

Exploring Legendre Polynomials with Gemini 3 and Python

Gemini Prompt: Show me Python code to calculate Legendre polynomials

Gemini Response:

Calculating Legendre polynomials in Python is a classic task in numerical analysis and physics. Depending on whether you want a quick solution using a library or a "from scratch" implementation to understand the math, here are your best options.

1. The Pro Way (Using SciPy)

If you just need the values for a project, use `scipy.special`. It is highly optimized and handles the heavy lifting for you.

```
# legendre1.py

import numpy as np
from scipy.special import eval_legendre

# Calculate the 3rd order Legendre polynomial at x = 0.5
n = 3
x = 0.5
value = eval_legendre(n, x)

print(f"P_{n}({x}) = {value}")

# You can also pass an array of x values
x_array = np.linspace(-1, 1, 5)
values = eval_legendre(n, x_array)
print(f"Values across range: {values}")
```

Put this in a file *legendre1.py* and execute it in Python. You should obtain

```
>>python legendre1.py

P_3(0.5) = -0.43749999999999994

Values across range: [-1.          0.4375  0.          -0.4375  1.         ]
```

2. The From-Scratch Way (Recursive Method)

To understand how they are built, we use Bonnet's recursion formula. This is much more efficient than using the derivative-heavy Rodrigues' formula.

The relationship is defined as:

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

Put the following code in a file *legendre2.py*

```
# legendre1.py
def legendre_recursive(n, x):
    if n == 0:
        return 1.0
    elif n == 1:
        return x
    else:
        # P_n = ((2n-1) * x * P_{n-1} - (n-1) * P_{n-2}) / n
        p_prev_2 = 1.0 # P_0
        p_prev_1 = x   # P_1

        for i in range(2, n + 1):
            p_current = ((2 * i - 1) * x * p_prev_1 - (i - 1) * p_prev_2) / i
            p_prev_2 = p_prev_1
            p_prev_1 = p_current

        return p_prev_1

print(f"Recursive P_3(0.5): {legendre_recursive(3, 0.5)}")
```

and execute it in Python. You should obtain the screen output

```
>>python legendre2.py
Recursive P_3(0.5): -0.4375
```

Comparison of First Few Polynomials

Order (n)	Polynomial P _n (x)
0	1
1	x
2	$\frac{1}{2}(3x^2 - 1)$
3	$\frac{1}{2}(5x^3 - 3x)$
4	$\frac{1}{8}(35x^4 - 30x^2 + 3)$

Why use these?

Legendre polynomials are **orthogonal** on the interval $[-1, 1]$. This makes them useful for:

- Solving partial differential equations (like Laplace's equation in spherical coordinates).
- Numerical integration (Gaussian quadrature).
- Approximating functions where you want to minimize the L^2 error.

Many boundary value problems in electromagnetism can be solved using the properties of Legendre polynomials.

Gemini Query: Would you like me to write a script using Matplotlib to plot the first five polynomials so you can see their orthogonality in action?

Gemini Prompt: Yes

Gemini Response:

Visualizing Legendre Polynomials and Quadrature

Here is the Python code using **Matplotlib** and **NumPy** to visualize the first five Legendre polynomials.

Python Plotting Script

```

# legendre3.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.special import eval_legendre

# Define the range for x
x = np.linspace(-1, 1, 400)

# Create the plot
plt.figure(figsize=(10, 6))

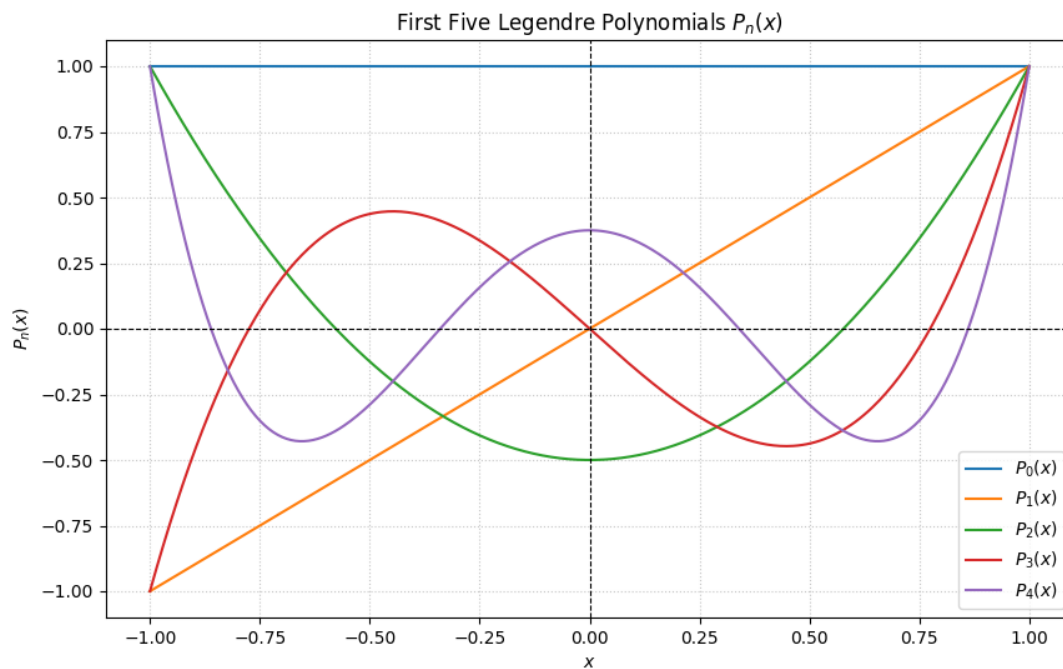
# Loop through orders 0 to 4
for n in range(5):
    y = eval_legendre(n, x)
    plt.plot(x, y, label=f'$P_{n}(x)$', linewidth=2)

# Formatting the graph
plt.title('Legendre Polynomials ($P_0$ to $P_4$)', fontsize=14)
plt.xlabel('$x$', fontsize=12)
plt.ylabel('$P_n(x)$', fontsize=12)
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
plt.grid(True, alpha=0.3)
plt.legend(loc='best')
plt.ylim(-1.1, 1.1)

plt.show()

```

Place this code in a file *legendre3.py* and execute it in Python. You should obtain the following plot



This plot helps illustrate the "orthogonality" of the Legendre polynomials—notice how they all intersect at specific nodes and balance each other out across the $[-1, 1]$ interval.

