around 1.85 degrees, so the adjusted angle becomes approximately −19.85 degrees. This adjustment ensures that the calculated Isha time matches the true observable darkness experienced by someone at that elevation. Without applying this correction, the prayer time would be inaccurately delayed, especially in mountainous regions. From the output, the sun altitude reach angle below −18 degrees at hour 12. Then proceed with a minute-level time loop to more precisely determine the exact minute when the solar altitude is first reached.

```
# Once the condition is met for hours, move to minutes
h_isya = h - 1
test=1

while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_isya, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun

  if sun_alt.degrees >= 0:
    break
  if test > 1440:
    break
```

```
    if sun_alt.degrees <= isha_angle_actual:
      break # Exit the loop if the solar altitude
      located below -18 degree
    m += 1
    print(f'Sun Altitude {sun_alt.degrees}, at minute
    {m}')

    # Increment time in minutes
m_isya = m - 1
Sun Altitude [-14.26400679], at minute [49.]
Sun Altitude [-14.47642833], at minute [50.]
Sun Altitude [-14.68866518], at minute [51.]
Sun Altitude [-14.90071568], at minute [52.]
Sun Altitude [-15.11257813], at minute [53.]
Sun Altitude [-15.32425085], at minute [54.]
Sun Altitude [-15.53573212], at minute [55.]
Sun Altitude [-15.74702021], at minute [56.]
Sun Altitude [-15.95811338], at minute [57.]
Sun Altitude [-16.16900984], at minute [58.]
Sun Altitude [-16.37970783], at minute [59.]
Sun Altitude [-16.59020554], at minute [60.]
Sun Altitude [-16.80050114], at minute [61.]
Sun Altitude [-17.0105928], at minute [62.]
Sun Altitude [-17.22047865], at minute [63.]
Sun Altitude [-17.43015683], at minute [64.]
Sun Altitude [-17.63962543], at minute [65.]
Sun Altitude [-17.84888254], at minute [66.]
Sun Altitude [-18.05792617], at minute [67.]
Sun Altitude [-18.26675447], at minute [68.]
```

From the printed output above, we can see that the sun altitude reaches below the required Isya' depression degree at minute 68. Therefore, we step back one minute and proceed with a second-level time loop to more precisely determine the exact minute when the required Isya' depression degree is first reached.

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_isya, m_isya, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
```

```
    if sun_alt.degrees >= 0:
      break
    if test > 86400:
      break

    if sun_alt.degrees <= isha_angle_actual:
      break # Exit the loop if the solar altitude
      located below -18 degree
    s += 1
    print(f'Sun Altitude {sun_alt.degrees}, at second
    {s}')

    # Increment time in seconds

  s_isya = s
```

```
Sun Altitude [-18.26675447], at second [37.56323494]
Sun Altitude [-18.2702331], at second [38.56323494]
Sun Altitude [-18.27371168], at second [39.56323494]
Sun Altitude [-18.27719019], at second [40.56323494]
Sun Altitude [-18.28066865], at second [41.56323494]
Sun Altitude [-18.28414704], at second [42.56323494]
Sun Altitude [-18.28762538], at second [43.56323494]
Sun Altitude [-18.29110365], at second [44.56323494]
Sun Altitude [-18.29458186], at second [45.56323494]
Sun Altitude [-18.29806001], at second [46.56323494]
Sun Altitude [-18.30153811], at second [47.56323494]
Sun Altitude [-18.30501614], at second [48.56323494]
Sun Altitude [-18.30849411], at second [49.56323494]
Sun Altitude [-18.31197202], at second [50.56323494]
Sun Altitude [-18.31544987], at second [51.56323494]
Sun Altitude [-18.31892766], at second [52.56323494]
Sun Altitude [-18.32240539], at second [53.56323494]
Sun Altitude [-18.32588306], at second [54.56323494]
Sun Altitude [-18.32936066], at second [55.56323494]
Sun Altitude [-18.33283821], at second [56.56323494]
Sun Altitude [-18.3363157], at second [57.56323494]
Sun Altitude [-18.33979312], at second [58.56323494]
Sun Altitude [-18.34327049], at second [59.56323494]
Sun Altitude [-18.34674779], at second [60.56323494]
Sun Altitude [-18.35022504], at second [61.56323494]
Sun Altitude [-18.35370222], at second [62.56323494]
Sun Altitude [-18.35717935], at second [63.56323494]
```

From the printed output above, we can see that the solar altitude exceeds the required Isya' Solar Depression at second 63. Therefore, we step back one second. The Isya' time, after the time loop, with timezone correction is

```
isya_time = (h_isya + (m_isya) / 60 + s_isya /
3600)+timezone
print(isya_time)
[20.13444444]
```

To correct this and ensure the resulting prayer time remains within a 24-hour format,

```
isya_time %= 24 # Ensure 24-hour clock format
print(isya_time)
```

To convert into time format,

```
isya_time = float(isya_time)
degrees = int(isya_time)
decimal_part = isya_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    isya = "Isya' Does Not Occur"
else:
    isya = f"Isya' Occurs at {degrees}° {minutes}'
    {seconds}""

print(isya)
Isya' Occurs at 20° 8' 4"
```

Isya prayer time at Bangkok where Latitude 13.7563° North, Longitude 100.5018° East and Time Zone GMT+7, with elevation of 150 meter on 16 June 2025 for 18 degree of solar depression is Isya' is at 20:08:04.

Exercise 1

Determine the 'Isya prayer time at Jakarta, Indonesia, where Latitude: 6.2088° South, Longitude: 106.8456° East, Time Zone: GMT+7, Elevation: 50 meters, Date: 21 July 2025, Assume 'Isya begins at a solar depression of –17°.

First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = -6.2088
long_location = 106.8456
timezone = 7
day = 21
month = 7
year = 2025
ele = 50
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
```

```
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunset

```
t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
print(h,m,s)
[10.] [54.] [23.66874027]
```

Since the 'Isha (Isya') prayer occurs after sunset, it is necessary to perform a time-based loop starting from the moment of sunset and continuing until the solar altitude reaches 18 degrees of solar depression. The time loop operation is similar with the previous asar prayer time loop operation. First determine the solar altitude,

```
sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_alt)
-00deg 58' 37.5"
```

To develop with hour time loop, with the rule of the loop the solar degree does not pass the altitude of – 17.

```
# Start with hour

test = 1
while True:

  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h+1, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  isha_angle = 17
  elevation_correction = 0.0293 * math.sqrt(ele)
  isha_angle_actual = -isha_angle - elevation_correction

  if sun_alt.degrees >= 0:
    break
  if test > 24:
    break
```

```
    if sun_alt.degrees <= isha_angle_actual:
      break # Exit the loop if the solar altitude
      located below -18 degree
    h = h + 1
```

Then proceed with a minute-level time loop to more precisely determine the exact minute when the solar altitude is first reached.

```
# Once the condition is met for hours, move to minutes
h_isya = h - 1
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_isya, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun

  if sun_alt.degrees >= 0:
    break
  if test > 1440:
    break

  if sun_alt.degrees <= isha_angle_actual:
     break # Exit the loop if the solar altitude
     located below -18 degree
  m += 1

  # Increment time in minutes
m_isya = m - 1
```

Proceed with a second-level time loop to more precisely determine the exact minute when the required Isya' depression degree is first reached.

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_isya, m_isya, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
```

```
    if sun_alt.degrees >= 0:
      break
    if test > 86400:
      break

    if sun_alt.degrees <= isha_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
    s += 1
    print(f'Sun Altitude {sun_alt.degrees}, {isha_angle_
    actual} at second {s}')

    # Increment time in seconds

s_isya = s
```

Then, we step back one second. The Isya' time, after the time loop, with time-zone correction is

```
isya_time = (h_isya + (m_isya) / 60 + s_isya /
3600)+timezone
print(isya_time)
[19.05666667]
```

To correct this and ensure the resulting prayer time remains within a 24-hour format,

```
isya_time %= 24 # Ensure 24-hour clock format
print(isya_time)
[19.05666667]
```

To convert into time format,

```
isya_time = float(isya_time)
degrees = int(isya_time)
decimal_part = isya_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    isya = "Isya' Does Not Occur"
else:
    isya = f"Isya' Occurs at {degrees}° {minutes}'
    {seconds}""
```

```
print(isya)
Isya′ Occurs at 19° 3′ 25″
```

‘Isya prayer time at Jakarta, Indonesia, where Latitude: 6.2088° South, Longitude: 106.8456° East, Time Zone: GMT+7, Elevation: 50 meters, Date: 21 July 2025, Assume ‘Isya begins at a solar depression of –17 is at 19:3:24.

Exercise 2

Determine the ‘Isya prayer time at Istanbul, Turkey, where Latitude: 41.0082° North, Longitude: 28.9784° East, Time Zone: GMT+3, Elevation: 40 meters, Date: 15 August 2025. Use a solar depression angle of –16° to mark the beginning of ‘Isya prayer.

Exercise 3

Determine the ‘Isya prayer time at Tokyo, Japan, with the following details, Latitude: 35.6762° North, Longitude: 139.6503° East, Elevation: 40 meters, Time Zone: GMT+9, Date: 16 June 2025. Assume ‘Isya begins when the solar depression angle reaches 20° below the horizon.

SYURUK

Syuruk is the end of Subh prayer time. It is based on the timing of the sunrise. This originates from the prophetic saying;

> *...The beginning of the time for Fajr is when Fajr begins, and its end is when the sun rises.*

Sunrise from the Syuruk is the first visibility of the upper limb of the sun. This means that the upper limb of the sun must be fully visible from view. Similarly, the Maghrib refraction and elevation factor also need to be considered. To determine the time of the Syuruk, assuming the user is located at Sydney Tower Eye, with Latitude 33.8688° South, Longitude 151.2093° East, on 23 March 2025, where the time zone in effect is GMT + 11, with elevation of 350 meter. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = -33.8688
long_location = 151.2093
timezone = 11
day = 23
month = 3
year = 2025
ele = 350
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day-1)
t1 = ts.utc(year, month, day)
```

Since Syuruk occurs in the morning, we assume the calculation from the day before. Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
```

```
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunrise

```
t, y = almanac.find_risings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
print(h,m,s)
[19.] [57.] [29.52933632]
```

The time of the sunset is 19:57:29, at UTC time. To convert into local time with GMT + 11.

```
syuruk_time = float(h + m / 60 + s / 3600 + timezone)
print(syuruk_time)
30.958202593423238
```

When calculating times near sunrise, there may be instances where the computed time exceeds 24 hours; the above example is the case. To correct this and ensure the resulting prayer time remains within a 24-hour format, the following code is used to normalize the time

```
syuruk_time %= 24 # Ensure 24-hour clock format
print(syuruk_time)
6.958202593423238
```

From this moment, we can convert into time format,

```
syuruk_time = float(syuruk_time)
degrees = int(syuruk_time)
decimal_part = syuruk_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
sun_astro = location.at(ts.utc(year, month, day, h, m,
s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()
sun_alt, _, _ = sun_astro.apparent().altaz()
if sun_alt.degrees >= 0:
    syuruk = "Syuruk Does Not Occur"
else:
```

```
        syuruk = f"Syuruk Occurs at {degrees}° {minutes}'
        {seconds}"

print(syuruk)
Syuruk Occurs at 6° 57' 30
```

This means that syuruk at Sydney Tower Eye, with Latitude 33.8688° South, Longitude 151.2093° East, on 23 March 2025, where the timezone in effect is GMT + 11, with elevation of 350 meters occurs at 06:57:30.

Exercise 1

Determine the Syuruk (sunrise end) time on 15 July 2025 for a user located in central Tokyo, Japan, with the following location details:

- Latitude: 35.6762° North
- Longitude: 139.6503° East
- Elevation: 40 meters
- Time Zone: GMT +9

Exercise 2

Calculate the Syuruk time for Cape Town, South Africa, on 10 October 2025. Use the following coordinates:

- Latitude: 33.9249° South
- Longitude: 18.4241° East
- Elevation: 15 meters
- Time Zone: GMT +2

Exercise 3

On 1 January 2025, determine the Syuruk prayer time for a user located in Rio de Janeiro, Brazil:

- Latitude: 22.9068° South
- Longitude: 43.1729° West
- Elevation: 5 meters
- Time Zone: GMT −3

Exercise 4

Compute the Syuruk time for Berlin, Germany, on 28 May 2025 using the following details:

- Latitude: 52.5200° North
- Longitude: 13.4050° East
- Elevation: 34 meters
- Time Zone: GMT +2 (Daylight Saving Time in effect)

SUBH

The beginning of Subh' prayer time is based on the position of the sky condition before sunrise. This is based on the prophetic saying.

> *...Then he prayed Fajr when Fajr (dawn) began...*
> *...then he prayed Subh when the land glowed*

The appearance of twilight from the sky is caused by the sun's continued ascent below the horizon before sunrise. At a certain angle, known as the solar depression angle, the sun's rays begin to be refracted by the atmosphere in a way that illuminates the night sky. When this critical angle is reached, the first dim light appears horizontally in the sky, marking the beginning of the Subh prayer time. However, Islamic scholars differ in their opinions regarding the exact degree of solar depression that signifies the start of Subh. While many adopt angles between 15° and 20° below the horizon, there is no unanimous agreement, and various regions apply different standards based on jurisprudential reasoning and observational studies. To determine the Subh prayer time in Bangkok, where Latitude 13.7563° North, Longitude 100.5018° East and Time Zone GMT+7, with elevation of 150 meter on 16 June 2025 for 18 degrees of solar depression during Subh, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = 13.7563
long_location = 100.5018
timezone = 7
day = 16
month = 6
year = 2025
ele = 150
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day-1)
t1 = ts.utc(year, month, day)
```

Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunrise

```
t, y = almanac.find_risings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
print(h,m,s)
[22.] [48.] [48.62638382]
```

Since the Subh prayer occurs before sunrise, it is necessary to perform a time-based loop starting from the moment of sunrise and backtrack until the solar altitude reaches 18 degrees of solar depression. The time loop operation is similar with the previous Asar prayer time loop operation. First determine the solar altitude,

```
sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_alt)
-01deg 27' 11.4"
```

To develop with hour time loop, with the rule of the loop the solar degree does not pass the altitude of – 18.

```
# Start with hour
test = 1
while True:

  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day, h,
  m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  subh_angle = 18
  elevation_correction = 0.0293 * math.sqrt(ele)
  subh_angle_actual = -subh_angle
  - elevation_correction

  if sun_alt.degrees >= 0:
    break
  if test > 24:
    break
```

```
    if sun_alt.degrees <= subh_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
    h -= 1
    print(f'Sun Altitude {sun_alt.degrees}, at hour
    {h}')

Sun Altitude [-1.27408764], at hour [21.]
Sun Altitude [-14.3007989], at hour [20.]
```

In the above code, if sun_alt.degrees <= subh_angle_actual is used to determine whether the solar altitude has reach below 18 degree of depression. From the output, the sun altitude reach angle below −18 degrees at hour 20. Then proceed with a minute-level time loop to more precisely determine the exact minute when the solar altitude is first reached.

```
# Once the condition is met for hours, move to minutes
h_subh = h + 1
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_subh, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun

  if sun_alt.degrees >= 0:
    break
  if test > 1440:
    break

  if sun_alt.degrees <= subh_angle_actual:
         break # Exit the loop if the solar altitude
         located below
  m -= 1
  print(f'Sun Altitude {sun_alt.degrees}, at minute
  {m}')

  # Increment time in minutes
m_subh = m + 1

Sun Altitude [-14.3007989], at minute [47.]
Sun Altitude [-14.51317791], at minute [46.]
Sun Altitude [-14.72537202], at minute [45.]
Sun Altitude [-14.93737954], at minute [44.]
```

```
Sun Altitude [-15.14919881], at minute [43.]
Sun Altitude [-15.36082811], at minute [42.]
Sun Altitude [-15.57226573], at minute [41.]
Sun Altitude [-15.78350995], at minute [40.]
Sun Altitude [-15.994559], at minute [39.]
Sun Altitude [-16.20541111], at minute [38.]
Sun Altitude [-16.41606451], at minute [37.]
Sun Altitude [-16.62651737], at minute [36.]
Sun Altitude [-16.83676789], at minute [35.]
Sun Altitude [-17.04681421], at minute [34.]
Sun Altitude [-17.25665447], at minute [33.]
Sun Altitude [-17.4662868], at minute [32.]
Sun Altitude [-17.67570929], at minute [31.]
Sun Altitude [-17.88492002], at minute [30.]
Sun Altitude [-18.09391709], at minute [29.]
Sun Altitude [-18.30269847], at minute [28.]
```

From the printed output above, we can see that the sun altitude reaches below the required Subh depression degree at minute 28. Therefore, we step back one minute and proceed with a second-level time loop to more precisely determine the exact minute when the required Subh depression degree is first reached.

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_subh, m_subh, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  if sun_alt.degrees >= 0:
    break
  if test > 86400:
    break

  if sun_alt.degrees <= subh_angle_actual:
        break # Exit the loop if the solar altitude
        located below
  s -= 1
  print(f'Sun Altitude {sun_alt.degrees}, at second
  {s}')

  # Increment time in seconds

s_subh = s +1
```

```
Sun Altitude [-18.30269847], at second [47.62638382]
Sun Altitude [-18.30617632], at second [46.62638382]
Sun Altitude [-18.30965412], at second [45.62638382]
Sun Altitude [-18.31313185], at second [44.62638382]
Sun Altitude [-18.31660952], at second [43.62638382]
Sun Altitude [-18.32008713], at second [42.62638382]
Sun Altitude [-18.32356467], at second [41.62638382]
Sun Altitude [-18.32704216], at second [40.62638382]
Sun Altitude [-18.33051959], at second [39.62638382]
Sun Altitude [-18.33399696], at second [38.62638382]
Sun Altitude [-18.33747427], at second [37.62638382]
Sun Altitude [-18.34095151], at second [36.62638382]
Sun Altitude [-18.3444287], at second [35.62638382]
Sun Altitude [-18.34790582], at second [34.62638382]
Sun Altitude [-18.35138289], at second [33.62638382]
Sun Altitude [-18.35485989], at second [32.62638382]
Sun Altitude [-18.35833683], at second [31.62638382]
```

From the printed output above, we can see that the solar altitude exceeds the required Isy' Solar Depression at second 31. Therefore, we step back one second. The Isya' time, after the time loop, with timezone correction is

```
subh_time = (h_subh + (m_subh) / 60 + s_subh /
3600)+timezone
print(subh_time)
[28.4925]
```

To correct this and ensure the resulting prayer time remains within a 24-hour format,

```
subh_time %= 24 # Ensure 24-hour clock format
print(subh_time)
[4.4925]
```

To convert into time format,

```
subh_time = float(subh_time)
degrees = int(subh_time)
decimal_part = subh_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    subh = "Subuh Does Not Occur"
else:
```

```
        subh = f"Subuh Occurs at {degrees}° {minutes}'
        {seconds}""

print(subh)
Subuh Occurs at 4° 29' 33"
```

Subh prayer time at Bangkok where Latitude 13.7563° North, Longitude 100.5018° East and Time Zone GMT+7, with elevation of 150 meter on 16 June 2025 for 18 degree of solar depression is at 04:31:16.

Exercise 1

Determine the Subh prayer time at Jakarta, Indonesia, where Latitude: 6.2088° South, Longitude: 106.8456° East, Time Zone: GMT+7, Elevation: 50 meters, Date: 21 July 2025, Assuming Subh begins at a solar depression of –17°. First, we need to import function that are needed for the calculation.

```
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
```

Put input into the variable

```
lat_location = -6.2088
long_location = 106.8456
timezone = 7
day = 21
month = 7
year = 2025
ele = 50
```

Load ephemeris.