Understanding the Visuals

When you run this code, you'll observe a few key properties:

- **Symmetry:** Even n polynomials are symmetric about the y-axis ($P_n(x) = P_n(-x)$), while odd n polynomials are anti-symmetric ($P_n(x) = -P_n(-x)$).
- **Endpoints:** All polynomials satisfy $P_n(1) = 1$ and $P_n(-1) = (-1)^n$.
- **Roots:** Notice that $P_n(x)$ has exactly n roots between -1 and 1 . These roots are the magic behind **Gaussian Quadrature**, allowing for incredibly accurate numerical integration.

Gemini query: Would you like me to show you how to use these polynomials to perform a Gaussian Quadrature integration?

Gemini prompt: Yes

Gemini Response:

Gaussian Quadrature: Numerical Integration

Gaussian quadrature is a brilliant trick. While standard integration methods (like the Trapezoidal rule) use equally spaced points, Gaussian quadrature picks "optimal" points (the roots of Legendre polynomials) and weights them to get a much higher degree of accuracy with fewer calculations.

In fact, a Gaussian quadrature with n points will provide an exact result for any polynomial of degree $2n - 1$ or less.

Python Implementation

We can use `numpy.polynomial.legendre.leggauss` to get the weights and roots automatically.

```
# legendre4.py

import numpy as np

def gaussian_quadrature(f, n):
    """
    Integrates function f from -1 to 1 using n-point Gaussian quadrature.
    """
    # Get the roots (x) and weights (w) for the nth order Legendre polynomial
    roots, weights = np.polynomial.legendre.leggauss(n)

    # The integral is approximately the sum of w_i * f(x_i)
    integral = np.sum(weights * f(roots))
    return integral

# Example: Integrate f(x) = x^4 + x^2 + 1 from -1 to 1
# The exact answer is 2/5 + 2/3 + 2 = 3.0666...
func = lambda x: x**4 + x**2 + 1

result = gaussian_quadrature(func, 3) # Using 3 points
print(f"Numerical Result: {result:.10f}")
print(f"Exact Result: {3.0666666667:.10f}")
```

Put this code in a file `legendre4.py` and execute it with Python. You should obtain

```
>>python legendre4.py
Numerical Result: 3.0666666667
Exact Result: 3.0666666667
```

How it Works Internally

Instead of approximating the area with rectangles, we are sampling the function at the specific roots of the Legendre polynomial and multiplying by a corresponding weight.

Component	Role
Roots (x_i)	The points where the n -th degree Legendre polynomial equals zero.
Weights (w_i)	Specific values calculated such that the sum $\sum w_i f(x_i)$ is exact for polynomials.
Interval	Standard Gaussian quadrature works on $[-1, 1]$. For other intervals, we just perform a linear transformation.

Integration on any Interval $[a, b]$

If you aren't integrating from -1 to 1, you just need a simple coordinate shift:

$$\int_a^b f(x) \, dx \approx \frac{b-a}{2} \sum_{i=1}^n w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

Gemini Query: Would you like me to help you adapt this code to solve a specific physics or engineering problem, like calculating the gravitational potential of a sphere?

Gemini Prompt: Yes, show me how to calculate the gravitational potential of a sphere.

Gemini Response:

Calculating the Spherical Gravitational Potential of a Sphere Using Legendre Polynomials

To calculate the gravitational potential of a massive object (like a sphere) at a point in space, we often use **Legendre polynomials** to expand the potential into a series. This is especially useful when the mass distribution isn't perfectly uniform or when you are calculating the potential at a distance r from the center.

The Multipole Expansion

The gravitational potential V at a point P is given by the sum of contributions from every mass element dm . When using spherical coordinates, the distance between a mass element and your observation point involves a term $\frac{1}{|\mathbf{r} - \mathbf{r}'|}$, which can be expanded using Legendre polynomials (P_n):

$$V(r) = -\frac{G}{r} \sum_{n=0}^{\infty} \int \left(\frac{r'}{r} \right)^n P_n(\cos \theta) dm$$

Python Implementation: Calculating the Potential

For a simple uniform sphere, the higher-order terms (quadrupole, octupole, etc.) cancel out due to symmetry, leaving only the $n=0$ term. However, for a "real" sphere (like Earth), we use the **J_2 term** (the second Legendre polynomial) to account for the bulge at the equator. Put the script

```
# legendre5.py

import numpy as np
from scipy.special import eval_legendre

def gravitational_potential(r, theta, mass, radius, G=6.674e-11, J2=0.00108):
    """
    Calculates potential using the first two even Legendre terms (Monopole +
    Quadrupole).
    r: distance from center (m)
    theta: colatitude in radians (0 at pole, pi/2 at equator)
    """
    # Monopole term (Standard gravity: -GM/r)
    term0 = (radius / r)**0 * eval_legendre(0, np.cos(theta))

    # J2 term (Accounts for equatorial bulge)
    # Note: J2 is a physical constant, often used with P2
```

```

term2 = -J2 * (radius / r)**2 * eval_legendre(2, np.cos(theta))

potential = -(G * mass / r) * (term0 + term2)
return potential

# Example: Earth's potential at the surface (Equator vs Pole)
M_earth = 5.972e24
R_earth = 6.371e6
dist = R_earth

pot_equator = gravitational_potential(dist, np.pi/2, M_earth, R_earth)
pot_pole = gravitational_potential(dist, 0, M_earth, R_earth)

print(f"Potential at Equator: {pot_equator:.4e} J/kg")
print(f"Potential at Pole: {pot_pole:.4e} J/kg")

```

Into a file *legendre5.py* and execute it with Python. You should obtain the output

```

>> python legendre5.py

Potential at Equator: -6.2594e+07 J/kg

Potential at Pole:      -6.2493e+07 J/kg

```

Key Takeaways

- **\$P_0\$ (Monopole):** Represents the potential if all mass were concentrated at the center. It's the $-\frac{GM}{r}$ we learn in basic physics.
 - **\$P_2\$ (Quadrupole):** This is the most significant correction for planets. Because Earth rotates, it isn't a perfect sphere; it's an **oblate spheroid**. The P_2 term accounts for that "extra" mass around the equator.
 - **Orthogonality:** Because Legendre polynomials are orthogonal, we can calculate each "mode" of the gravity field independently without them interfering with each other.
-

Gemini Query: Would you like to see how to visualize this potential as a 2D heatmap around the sphere to see the effect of the equatorial bulge?