```
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
```

Feed info of user location

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
```

Skyfield require range of calculate date, this is written as;

```
t0 = ts.utc(year, month, day-1)
t1 = ts.utc(year, month, day)
```

Then, include the variable of refraction and location elevation

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Then, determine the time of the sunrise

```
t, y = almanac.find_risings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
```

Since the Subh prayer occurs before sunrise, it is necessary to perform a time-based loop starting from the moment of sunrise and backtrack until the solar altitude reaches 17 degrees of solar depression. The time loop operation is similar with the previous Asar prayer time loop operation. First determine the solar altitude,

```
sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
```

To develop with hour time loop, with the rule of the loop the solar degree does not pass the altitude of – 17.

```
# Start with hour

test = 1
while True:

  # Calculate the Solar Altitude
```

```
    sun_astro = location.at(ts.utc(year, month, day, h,
    m, s)).observe(sun)
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    subh_angle = 17
    elevation_correction = 0.0293 * math.sqrt(ele)
    subh_angle_actual = -subh_angle
    - elevation_correction
    if sun_alt.degrees >= 0:
      break
    if test > 24:
      break

    if sun_alt.degrees <= subh_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
    h -= 1
```

In the above code, if sun_alt.degrees <= subh_angle_actual:is used to determine whether the solar altitude has reached below, then proceed with a minute-level time loop to more precisely determine the exact minute when the solar altitude is first reached.

```
# Once the condition is met for hours, move to minutes
h_subh = h + 1
test =1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_subh, m, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  if sun_alt.degrees >= 0:
    break
  if test > 1440:
    break

  if sun_alt.degrees <= subh_angle_actual:

     break # Exit the loop if the solar altitude
     located below -18 degree
  m -= 1

  # Increment time in minutes
m_subh = m + 1
```

Then proceed with a second-level time loop to more precisely determine the exact minute when the required Subh depression degree is first reached.

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
  # Calculate the Solar Altitude
  sun_astro = location.at(ts.utc(year, month, day,
  h_subh, m_subh, s)).observe(sun)
  sun_alt, _, _ = sun_astro.apparent().altaz() # Get
  the altitude of the sun
  if sun_alt.degrees >= 0:
    break
  if test > 86400:
    break

  if sun_alt.degrees <= subh_angle_actual:
     break # Exit the loop if the solar altitude
     located below -18 degree
  s -= 1

  # Increment time in seconds

  s_subh = s +1
```

The Isya' time, after the time loop, with timezone correction is

```
subh_time = (h_subh + (m_subh) / 60 + s_subh /
3600)+timezone
print(subh_time)
[28.91295064]
```

To correct this and ensure the resulting prayer time remains within a 24-hour format,

```
subh_time %= 24 # Ensure 24-hour clock format
print(subh_time)
4.91295064
```

To convert into time format,

```
subh_time = float(subh_time)
degrees = int(subh_time)
```

```
decimal_part = subh_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    subh = "Subuh Does Not Occur"
else:
    subh = f"Subuh Occurs at {degrees}° {minutes}'
    {seconds}""

print(subh)
Subuh Occurs at 4° 54' 47"
```

Subh prayer time at Jakarta, Indonesia, where Latitude: 6.2088° South, Longitude: 106.8456° East, Time Zone: GMT+7, Elevation: 50 meters, Date: 21 July 2025, Assuming Subh begins at a solar depression of –17° is at 04:54:47.

Exercise 2

Determine the Subh (Fajr) prayer time at Khartoum, Sudan on 21 July 2025, assuming Subh begins when the sun reaches –15° below the horizon. Location details:

- Latitude: 15.5007° North
- Longitude: 32.5599° East
- Elevation: 380 meters
- Time Zone: GMT+2

Exercise 3

Calculate the Subh prayer time in Manila, Philippines for 21 July 2025, using a –19° solar depression angle. Location details:

- Latitude: 14.5995° North
- Longitude: 120.9842° East
- Elevation: 16 meters
- Time Zone: GMT+8

Exercise 4

Compute the Subh prayer time at San Salvador, El Salvador on 21 July 2025, based on a −20° solar depression. Location details:

- Latitude: 13.6929° North
- Longitude: −89.2182° West
- Elevation: 658 meters
- Time Zone: GMT−6

Moonsighting Observation Data Computation

8

The determination of new Hijri months is based on the position of the moon during 29th day of a Hijri month (Mustapha et al., 2024). Either the new Hijri month is determined using moonsighting, astronomical calculations, or lunar crescent visibility criteria; knowing the geometric position of the moon is vital. In the past, the endeavor to calculate the position of the moon during 29th day of Hijri month is an arduous task, and only selected person can carry out the task. Nowadays, due to the invention of Python programming language and extensive libraries, the task of calculating the position of the moon can be conducted with a few lines of code. In this case, the task of calculating the position of the moon can be calculated using Skyfield.

Let us say, user observation site is Kigali, Rwanda, with Latitude 1.9577° South, Longitude: 30.1127° East, Elevation: 1,567 meters and Time Zone: GMT+2. Determine the geometrical position of the moon during for the Hijri month observation on 27 May 2025. First, install skyfield, and other related libraries.

```
pip install skyfield numpy tabulate
```

Import-related functions

```
from skyfield import almanac
from skyfield.api import Topos,load
from skyfield import api
import numpy as np
from skyfield.api import N, S, E, W, load, wgs84
from skyfield.api import Topos, load,
Angle,GREGORIAN_START
import math
```

DOI: 10.1201/9781003649120-8

```
from scipy.ndimage import rotate
import calendar
from tabulate import tabulate
from matplotlib.patches import Arc
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import matplotlib.image as mpimg
from matplotlib.offsetbox import OffsetImage,
AnnotationBbox
```

Load ephemeris and function

```
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']
moon = planets['moon']
h_maghrib = 0
m_maghrib = 0
ts = load.timescale()
eph = api.load('de440s.bsp')
```

The moon is loaded since we calculate the position of the moon. h_maghrib = 0, m_maghrib = 0

is declared to mitigate some issue later. Then, input the related variables.

```
lat_location = -1.9577
long_location = 30.1127
timezone = 8
year = 2025
month = 3
day = 30
ele = 100
```

Input the location info and refraction and elevation parameter.

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction
altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
```

```
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)
```

Determine the time of sunset.

```
t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h_sunset, m_sunset, s_sunset = t.utc.hour, t.utc.minu
te, t.utc.second
```

In moonsighting calculations, the sunset time is stored using the variables h_ sunset, m_sunset, s_sunset. This distinction is made to clearly differentiate it from the moonset time, which will be calculated separately. Then convert sunset time in time format

```
sunset_time = float(h_sunset + m_sunset / 60 + s_
sunset / 3600 + timezone)
sunset_time %= 24 # Ensure 24-hour clock format
sunset_time = float(sunset_time)
degrees = int(sunset_time)
decimal_part = sunset_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
sunset = f"{degrees}° {minutes}' {seconds}""
print(sunset)

18° 7' 58"
```

We can combine the result into a variable using an f-string, as shown above. In this example, the result is stored in the sunset variable and printed in degree-minute-second (DMS) format. This corresponds to 18:07:58, which indicates the time of sunset. Once the sunset time is determined, we can proceed to calculate the moonset time, which will follow a similar method.

```
t, y = almanac.find_settings(location, moon, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h_moonset, m_moonset, s_moonset = t.utc.hour, t.utc
.minute, t.utc.second
```

Note the changes from sun to moon in comparison to the previous code. This is because the previous code is to calculate the sun position, while this code is used to calculate the moon position. Then convert moonset time in time format

```
moonset_time = float(h_moonset + m_moonset / 60 +
s_moonset / 3600 + timezone)
moonset_time %= 24 # Ensure 24-hour clock format
moonset_time = float(moonset_time)
degrees = int(moonset_time)
decimal_part = moonset_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
moonset = f"{degrees}° {minutes}' {seconds}""
print(moonset)
19° 4' 59"
```

Therefore, the moonset occurs at 19:04:59. Then the value of lag time, the difference of time between moonset and sunset, can be extracted.

```
lag_time = abs(moonset_time - sunset_time)
print(lag_time)
0.9504272807035683
```

To calculate lag time, subtract **moonset_time** and **sunset_time**. Then make sure the result is always positive; use the function abs() inside the subtraction operation. The result is in decimal format; to convert into time format,

```
degrees = int(lag_time)
decimal_part = lag_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
lagtime= f"{degrees}° {minutes}' {seconds}""
print(lagtime)
0° 57' 2"
```

The lag time between moonset and sunset is 57 minute and 2 seconds. Next is to determine the geometrical position of the moon. The geometrical position of the moon is calculated using a topocentric reference. This means that the position of the moon is calculated in reflect to the user position on the surface of the Earth. The calculation is conducted based on the position of the moon during sunset. To perform the calculation, first we calculate the altitude of the moon during sunset.

```
moon_astro = location.at(ts.utc(year, month, day,
h_sunset, m_sunset)).observe(moon)
```

```
moon_app = moon_astro.apparent()
moon_alt, moon_az, distance = moon_app.altaz()
print(moon_alt)
12deg 27' 16.3"
```

The code structure used to calculate moonset is like the code for calculating prayer times, such as sunset. The only difference lies in the celestial object being observed, the sun for prayer times, and the moon for moonsighting. In this case, the same variables as h_sunset and m_sunset can be reused for determining the moon's position during sunset. The moon altitude during sunset is 12 deg 27' 16.3". This format is readable for humans but not suitable for numerical calculations. To perform further computations, like determining moon visibility, we need the altitude in decimal degrees. To do this, simply access the .degrees attribute of the moon_alt variable:

```
print(moon_alt.degrees)
[12.4545324]
```

The next important parameter for new moon sighting is arc of vision. The arc of vision is the altitude difference between the sun's altitude and moon's altitude. First, determine the sun's altitude during sunset.

```
sun_astro = location.at(ts.utc(year, month, day,
h_sunset, m_sunset)).observe(moon)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_alt.degrees)
[-0.92325118]
```

Next, the difference between sun-moon altitude or Arc of Vision,

```
arc_of_vision = abs(moon_alt.degrees - sun_alt.
degrees)
print(arc_of_vision)
[13.37778358]
```

The arc of vision is 13.3778358. Next is the arc of light. The arc of light is the angle of separation between sun and moon, or the elongation angle. The arc of light can be determined with code,

```
arc_of_light = sun_app.separation_from(moon_app)
print(arc_of_light.degrees)
[16.57460685]
```

The arc of light is 13.3778358. Next is the difference in azimuth. The difference in azimuth is the azimuthal difference between sun azimuth and moon azimuth. The code to determine the difference in azimuth is as follows

```
difference_azimuth =
abs(moon_az.degrees-sun_az.degrees)
print(difference_azimuth)
[9.85833904]
```

The difference in azimuth between the sun and the moon is 9.85833904°. The final parameter to consider is the moon age, which refers to the elapsed time between the moment of conjunction (new moon) and the sunset at the observation location. This age is a crucial factor in evaluating the likelihood of crescent visibility. To determine moon age, first determine the Julian date of the sunset time,

```
t, y = almanac.find_settings(location, moon, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
jd_sunset = t.tt

jd_sunset= jd_sunset[0]
print(jd_sunset)
2460760.037716627
```

The code is similar with find sunset code, however the output is changed to t.tt, as it is in Julian date format. Next, determine the time of moon conjunction using almanac function,

```
t0 = ts.utc((year), (month), (day-5))
t1 = ts.utc((year), (month), (day+5))
f = almanac.oppositions_conjunctions(eph, eph['Moon'])
t, y = almanac.find_discrete(t0, t1, f)

for ti, yi in zip(t, y):
   if yi == 1:
       jd_moon_conjuction = format(ti.tt)
   else:
       None
jd_moon_conjuction= float(jd_moon_conjuction)
```

The time of moon conjunction is computed using almanac.oppositions_conjunctions function. This function can determine the timing of conjunction

or opposition, with yi == 1 for conjunction, and yi == 0 for moon opposition or full moon. −5 and +5 is performed at t0 and t1 since the conjunction usually takes a 0 to a few days before the moon observation. Then, the moon age is computed as the Julian date difference between sunset and moon conjunction.

```
moonage = (jd_sunset -jd_moon_conjuction)*24
print(moonage)
```

There! We got the geometric position of the moon during observation.

Exercise 1

An observer in Istanbul, Turkey, located at 41.0082° North latitude and 28.9784° East longitude, at an elevation of 39 meters and under the GMT+3 time zone, intends to sight the new moon that would indicate the beginning of the month of Shawal. The observation is to be made on 29 March 2025, which corresponds to the 29th day of Ramadan 1446H. Determine the geometrical position of the moon during this date and time.

Exercise 2

In Cape Town, South Africa, an observer situated at 33.9249° South latitude and 18.4241° East longitude, at an elevation of 15 meters and within the GMT+2 time zone, will attempt to observe the crescent moon on 6 June 2025. This date aligns with the 29th day of Zulkaedah 1446H and is critical for determining the beginning of the month of Zulhijjah. Calculate the geometrical position of the moon at this location and date.

Exercise 3

Assume a user in Jakarta, Indonesia, with coordinates of 6.2088° South latitude and 106.8456° East longitude, observes the moon from an elevation of 50 meters above sea level. The local time zone is GMT+7. The observation is planned for 29 March 2025, corresponding to 29 Ramadhan 1446H. The task is to determine the geometrical position of the moon for Shawwal crescent sighting on that evening.

Exercise 4

In Los Angeles, USA, an observer located at 34.0522° North latitude and 118.2437° West longitude, at an elevation of 71 meters and within the GMT–7 time zone, intends to observe the moon on 6 June 2025. This corresponds to the 29th day of Zulkaedah 1446H. Determine the geometrical position of the moon during sunset at this location to evaluate the possibility of crescent sighting for Zulhijjah.

Qiblah Compass Visualization

9

The determination of Rashdul Kiblat is universally suitable for all days in a year. There are times when the Rashdul Kiblat is not applicable for a particular day or time. This is due to the solar position. Therefore, another alternative is to determine the Qibla is based on the degree of solar azimuth. So, to create the visualization of the Qibla compass, it must be based on the solar azimuth degree. The solar azimuth degree can be calculated using Skyfield, while the Qibla direction can be calculated using the previous formula; therefore, the angle between the solar azimuth degree and Qibla direction can be determined using the subtraction between the solar azimuth degree and Qibla direction degree. This angle of subtraction can be visualized.

Visualization of the angle between the sun azimuth and the direction of the Qibla can be determined using matplotlib function of polar plot. Matplotlib is a visualization function that is embedded with Python. Matplotlib can be used to create bar graphs, line graph, and many more graphs. Matplotlib can visualize the angle of difference between solar azimuth and Qibla direction using polar plot. Polar plot is plot of magnitude $|G(j\omega)H(j\omega)|$ versus phase angle $\angle(G(j\omega)H(j\omega))$ in polar coordinates, and the value of frequency i.e. ω is varied from 0 to ∞. In polar plot magnitude of transfer function is plotted to distance from origin and phase angle is plotted from positive real x-axis. Polar plot is used in Nyquist plot to determine the stability of closed loop control system from its open loop frequency response. The polar plot has the angle scale from 0 degree to 360 degree. This would be perfect for the determination of Qibla direction from the position of the solar azimuth.

DOI: 10.1201/9781003649120-9

QIBLA DIRECTION VISUALIZATION ON POLAR PLOT

The instruction to use Polar Plots for Qibla direction visualization in Python is

```
fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
where
fig: The figure object, which is the overall container
for the plot.
ax: The axes object, which represents the polar
subplot.
subplot_kw={'projection': 'polar'}: Configures the
axes to have a polar projection. The output for the
code
import matplotlib.pyplot as plt
# Create the polar plot
fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
```

This code will output as shown in Figure 9.1.

Notice that the direction of 0 degree start from the right side of the compass. To rearrange the polar plot, with 0 degree located at the forward/upside of the compass, the code is

```
import matplotlib.pyplot as plt
# Create the polar plot
fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
ax.set_theta_direction(-1) # Set clockwise direction
ax.set_theta_zero_location('N') # Set 0° to the top
(North)
```

Where ax.set_theha_zero_location ('N') will set the top as the starting location for the compass, which is the **North**. While ax.set_theta_direction(–1), will set the direction of the compass to be in clockwise direction, following the actual compass direction.

The direction of the Qibla follows the previous Qibla equation; let's say the direction of the Qibla is 291. In the code, this can be expressed as

```
qibla_direction_deg = 291
```

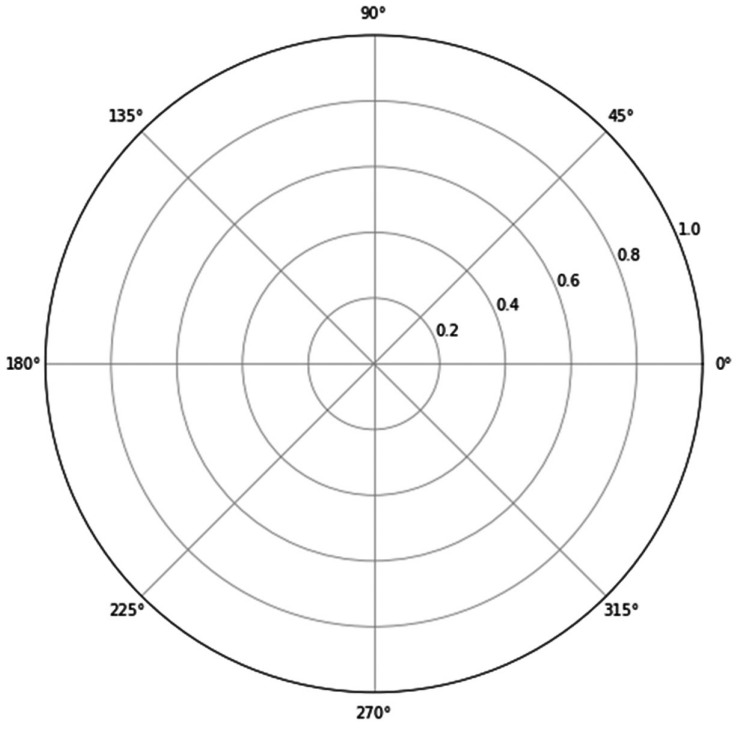


FIGURE 9.1 Generic polar plot.

The direction of the Qibla in the polar plot is read in radians form; therefore, to convert into radians form, the code is expressed as

```
qibla_direction_rad = np.deg2rad(qibla_direction_deg)
```

this will convert the angle degree to radians form. After that, to visualize the Qibla direction in the polar plot, the code is as follows

```
ax.plot([0, qibla_direction_rad], [0, 1], label='Qibla
Direction', color='blue', linewidth=2)
```

where,

```
ax.plot:
```

This is a method to plot data on the specified axes (ax), in this case, a polar axis created earlier.

`Arguments: [0, qibla_direction_rad] and [0, 1]:`

- [0, qibla_direction_rad]: This is the radial angle data in radians. The line starts at 0 radians (the center) and extends to qibla_direction_rad, which represents the direction of the Qibla in polar coordinates.
- [0, 1]: This is the radius data. The line starts at a radius of 0 (the center) and extends outward to a radius of 1.

`label='Qibla Direction':`

- This provides a label for the line, which can be displayed in a legend if added to the plot.

`color='blue':`

- Sets the color of the line to blue.

`linewidth=2:`

- Specifies the thickness of the line as 2 units.

The output for this code is shown in Figure 9.2.

Exercise 1: Visualize a Qibla Direction Using Location Latitude of 39.12 North, and Longitude of 80.11 East

First, determine the Qibla direction. To do this, first we insert the variable required for the calculation.

```
φ_Location = 39.12
λ_Location = 80.11
φ_Kaabah = 21.4225
λ_Kaabah = 39.8262
Difference_Longitude = abs(λ_Location-λ_Kaabah)
```

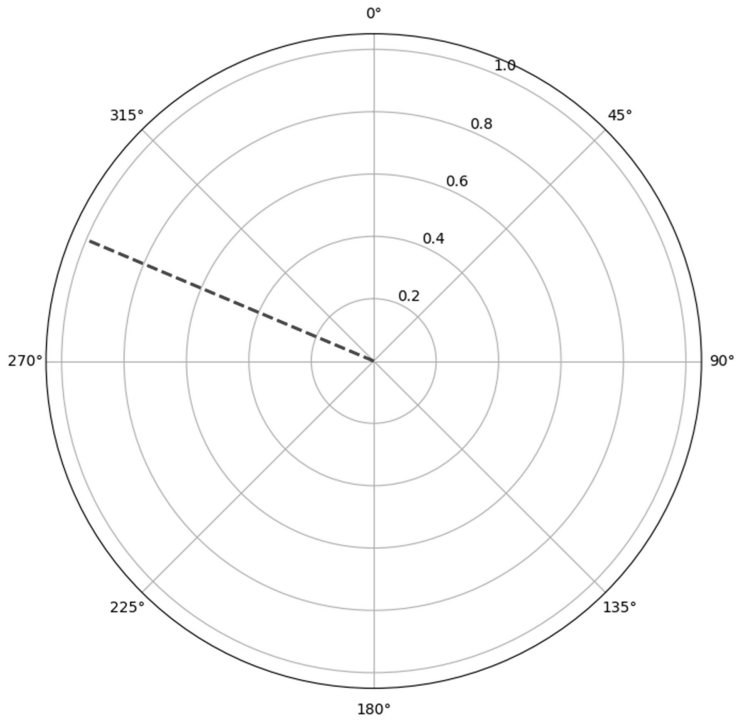


FIGURE 9.2 Polar plot with Qibla location.

Then, we performed the calculation.

```
#Calculation of Qibla Direction
import math

A = math.sin(math.radians(abs(Difference_Longitude)))
B = math.cos(math.radians(φ_Location))*math.tan(math.r
    adians(φ_Kaabah))
C = math.sin(math.radians(φ_Location)) * math.cos(m
    ath.radians(Difference_Longitude))
D = A/(B-C)
θ = math.degrees(math.atan(D))

#Determine the Azimuth of the Qibla
if Difference_Longitude > 180:
    delta_λ = 360 - Difference_Longitude
```

```
else:
    delta_λ = Difference_Longitude

if θ > 0:
    if λ_Location > λ_Kaabah:
        quadrant = "UB" # Utara Barat
    elif λ_Location <= λ_Kaabah:
        quadrant = "UT" # Utara Timur
    elif λ_Location < 0:
        if c >= 180:
          quadrant = "UB"
        else:
          quadrant = "UT"
elif θ < 0:
    if λ_Location > λ_Kaabah:
        quadrant = "SB" # Selatan Barat
    elif λ_Location <= λ_Kaabah:
        quadrant = "ST" # Selatan Timur
    elif λ_Location < 0:
        if c >= 180:
          quadrant = "SB"
        else:
          quadrant = "ST"

if quadrant == "UB":
    azimuth_kiblat = 360 - θ
elif quadrant == "SB":
    azimuth_kiblat = 180 - θ
elif quadrant == "UT":
    azimuth_kiblat = θ
elif quadrant == "ST":
    azimuth_kiblat = 180 + θ
# To Convert in Degree Form

degrees = int(azimuth_kiblat)
decimal_part = azimuth_kiblat - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
print(f'The azimuth of the Qibla for Location with
coordinate {φ_Location} Latitude, {λ_Location}
Longitude, is {degrees}° {minutes}' {seconds}"')
```

```
The azimuth of the Qibla for Location with coordinate
39.12 Latitude, 80.11 Longitude, is 254° 41′ 48″
```

The Qibla direction for the given location is 254° 41′ 48″. Next, is to create the polar plot. First, create an empty polar plot.

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
```

The above code will result in an empty polar plot. And then, to align the top of the polar plot as North, or 0 degree.

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

fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
ax.set_theta_direction(-1) # Set clockwise rotation
ax.set_theta_zero_location('N') # Set 0° (North) at
the top
```

Ok now, the top of the polar plot is 0 degree, similar with the alignment that we usually found on magnetic compass. Then is to label the Qibla direction based on the Qibla direction calculated before.

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

fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
ax.set_theta_direction(-1) # Set clockwise rotation
ax.set_theta_zero_location('N') # Set 0° (North) at
the top

# Convert Qibla direction to radians for plotting
qibla_direction_rad = np.deg2rad(azimuth_kiblat)
ax.plot([0, qibla_direction_rad], [0, 1], label=
f'Qibla: {azimuth_kiblat:.2f}°', color='blue',
linewidth=2)
```

This will label the Qibla direction (Figure 9.3).

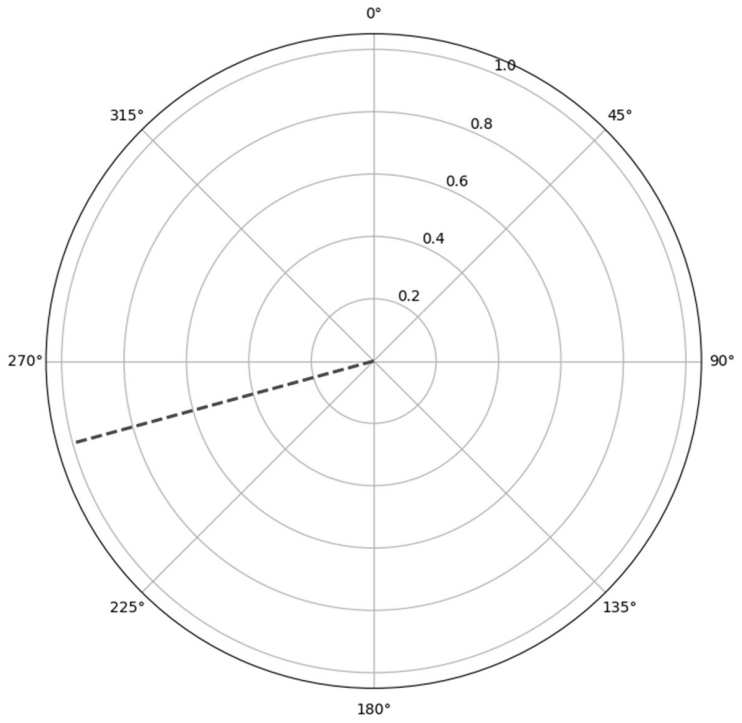


FIGURE 9.3 Polar plot with Qibla direction.

SUN AZIMUTH VISUALIZATION ON POLAR PLOT

After the Qibla direction can be computed and visualized in the polar plot, now it is for the solar azimuth computation. The solar azimuth computation can be performed using Skyfield. To calculate the sun azimuth at any given time and location, is as follows:

```
from skyfield.api import load
from skyfield.api import N,S,E,W, wgs84
location = earth + wgs84.latlon(location_latitud * N,
location_longitud * E, elevation_m=0)
ts = load.timescale()
```

```
    eph = load('de440s.bsp')
    planets = load('de440s.bsp')
    earth = planets['earth']
    sun = planets['sun']
    astro = location.at(ts.utc(year, month, day,
    hour-timezone,minute)).observe(sun)
    sun_app = astro.apparent()
    sun_alt, sun_az, = sun_app.altaz()
```

Explanation of Each Line

```
from skyfield.api import load:
```

- Imports the load function from the skyfield.api module.
- The load function is used to fetch data files (e.g., timescales, ephemeris files) necessary for astronomical computations.

```
from skyfield.api import N, S, E, W, wgs84:
```

- Imports the constants N, S, E, W (representing the cardinal directions: North, South, East, and West) and the wgs84 object from the skyfield.api module.
- wgs84 is a geodetic model used for Earth-related computations, such as converting latitudes and longitudes to 3D coordinates.

```
ts = load.timescale():
```

- Loads a timescale object, which is used to work with time in Skyfield.
- The timescale object provides methods to define and manipulate time (e.g., UTC, TT).

```
eph = load('de421.bsp'):
```

- Loads the JPL (Jet Propulsion Laboratory) **DE421 ephemeris file**. This file contains precise positions and velocities for celestial bodies in the solar system.
- 'de421.bsp' is a binary file that Skyfield uses to compute positions of planets and other bodies.

```
planets = load('de421.bsp'):
```

- This line is redundant (repeats the loading of 'de421.bsp') but essentially assigns the same ephemeris data to a new variable planet.
- It's commonly done to improve code readability (e.g., distinguishing between different use cases for the same data file).

```
earth = planets['earth']:
```

- Extracts the Earth object from the ephemeris data. This object allows calculations involving Earth's position, including observer locations.

```
sun = planets['sun']:
```

- Extracts the sun object from the ephemeris data. This object can be used to calculate the sun's position relative to Earth or other bodies.

location = earth + wgs84.latlon(location_latitud * N, location_longitud * E, elevation_m=0)

```
Earth:
```

- Represents the Earth in the solar system as defined by the loaded ephemeris (de421.bsp).
- This object is used as the base reference for observer locations.

```
wgs84.latlon(location_latitud * N, location_longitud * E,
elevation_m=0):
```

- A geodetic model representing Earth's shape (latitude, longitude, elevation).
- Converts the geographical coordinates into a 3D position in space relative to Earth's center.

```
location_latitud * N:
```

- Multiplies the latitude value (location_latitud) by N (North), ensuring it's interpreted as a northern hemisphere coordinate. If in the southern hemisphere, you'd use S instead.

```
location_longitud * E:
```

- Multiplies the longitude value (location_longitud) by E (East). If in the western hemisphere, you'd use W.

`elevation_m=0:`

- Specifies the elevation (in meters) of the location. Here, it is set to 0, which corresponds to sea level.

`earth + ...:`

- Combines the Earth's position in the solar system with the observer's location on Earth. The resulting location object represents a specific point on Earth as it moves through space.

```
astro = location.at(ts.utc(year, month, day, hour,minu
te)).observe(sun)
```

`location.at(ts.utc(year, month, day, hour, minute)):`

- Computes the position of the specified location on Earth at the given **UTC time**.
- ts.utc(year, month, day, hour, minute): Defines the time in Coordinated Universal Time (UTC).
- This step determines where the Earth (and hence the observer) is in space at that moment.

`.observe(sun):`

- Calculates the **apparent position** of the sun as seen from the location at the specified time.
- Considers the relative positions of the observer, Earth, and sun.

```
sun_app = astro.apparent()
```

`astro.apparent():`

- Converts the **geometric position** of the sun (as computed by astro) into its **apparent position** by:

  1. Accounting for **light-time delay**: The sun's observed position includes the time it takes for light to travel from the sun to the observer.
  2. Including **aberration of light**: Adjusts for the motion of the Earth while observing the sun.

`sun_app:`

- This variable now contains the sun's **apparent position** as seen by the observer, accounting for these effects.

```
sun_alt, sun_az, sun_distance = sun_app.altaz()
```

Explanation of the Outputs:

`sun_alt (Altitude):`

- The angular height of the sun above or below the horizon (in degrees).
- Positive values: The sun is above the horizon.
- Zero: The sun is on the horizon (sunrise or sunset).
- Negative values: The sun is below the horizon (nighttime).

`sun_az (Azimuth):`

- The compass direction to the sun (in degrees).
- Measured clockwise from North:
  - 0° = North
  - 90° = East
  - 180° = South
  - 270° = West

`sun_distance (Distance):`

The distance to the sun from the observer's location, measured in Astronomical Units (AU).

1 AU ≈ 149.6 million kilometers, which is the average Earth-sun distance.

Exercise 2: Calculate the Solar Azimuth at the Latitude of 39.12 North, and Longitude of 80.11 East, in 6+ Timezone, on 13 April 2024, during 15:34

First, determine the sun azimuth. The initial step is to input the required variable.

```
import math
from skyfield.api import load, N, E, wgs84 # Skyfield
for astronomical calculations
import calendar # For handling dates and times

lat_location = 39.12
long_location = 80.11
timezone = 6
year = 2025
month = 4
day = 13
ele = 100
```

The idea is that the user can use the position of the sun to determine the direction of the Qibla without requiring timing of the Rashdul Qibla. We can use any time of the day when the sun is located above the horizon. For this case, we use the position of sun azimuth at 15:34. To input the variable for hour and minute,

```
hour = 15
minute = 34
```

And then, to determine the sun azimuth

```
# Load planetary ephemeris data (precise astronomical
positions)
eph = load('de440s.bsp')
planets = load('de440s.bsp')

# Get Earth and Sun objects from the ephemeris
earth = planets['earth']
sun = planets['sun']

location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)

# Create timescale object and set observation time
ts = load.timescale()
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

```
sun_astro = location.at(ts.utc(year, month, day,
hour-timezone, minute)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_az.degrees)
```

From the sun azimuth, we can visualize the sun azimuth using the polar plot.

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

# Create polar plot (compass-style visualization)
fig, ax = plt.subplots(
    subplot_kw={'projection': 'polar'}, # Polar
    coordinate system
    figsize=(8, 8)              # 8x8 inch figure
)

# Configure polar plot:
ax.set_theta_direction(-1)       # Clockwise rotation
                                 (standard for compass
                                 bearings)
ax.set_theta_zero_location('N')  # 0° at top (North)

# Convert sun direction from degrees to radians for
plotting
sun_direction_rad = np.deg2rad(sun_az.degrees)

# Plot a line from center (0,0) to edge (1) in sun's
direction
ax.plot(
    [0, sun_direction_rad],       # Angle in radians
    [0, 1],                 # Distance from center
    label=f'Sun Az: {sun_az.degrees:.2f}°',    # Legend
                                               label
    color='orange',           # Color of line
    linewidth=2               # Line thickness
)

# Display the plot
plt.show()
```

This results as shown in Figure 9.4.

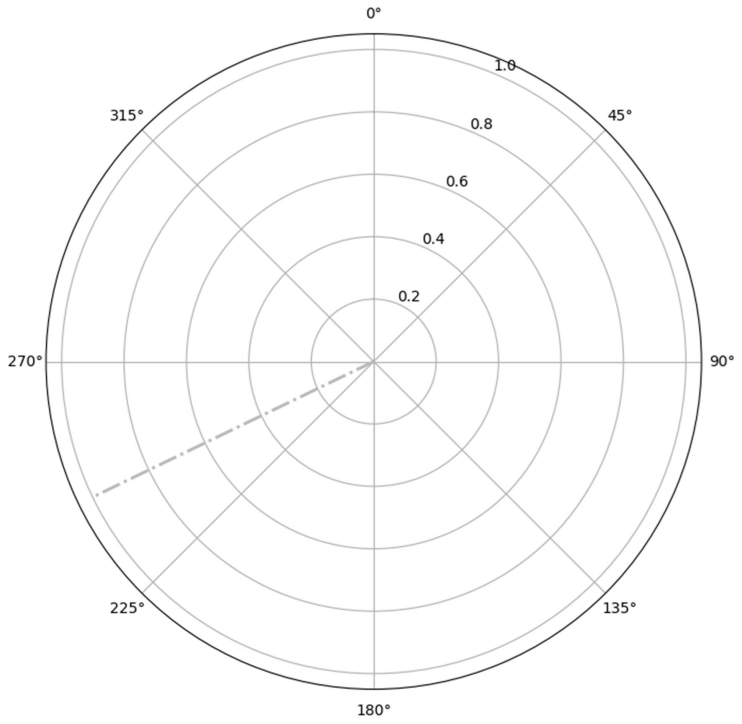


FIGURE 9.4 Polar plot with sun azimuth direction.

SUN AZIMUTH AND QIBLA DIRECTION VISUALIZATION ON POLAR PLOT

Now, let's combine polar plot for Qibla direction and sun azimuth.

Exercise 3: Visualize Qibla Direction and the Solar Azimuth on 13 April 2024, during 15:34, Using Location Latitude of 39.12 North, and Longitude of 80.11 East

The sun azimuth and Qibla direction visualization can be combined when both visualization codes are run on the same command console. First, run the Qibla direction calculation code.

```
φ_Location = 39.12
λ_Location = 80.11
φ_Kaabah = 21.4225
λ_Kaabah = 39.8262
Difference_Longitude = abs(λ_Location-λ_Kaabah)

#Calculation of Qibla Direction
import math

A = math.sin(math.radians(abs(Difference_Longitude)))
B = math.cos(math.radians(φ_Location))*math.tan(math.r
    adians(φ_Kaabah))
C = math.sin(math.radians(φ_Location)) * math.cos(m
    ath.radians(Difference_Longitude))
D = A/(B-C)
θ = math.degrees(math.atan(D))

#Determine the Azimuth of the Qibla
if Difference_Longitude > 180:
    delta_λ = 360 - Difference_Longitude
else:
    delta_λ = Difference_Longitude

if θ > 0:
    if λ_Location > λ_Kaabah:
        quadrant = "UB" # Utara Barat
    elif λ_Location <= λ_Kaabah:
        quadrant = "UT" # Utara Timur
    elif λ_Location < 0:
        if c >= 180:
            quadrant = "UB"
        else:
            quadrant = "UT"
elif θ < 0:
    if λ_Location > λ_Kaabah:
        quadrant = "SB" # Selatan Barat
    elif λ_Location <= λ_Kaabah:
        quadrant = "ST" # Selatan Timur
    elif λ_Location < 0:
        if c >= 180:
            quadrant = "SB"
        else:
            quadrant = "ST"

if quadrant == "UB":
    azimuth_kiblat = 360 - θ
```

```
elif quadrant == "SB":
    azimuth_kiblat = 180 - θ
elif quadrant == "UT":
    azimuth_kiblat = θ
elif quadrant == "ST":
    azimuth_kiblat = 180 + θ

# To Convert in Degree Form

degrees = int(azimuth_kiblat)
decimal_part = azimuth_kiblat - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)
print(f'The azimuth of the Qibla for Location with
coordinate {φ_Location} Latitude, {λ_Location}
Longitude, is {degrees}° {minutes}' {seconds}"')
```

Then the Sun Azimuth code

```
# Import required libraries
import math
from skyfield.api import load, N, E, wgs84 # Skyfield
for astronomical calculations
import calendar # For handling dates and times

lat_location = 39.12
long_location = 80.11
timezone = 6
year = 2025
month = 4
day = 13
ele = 100

hour = 15
minute = 34

# Load planetary ephemeris data (precise astronomical
positions)
eph = load('de440s.bsp')
planets = load('de440s.bsp')

# Get Earth and Sun objects from the ephemeris
earth = planets['earth']
sun = planets['sun']
```

```
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)

# Create timescale object and set observation time
ts = load.timescale()
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)

sun_astro = location.at(ts.utc(year, month, day,
hour-timezone, minute)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
print(sun_az.degrees)
```

Then run the combination of the visualization code. The combination code requires two important variables, sun azimuth as sun_az.degrees, and Qibla Direction as azimuth_kiblat.

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

fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
ax.set_theta_direction(-1)
ax.set_theta_zero_location('N')

# Plot Qibla Direction (blue line)
qibla_direction_rad = np.deg2rad(azimuth_kiblat)
ax.plot([0, qibla_direction_rad], [0, 1], label=
f'Qibla: {azimuth_kiblat:.2f}°', color='blue',
linewidth=2)

# Plot Sun Azimuth (orange line)
sun_direction_rad = np.deg2rad(sun_az.degrees)
# Plot a line from center (0,0) to edge (1) in sun's
direction
ax.plot(
   [0, sun_direction_rad],       # Angle in radians
   [0, 1],                 # Distance from center
   label=f'Sun Az: {sun_az.degrees:.2f}°',     # Legend
                                               label
   color='orange',            # Color of line
   linewidth=2                # Line thickness
)
```

The resulting code generates a compass displaying both the Sun's azimuth and the Qibla direction. However, it currently lacks labels to help users interpret the visualization. To enhance its usability, the visual output should be updated with clear, descriptive labels.