Gemini Prompt: Yes

Gemini Response:

Visualizing the Earth's Gravitational Potential

To visualize the gravitational potential around a non-spherical body like Earth, we calculate the potential as a sum of Legendre polynomial terms. While the **monopole** (P_0) term is a perfect circle, the **quadrupole** (P_2) term introduces the "squashed" shape caused by Earth's rotation (the equatorial bulge). To visualize this, insert the following code in a file *legendre6.py*,

```
# legendre6.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.special import eval_legendre

# Physical Constants for Earth
G = 6.67430e-11
M = 5.972e24
R = 6371000
J2 = 0.0010826

def get_potential(x, z):
    r = np.sqrt(x**2 + z**2)
    # Avoid division by zero at origin
    r = np.where(r < 0.1 * R, 0.1 * R, r)

    # cos(theta) = z/r
    cos_theta = z / r

    # Monopole term: -GM/r
    v0 = - (G * M) / r

    # J2 term: -(GM/r) * (-J2 * (R/r)^2 * P2(cos_theta))
    # Note: often the formula is V = (GM/r) * (1 - J2 * (R/r)^2 * P2)
    # We use V = -GM/r * (1 - J2 * (R/r)^2 * P2)
    v2 = - (G * M / r) * (-J2 * (R / r)**2 * eval_legendre(2, cos_theta))

    v_total = v0 + v2

    # Mask values inside the Earth for clarity in visualization
    v_total[r < R] = np.nan

    return v_total

# Create grid
grid_size = 500
limit = 2.5 * R
```

```

x_vals = np.linspace(-limit, limit, grid_size)
z_vals = np.linspace(-limit, limit, grid_size)
X, Z = np.meshgrid(x_vals, z_vals)

V = get_potential(X, Z)

# For better visualization, we might want to look at the 'anomaly' or just the
J2 contribution