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

fig, ax = plt.subplots(subplot_kw={'projection':
'polar'}, figsize=(8, 8))
ax.set_theta_direction(-1)
ax.set_theta_zero_location('N')

# Plot Qibla Direction (blue line)
qibla_direction_rad = np.deg2rad(azimuth_kiblat)
ax.plot([0, qibla_direction_rad], [0, 1], label=
f'Qibla: {azimuth_kiblat:.2f}°', color='blue',
linewidth=2)

# Plot Sun Azimuth (orange line)
sun_direction_rad = np.deg2rad(sun_az.degrees)
# Plot a line from center (0,0) to edge (1) in sun's
direction
ax.plot(
            [0, sun_direction_rad],[0, 1],label=f'Sun
            Az: {sun_az.degrees:.2f}°',
            color='orange',linewidth=2
)

plt.title(f'Visualization of Qibla and Sun Azimuth\
nLat: {lat_location}° | Long: {long_location}° \n{day}
{month} {year}, {hour}:{minute} Local Time\n', pad=20)
plt.legend(loc='upper right')
plt.show()
```

Figure 9.5 shows a polar plot; it's like a compass that shows two important directions: where the Qibla is (marked by the dashed line) and where the sun is at a particular time (shown with the dashed and dotted line). Right above the plot, you'll see the exact location it's based on, with the latitude and longitude, as well as the local date and time. That information is important because the direction of both the Qibla and the sun changes depending on where you are and what time it is.

Now, if you look around the circle, you'll notice numbers like 0°, 90°, 180°, and 270°. These represent directions: 0° is North, 90° is East, 180° is

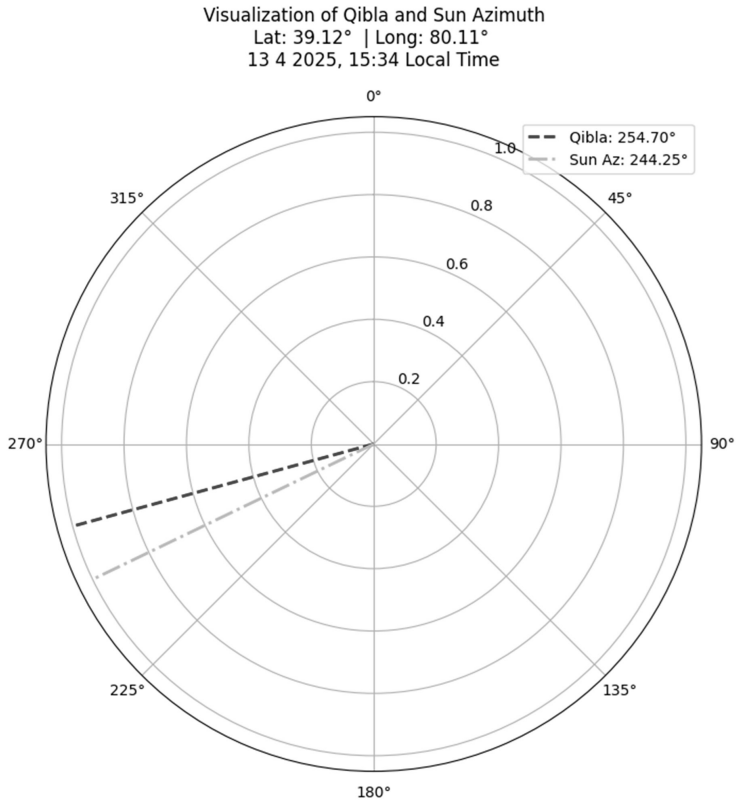


FIGURE 9.5 Polar plot of the sun azimuth and Qibla direction.

South, and 270° is West. So, imagine this as if you're standing inside a compass, and each degree tells you which way you're facing.

To use this, start by focusing on the blue line; that's the Qibla direction for your location. It tells you the exact angle you need to face to pray toward the Kaaba in Makkah. If you have a compass or a compass app on your phone, just rotate yourself until you're aligned with that degree. That's your Qibla.

Now here's where it gets even more helpful, notice the orange line? That shows where the sun is at the given time. If you're outdoors and the sun is visible, you can use its position to help you find the Qibla without needing a compass. One simple way is to use something like a water bottle or any object that stands upright. Place it on a flat surface and observe where its shadow falls. The shadow will point directly away from the sun's direction.

Now, look at the diagram again and find the orange line; that's the sun's azimuth. Then compare it to the blue line, which shows the Qibla direction. The angle between these two lines tells you how much you need to rotate from the sun's position to face the Qibla. For example, if the Qibla is to the left of the sun's position on the plot, you would turn your body that same amount to the left from the shadow's direction. This way, the shadow becomes your guide, and with just a bottle and this plot, you can figure out the Qibla direction even without technology. Pretty handy, right?

So, in short, this plot helps you see both where the Qibla is and where the sun is at a specific time, which can be useful, especially when you don't have a digital compass handy but can see the sun.

Exercise 4

Visualize the Qibla direction and the solar azimuth on August 17, 2025, at 14:40 local time, for a location with latitude 36.74° South and longitude 71.06° West (near Chillán, Chile). The location has an elevation of 150 meters above sea level, and the local timezone is UTC−4 (Chile Standard Time). Use this information to generate a polar plot showing both the direction of the Qibla and the azimuth of the sun at that moment.

Exercise 5

Visualize the Qibla direction and the solar azimuth on August 31, 2025, at 09:23 local time, for a location with latitude 48.85° North and longitude 2.35° East (Paris, France). The location has an elevation of 35 meters above sea level, and the local timezone is UTC+2 (Central European Summer Time). Generate a polar plot to visualize both the Qibla direction and the position of the sun at that specific time.

Exercise 6

Visualize the Qibla direction and the solar azimuth on October 10, 2025, at 16:10 local time, for a location with latitude 1.29° North and longitude 103.85° East (Singapore). The location has an elevation of 15 meters, and the local timezone is UTC+8 (Singapore Standard Time). Generate a plot that shows the sun's azimuth alongside the Qibla direction.

Exercise 7

Visualize the Qibla direction and the solar azimuth on December 5, 2025, at 07:00 local time, for a location with latitude 40.71° North and longitude –74.01° West (New York City, USA). The location has an elevation of 10 meters above sea level, and the local timezone is UTC–5 (Eastern Standard Time). Create a visualization to compare the sun's position with the Qibla direction during sunrise hours.

Sun Position during Prayer Times Visualization 10

VISUALIZING THE SUN POSITION

Prayer times are determined by the sun's position relative to the observer. Each prayer time corresponds to a specific solar event:

1. Zuhur begins when the sun reaches its highest point (zenith) in the sky, directly above the observer's location (solar noon).
2. Asr starts when the length of an object's shadow equals its actual height (or twice its height, depending on the school of thought).
3. Maghrib begins at sunset, when the sun completely disappears below the horizon.
4. Isha and Fajr depend on atmospheric twilight caused by the sun's diffraction below the horizon.
    - Isha begins when the sky is fully dark (astronomical twilight ends).
    - Fajr begins at dawn when the first light appears (astronomical twilight begins).
5. Sunrise marks the end of Fajr and the beginning of daytime.

This visualization helps illustrate these key solar positions, making it easier to understand the astronomical basis of Islamic prayer times. Before diving into the code, let's first understand the objective and the expected output:

DOI: 10.1201/9781003649120-10

A. Objective

We want to create a simple visualization that shows:

1. The position of an observer (stick figure) at a certain location.
2. The position of the sun in the sky at a specific altitude angle.
3. The horizon line as a reference (0°).

B. Expected Output

A plot containing the following elements:

1. Green line: Horizon (ground level).
2. Simple human figure: Representation of the observer.
3. Orange circle: The sun at a certain elevation.
4. Dashed line: Line of sight from the observer to the sun.

To visualize the position of the sun based on the observer, the step is as follows:

1. Import Libraries: Use matplotlib for plotting and numpy for mathematical calculations.
2. Setup Positions:
   - Observer coordinates (observer_x, observer_y).
   - Sun's altitude angle (altitude_angle).
3. Calculate Sun Position Based on the Angle (sun_x, sun_y).
4. Plot the Visualization:
   - Draw the horizon as a straight line.
   - Represent the sun as an orange dot.
   - Draw a stick figure (if an image is available).

Here is the Complete Code with Inline Explanations, but first, install related libraries.

Import Library

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
```

```
import requests
import io # Import io to handle the image data in
memory
```

Set Observer and Sun Positions

```
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = 40 # Angle in degrees (negative for
below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun
```

Calculate the Sun's Position Based on the Angle

```
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position
```

Plot the Visualization

```
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"

# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
```

```
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

# Setup position of stick figure (on the horizon)
stick_x, stick_y = 5, 0
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")

# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y, f"Sun Position ({altitude_
angle}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)

# Add labels and legend
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title("Sun's Altitude Visualization",
fontsize=14)

plt.show()
```

The result of the code implementation above is shown in Figure 10.1.

This visualization demonstrates how solar altitude angles (like those used to determine prayer times) can be represented geometrically from an observer's perspective.

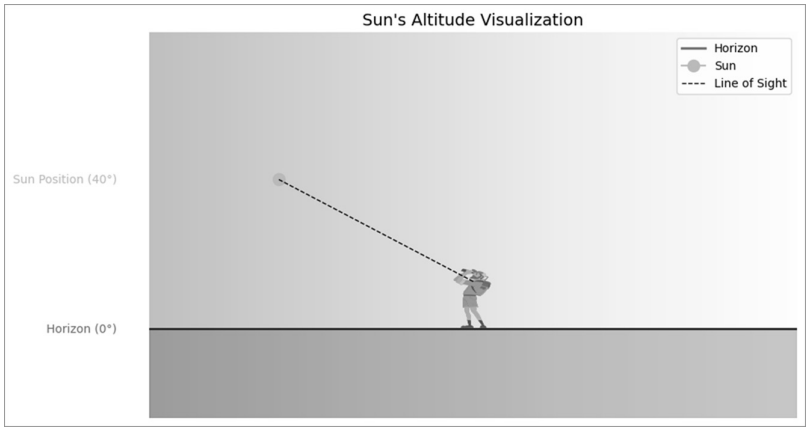


FIGURE 10.1 Visualization of sun position in respect to an observer.

Exercise 1: Calculating Sun's Altitude Angles for Prayer Times

Use Python's skyfield library to calculate the sun's altitude angle at specific prayer times (e.g., Asr) for given coordinates and dates.

For example, if asar prayer time on 13 November 2024, lat 3, long 101, timezone 8 is 4:32, the determination of the sun's altitude angle is:

Skyfield Installation

```
!pip install skyfield

# 2. Library Imports
from skyfield.api import load, wgs84
from skyfield.almanac import find_transits,find_
settings, find_risings
from datetime import datetime, timedelta
import math

# 3. Load Astronomical Data
ts = load.timescale()
eph = load('de440s.bsp')
sun = eph['Sun']
```

```
# 4. Define Observer's Location
latitude=3
longitude = 101
timezone = 8

# 5. Define Date and Time
day = 13
month = 11
year = 2024
hour = 16
minute = 32
month_name = calendar.month_name[month]

# 6. Initialize Observer Location
observer = eph['Earth'] + wgs84.latlon(latitude,
longitude)

# 7. Calculate Sun's Apparent Position
sun_astro = observer.at(ts.utc(year, month, day, hour,
minute)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()

# 8. Output the Sun's Altitude
print(f'the Altitude of Sun at Asr prayer time on
{day} {month_name} {year}, at coordinate lat:
{latitude}, long: {longitude}, tz: {timezone}, at
{hour}:{minute} Local Time is {sun_alt.degrees}')
```

This Python code calculates the **altitude of the Sun** (in degrees) at a specific location, date, and time using the **Skyfield library**. Here's how the code works:

Library Imports

- **skyfield.api**: Provides tools to load astronomical data and calculate celestial positions.
- **skyfield.almanac**: Contains functions for calculating astronomical events, though not directly used here.
- **datetime, timedelta**: Handles date and time manipulation.
- **math**: Not used in this snippet but available for rounding operations if needed.

Load Astronomical Data

```
ts = load.timescale()
eph = load('de440s.bsp')
sun = eph['Sun']
```

- **ts = load.timescale()**: Initializes a timescale object for handling time-related calculations.
- **eph = load('de440s.bsp')**: Loads the DE440s ephemeris file containing accurate positional data for celestial objects.
- **sun = eph['Sun']**: Loads positional data for the Sun.

Define Observer's Location

```
latitude = 3
longitude = 101
timezone = 8
```

- **latitude and longitude**: Specify the geographic coordinates of the observer. These coordinates correspond to a location in Malaysia.
- **timezone**: The local timezone offset from UTC (Malaysia = +8).

Define Date and Time

```
day = 13
month = 6
year = 2024
hour = 16
minute = 32
```

- Specifies the exact **date** (June 13, 2024) and **time** (16:32 in local time) for which the sun's altitude will be calculated.

Initialize Observer Location

```
observer = eph['Earth'] + wgs84.latlon(latitude,
longitude)
```

- Combines the Earth's position with the observer's location using the **WGS84 geodetic system**.

Calculate Sun's Apparent Position

```
sun_astro = observer.at(ts.utc(year, month, day, hour
- timezone, minute)).observe(sun)
sun_app = sun_astro.apparent()
```

- **ts.utc(year, month, day, hour – timezone, minute)**: Converts the local time to UTC by subtracting the timezone offset (16:32 local time → 08:32 UTC).
- **observer.at()**: Specifies the observer's position at the given UTC time.
- **observe(sun)**: Calculates the Sun's relative position in the sky from the observer's location.
- **apparent()**: Adjusts for atmospheric effects (e.g., refraction) to provide the apparent position of the Sun.

Calculate Sun's Altitude

```
sun_alt, sun_az, distance = sun_app.altaz()
```

- **altaz()**: Computes the Sun's **altitude**, **azimuth**, and **distance** from the observer.
  - **sun_alt**: The altitude of the Sun in degrees (angle above the horizon).
  - **sun_az**: The azimuth of the Sun in degrees (angle measured clockwise from true north).
  - **distance**: The distance between the observer and the Sun (not used here).

Output the Sun's Altitude

```
print(f'the Altitude of Sun during Asr prayer time on
{day} {month_name} {year}, at coordinate lat:
{latitude}, long: {longitude}, tz: {timezone}, at
{hour}:{minute} Local Time is {sun_alt.degrees}')
```

- **sun_alt.degrees**: Converts the altitude from radians (default Skyfield unit) to degrees and prints it.
- The value represents the sun's angular elevation above the horizon:

```
the Altitude of sun during Asr prayer time on 13
November 2024, at coordinate lat: 3, long: 101, tz: 8,
at 16:32 Local Time is 33.77219002658052
```

VISUALIZATION OF SUN POSITION DURING ZUHUR PRAYER TIME

Exercise 2: Visualizing the Sun's Altitude at Zuhur Prayer Time for the Given Location and Date

Create a visualization that shows the sun's position in the sky at Zuhur prayer time for the given coordinates (latitude: 39.9°N, longitude: 116.4°E, time zone: UTC+8) on December 19, 2024, with elevation of 100 m.

First, determine the time of Zuhur using the given location.

```
#Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac

#Load Ephemeris Data and Planet Objects
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

# Variable Input

lat_location = 39.9
long_location = 116.4
timezone = 8
day = 19
month = 12
year = 2024
ele = 100
```

```
#input into
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)

#Range of Data

t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)

# Time of Solar Transit
t = almanac.find_transits(location, sun, t0, t1)
hour_solar_transit = t.utc.hour
minutes_solar_transit = t.utc.minute
second_solar_transit = t.utc.second

zuhur_time = hour_solar_transit + (minutes_solar_
transit / 60) + (second_solar_transit / 3600 ) +
timezone + 0.017778

zuhur_time = float(zuhur_time)
degrees = int(zuhur_time )
decimal_part = zuhur_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)

seconds = round((minutes_total - minutes) * 60)
sun_astro = location.at(ts.utc(year, month, day,
hour_solar_transit, minutes_solar_transit, second_
solar_transit)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()
# Check if the sun is above the horizon at zuhur time
if sun_alt.degrees <= 0:
    zuhur = "Zuhur Does Not Occur"
else:
    zuhur = f"Zuhur Occurs at {degrees}° {minutes}'
    {seconds}""

print(zuhur)
Zuhur Occurs at 12° 12' 36"
```

The calculated Zuhur prayer time for the specified location is 12:12:36. This result is derived from the values stored in the variables: degrees = 12, minutes = 12, and seconds = 36. These values represent the time when the sun has just passed its daily zenith. It is important to note that this calculation assumes conversion based on the local time (LTC) using the appropriate time zone offset.

However, if we intend to use these values for further astronomical calculations, such as determining the solar position, we must ensure that the computation is based on UTC (Universal Coordinated Time). This is crucial because astronomical algorithms, such as those used to determine solar altitude or azimuth, typically rely on standardized time references like UTC to maintain accuracy and consistency across different locations and dates.

From the given time, determine the position of the sun.

```
# Local time values
h = degrees
m = minutes
s = seconds

# Adjust local time to UTC
h_utc = h - timezone

# Compute observation time using Skyfield
sun_astro = location.at(ts.utc(year, month, day,
h_utc, m, s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()

print(sun_alt)
26deg 40' 41.6"
```

Now we get the position of the sun is 26deg 40' 41.6" of solar altitude. We can use this value to visualize the position of the sun. Visualization of the sun can be used using matplotlib. The code could be a little overwhelming, but the important thing is the variable.

```
# 1. Import Library
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import io # Import io to handle the image data in
memory

# 2. Set Observer and Sun Positions
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = sun_alt.degrees[0] # Angle in degrees
(negative for below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun
```

```
# 3. Calculate the Sun's position based on the angle
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position

# 4. Plot the Visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"
# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

stick_x, stick_y = 5, 0 # Position of stick figure (on
the horizon)
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Add sky gradient (blue)
sky_gradient = np.linspace(1, 0, 256).reshape(1, -1)
sky_gradient = np.vstack((sky_gradient, sky_gradient))
ax.imshow(sky_gradient, extent=[0, 10, -1.5, 5],
cmap='Blues', alpha=0.3, aspect='auto')

# Add ground (green)
ground = plt.Rectangle((0, -1.5), 10, 1.5,
color='darkgreen', alpha=0.3)
ax.add_patch(ground)
```

```
# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")

# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y, f"Sun Position ({altitude_
angle:.4f}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)

# Add labels and legend
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title(f"Sun's Altitude Visualization at Zuhur
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month_name}
{year} {degrees}: {minutes}: {seconds}", fontsize=14)

plt.show()
```

In the code above, the most important variable is sun_alt.degrees, which represents the sun's altitude in degrees at the specified moment. This value is critical for determining whether the sun meets the required condition for a given prayer time. The remaining code functions primarily as a template, providing structure for observation and visualization, and does not directly affect the outcome of the sun's altitude calculation. Additionally, when visualizing different prayer times (e.g., Subh, Zuhur, Asar, etc.), ensure that the graph title is updated accordingly by modifying the line:

```
ax.set_title(f"Sun's Altitude Visualization at Zuhur
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month_name} {year}
{degrees}: {minutes}: {seconds}", fontsize=14)
```

to reflect the **specific prayer time** being analyzed. The result of the coding visualization is shown in Figure 10.2.

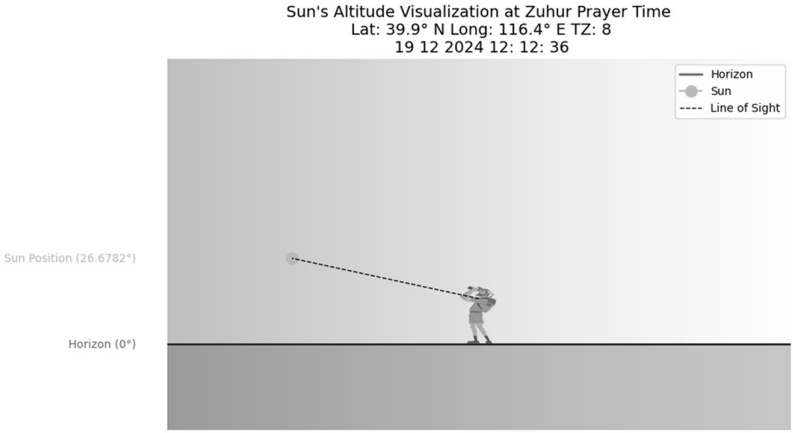


FIGURE 10.2 Visualization of sun position during Zuhr prayer time.