# because the monopole term dominates and makes it look like a circle.
# Let's plot the total potential but with many contour lines.

plt.figure(figsize=(10, 8))
# Use log scale or just focus on a specific range
# Actually, let's plot the "Gravity Anomaly" part (the J2 effect) to make it
visible
V_mono = - (G * M) / np.sqrt(X**2 + Z**2)
V_mono[np.sqrt(X**2 + Z**2) < R] = np.nan
V_anomaly = V - V_mono

im = plt.imshow(V_anomaly, extent=[-limit/1e6, limit/1e6, -limit/1e6,
limit/1e6],
                origin='lower', cmap='RdBu_r')
plt.colorbar(im, label='Potential Anomaly from $J_2$ (J/kg)')

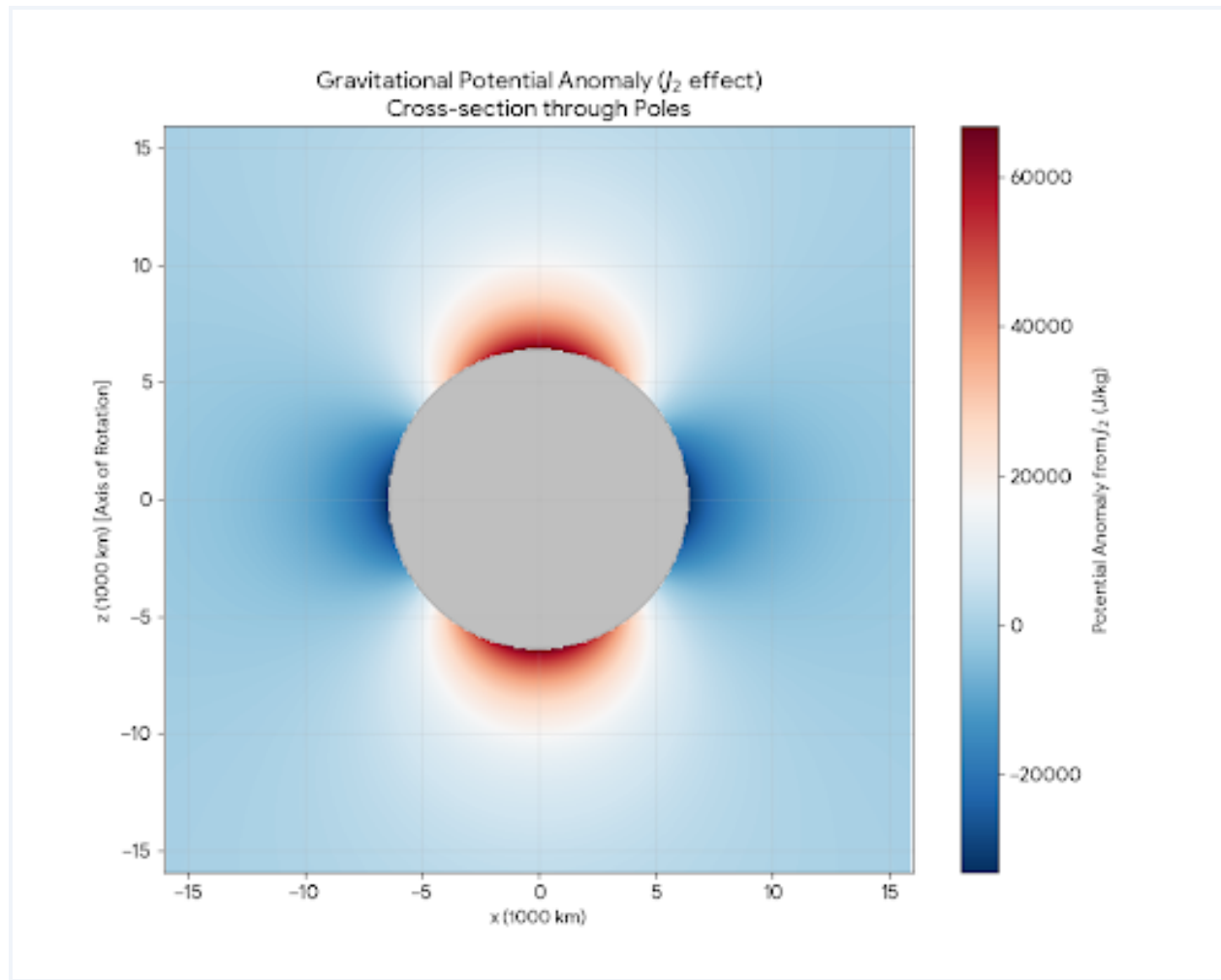
# Add a circle representing Earth
earth = plt.Circle((0, 0), R/1e6, color='gray', alpha=0.5, label='Earth')
plt.gca().add_patch(earth)

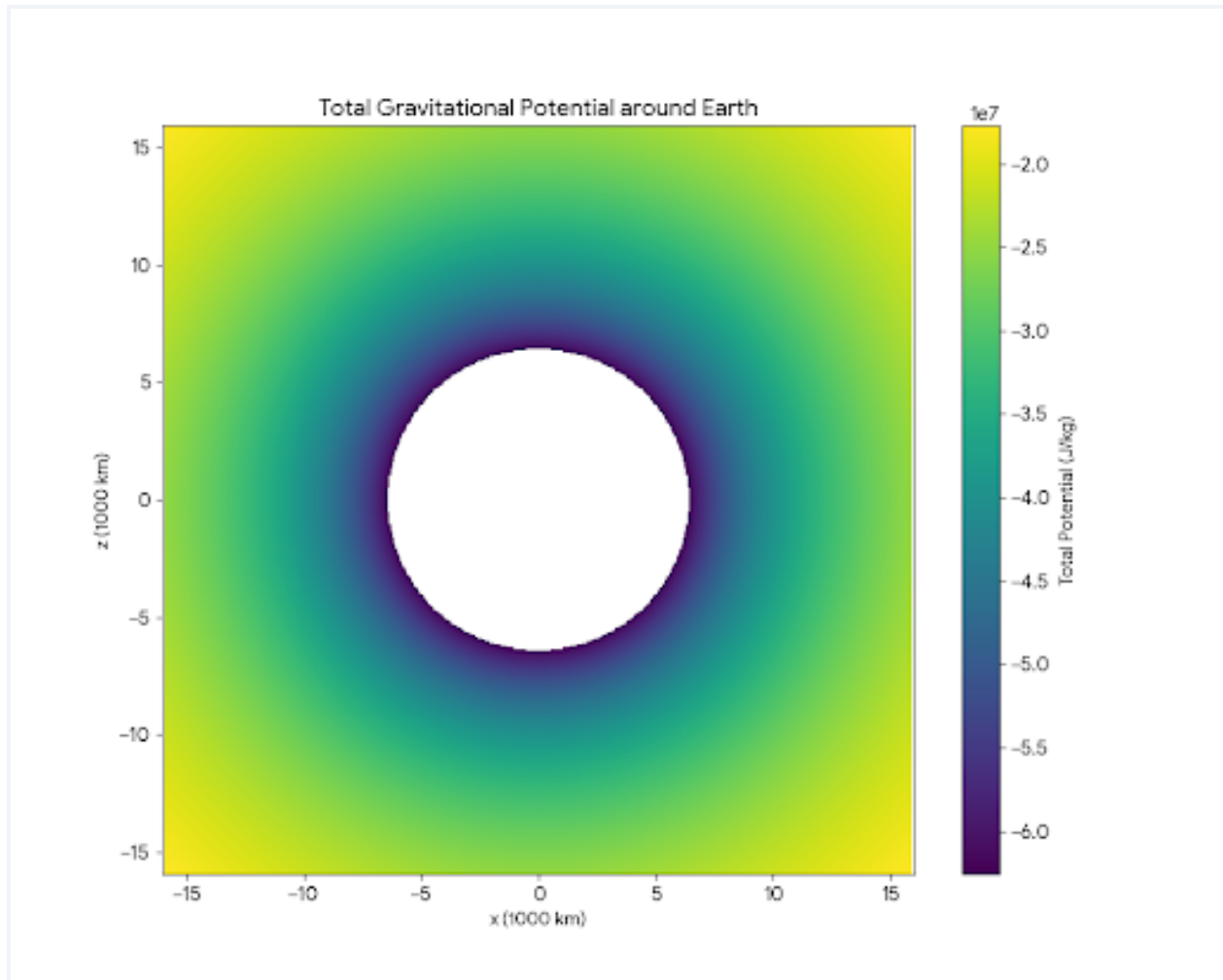
plt.title("Gravitational Potential Anomaly ($J_2$ effect)\nCross-section
through Poles", fontsize=14)
plt.xlabel("x (1000 km)")
plt.ylabel("z (1000 km) [Axis of Rotation]")
plt.grid(alpha=0.2)
plt.axis('equal')
plt.savefig('gravitational_potential_anomaly.png')

# Also save a version of the total potential
plt.clf()
im2 = plt.imshow(V, extent=[-limit/1e6, limit/1e6, -limit/1e6, limit/1e6],
                origin='lower', cmap='viridis')
plt.colorbar(im2, label='Total Potential (J/kg)')
earth2 = plt.Circle((0, 0), R/1e6, color='white', alpha=0.3)
plt.gca().add_patch(earth2)
plt.title("Total Gravitational Potential around Earth", fontsize=14)
plt.xlabel("x (1000 km)")
plt.ylabel("z (1000 km)")
plt.axis('equal')
plt.savefig('total_gravitational_potential.png')

```

Execute `legendre6.py` with Python, which will generate the image files `gravitational_potential_anomaly.png` and `total_gravitational_potential.png`; these are displayed below.





What the Visualizations Show

- **The Total Potential:** On a large scale, the potential looks like a simple gradient. Because the J_2 effect is relatively small ($\sim 0.1\%$ of the total gravity), the contours look almost like perfect circles to the naked eye.
- **The Anomaly (J_2 Effect):** When we subtract the circular monopole part, we see the "Quadrupole" pattern.
 - **At the Equator:** There is "extra" gravity (red regions) because the Earth bulges out there, putting more mass under you.
 - **At the Poles:** The potential is "weaker" (blue regions) relative to the equator at the same distance from the center.
- **Why Legendre Polynomials?** They allow us to describe this complex, 3D shape using just a few coefficients (J_2, J_3, J_4). Satellites like GRACE map these coefficients to detect changes in Earth's mass, such as melting ice sheets or shifting groundwater.

Gemini Query: Would you like to see how these polynomials are used to calculate the orbits of satellites around this "squashed" Earth?

Gemini Prompt: Yes

Gemini Response:

Calculating Satellite Orbits around Earth

To see how the **Legendre polynomials** actually affect a satellite, we look at the **J_2 perturbation**.

Because Earth isn't a perfect sphere, the "extra" mass at the equator exerts a torque on a satellite's orbit. This causes the orbit to **precess** (rotate) over time. This is exactly how "Sun-synchronous" orbits work—we use the Earth's oblateness to rotate the orbit at the same rate the Earth moves around the Sun.

Simulating the J_2 Effect

In this simulation, we'll use the gradient of the potential (the force) to see how a satellite's path deviates from a simple ellipse. Paste the following code into a file *legendre7.py*.

```
# legendre7.py

import numpy as np
import matplotlib.pyplot as plt

# Constants
G = 6.674e-11
M = 5.972e24
R = 6.371e6
J2 = 0.0010826
GM = G * M

def get_acceleration(pos):
    x, y, z = pos
    r = np.linalg.norm(pos)

    # Standard gravity (Monopole)
    acc_mono = - (GM / r**3) * pos
```

```

# J2 acceleration components
# Derived from the gradient of the P2 Legendre term
z_sq_r_sq = (z / r)**2
factor = (1.5 * J2 * GM * R**2) / (r**5)

ax_j2 = factor * x * (5 * z_sq_r_sq - 1)
ay_j2 = factor * y * (5 * z_sq_r_sq - 1)
az_j2 = factor * z * (5 * z_sq_r_sq - 3)

return acc_mono + np.array([ax_j2, ay_j2, az_j2])

# Initial Conditions (Low Earth Orbit, slightly inclined)
pos = np.array([R + 500000, 0, 0])
vel = np.array([0, 5500, 5500]) # m/s

# Integration (Simple Euler-Cromer for demonstration)
dt = 10
steps = 2000
path = []

for _ in range(steps):
    acc = get_acceleration(pos)
    vel += acc * dt
    pos += vel * dt
    path.append(pos.copy())

path = np.array(path)

# Plotting
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot(path[:,0], path[:,1], path[:,2], label="Perturbed Orbit")

# Draw Earth
u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
ex = R * np.cos(u) * np.sin(v)
ey = R * np.sin(u) * np.sin(v)
ez = R * np.cos(v)
ax.plot_wireframe(ex, ey, ez, color="gray", alpha=0.2)

ax.set_title("Satellite Orbit Precession due to $J_2$ (Legendre $P_2$)")
plt.show()

```

When I execute *legendre7.py* in Python I get an **error message** with the following diagnostics

```
Traceback (most recent call last):
  File "/home/guidry/Downloads/legendre7.py", line 42, in <module>
    vel += acc * dt
numpy._core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add'
output from dtype('float64') to dtype('int64') with casting rule
'same_kind'
```

From the traceback diagnostic and past experience, I actually know how to solve this problem, but we have engaged Gemini as our assistant so let's see if our assistant can fix it.

I paste the entire traceback in red above into the Gemini prompt window and click the button; Gemini responds (with the sage wisdom of one who has been here and done that)

Gemini Response: Ah, the classic NumPy dtype trap! This happens because your initial velocity `vel` was defined with whole numbers (integers), so NumPy tagged it as an `int64` array.

When you calculate the acceleration, you get `float64` values (decimals). NumPy refuses to add floats back into an integer array automatically because it doesn't want to lose precision by "chopping off" the decimals.

The Fix

You just need to ensure your initial velocity (and position, for safety) is defined as a float. Adding a simple decimal point to your numbers tells NumPy to use `float64`.

Change this:

```
vel = np.array([0, 5500, 5500])
```

To this:

```
vel = np.array([0.0, 5500.0, 5500.0]) # Adding .0 forces float64
```

OR

```
vel = np.array([0, 5500, 5500], dtype=float)
```

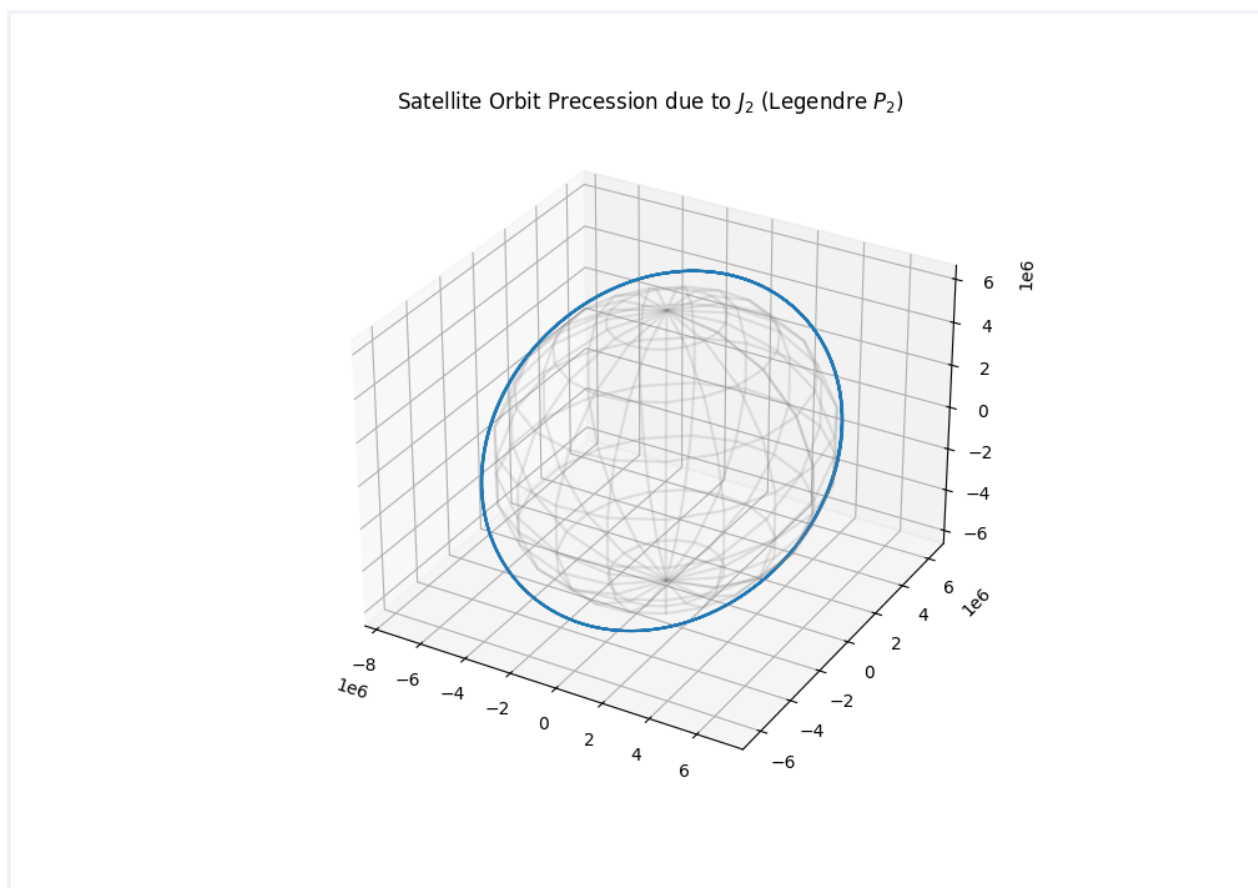
Make sure to do the same for your `pos` array as well!