VISUALIZATION OF SUN POSITION DURING ASAR PRAYER TIME

Exercise 3: Visualizing the Sun's Altitude at Asar Prayer Time for the Given Location and Date

Create a visualization that shows the sun's position in the sky at Asar prayer time for the given coordinates (latitude: 39.9°N, longitude: 116.4°E, time zone: UTC+8) on December 19, 2024. In exercise 3, the given data only includes location coordinates and dates. Therefore, we need to calculate the sun's altitude at the time of Asar as explained in Chapter 7. After that, we can then create a visualization of the sun's altitude at the time of Asar. First, calculate the time of Asar prayer time

```
#Calculate Asr Prayer Time and Sun's Altitude
# 1. Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math
import calendar
```

```
# 2. Define Observer's Location
lat_location = 39.9
long_location = 116.4
timezone = 8

# 3. Define Date
day = 19
month = 12
year = 2024
month_name = calendar.month_name[month]

# 4. Load Astronomical Data
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

# 6. Initialize Observer Location
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)

# 7. Setup range of calculate date for calculating
solar transit
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)

# 8. Calculate the time of solar transit
t = almanac.find_transits(location, sun, t0, t1)

# 9. The position of sun altitude at the time of the
solar transit
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()

# 10. Calculate the length of the sun shadow during
transit
sun_shadow_transit = 1/(math.tan(math.radians(sun_a
lt.degrees)))

# 11. Calculate the length of the sun shadow at Asr
Prayer Time
sun_shadow_asar = 1 + sun_shadow_transit
```

```
# 12. Loop the sun shadow does not pass the length of
the asar sun shadow
# Start with hour
# Start with hour

test = 1
while True:

   # Calculate the shadow length
   sun_astro = location.at(ts.utc(year, month, day, h,
   m, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun
   sun_shadow= 1 / math.tan(math.radians(sun_alt.
   degrees))

   if sun_alt.degrees <= 0:
     break
   if test > 24:
     break

   if sun_shadow >= sun_shadow_asar:
     break # Exit the loop if the shadow length
     matches or exceeds the desired length
   h += 1

# Once the condition is met for hours, move to minutes
h_asar = h - 1
test=1
while True:
   # Calculate the shadow length
   sun_astro = location.at(ts.utc(year, month, day,
   h_asar, m, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun
   sun_shadow = 1 / math.tan(math.radians(sun_alt.
   degrees))

   if sun_alt.degrees <= 0:
     break
   if test > 1440:
     break

   if sun_shadow >= sun_shadow_asar:
     break # Exit the loop if the shadow length
     matches or exceeds the desired length
   m += 1
```

```
    # Increment time in minutes
m_asar = m - 1
test = 1

# Once the condition is met for minutes, move to
seconds
while True:
    # Calculate the shadow length
    sun_astro = location.at(ts.utc(year, month, day,
    h_asar, m_asar, s)).observe(sun)
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    sun_shadow = 1 / math.tan(math.radians(sun_alt.
    degrees))

    if sun_alt.degrees <= 0:
      break
    if test > 86400:
      break

    if sun_shadow >= sun_shadow_asar:
      break # Exit the loop if the shadow length
      matches or exceeds the desired length
    s += 1

    # Increment time in seconds

s_asar = s

asar_time = (h_asar + (m_asar) / 60 + s_asar / 3600) +
timezone
asar_time = float(asar_time)
degrees = int(asar_time)
decimal_part = asar_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees <= 0 or test >86400:
    asar = "Asar Does Not Occur"
else:
    asar = f"Asar Occurs at {degrees}° {minutes}'
    {seconds}""

print(asar)
Asar Occurs at 14° 34' 7"
```

Second, sun's altitude visualization,

```
# 1. Import Library
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import io # Import io to handle the image data in
memory

# 2. Set Observer and Sun Positions
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = sun_alt.degrees[0] # Angle in degrees
(negative for below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun

# 3. Calculate the Sun's position based on the angle
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position

# 4. Plot the Visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"
# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

stick_x, stick_y = 5, 0 # Position of stick figure (on
the horizon)
```

```
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Add sky gradient (blue)
sky_gradient = np.linspace(1, 0, 256).reshape(1, -1)
sky_gradient = np.vstack((sky_gradient, sky_gradient))
ax.imshow(sky_gradient, extent=[0, 10, -1.5, 5],
cmap='Blues', alpha=0.3, aspect='auto')

# Add ground (green)
ground = plt.Rectangle((0, -1.5), 10, 1.5,
color='darkgreen', alpha=0.3)
ax.add_patch(ground)
# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")

# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y, f"Sun Position ({altitude_
angle:.4f}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)

# Add labels and legend
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title(f"Sun's Altitude Visualization at Asr
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month_name}
{year} {degrees}: {minutes}: {seconds}", fontsize=14)

plt.show()
```

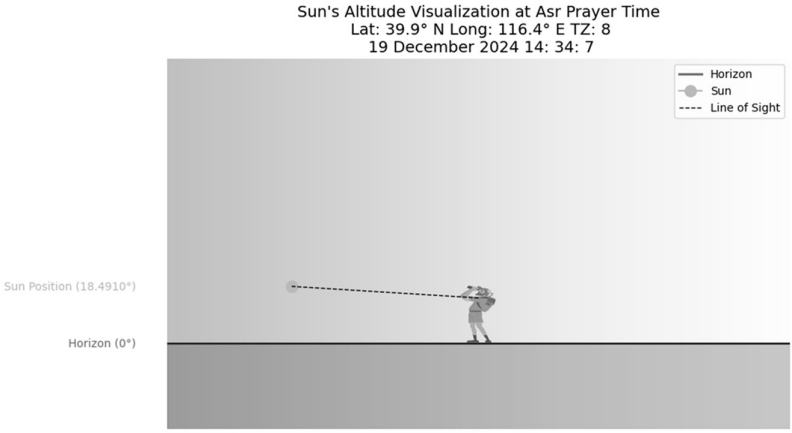


FIGURE 10.3 Visualization of sun position during Asar prayer time.

The result of the code implementation above is shown in Figure 10.3.

VISUALIZATION OF SUN POSITION DURING MAGHRIB PRAYER TIME

Exercise 4: Visualizing the Sun's Altitude at Maghrib Prayer Time for the Given Location and Date

Create a visualization that shows the sun's position in the sky at Maghrib prayer time for the given coordinates (latitude: 39.9°N, longitude: 116.4°E, time zone: UTC+8) on December 19, 2024, with elevation of 100 m. First, determine the maghrib prayer time.

```
#Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac

#Load Ephemeris Data and Planet Objects
ts = load.timescale()
```

```
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

# Variable Input
lat_location = 39.9
long_location = 116.4
timezone = 8
day = 19
month = 12
year = 2024
ele = 100

#input into
location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=0)

#Range of Data

t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)

from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)

t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second

maghrib_time = float(h + m / 60 + s / 3600 + timezone)
maghrib_time %= 24 # Ensure 24-hour clock format
maghrib_time = float(maghrib_time)
degrees = int(maghrib_time)
decimal_part = maghrib_time - degrees
```

```
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

sun_astro = location.at(ts.utc(year, month, day, h, m,
s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()
if sun_alt.degrees >= 0:
    maghrib = "Maghrib Does Not Occur"
else:
    maghrib = f"Maghrib Occurs at {degrees}° {minutes}'
    {seconds}""

print(maghrib)
Maghrib Occurs at 16° 53' 43"
```

First, determine sun altitude.

```
# Local time values
h = degrees
m = minutes
s = seconds

# Adjust local time to UTC
h_utc = h - timezone

# Compute observation time using Skyfield
sun_astro = location.at(ts.utc(year, month, day,
h_utc, m, s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()

print(sun_alt)
-01deg 09' 42.5"
```

Then visualize the position of the sun

```
# 1. Import Library
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import io # Import io to handle the image data in
memory
```

```
# 2. Set Observer and Sun Positions
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = sun_alt.degrees # Angle in degrees
(negative for below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun

# 3. Calculate the Sun's position based on the angle
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position

# 4. Plot the Visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"
# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

stick_x, stick_y = 5, 0 # Position of stick figure (on
the horizon)
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Add sky gradient (blue)
sky_gradient = np.linspace(1, 0, 256).reshape(1, -1)
```

```
sky_gradient = np.vstack((sky_gradient, sky_gradient))
ax.imshow(sky_gradient, extent=[0, 10, -1.5, 5],
cmap='Blues', alpha=0.3, aspect='auto')

# Add ground (green)
ground = plt.Rectangle((0, -1.5), 10, 1.5,
color='darkgreen', alpha=0.3)
ax.add_patch(ground)

# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")

# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y-1, f"Sun Position ({altitude_
angle:.4f}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)
# Add labels and legend
```

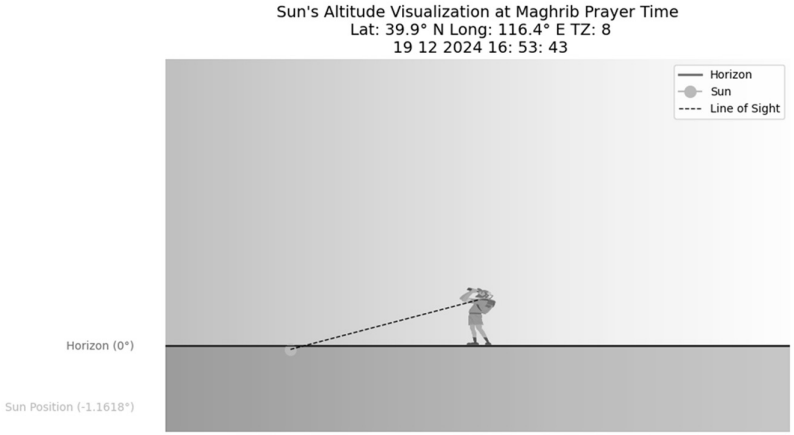


FIGURE 10.4 Visualization of sun position during Maghrib prayer time.

```
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title(f"Sun's Altitude Visualization at Maghrib
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month} {year}
{degrees}: {minutes}: {seconds}", fontsize=14)

plt.show()
```

The result of the code implementation above is shown in Figure 10.4.

VISUALIZATION OF SUN POSITION DURING ISYA' PRAYER TIME

Exercise 5: Visualizing the Sun's Altitude at Isya' Prayer Time for the Given Location and Date

Create a visualization that shows the sun's position in the sky at Isya' prayer time for the given coordinates (latitude: 39.9°N, longitude: 116.4°E, time zone: UTC+8) on December 19, 2024, for solar depression degree of 16. First, determine the Isya' prayer time

```
#Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math

#Load Ephemeris Data and Planet Objects
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

# Variable Input

lat_location = 39.9
long_location = 116.4
```

```
timezone = 8
day = 19
month = 12
year = 2024
ele = 100

location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)

from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)

t, y = almanac.find_settings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second
#print(h,m,s)

m=1
s=1
h=h+1

sun_astro = location.at(ts.utc(year, month, day, h,
m)).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, distance = sun_app.altaz()
#print(sun_alt)

# Start with hour

isya_angle = 16
elevation_correction = 0.0293 * math.sqrt(ele)
isha_angle_actual = -isya_angle -elevation_correction
#print(isya_angle_corrected)
```

```
# Start with hour

test = 1
while True:

   # Calculate the Solar Altitude
   sun_astro = location.at(ts.utc(year, month, day,
   h+1, m, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun

   if sun_alt.degrees >= 0:
     break
   if test > 24:
     break

   if sun_alt.degrees <= isha_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
   h =h+ 1

# Once the condition is met for hours, move to minutes
h_isya = h - 1
test = 1
while True:
   # Calculate the Solar Altitude
   sun_astro = location.at(ts.utc(year, month, day,
   h_isya, m, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun

   if sun_alt.degrees >= 0:
     break
   if test > 1440:
     break

   if sun_alt.degrees <= isha_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
   m += 1

   # Increment time in minutes
m_isya = m - 1
```

```
# Once the condition is met for minutes, move to
seconds
test = 1
while True:
   # Calculate the Solar Altitude
   sun_astro = location.at(ts.utc(year, month, day,
   h_isya, m_isya, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun

   if sun_alt.degrees >= 0:
     break
   if test > 86400:
     break

   if sun_alt.degrees <= isha_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
   s += 1

   # Increment time in seconds

s_isya = s

isya_time = float(h_isyak + m_isyak / 60 + s_isyak /
3600 + timezone)
isya_time %= 24 # Ensure 24-hour clock format
isya_time = float(isya_time)
degrees = int(isya_time)
decimal_part = isya_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    isya = "Isya' Does Not Occur"
else:
    isya = f"Isya' Occurs at {degrees}° {minutes}'
    {seconds}""

print(isya)
Isya' Occurs at 18° 19' 59"
```

Determine the sun altitude during Isya' Prayer Time

```
# Local time values
h = degrees
m = minutes
s = seconds

# Adjust local time to UTC
h_utc = h - timezone

# Compute observation time using Skyfield
sun_astro = location.at(ts.utc(year, month, day,
h_utc, m, s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()

print(sun_alt)
-16deg 17' 40.3"
```

Visualize the position of the sun during Isya' Prayer Time

```
# 1. Import Library
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import io # Import io to handle the image data in
memory

# 2. Set Observer and Sun Positions
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = sun_alt.degrees # Angle in degrees
(negative for below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun

# 3. Calculate the Sun's position based on the angle
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position
```

```
# 4. Plot the Visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"
# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

stick_x, stick_y = 5, 0 # Position of stick figure (on
the horizon)
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Add sky gradient (blue)
sky_gradient = np.linspace(1, 0, 256).reshape(1, -1)
sky_gradient = np.vstack((sky_gradient, sky_gradient))
ax.imshow(sky_gradient, extent=[0, 10, -1.5, 5],
cmap='Blues', alpha=0.3, aspect='auto')

# Add ground (green)
ground = plt.Rectangle((0, -1.5), 10, 1.5,
color='darkgreen', alpha=0.3)
ax.add_patch(ground)

# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")
```

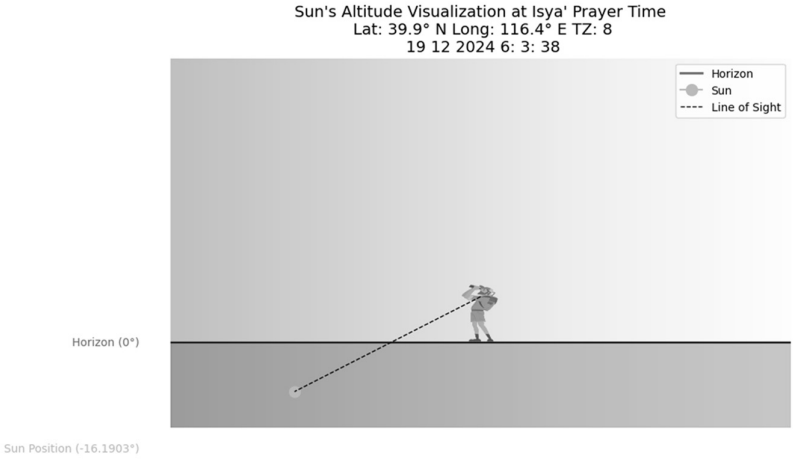


FIGURE 10.5 Visualization of sun position during Isya' prayer time.

```
# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y-1, f"Sun Position ({altitude_
angle:.4f}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)
# Add labels and legend
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title(f"Sun's Altitude Visualization at Isya'
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month} {year}
{degrees}: {minutes}: {seconds}", fontsize=14)

plt.show()
```

The result of the code implementation above is shown in Figure 10.5.

VISUALIZATION OF SUN POSITION DURING SUBH PRAYER TIME

Exercise 6: Visualizing the Sun's Altitude at Subh Prayer Time for the Given Location and Date

Create a visualization that shows the sun's position in the sky at Subh prayer time for the given coordinates (latitude: 39.9°N, longitude: 116.4°E, time zone: UTC+8) on December 19, 2024, for solar depression degree of 15. First, determine the Subh prayer time

```
#Import Necessary Function
from skyfield.api import load
from skyfield.api import N, S, E, W, wgs84
from skyfield import almanac
import math

#Load Ephemeris Data and Planet Objects
ts = load.timescale()
eph = load('de440s.bsp')
planets = load('de440s.bsp')
earth = planets['earth']
sun = planets['sun']

# Variable Input

lat_location = 39.9
long_location = 116.4
timezone = 8
day = 19
month = 12
year = 2024
ele = 100

location = earth + wgs84.latlon(lat_location, long_
location, elevation_m=ele)
t0 = ts.utc(year, month, day)
t1 = ts.utc(year, month, day + 1)
```

```
from skyfield.units import Angle
from numpy import arccos
from skyfield.earthlib import refraction

altitude_m = ele
earth_radius_m = 6378136.6
side_over_hypotenuse = earth_radius_m / (earth_
radius_m + altitude_m)
h = Angle(radians=-arccos(side_over_hypotenuse))
solar_radius_degrees = 16 / 60
r = refraction(0.0, temperature_C=15.0,
pressure_mbar=1030.0)

t, y = almanac.find_risings(location, sun, t0, t1,
horizon_degrees=-r + h.degrees - solar_radius_degrees)
h, m, s = t.utc.hour, t.utc.minute, t.utc.second

subh_angle = 16
elevation_correction = 0.0293 * math.sqrt(ele)
subh_angle_actual = -subh_angle - elevation_correction

# Start with hour

test = 1
while True:

   # Calculate the Solar Altitude
   sun_astro = location.at(ts.utc(year, month, day, h,
   m, s)).observe(sun)
   sun_alt, _, _ = sun_astro.apparent().altaz() # Get
   the altitude of the sun

   if sun_alt.degrees >= 0:
     break
   if test > 24:
     break

   if sun_alt.degrees <= subh_angle_actual:
       break # Exit the loop if the solar altitude
       located below -18 degree
   h -= 1

# Once the condition is met for hours, move to minutes
h_subh = h + 1
test =1
```

```
while True:
    # Calculate the Solar Altitude
    sun_astro = location.at(ts.utc(year, month, day,
    h_subh, m, s)).observe(sun)
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    if sun_alt.degrees >= 0:
      break
    if test > 1440:
      break

    if sun_alt.degrees <= subh_angle_actual:
        break # Exit the loop if the solar altitude
        located below -18 degree
    m -= 1

    # Increment time in minutes
m_subh = m + 1

# Once the condition is met for minutes, move to
seconds
test = 1
while True:
    # Calculate the Solar Altitude
    sun_astro = location.at(ts.utc(year, month, day,
    h_subh, m_subh, s)).observe(sun)
    sun_alt, _, _ = sun_astro.apparent().altaz() # Get
    the altitude of the sun
    if sun_alt.degrees >= 0:
      break
    if test > 86400:
      break

    if sun_alt.degrees <= subh_angle_actual:
        break # Exit the loop if the solar altitude
        located below -18 degree
    s -= 1

      # Increment time in seconds

    s_subh = s +1

subh_time = float(h_subh + (m_subh) / 60 + s_subh /
3600 + timezone)
```

```
subh_time %= 24 # Ensure 24-hour clock format
subh_time = float(subh_time)
degrees = int(subh_time)
decimal_part = subh_time - degrees
minutes_total = decimal_part * 60
minutes = int(minutes_total)
seconds = round((minutes_total - minutes) * 60)

if sun_alt.degrees >= 0 or test >86400:
    subh = "Subuh Does Not Occur"
else:
    subh = f"Subuh Occurs at {degrees}° {minutes}'
    {seconds}""

print(subh)
```

Determine the sun altitude of Subh

```
# Local time values
h = degrees
m = minutes
s = seconds

# Adjust local time to UTC
h_utc = h - timezone

# Compute observation time using Skyfield
sun_astro = location.at(ts.utc(year, month, day,
h_utc, m, s)).observe(sun)
sun_alt, _, _ = sun_astro.apparent().altaz()

print(sun_alt)
-16deg 11' 25.1"
```

Determine and visualize the position of the sun during Subh prayer time.

```
# 1. Import Library
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import io # Import io to handle the image data in
memory
```

```
# 2. Set Observer and Sun Positions
observer_x, observer_y = 5, 0 # Observer's position on
the horizon
altitude_angle = sun_alt.degrees # Angle in degrees
(negative for below horizon)
distance_to_sun = 3 # Arbitrary horizontal distance to
the Sun

# 3. Calculate the Sun's position based on the angle
sun_x = observer_x - distance_to_sun # Place the Sun
to the left of the observer
sun_y = observer_y + np.tan(np.radians(altitude_
angle)) * distance_to_sun # Calculate vertical
position

# 4. Plot the Visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Draw the horizon (straight line)
ax.plot([0, 10], [0, 0], color="green", linewidth=2,
label="Horizon")

# Add stick figure to the plot using direct download
link from Google Drive URL
image_url = "https://drive.google.com/uc?export
=download&id=1T7pLZNW6dF9PKxdOZ9UXZDbA84teRSdx"
# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
stick_figure = Image.open(io.BytesIO(response.
content))

stick_x, stick_y = 5, 0 # Position of stick figure (on
the horizon)
ax.imshow(stick_figure, extent=(stick_x - 0.3, stick_x
+ 0.3, stick_y, stick_y + 1))

# Add the Sun as an orange dot
ax.plot(sun_x, sun_y, marker="o", color="orange",
markersize=10, label="Sun")

# Add sky gradient (blue)
sky_gradient = np.linspace(1, 0, 256).reshape(1, -1)
sky_gradient = np.vstack((sky_gradient, sky_gradient))
```

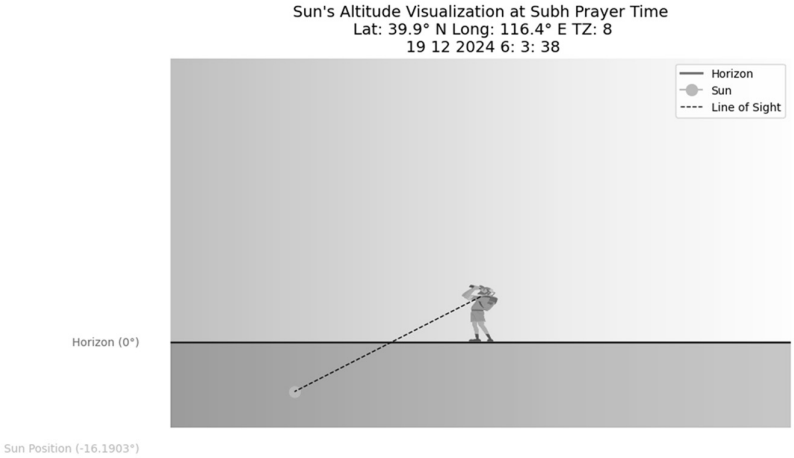


FIGURE 10.6 Visualization of sun position during Subh prayer time.

```
ax.imshow(sky_gradient, extent=[0, 10, -1.5, 5],
cmap='Blues', alpha=0.3, aspect='auto')

# Add ground (green)
ground = plt.Rectangle((0, -1.5), 10, 1.5,
color='darkgreen', alpha=0.3)
ax.add_patch(ground)

# Draw the line of sight (dashed line)
ax.plot([observer_x, sun_x], [observer_y + 0.8,
sun_y], color="black", linestyle="--", linewidth=1,
label="Line of Sight")

# Add altitude scale
ax.axhline(0, color="black", linestyle="-",
linewidth=1)
ax.text(-0.5, 0, "Horizon (0°)", va="center",
ha="right", fontsize=10, color="green")
ax.text(-0.5, sun_y-1, f"Sun Position ({altitude_
angle:.4f}°)", va="center", ha="right", fontsize=10,
color="orange")

# Adjust the plot
ax.set_xlim(0, 10)
ax.set_ylim(-1.5, 5)
# Add labels and legend
```

```
ax.axis("off")
ax.legend(loc="upper right")
ax.set_title(f"Sun's Altitude Visualization at Subh
Prayer Time\n Lat: {lat_location}° N Long: {long_
location}° E TZ: {timezone}\n {day} {month} {year}
{degrees}: {minutes}: {seconds}", fontsize=14)

plt.show()
```

The result of the code implementation above is shown in Figure 10.6.

Exercise 1: Subh

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Subuh (Fajr) prayer time for the date 27 December 2025 using a solar depression angle of −18 degrees, and visualize the sun's altitude when it reaches this angle before sunrise.

Exercise 2: Syuruk

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Syuruk (sunrise) time for the date 27 December 2025 and visualize the sun's altitude curve around sunrise.

Exercise 3: Zuhur

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Zuhur (prayer time) for the date 27 December 2025, and visualize the sun's altitude at its highest point on that day.

Exercise 4: Asar

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Asar prayer time for the date 27 December 2025 using both the standard shadow ratio (1×) and the Hanafi method (2×), and visualize the sun's altitude now of each Asar time.

Exercise 5: Maghrib

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Maghrib prayer time for the date 27 December 2025 based on the time of sunset and visualize the sun's altitude as it crosses the horizon.

Exercise 6: Isya'

Using the coordinates of Tokyo, which are 35.6894° North, 139.6916° East, elevation 100 m, and a time zone of GMT+9, calculate the Isya' (Isha) prayer time for the date 27 December 2025 using a solar depression angle of −18 degrees, and visualize the sun's altitude as it reaches that angle after sunset.

Lunar Crescent Observation Data Visualization

11

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension, NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble MATLAB, though its use is discouraged. SciPy uses Matplotlib. Matplotlib was originally written by John D. Hunter. Since then, it has developed an active community and is distributed under a BSD-style license. Michael Droettboom was appointed as the lead developer of Matplotlib shortly before John Hunter's passing in August 2012 and was later joined by Thomas Caswell. Matplotlib can be used to create maps, plot various types of graphs, and is highly flexible for customization. Therefore, in this class, we will learn how to use Matplotlib for generating visualizations of crescent moon (*hilal*) data.

FIRST PRACTICE: PENANG MALAYSIA

A. Horizon Generation

The horizon generation must cover 360 degrees in azimuth and 180 degrees in altitude. This ensures that the generated horizon can visualize the visibility data of the crescent moon (*hilal*) across various locations and dates. The determination for horizon generation is as follows:

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

fig, ax = plt.subplots(figsize=(20,10))
```


DOI: 10.1201/9781003649120-11

```
horizon_angles = np.linspace(0, 360, 720)
horizon_altitudes = np.zeros_like(horizon_angles)
ax.plot(horizon_angles, horizon_altitudes,
color='black', linestyle='-', linewidth=1)
```

The result of the above programming is shown in Figure 11.1.
The generated line represents the observer's horizon line.

B. Solar Position Visualization and Graph Labelling
To display the sun's position (represented in yellow), the following programming steps are implemented:

```
# Sun Position Determination
sun_az = 287
sun_alt = -1

moon_az = 283
moon_alt = 12.00474
daz = abs(sun_az-moon_az)
arcl = 13.03075
Location = "Penang, Malaysia"
day = 12
month = 1
year = 2024
month_name = calendar.month_name[month]

The graph is annotated using these key methods:
ax.set_xlabel('Azimuth (degrees)')
ax.set_ylabel('Altitude (degrees)')
ax.set_title(' The Position of the Moon and Sun During
Observation ')
```

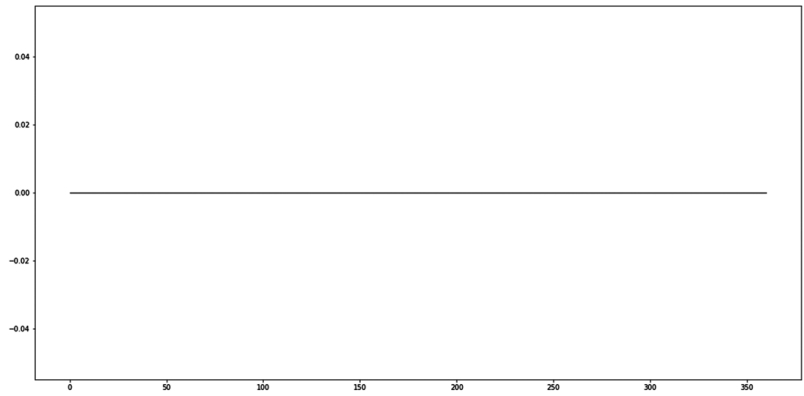


FIGURE 11.1 Observer horizon line.

With this, the altitude and azimuth have been labelled for easier interpretation. Typically, we observe the sun and the central reference line aligned at the midpoint. To ensure the black horizon line and the sun are centered, the following adjustments are made (Figure 11.2):

```
xlim_max = max(sun_az - (daz * 2), sun_az + (daz * 2))
xlim_min = min(sun_az - (daz * 2), sun_az + (daz * 2))

ax.set_xlim((xlim_min, xlim_max))
ax.set_ylim((sun_alt - 2), (moon_alt + 5))
```

C. Next is to display the moon's position. The moon's shape uses a crescent moon image. Please download the crescent moon position file from Google image search with a transparent background. Upload the crescent moon image to Google Drive with public access. The programming to determine the crescent moon's position is as follows:

```
from PIL import Image
import requests
import io # Import io to handle the image data in
memory
from matplotlib.offsetbox import OffsetImage,
AnnotationBbox
# Add the crescent moon image as a marker
opposite = moon_alt - sun_alt
adjacent = (moon_az - sun_az)
```

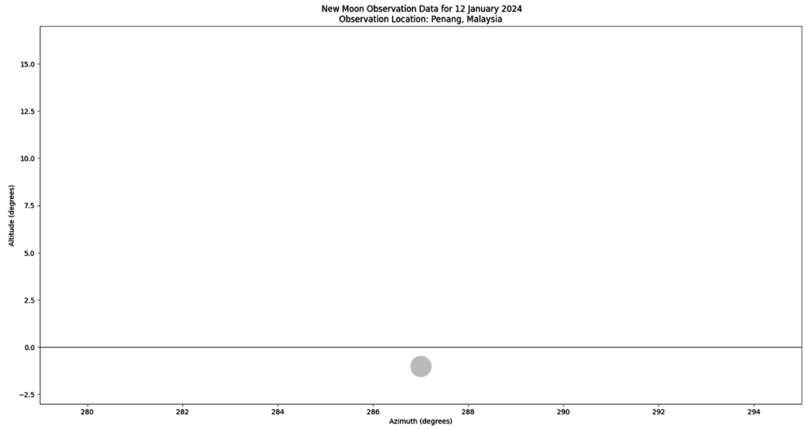


FIGURE 11.2 Observer horizon line with sun.

```
# Calculate the angle using the tangent function (TOA)
angle_rad = math.atan2(opposite, adjacent)

# Convert the angle to degrees
angle_degrees = (math.degrees(angle_rad))

# Load the crescent moon image
# Modified the Google Drive URL to get the direct
download link
image_url = "https://drive.google.com/uc?export
=download&id=1ZFbJ5pWYv3ZE4SY50Rme7w8iejzKRik4"

# Download the image content from the URL
response = requests.get(image_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
crescent_img = Image.open(io.BytesIO(response.
content))

# Rotate the crescent moon image based on the
calculated angle
rotated_img = crescent_img.rotate(angle_degrees)

# Add the crescent moon image as a marker
imagebox = OffsetImage(rotated_img, zoom=0.03) #
Adjust zoom as needed
# Using the correct moon position variables moon_az
and moon_alt
ab = AnnotationBbox(imagebox, (moon_az, moon_alt),
frameon=False)
ax.add_artist(ab)

plt.show() # Added this line to display the plot
```

Thus, the crescent moon's position relative to the sun can be clearly illustrated (Figure 11.3).

D. Sky Background Generation
The next step involves creating the celestial background. The implementation requires these steps:

```
from matplotlib.colors import LinearSegmentedColormap
# Import LinearSegmentedColormap
```

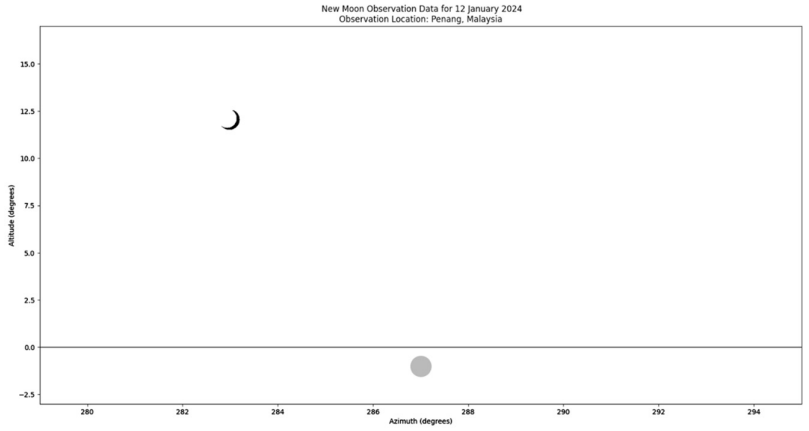


FIGURE 11.3 Observer horizon line with sun and moon.

```
# Sky Background Generation
sky = LinearSegmentedColormap.from_list('sky',
['blue','white', 'yellow', 'orange'])
extent = ax.get_xlim() + ax.get_ylim() # Adjusted to
use current xlim and ylim
ax.imshow([[0, 0], [1, 1]], cmap=sky,
interpolation='bicubic', extent=extent)
```

E. Next is to display the MABIMS criteria on the visualization. First, we'll visualize the elongation criterion. On this plot, elongation is represented as a radial distance from the sun's center. The implementation code is as follows (Figure 11.4):

```
from matplotlib.patches import Arc # Import the Arc
class
# Visualization of Elongation Criteria
sun_az_degrees = sun_az
sun_alt_degrees = sun_alt
moon_az_degrees = moon_az
moon_alt_degrees = moon_alt

kriteria_elongasi = 6.4
circle_radius = kriteria_elongasi

# Sun center coordinates
sun_center_x = sun_az
sun_center_y = sun_alt
```

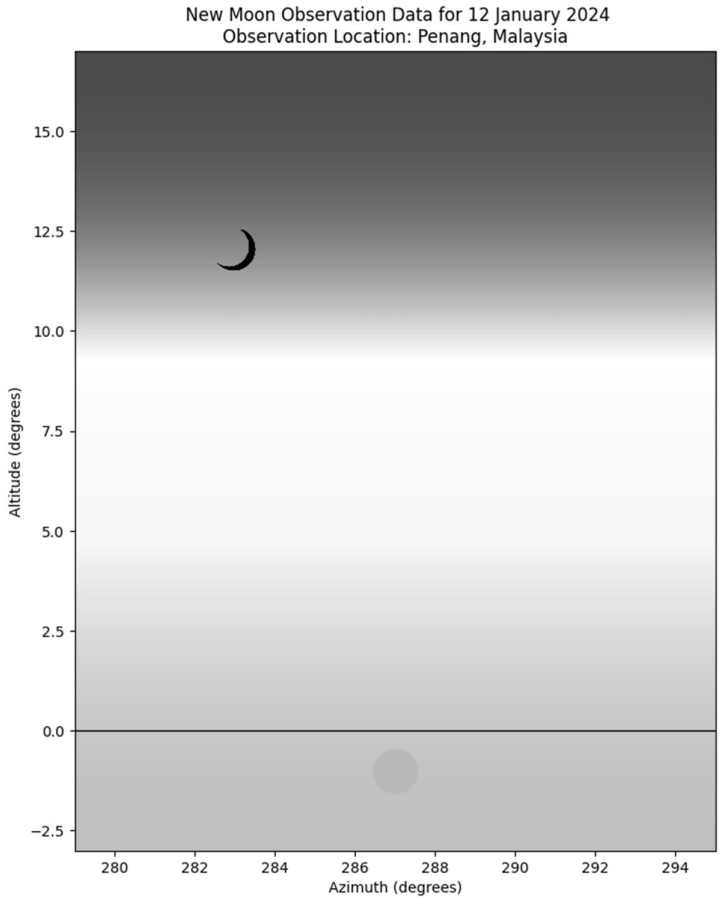


FIGURE 11.4 Visualization of observer horizon with sun and moon position.

```
# Semicircle parameters
radius = kriteria_elongasi
x1 = sun_az-kriteria_elongasi
x2 = sun_az+kriteria_elongasi

# Calculate the starting and ending angles of the
semicircle

arccos_value_start_angle = np.arccos(((x1 - sun_
center_x) / radius))
arccos_value_end_angle = np.arccos(((x2 - sun_
center_x) / radius))
```

```
if np.isnan(np.arccos((arccos_value_start_angle))):
    start_angle = 0.000001
else:
    start_angle = 180
    - np.degrees(arccos_value_start_angle)

if np.isnan(np.arccos((arccos_value_end_angle))):
    end_angle = 180+0.000001
else:
    end_angle = 180
    - np.degrees(arccos_value_end_angle)

print(start_angle,end_angle)
# Create the semicircle patch
semicircle_patch = Arc((sun_center_x, sun_center_y), 2
* radius, 2 * radius, theta1=start_angle,
theta2=end_angle,
             fill=False, color='blue', linestyle='--')
# Add the semicircle patch to the plot
ax.add_patch(semicircle_patch)
```

F. Next is the visualization of the Altitude Criterion. Altitude Criteria are calculated relative to the horizon (0° altitude). The implementation will display as shown in Figures 11.5–11.8:

```
# Visualization of Altitude Criteria
kriteria_altitude = 3
horizontal_line_y = kriteria_altitude
x1 = 0
x2 = sun_az-kriteria_elongasi
ax.hlines(y=horizontal_line_y,xmin=x1,
xmax=x2,color='red', linestyle='--')

x11 = sun_az+kriteria_elongasi
x22 = 360
ax.hlines(y=horizontal_line_y,xmin=x11,
xmax=x22,color='red', linestyle='--')
```

G. Add Logo

```
# Load the Logo
# Modified the Google Drive URL to get the direct
download link
logo_url = "https://drive.google.com/uc?export
=download&id=1_HEq0C3bv33i54HlbrJKD4Zmdxap8Q8a"
```

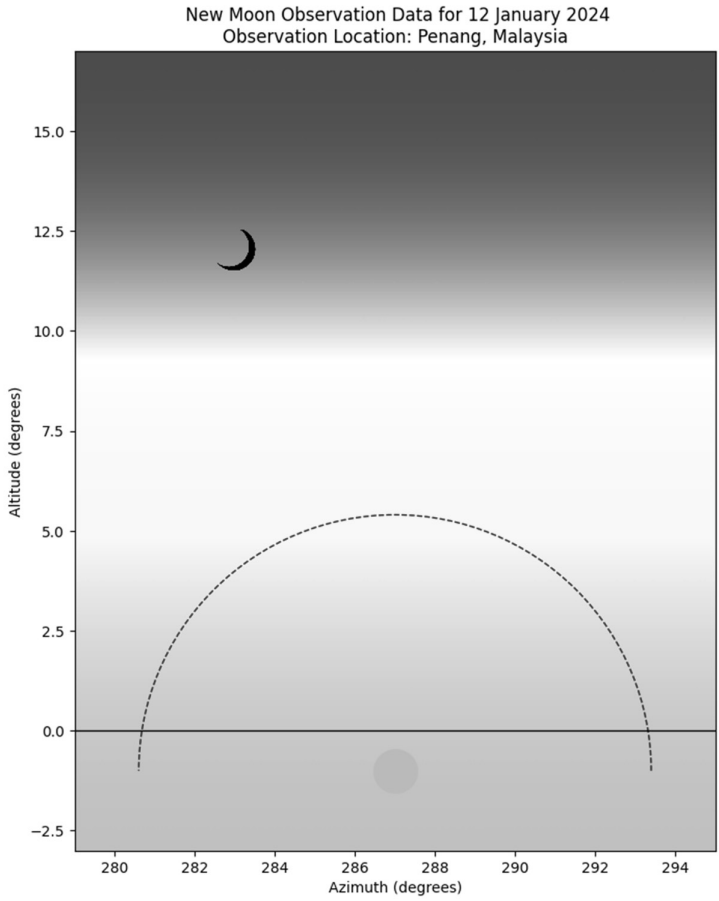


FIGURE 11.5 Visualization of observer horizon with sun and moon position with elongation criterion.