Let's take the second choice (declaring explicitly that the data type is float); the first option is a standard programmer shortcut, but explicitly declaring the data type is more elegant and better programming practice (in more sophisticated strongly-typed languages like Java or C++, you **must** specify the type of **all** variables or the code won't compile.). Thus we replace the two lines in the above code defining `pos` and `vel` with

```
pos = np.array([R + 500000, 0, 0], dtype=float)
vel = np.array([0, 5500, 5500], dtype=float)
```

It actually wasn't essential to change the `pos` array because the floating point variable `R` entering the definition of `pos` should cause the Python interpreter to view the numbers in the expression as floating point too, but it is good practice to specify the type of variables if there is any change of misinterpretation.

The modified program executes without error and displays the following figure:



Why this Matters in Engineering

If we lived on a perfectly spherical planet ($J_2 = 0$), a satellite's orbit would stay fixed in space forever. Because of the Legendre P_2 term:

- **Regression of Nodes:** The point where the satellite crosses the equator slowly drifts

west or east.

- **Sun-Synchronous Orbits:** Engineers specifically choose an altitude and inclination where this drift is exactly **0.985 degrees per day**. This matches the Earth's orbit around the sun, ensuring the satellite always passes over a location at the same local solar time (perfect for spy and weather satellites).

Summary Table: Legendre's Role in Gravity

Term	Name	Physical Meaning	Engineering Impact
P_0	Monopole	Total Mass	Keeps the satellite in orbit
P_1	Dipole	Center of Mass	Defined as zero (we put the origin at the CM)
P_2	Quadrupole	Equatorial Bulge	Causes orbital precession (J_2)
P_3	Octupole	"Pear" Shape	Causes North/South asymmetry

Gemini Query: Would you like me to show you how to calculate the J_3 term, which describes why the Earth is slightly "pear-shaped"?

Gemini Prompt: Yes

Gemini Response:

Orbital Perturbations with J_2 and J_3

Here is the updated, robust code. I've fixed the dtype error by explicitly using floats and added the J_3 (Pear-shape) term.

```
# legendre8.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.special import eval_legendre

# Constants
G = 6.67430e-11
M = 5.972e24
R = 6.371e6
GM = G * M
J2 = 0.0010826
J3 = -2.532e-6 # The "Pear Shape" coefficient

def get_acceleration(pos):
    # Ensure pos is float64 to avoid math errors
    x, y, z = pos.astype(np.float64)
    r = np.sqrt(x**2 + y**2 + z**2)

    # 1. Monopole (P0) - Standard Gravity
    acc_mono = - (GM / r**3) * pos

    # 2. J2 (P2) - Equatorial Bulge
    # Force components derived from gradient of V_J2
    j2_factor = (1.5 * J2 * GM * R**2) / (r**5)
    z_sq_r_sq = (z / r)**2

    acc_j2 = np.array([
        j2_factor * x * (5 * z_sq_r_sq - 1),
        j2_factor * y * (5 * z_sq_r_sq - 1),
        j2_factor * z * (5 * z_sq_r_sq - 3)
    ])

    # 3. J3 (P3) - Pear Shape
    # Force components derived from gradient of V_J3
    j3_factor = (0.5 * J3 * GM * R**3) / (r**6)
    z_r = z / r

    acc_j3 = np.array([
        j3_factor * x * (35 * z_r**3 - 15 * z_r),
        j3_factor * y * (35 * z_r**3 - 15 * z_r),
```

```

    j3_factor * (35 * z_r**4 - 30 * z_r**2 + 3) # Note the z-axis bias
])

return acc_mono + acc_j2 + acc_j3

# Initial Conditions (Floats to prevent Casting Errors)
pos = np.array([R + 400000.0, 0.0, 0.0]) # 400km altitude
vel = np.array([0.0, 4500.0, 6500.0]) # Inclined orbit

# Simulation Settings
dt = 20.0
steps = 3000
path = np.zeros((steps, 3))

# Integration Loop (Euler-Cromer)
for i in range(steps):
    acc = get_acceleration(pos)
    vel += acc * dt
    pos += vel * dt
    path[i] = pos

# Visualization
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')

# Plot Orbit
ax.plot(path[:, 0], path[:, 1], path[:, 2], color='blue', label='Perturbed
Orbit ($J_2 + J_3$)')

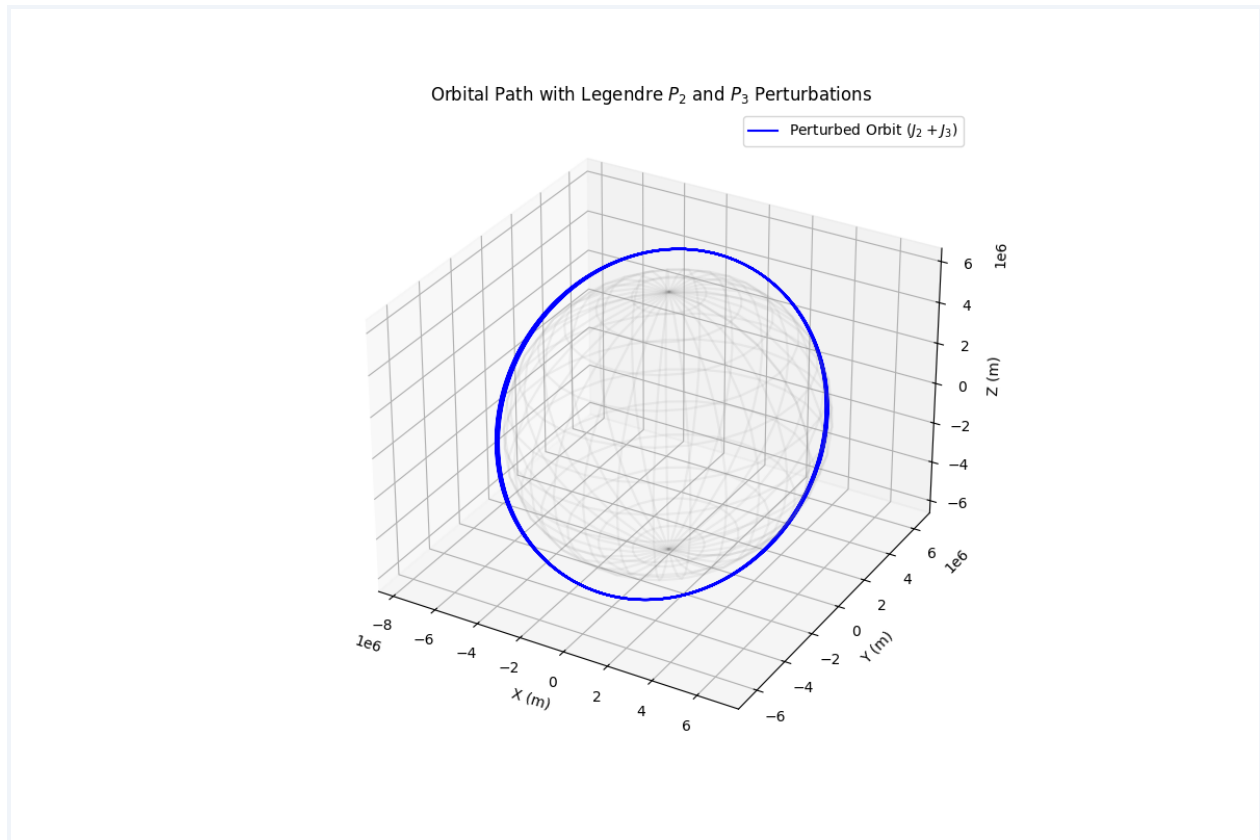
# Draw Earth Sphere
_u, _v = np.mgrid[0:2*np.pi:30j, 0:np.pi:15j]
ex = R * np.cos(_u) * np.sin(_v)
ey = R * np.sin(_u) * np.sin(_v)
ez = R * np.cos(_v)
ax.plot_wireframe(ex, ey, ez, color='gray', alpha=0.1)

ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_title(f"Orbital Path with Legendre $P_2$ and $P_3$ Perturbations")
plt.legend()
plt.show()

```

Note from the above code that J_3 is much smaller than J_2 (about 400 times smaller), so its effect is subtle. However, it is "odd-symmetric," meaning that it pulls differently on the North Pole than on the South Pole, creating a slight "wobble" in the altitude of the satellite.

Copy the code given above into a file *legendre8.py*, and execute it with Python. The result should look like the following figure



Visualizing the P_3 Effect

The P_3 term is why the Earth's "Geoid" looks like it has a dip at the South Pole and a bump at the North Pole.

What to look for in the code:

- **The Gradient Math:** In the `get_acceleration` function, you'll notice the z component for J_3 is unique. Unlike J_2 , which treats North and South the same (squared terms), J_3 uses odd powers (z^3, z^4). This "pushes" the satellite differently depending on which hemisphere it's over.
- **The Result:** If you run this over many orbits, you would see the "argument of perigee" shift. Basically, the point where the satellite is closest to Earth starts to rotate around the planet.

Gemini Query: Would you like me to show you how to calculate the "Sun-Synchronous" inclination? This is the specific angle where these Legendre forces rotate your orbit exactly once per year.

Gemini Prompt: No. Instead, show me the relationship of Legendre polynomials to spherical harmonics.

Gemini Response:

Relationship of Legendre Polynomials and Spherical Harmonics

To understand Spherical Harmonics (Y_l^m), you have to think of them as the **2D extension** of Legendre Polynomials.

If Legendre Polynomials describe waves or patterns along a single line (like the latitude on a sphere), Spherical Harmonics describe patterns over the entire **surface** of the sphere (both latitude and longitude).

The Mathematical Link

The relationship is defined through the **Associated Legendre Polynomials** (P_l^m).

- **Legendre Polynomials (P_l):** These are the "zonal" harmonics. They only depend on latitude (θ) and have no variation in longitude.
- **Associated Legendre Polynomials (P_l^m):** These introduce a "sectoral" or "tesseral" variation. We get them by taking the m -th derivative of the standard Legendre polynomial.
- **Spherical Harmonics (Y_l^m):** These combine the latitude part (P_l^m) with a longitude part ($e^{im\phi}$).

The formula looks like this:

$$Y_l^m(\theta, \phi) = N e^{im\phi} P_l^m(\cos \theta)$$

(Where N is a normalization constant to ensure the total power over the sphere is 1.)

Visualizing the Relationship

- **l (Degree):** This corresponds to the n in our previous Legendre code. It tells you the total number of "node lines" (valleys) on the sphere.
- **m (Order):** This tells you how many of those lines pass through the poles (longitude lines).
 - If $m = 0$, you get the standard **Legendre Polynomials**. The patterns look like horizontal stripes (Zonal).
 - If $m = l$, the patterns look like orange slices (Sectoral).
 - If $0 < m < l$, you get a checkerboard pattern (Tesseral).