```
# Download the image content from the URL
response = requests.get(logo_url)
response.raise_for_status() # Raise an exception for
bad status codes
# Open the image from the downloaded content
logo_img = Image.open(io.BytesIO(response.content))

# Add the logo to the plot (bottom right corner)
logo_box = OffsetImage(logo_img, zoom=0.1) # Adjust
zoom as needed
```

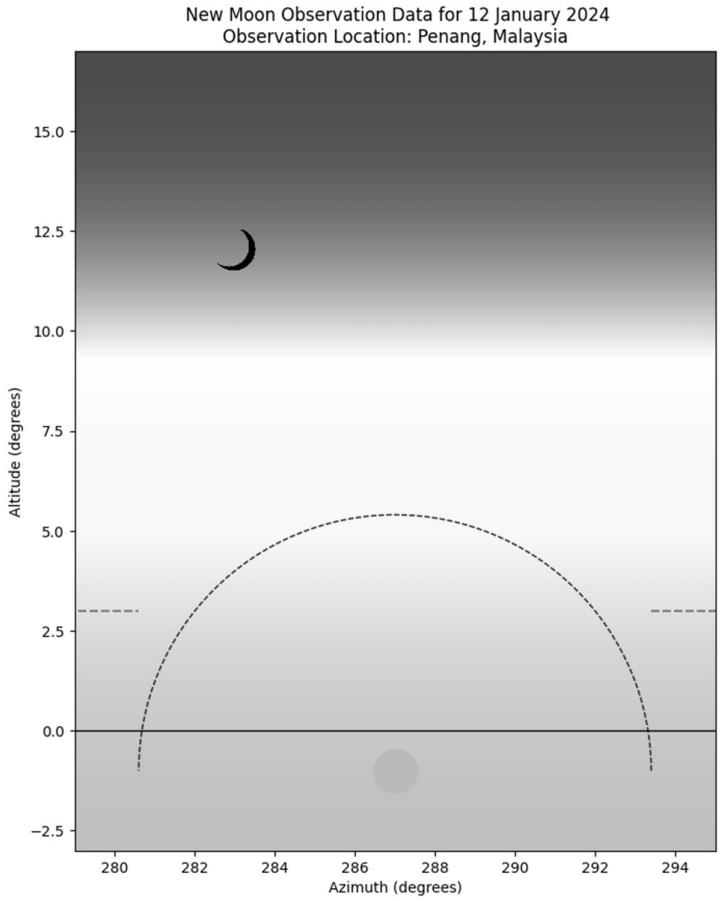


FIGURE 11.6 Visualization of observer horizon with sun and moon position with elongation and altitude criterion.

```
logo_ab = AnnotationBbox(logo_box, (xlim_max - 2.5,
sun_alt - 1), frameon=False) # Position adjusted based
on plot limits
ax.add_artist(logo_ab)
```

H. Plot the Moon's Altitude Line and Elongation

```
# Altitude Line
ax.vlines(x=moon_az, ymin=0, ymax=moon_alt,
color='blue', linestyle='--')
```

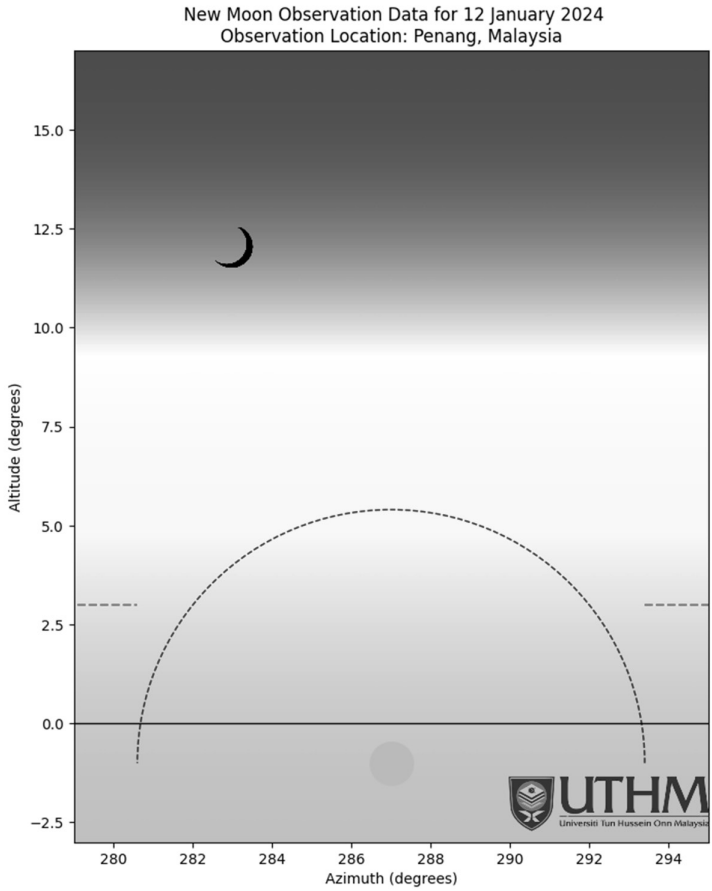


FIGURE 11.7 Visualization of observer horizon with sun and moon position with elongation and altitude criterion, added with logo.

```
ax.text(moon_az+0.3, moon_alt/2, f'Moon Altitude :
{moon_alt:.2f}', color='blue', fontsize=10, ha='left')

# Elongation Line
ax.plot([moon_az, sun_az], [moon_alt, sun_alt],
color='green', linestyle='--')
ax.text(sun_az-2, arcl/2-1, f'Elongation: {arcl:.2f}',
color ='green', fontsize=10, ha='left')

plt.show() # Added this line to display the plot
```

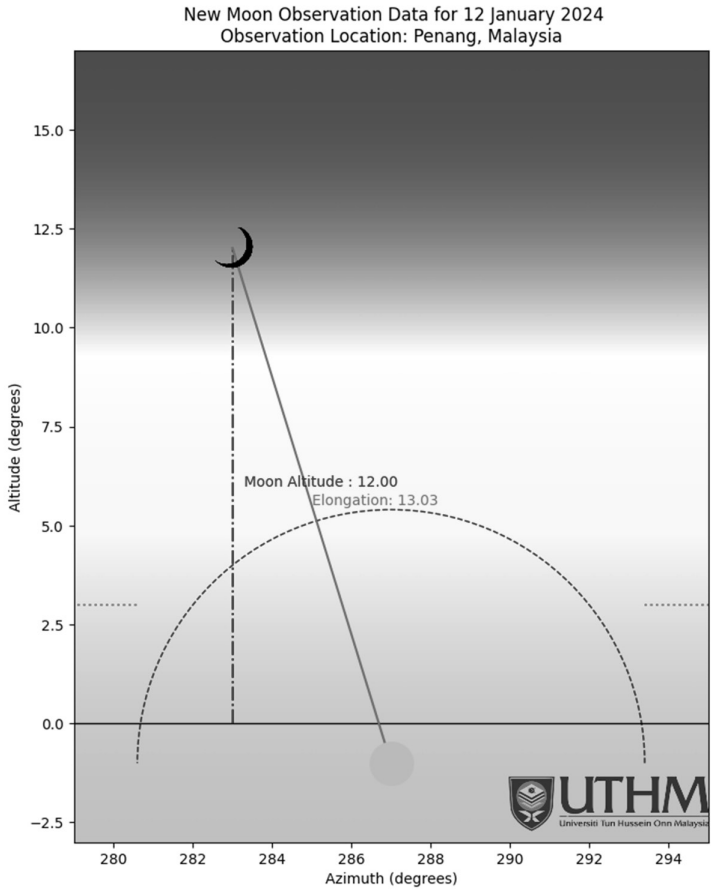


FIGURE 11.8 Visualization of observer horizon with sun and moon position with elongation and altitude criterion, added with logo and dash line.

SECOND PRACTICE: BANDA ACEH INDONESIA

Exercise 1

Visualize new moon observation data for May 27, 2025, on Observation Location: Banda Aceh, Indonesia, Lat: 5.548290 N, 95.323753 East, UTC+7.

```
# Install required libraries
!pip install skyfield # Astronomical calculations
!pip install numpy # Numerical operations
!pip install scipy # Scientific computing
!pip install matplotlib # Plotting library
!pip install tabulate # Pretty-print tables

# Import necessary modules
from skyfield import almanac
from skyfield.api import Topos,load
from skyfield import api
import numpy as np
from skyfield.api import N, S, E, W, load, wgs84
from skyfield.api import Topos, load,
Angle,GREGORIAN_START
import math
from scipy.ndimage import rotate
import calendar
from tabulate import tabulate
from matplotlib.patches import Arc
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import matplotlib.image as mpimg
from matplotlib.offsetbox import OffsetImage,
AnnotationBbox
from PIL import Image
import requests
import io # Import io to handle the image data in
memory

# Load planetary ephemeris data
planets = load('de421.bsp')
earth = planets['earth']
sun = planets['sun']
moon = planets['moon']
h_maghrib = 0 # Maghrib hour
m_maghrib = 0 # Maghrib minute

# Initialize time scale and ephemeris
ts = load.timescale()
eph = api.load('de421.bsp')

# Set observation location and time parameters
Lokasi = "Banda Aceh, Indonesia"
lat_titik1 = 5.548290 # Latitude of observation point
```

```
lon_titik1 = 95.323753 # Longitude of observation
point
tz = 7 # Timezone offset (UTC+7)
year = 2025 # Year of observation
month = 5 # Month of observation
day = 27 # Day of observation

# Create observer location object
location_titik1 = wgs84.latlon(lat_titik1 * N, lon_
titik1 * E)
observer_titik1 = eph['Earth'] + location_titik1
print(observer_titik1)

def settime(year, month, day, observer_titik, x):
  """Calculate setting time of celestial body with
  timezone adjustment"""
  # Set time range for search (current day to next day)
  t0 = ts.utc(year,month,day)
  t1 = ts.utc(year, month,day+1)

  # Find setting time of celestial body x
  t,y = almanac.find_settings(observer_titik, x, t0, t1)

  # Extract UTC time components
  h_set_transit_notz,m_set_transit_notz,s_set_transi
  t_notz = (int(t.utc.hour)),(int(t.utc.minute)),(i
  nt(t.utc.second))

  # Convert to decimal hours and add timezone offset
  time_set = (h_set_transit_notz + ((m_set_transit_
  notz) / 60 + s_set_transit_notz / 3600))+tz

  # Convert back to hours, minutes, seconds
  h_set, d = divmod(time_set, 1)
  h_set = int(h_set)
  m_set, s = divmod(d * 60, 1)
  m_set = int(m_set+1+4/60) # Add small adjustment
  s_set = int(s * 60)
  # Handle overflow in seconds
  if s_set >= 60:
      m_set += s_set // 60
      s_set = s_set % 60

  # Handle overflow in minutes
  if m_set >= 60:
```

```
        h_set += m_set // 60
        m_set = m_set % 60

    # Ensure hours wrap around a 24-hour clock
    h_set = h_set % 24
    waktu_terbenam = f"{h_set}:{m_set:02}:{s_set:02}"

    return waktu_terbenam

# Calculate sunset time
Objek = 'Matahari'
sun_set = settime(year, month, day, observer_titik1,
sun)

# Calculate moonset time
Objek = 'Bulan'
moon_set = settime(year, month, day, observer_titik1,
moon)

# Set Julian calendar cutoff for historical dates
ts.julian_calendar_cutoff = GREGORIAN_START
location = Topos(latitude_degrees=lat_titik1,
longitude_degrees=lon_titik1, elevation_m=1)

def settime(year, month, day, observer_titik, x):
    """Alternative version that returns hours and
    minutes separately"""
    # Similar to previous function but returns numeric
    values
    t0 = ts.utc(year,month,day)
    t1 = ts.utc(year, month,day+1)
    t,y = almanac.find_settings(observer_titik, x, t0,
    t1)
    h_set_transit_notz,m_set_transit_notz,s_set_transi
    t_notz = (int(t.utc.hour)),(int(t.utc.minute)),(i
    nt(t.utc.second))
    time_set = (h_set_transit_notz + ((m_set_transit_
    notz) / 60 + s_set_transit_notz / 3600))

    h_set, d = divmod(time_set, 1)
    h_set = int(h_set)
    m_set, s = divmod(d * 60, 1)
    m_set = int(m_set+1+4/60)
    s_set = int(s * 60)
```

```
    if s_set >= 60:
        m_set += s_set // 60
        s_set = s_set % 60

    if m_set >= 60:
        h_set += m_set // 60
        m_set = m_set % 60

    h_set = h_set % 24
    waktu_terbenam = f"{h_set}:{m_set:02}:{s_set:02}"

    return h_set,m_set

# Get sunset time in numeric format
hsunset,msunset = settime(year, month, day, observer_
titik1, sun)

# Create observation location object
boston = earth + Topos(latitude_degrees=lat_titik1,
longitude_degrees=lon_titik1, elevation_m=0)

# Calculate sun position at sunset
sun_astro = boston.at(ts.utc((year), (month), (day),
(hsunset), (msunset))).observe(sun)
sun_app = sun_astro.apparent()
sun_alt, sun_az, sun_distance = sun_app.altaz()

# Calculate moon position at sunset
moon_astro = boston.at(ts.utc((year), (month), (day),
(hsunset), (msunset))).observe(moon)
moon_app = moon_astro.apparent()
moon_alt, moon_az, moon_distance = moon_app.altaz()

# Calculate differences between moon and sun positions
beza_altitud_bulan_matahari = abs(moon_alt.degrees-
sun_alt.degrees)
daz = abs(moon_az.degrees-sun_az.degrees)
str_date = f'{day}/{month}/{year}'

# Store observation data
altitud_bulan = moon_alt.degrees
elongasi = sun_app.separation_from(moon_app).degrees

# Prepare data for table display
data = [["Date", str_date],
```

```
        ["Location", Lokasi],
        ["Sunset Time", sun_set],
        ["Moonset Time", moon_set],
        ["Moon Altitude", altitud_bulan],
        ["Elongation", elongasi],
]

col_names = ["Moon-Sun Data", "Value"]

# Create formatted table
data_table = (tabulate(data, headers=col_names))

# Create visualization figure
fig, ax = plt.subplots(figsize=(20,10))

# Draw horizon line
horizon_angles = np.linspace(0, 360, 720)
horizon_altitudes = np.zeros_like(horizon_angles)
ax.plot(horizon_angles, horizon_altitudes,
color='black', linestyle='-', linewidth=1)

# Get sun and moon positions in degrees
sun_az = sun_az.degrees
sun_alt = sun_alt.degrees
moon_az = moon_az.degrees
moon_alt = moon_alt.degrees

arcl = elongasi
Location = Lokasi
month_name = calendar.month_name[month]

# Plot sun position
ax.scatter(sun_az, sun_alt, color='orange',
label='Sun', zorder=10, s=900)
ax.set_xlabel('Azimuth (degrees)')
ax.set_ylabel('Altitude (degrees)')
ax.set_title(f'New Moon Observation Data for {day}
{month_name} {year}\nObservation Location:
{Location}')

# Set plot limits based on sun-moon positions
xlim_max = max(sun_az - (daz * 2), sun_az + (daz * 2))
xlim_min = min(sun_az - (daz * 2), sun_az + (daz * 2))
ax.set_xlim((xlim_min, xlim_max))
ax.set_ylim((sun_alt - 2), (moon_alt + 5))
```

```
# Calculate angle for moon crescent orientation
opposite = moon_alt - sun_alt
adjacent = (moon_az - sun_az)
angle_rad = math.atan2(opposite, adjacent)
angle_degrees = (math.degrees(angle_rad))

# Load and rotate crescent moon image
image_url = "https://drive.google.com/uc?export
=download&id=1ZFbJ5pWYv3ZE4SY50Rme7w8iejzKRik4"
response = requests.get(image_url)
response.raise_for_status()
crescent_img = Image.open(io.BytesIO(response.
content))
rotated_img = crescent_img.rotate(angle_degrees)

# Add rotated moon image to plot
imagebox = OffsetImage(rotated_img, zoom=0.03)
ab = AnnotationBbox(imagebox, (moon_az, moon_alt),
frameon=False)
ax.add_artist(ab)

# Create sky background gradient
sky = LinearSegmentedColormap.from_list('sky',
['blue','white', 'yellow', 'orange'])
extent = ax.get_xlim() + ax.get_ylim()
ax.imshow([[0, 0], [1, 1]], cmap=sky,
interpolation='bicubic', extent=extent)

# Visualization of elongation criteria (6.4°
semicircle)
kriteria_elongasi = 6.4
circle_radius = kriteria_elongasi
sun_center_x = sun_az
sun_center_y = sun_alt
radius = kriteria_elongasi
x1 = sun_az-kriteria_elongasi
x2 = sun_az+kriteria_elongasi

# Calculate angles for semicircle
arccos_value_start_angle = np.arccos(((x1 - sun_
center_x) / radius))
arccos_value_end_angle = np.arccos(((x2 - sun_
center_x) / radius))

if np.isnan(np.arccos((arccos_value_start_angle))):
    start_angle = 0.000001
```

```
else:
    start_angle = 180
    - np.degrees(arccos_value_start_angle)

if np.isnan(np.arccos((arccos_value_end_angle))):
    end_angle = 180+0.000001
else:
    end_angle = 180
    - np.degrees(arccos_value_end_angle)

# Draw semicircle around sun position
semicircle_patch = Arc((sun_center_x, sun_center_y), 2
* radius, 2 * radius,
               theta1=start_angle, theta2=end_angle,
               fill=False, color='blue',
               linestyle='--')
ax.add_patch(semicircle_patch)

# Visualization of altitude criteria (3° line)
kriteria_altitude = 3
horizontal_line_y = kriteria_altitude
x1 = 0
x2 = sun_az-kriteria_elongasi
ax.hlines(y=horizontal_line_y, xmin=x1, xmax=x2,
color='red', linestyle='--')

x11 = sun_az+kriteria_elongasi
x22 = 360
ax.hlines(y=horizontal_line_y, xmin=x11, xmax=x22,
color='red', linestyle='--')

# Load and add logo
logo_url = "https://drive.google.com/uc?export
=download&id=1_HEq0C3bv33i54HlbrJKD4Zmdxap8Q8a"
response = requests.get(logo_url)
response.raise_for_status()
logo_img = Image.open(io.BytesIO(response.content))
logo_box = OffsetImage(logo_img, zoom=0.1)
logo_ab = AnnotationBbox(logo_box, (xlim_max - 2.5,
sun_alt - 1), frameon=False)
ax.add_artist(logo_ab)

# Draw moon altitude line and label
ax.vlines(x=moon_az, ymin=0, ymax=moon_alt,
color='blue', linestyle='--')
```

```
ax.text(moon_az+0.3, moon_alt/2, f'Moon Altitude:
{moon_alt:.2f}°', color='blue', fontsize=10,
ha='left')

# Draw elongation line and label
ax.plot([moon_az, sun_az], [moon_alt, sun_alt],
color='green', linestyle='--')
ax.text(sun_az+2, arcl/2-1, f'Moon Elongation:
{arcl:.2f}°', color='green', fontsize=10, ha='left')

# Determine visibility based on MABIMS criteria
if moon_alt >= kriteria_altitude and arcl >=
kriteria_elongasi:
    kenampakan = "Visible"
    nama_kriteria = "MABIMS Criteria Met"
else:
    kenampakan = "Not Visible"
    nama_kriteria = "MABIMS Criteria Not Met"

# Add information box
info_text = (f'Moon Data:\n'
          f'Altitude: {moon_alt:.2f}°\n'
          f'Elongation: {arcl:.2f}°\n'
          f'Visibility: {kenampakan}\n'
          f'Criteria: {nama_kriteria}')
ax.text(xlim_max-7, moon_alt+3.5, info_text,
     fontsize=12, ha='left', va='top',
     bbox=dict(facecolor='white', alpha=0.8))

# Add horizon and criteria labels
ax.text(xlim_min+1, 0.1, "Horizon", fontsize=10,
ha='left')
ax.text(xlim_min+1, kriteria_altitude+0.1, "MABIMS
2021 Criteria", fontsize=10, ha='left')

plt.show()