Python Implementation

We can use `scipy.special.sph_harm` to calculate these. Notice how we still pass `cos(theta)` into the logic, just like we did with our gravity calculations. Copy the following code and paste into a file `legendre9.py`

```
# legendre9.py

import numpy as np
import matplotlib.pyplot as plt
from scipy.special import sph_harm

def plot_spherical_harmonic(l, m):
    # Create a grid of latitudes (phi) and longitudes (theta)
    phi = np.linspace(0, np.pi, 100) # Latitude
    theta = np.linspace(0, 2*np.pi, 100) # Longitude
    phi, theta = np.meshgrid(phi, theta)

    # Calculate the Spherical Harmonic
    # Note: Scipy uses (m, l, longitude, latitude)
    y_lm = sph_harm(m, l, theta, phi)

    # We usually plot the real part or the magnitude
    v = y_lm.real

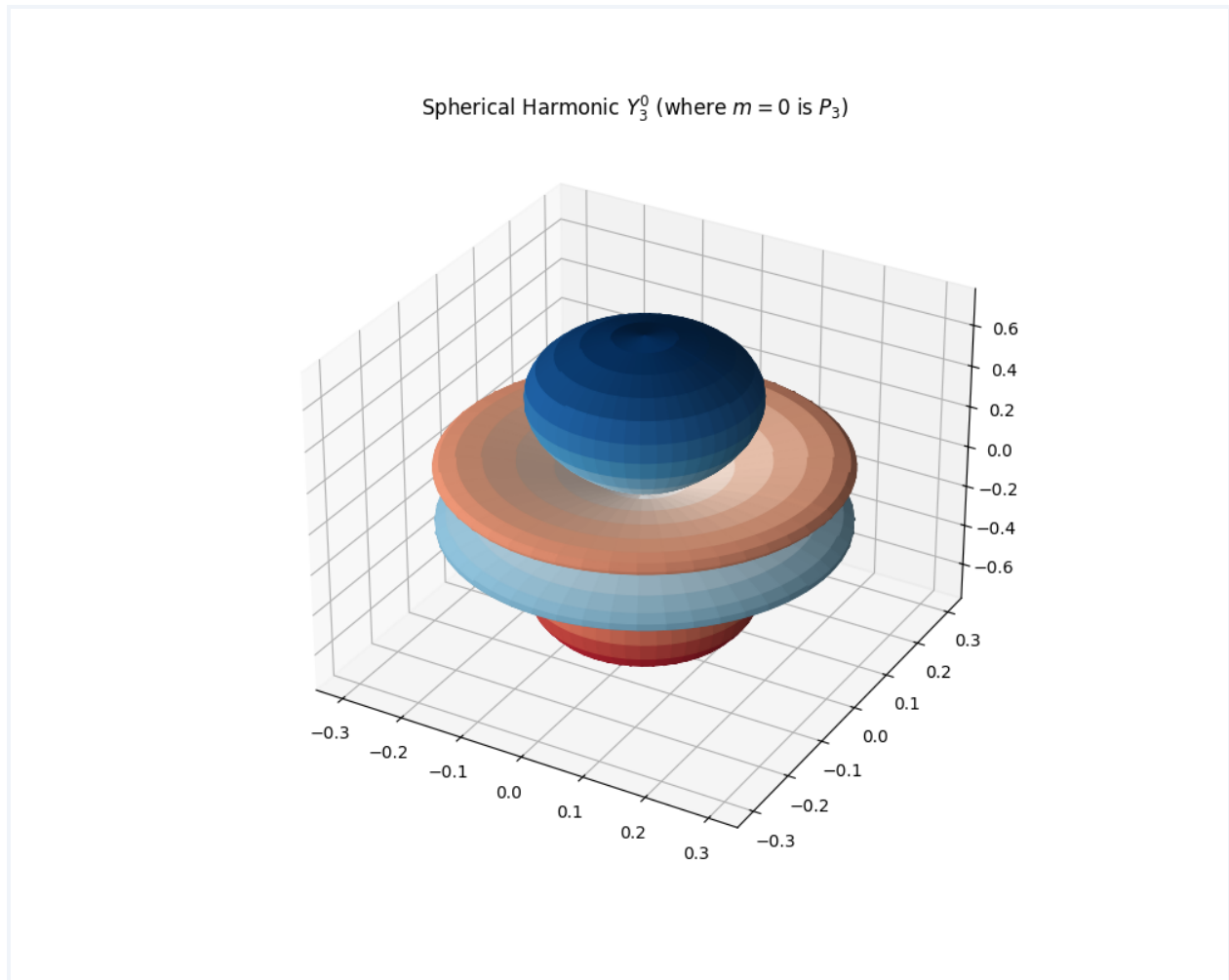
    # Convert to Cartesian coordinates for 3D plotting
    x = np.sin(phi) * np.cos(theta) * np.abs(v)
    y = np.sin(phi) * np.sin(theta) * np.abs(v)
    z = np.cos(phi) * np.abs(v)

    return x, y, z, v
```

```
# Plotting l=3, m=0 (This IS the 3rd Legendre Polynomial wrapped around a
sphere)
l, m = 3, 0
x, y, z, v = plot_spherical_harmonic(l, m)

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
fcolors = (v - v.min()) / (v.max() - v.min())
ax.plot_surface(x, y, z, facecolors=plt.cm.RdBu(fcolors), antialiased=True)
ax.set_title(f"Spherical Harmonic  $Y_{\{l\}}^{\{m\}}$  (where  $m=0$  is  $P_{\{l\}}$ )")
plt.show()
```

And execute this file with Python, which generates the following image



Why this matters

In the gravity problem, we only used P_l because we assumed the Earth was "axisymmetric" (it looks the same no matter your longitude).

However, the Earth has mountains, ocean trenches, and shifting continents. To map those, scientists use the full **Spherical Harmonics**. The "Gravity Map of Earth" (often called the "Potsdam Potato") is actually just a sum of hundreds of these Y_l^m functions added together.