# Print data table
print('\n')
print(data_table)
```

The result of the above code is shown in Figure 11.9.

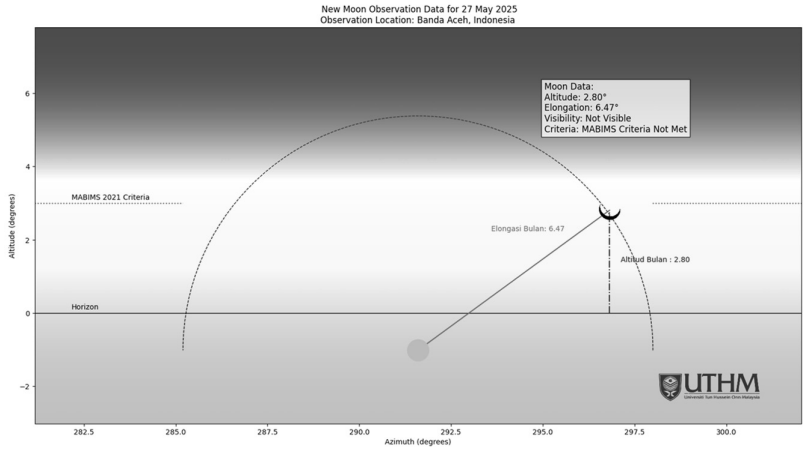


FIGURE 11.9 Visualization of sun position at Banda Acheh Indonesia.

Exercise 2

Visualize new moon observation data for June 25, 2025, on Observation Location: Singapore, Lat: 1.290270 N, 103.851959 E, UTC+8.

Exercise 3

Visualize new moon observation data for September 22, 2025, on Observation Location: Kuala Lumpur, Malaysia, Lat: 3.140853 N, 101.693207 E, UTC+8.

Exercise 4

Visualize new moon observation data for October 21, 2025, on Observation Location: Bandar Seri Begawan, Brunei, Lat: 4.890278 N, 114.942222 E, UTC+8.

Exercise 5

Visualize new moon observation data for December 20, 2025, on Observation Location: Naypyidaw, Myanmar, Lat: 19.7475 N, 96.115 E, UTC+6.5.

Conclusion

The fusion of Python programming and Islamic Astronomy presented in this textbook is not just a convergence of science and faith; it reflects the long-standing Islamic tradition that values precision, observation, and knowledge. Islamic Astronomy (Ilmu Falak) has historically served as a critical field to determine acts of worship such as prayer times, Qibla direction, and the Hijri calendar. In modern times, the use of computational tools such as Python enhances the accuracy, reproducibility, and clarity of these determinations.

Throughout this book, we have systematically explored how Python can be utilized to model and compute key astronomical phenomena relevant to Islamic practices. Beginning with fundamental programming skills, we built up to advanced computations including solar transit, shadow length analysis, and lunar crescent visibility prediction. We introduced the Skyfield library as a core tool, highlighting its ability to interface directly with astronomical data and simulate celestial motion with high precision.

In Chapters 5 through 7, we demonstrated how to calculate the direction of the Qibla and determine prayer times such as Zuhur, Asar, Maghrib, Isya', Syuruk, and Subh. These calculations were grounded in real astronomical parameters such as solar altitude, Equation of Time, and observer location. We emphasized the importance of considering elevation, atmospheric refraction, and solar declination, factors often overlooked in manual estimations. We also provided visualizations to help learners see the underlying logic and movement of the sun in relation to the Earth's surface.

In Chapter 8, we shifted to moonsighting, a subject of both scientific and sociological significance in the Muslim world. Here, readers were introduced to methods of determining the moon's altitude, age, elongation, and azimuth, all of which play crucial roles in validating the visibility of the lunar crescent (hilal). Through case studies and real-world examples, readers learned how to extract meaningful data that contributes to hilal visibility reports and the formulation of Hijri calendars.

Chapters 9 and 10 focused on visualization techniques that support understanding and teaching of Falak. With Python's graphical capabilities, learners could plot Qibla directions, solar azimuths, and sun positions at various prayer times. These visual outputs not only serve pedagogical purposes but also help in validating and communicating data effectively, particularly in official or community-level Falak determinations.

Finally, in Chapter 11, we explored lunar crescent observation data visualization using real observation scenarios from Malaysia and Indonesia. These practical examples tied together the themes of data, visibility, and Islamic calendar accuracy, showing that observation and computation must go hand-in-hand.

By combining modern computational skills with classical Islamic astronomical knowledge, this textbook aspires to produce a new generation of Muslim scholars, students, and professionals who are confident in applying scientific methods to religious obligations. This interdisciplinary skill set not only enhances one's understanding of Islamic rituals but also contributes to broader scientific literacy and critical thinking.

It is our hope that this book serves as a gateway for more Muslims, particularly students in higher education and religious institutions, to explore Islamic Astronomy as both an academic and spiritual pursuit. As we embrace the tools of modern science, we continue the legacy of great Muslim astronomers such as Al-Biruni, Al-Khwarizmi, and Al-Tusi, who exemplified the harmony between reason, revelation, and observation.

May this book inspire a deeper love for knowledge, a renewed commitment to precision in worship, and a stronger integration of technology in the service of our faith and community.

Wallāhu a'lam

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

References

Abas, A.-M., Safiai, M. H., Hasan, S. A., Azam, A. I., & Hussaini, R. (2022). Penentuan Waktu Ibadah Solat dan Puasa di Bangunan Pencakar Langit [Determining the times for prayer and fasting on skyscraper building]. *BITARA International Journal of Civilizational Studies and Human Sciences*, *6*(1), 53–59.

Adegoke, K. A. (2013). Ikhtilaf on moon sighting among Nigerian Muslims within the framework of Shari'ah. *Journal of Arabic and Islamic Studies*, *7*(2), 140–156.

Adegoke, K. A. (2017). Ādam al-Ilūrī and neo-ijtihād: An examination of his legal views on Ramadān fasting in Nigeria. *Journal of Arabic And Islamic Studies*, *20*(1), 11–25.

Ali, M. (2015). Is the British weather anti-Islamic? Prayer times, the ulama and application of the shari'a. *Contemporary Islam*, *9*(2), 171–187. https://doi.org/10.1007/s11562-014-0318-7

Al-Rajab, M., Loucif, S., & Al Risheh, Y. (2023). Predicting new crescent moon visibility applying machine learning algorithms. *Scientific Reports*, *13*(1), 6674. https://doi.org/10.1038/s41598-023-32807-x

Amin, M. F. (2018). Global Rasdhul Qibla: The probability of four times in a year study. *Jurnal Penelitian*, 15(2) 175–188.

Asrin, A., Hapsari, G. I., & Mutiara, G. A. (2018). Development of Qibla direction cane for blind using interactive voice command. *2018 6th International Conference on Information and Communication Technology (ICoICT)*, 216–221. https://doi.org/10.1109/ICoICT.2018.8528769

Blank, J., & Deb, K. (2020). Pymoo: Multi-objective optimization in Python. *IEEE Access*, *8(1)*, 89497–89509. https://doi.org/10.1109/ACCESS.2020.2990567

Elmhamdi, A., Roman, M. T., Peñaloza-Murillo, M. A., Pasachoff, J. M., Liu, Y., Al-Mostafa, Z. A., Maghrabi, A. H., Oloketuyi, J., & Al-Trabulsy, H. A. (2024). Impact of the eclipsed sun on terrestrial atmospheric parameters in desert locations: A comprehensive overview and two events case study in Saudi Arabia. *Atmosphere*, *15*(1), 62. https://doi.org/10.3390/atmos15010062

Faid, M. S., Husien, N., Shariff, N. N. M., Ali, M. O., Hamidi, Z. S., Zainol, N. H., & Sabri, S. N. U. (2016). Monitoring the level of light pollution and its impact on astronomical bodies naked-eye visibility range in selected areas in Malaysia using the sky quality meter. *2016 International Conference on Industrial Engineering, Management Science and Application (ICIMSA)*, 1–6. https://doi.org/10.1109/ICIMSA.2016.7504020

Faid, M. S., Md Shariff, N. N., & Hamidi, Z. S. (2019). The risk of light pollution on sustainability. *ASM Science Journal*, *12*(Special Issue 2), 134–142.

Faid, M. S., Mohd Nawawi, M. S. A., & Mohd Saadon, M. H. (2023). *HilalPy: Analysis tool for lunar crescent visibility criterion*. ascl:2307.031-ascl:2307.031.

Faid, M. S., Mohd Nawawi, M. S. A., & Mohd Saadon, M. H. (2024). *Design, development and analysis of lunar crescent visibility criterion with Python*. CRC Press.

Faid, M. S., Mohd Nawawi, M. S. A., & Norman, M. P. (2021). Issue of sustainability on light pollution from the perspective of Maqasid Shariah. *Journal of Fatwa Management and Research*, *26*(2), 1–9. https://doi.org/10.33102/jfatwa.vol26no2.390

Faid, M. S., Nawawi, M. M., & Saadon, M. M. (2023). Analysis tool for lunar crescent visibility criterion based on integrated lunar crescent database. *Astronomy and Computing*, *45(1)*, 100752.

Faid, M. S., Nawawi, M. S. A. M., Wahab, R. A., & Ahmad, N. (2023). HilalPy: Software to analyse lunar sighting criteria. *Software Impacts*, *18*, 100593.

Faid, M. S., Mohd Nawawi, M. S. A., Mohd Saadon, M. H., Wahid, K., & Norman, P. (2025). Methods in determining new Hijri month: A thematic review from Islamic jurisprudence perspective. *Malaysian Journal of Syariah and Law*, *13*(1), 75–99. https://doi.org/10.33102/mjsl.vol13no1.687

Faid, M. S., Nahwandi, M. S., Nawawi, M. S. A. B. M., Zaki, N. B. A., & Saadon, M. H. M. (2022). Development of Qibla direction determinant using sun shadow. *Online Journal of Research in Islamic Studies*, *9*(1), 89–102.

Faid, M. S., Nawawi, M. S. A. M., Saadon, M. H. M., Ahmad, N., & Ali @ Mat Zin, A. (2022). Islamic historical review on the middle age lunar crescent visibility criterion. *Journal of Al-Tamaddun*, *17*(1). https://doi.org/10.22452/JAT.vol17no1.9

Faid, M. S., Shariff, N. N. M., Hamidi, Z. S., Kadir, N., Ahmad, N., & Wahab, R. A. (2018). Semi empirical modelling of light polluted twilight sky brightness. *Jurnal Fizik Malaysia*, *39*(2), 30059–30067.

Faid, M. S., Shariff, N. N. M., Hamidi, Z. S., Sabri, S. N. U., Zainol, N. H., Husien, N. H., & Ali, M. O. (2016). Monitoring the level of light pollution and its impact on astronomical bodies naked-eye visibility range in selected areas in Malaysia using sky quality meter. *Journal of Industrial Engineering and Management Science*, *2016*(1), 1–18. https://doi.org/10.13052/jiems2446-1822.2016.007

Faid, M. S., Shariff, N. N. M., Hamidi, Z. S., Wahab, R. A., Ahmad, N., Mohd Nawawi, M. S. A., & Nahwandi, M. S. (2024). Alteration of twilight sky brightness profile by light pollution. *Scientific Reports*, *14*(1), 26682.

Faid, M. S., Nawawi, M. S. A. M., Saadon, M. H. M., Wahab, R. A., Ahmad, N., Nahwandi, M. S., Ahmed, I., & Mohamed, I. (2024). Assessment and review of modern lunar crescent visibility criterion. *Icarus*, *412*, 115970.

Faid, M. S., Nawawi, M. S. A. M., Saadon, M. H. M., Nahwandi, M. S., Shariff, N. N. M., Hamidi, Z. S., Wahab, R. A., Norman, M. P., & Ahmad, N. (2023). Confirmation methodology for a lunar crescent sighting report. *New Astronomy*, *103*, 102063–102063. https://doi.org/10.1016/j.newast.2023.102063

Gharaybeh, M. (2025). Jurisprudential reliance on astronomical calculations in determining the beginnings of the Hijri month. In H. M. K. Al Naimiy, H. M. Elmehdi, & I. A. Shehadi (Eds.), *Proceedings of the 14th Arabic Conference of the Arab Union for Astronomy and Space Sciences* (Vol. 420, pp. 160–177). Springer Nature Singapore. https://doi.org/10.1007/978-981-96-3276-3_13

Holwerda, B. W., Trenti, M., Clarkson, W., Sahu, K., Bradley, L., Stiavelli, M., Pirzkal, N., Marchi, G. D., Andersen, M., Bouwens, R., Ryan, R., Vledder, I. V., & Vlugt, D. V. D. (2016). Erratum: "Milky Way red dwarfs in the BoRG survey; Galactic scale-height and the distribution of dwarf stars in WFC3 imaging" (2014, ApJ, 788, 77). The Astrophysical Journal, 825(1), 82. https://doi.org/10.3847/0004-637X/825/1/82

Huda, N., Zaki, A., Ali, A. K., Wahab, R. A., & Niri, M. A. (2014). Penentuan waktu solat Subuh menggunakan Rubu‘ Mujayyab di Malaysia: The determination of Subuh prayer time by using Rubu‘ Mujayyab in Malaysia. Jurnal Fiqh, *11*(11), 97–118.

Ilyas, M. (1984). A modern guide to astronomical calculations of Islamic calendar, times and qibla. In *A modern guide to astronomical calculations of Islamic calendar*. https://ui.adsabs.harvard.edu/abs/1984mgta.book.....I

Ilyas, M. (1997). *Astronomy of Islamic calendar* (1st ed.). A. S. Nordeen.

Izzuddin, A., Imroni, M. A., Imron, A., & Mahsun, M. (2022). Cultural myth of eclipse in a Central Javanese village: Between Islamic identity and local tradition. *HTS Teologiese Studies / Theological Studies*, *78*(4). https://doi.org/10.4102/hts.v78i4.7282

Junaidi, A. (2022). Syahadah Rukyatulhilal using astro digital imaging: From subjectivity to objectivity. *De Jure: Jurnal Hukum Dan Syar'iah*, *14*(1), 58–74. https://doi.org/10.18860/j-fsh.v14i1.15062

King, D. A. (1993). *Astronomy in the service of Islam*. Routledge.

Lairgi, Y. (2025). *When astronomy meets AI: Manazel for crescent visibility prediction in Morocco* (Version 1). arXiv. https://doi.org/10.48550/ARXIV.2503.21634

Loucif, S., Al-Rajab, M., Abu Zitar, R., & Rezk, M. (2024). Toward a globally lunar calendar: A machine learning-driven approach for crescent moon visibility prediction. *Journal of Big Data*, *11*(1), 114. https://doi.org/10.1186/s40537-024-00979-6

Meeus, J. (1991). *Astronomical algorithms*. Willmann-Bell.

Mehare, H. B., Anilkumar, J. P., & Usmani, N. A. (2023). The Python programming language. In M. "Sufian" Badar (Ed.), *A guide to applied machine learning for biologists* (pp. 27–60). Springer International Publishing. https://doi.org/10.1007/978-3-031-22206-1_2

Mohd Nawawi, M. S. A., Faid, M. S., Saadon, M. H. M., Wahab, R. A., & Ahmad, N. (2024). Hijri month determination in Southeast Asia: An illustration between religion, science, and cultural background. *Heliyon*, *10*(20). Scopus. https://doi.org/10.1016/j.heliyon.2024.e38668

Mustapha, A. S., Zainol, N. Z. N., Mohammad, C. A., Khalim, M. A., Muhammed, N. K. W., & Faid, M. S. (2024). Al-Fatani's perspectives on Islamic family law: Insights from Hidayah Al-Muta'allim Wa'Umdah Al-Muta'alim. *Journal of Islamic Thought and Civilization*, *14*(1), 247–265. https://doi.org/10.32350/jitc.141.15

Muztaba, R., Malasan, H. L., & Djamal, M. (2023). Deep learning for crescent detection and recognition: Implementation of Mask R-CNN to the observational Lunar dataset collected with the Robotic Lunar Telescope System. *Astronomy and Computing*, *45*, 100757. https://doi.org/10.1016/j.ascom.2023.100757

Niri, M. A., Jamaludin, M. H., Nawawi, M. S. A. M., Zaki, N. A., & Wahab, R. A. (2023). Astronomy development since antiquity to Islamic civilization from the perspective of Islamic historiography. *Journal of Al-Tamaddun*, *18*(1), 169–177. Scopus. https://doi.org/10.22452/JAT.vol18no1.14

Ozgur, C., Colliau, T., Rogers, G., & Hughes, Z. (2021). MatLab vs. Python vs. R. *Journal of Data Science*, *15*(3), 355–372. https://doi.org/10.6339/JDS.201707_15(3).0001

Rabbat, N. (1996). Al-Azhar mosque: An architectural chronicle of Cairo's history. *Muqarnas*, *13*, 45. https://doi.org/10.2307/1523251

Rasyid, M., Aseri, M., Izzuddin, A., Faid, M. S., Aseri, A. F., & Sukarni, S. (2024). Study of the Falakiyah Fatwas of the Indonesian Ulema council. *Journal of Law and Governance*, *7*(1), 45–62.

Rasyid, M., Aseri, M., Sukarni, Akh. Fauzi Aseri, & Faid, M. S. (2023). Science and its role in changes in Islamic legal thought (An analysis of changes in the Fatwa of the Indonesian Ulema council due to recent scientific findings). *Syariah: Jurnal Hukum Dan Pemikiran*, *23*(2), 120–137.

Razzak, A. (2024). A geophysical analysis through a Python program to predict sunrise, sunset, and prayer timings. *Jurnal Fizik Malaysia*, *45*(1), 10019–10027.

Rhodes, B. C. (2011). *PyEphem: Astronomical ephemeris for Python. Astrophysics Source Code Library, record* Ascl:1112.014.

Rubin, U. (2017). Morning and evening prayers in early Islam. In G. Hawting (Ed.), *The development of Islamic ritual* (1st ed., pp. 105–129). Routledge. https://doi.org/10.4324/9781315240312-9

Schumm, W. R. (2020). How accurately could early (622-900 C.E.) Muslims determine the direction of prayers (Qibla)? *Religions*, *11*(3), 102. https://doi.org/10.3390/rel11030102

Shariff, N. N. M., Faid, M. S., & Hamidi, Z. S. (2016). Islamic new moon software: Current status. In K. J. Kim & N. Joukov (Eds.), *Information science and applications (ICISA) 2016* (Vol. 376, pp. 1069–1079). Springer Singapore. https://doi.org/10.1007/978-981-10-0557-2_102

Shariff, N. N. M., Hamidi, Z. S., & Faid, M. S. (2017). The impact of light pollution on Islamic new moon (hilal) observation. *International Journal of Sustainable Lighting*, *19*(1), 10–14. https://doi.org/10.26607/ijsl.v19i1.61

Syazwan Faid, M., Mohd Nawawi, M. S. A., Saadon, M. H. M., & Ahmad, N. (2025). A review on modern lunar crescent visibility criterion. *Malaysian Journal of Science*, *44*(1), 101–120. https://doi.org/10.22452/mjs.vol44no1.12

Wahidi, A., Yasin, N., & Kadarisman, A. (2019). The beginning of Islamic months determination in Indonesia and Malaysia: Procedure and social condition. *ULUL ALBAB Jurnal Studi Islam*, *20*(2), 322–345. https://doi.org/10.18860/ua.v20i2.5913

Wahyuni, S., Samsuddin, S., & Hamzah, E. (2022). Qibla direction accuracy analysis based on astronomy (Google Earth), perspective of Islamic law. *Journal of Islam and Science*, *9*(1), 39–45. https://doi.org/10.24252/jis.v9i1.30111

Yildirim, F., Kadi, F., & Sahin, S. L. (2024). Developing a new interface for Qibla direction application based on MATLAB GUI. *Survey Review*, 1–15. https://doi.org/10.1080/00396265.2024.2411186

Yusuf, H. (2010). *Caesarean moon births: Calculations, moon sighting and the prophetic way*. Zaytuna Institute.

Index

R

S

U

V

Z