4-30-2023

A Model Python Script to Calculate and Display in 3D Space the Inertial Properties of a Rectangular Prism with Linear Mass Density Distribution

Lorenzo Nii Antie Addy

A Model Python Script to Calculate and Display in 3D Space the Inertial Properties of a Rectangular Prism with Linear Mass Density Distribution

by

Lorenzo Nii Antie Addy

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

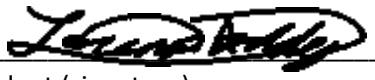
of

The University of Alabama in Huntsville

April 30, 2023

Honors Capstone Director: Dr. Massimiliano Bonamente

Professor of Physics and Astronomy

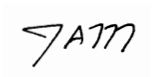


April 26, 2023

Student (signature) Date

Massimiliano Bonamente **(digitally signed April 30, 2023)**

Director (signature) Date



Digitally signed by James Miller
Date: 2023.04.30 17:43:51 -05'00'
Adobe Acrobat version: 2023.001.20143

Department Chair (signature) Date

William Wilkerson

Digitally signed by William Wilkerson
Date: 2023.05.02 12:11:13 -05'00'

Honors College Dean (signature) Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Lorenzo Addy

Student Name (printed)

Student Signature

April 26, 2023

Date

April 26, 2023

A Model Python Script to Calculate and Display in 3D Space the Inertial Properties of a Rectangular Prism with Linear Mass Density Distribution

Addy, Lorenzo

Advisor: Dr. Massimiliano Bonamente

Department of Physics and Astronomy, University of Alabama in Huntsville, Huntsville, AL, U.S.A.

ABSTRACT

The study of the rotation of rigid bodies sits at the foundation of classical mechanics, especially with the study of principal axes and moments of inertia for a variety of common objects. The goal of this project is to develop a Python program to calculate and display the inertial properties of a user-input rectangular prism with a nonuniform mass density distribution, here linear. The user is prompted to input the dimensions and density distribution of their prism. The program then calculates the total mass, center of mass, inertia tensor oriented about the center of mass, diagonalized inertia tensor, moments of inertia, and principal axes of rotation, indicating stability of rotation around a given axis. A 3D figure is generated containing the prism, its principal axes as a parametrization of the eigenvectors from the center of mass color coded according to stability of rotation, and, if the user chooses, the density distribution represented by a grid of points with variable opacity dispersed throughout the interior and/or surface of the prism. When inputs are normalized, this program is found to calculate each physical quantity accurately and effectively for test cases, including homogeneous and non-homogeneous prisms. The 3D display effectively depicts the prism with its principal axes intersecting at the center of mass and with a density display that includes a color gradient corresponding to the gradient of the density profile. Given the effectiveness of this program, it serves as a potential structural model to develop programs, with appropriate adaptations, to do the same for any object.

1. INTRODUCTION

The study of the rotation of rigid bodies sits at the foundation of classical mechanics. Various objects are studied to determine their inertial properties for rotational motion. Properties such as the principal axes of rotation and moments of inertia calculated help determine rotational energies and momenta as used in countless applications especially in engineering and technological contexts. These properties are known for common objects with uniform mass densities, including spheres, disk, cubes, etc., over axes of symmetry for these shapes, but not so commonly known for nonuniform objects. This project, adopts two goals: to develop a Python program to quickly calculate the inertial properties of a user-input rectangular prism with a nonuniform mass density profile; and to provide a visualization tool of these properties by plotting them in 3D space along with the prism itself.

In this program, the user is allowed to input an object by providing dimensions and the density distribution in the form

$$\rho(\vec{r}) = Ax + By + Cz + D$$

where $\rho(\vec{r})$ is the mass density at vector point $\vec{r} = (x, y, z)$ and $A, B, C,$ and D are coefficients input by the user. The program takes this information to calculate the rotational properties of the object via the inertia tensor whose elements are defined for continuous bodies by

$$I_{ij} = \int_V \rho(\vec{r}) \left(\delta_{ij} \sum_k x_k^2 - x_i x_j \right) dV$$

where $x_{i,j,k}$ represent spatial coordinates in the volume of the object, V, and $i, j,$ and k each span the three spatial coordinates[8]. The three moments of inertia and principal axes of rotation are

calculated as the eigenvalues and vectors of the inertia tensor. The program collects all this information and displays the object along with its principal axes of rotation centered on its center of mass. The program also has the capability to display an approximate density distribution by displaying an array of points either on the surface or within the interior of the prism. Furthermore, the program uses the moments of inertia corresponding to each principal axis to determine stability of rotation around that axis via the intermediate axis theorem, i.e., if an axis has a moment of inertia that is the intermediate of the three, rotation about that axis is unstable, and indicates this property in the display via color-coding of the axes.

2. METHODOLOGY

2.1 **Initial Inputs and Calculations**

This program is created using Python 3.8 (64-bit) in Microsoft Visual Studio 2019. As with any python script, it begins by importing necessary packages and functions: NumPy and Matplotlib's Pyplot modules as well as the toolkits Axes3D and Poly3DCollection from Matplotlib's Mplot3d module and the "nquad" and "isclose" functions from the SciPy Integrate and Math modules, respectively. The program itself begins with a quick introduction printed for the user describing its own function. In it, it is explained that the user can input the dimensions of the object as well as the density distribution as defined above, and then the program will calculate the total mass, center of mass, inertia tensor, moments of inertia, and principal axes of rotation, including an intermediate axis if applicable.

The user is then prompted to input the dimensions of their prism as float values, which are then mapped to individual variables for each dimension. For the density distribution, the user is asked to input each coefficient in the density profile individually, again as float values, where the origin occurs at a corner of the object and the object extends into the first octant of the coordinate space by the given dimensions. Only numerical inputs are allowed, so no units are taken into account. Once all the inputs have been entered, the dimensions and mass distribution are printed in the form "$(x\ dimension) \times (y\ dimension) \times (z\ dimension)$" for the dimensions for the density. Then the calculations begin.

First, the center of mass is calculated via

$$x_{i,cm} = \frac{\int_V \rho(\vec{r}) x_i dV}{\int_V \rho(\vec{r}) dV}$$

where $x_{i,cm}$ is the ith coordinate of the center of mass. The bottom integral, the integral of the density over the volume of the prism, is equivalent to the total mass and is collected as such. The integrals are evaluated in the script by defining the integrands as callable functions in preparation for integration. This and all of the following integration in this program is done using the "nquad" SciPy function, which approximates an integral using Gaussian quadrature wrapped over each spatial variable, takes the previously defined integrand function and limits on the dimensions of the object as functional arguments, and returns the evaluation of the integral as well as the associated error margin[7]. The upper integral is evaluated for each dimension, then each coordinate of the center of mass is found accordingly and collected into a NumPy array representing the center of mass point. The total mass and center of mass are then printed for the user.

In calculating the inertia tensor, the equation for a tensor element is separated into definitions for diagonal and non-diagonal elements according to

$$I_{ii} = \int_V \rho(\vec{r})\left(\Delta x_j^2 + \Delta x_k^2\right)dV$$

$$I_{ij} = \int_V \rho(\vec{r})\left(-\Delta x_i \Delta x_j\right)dV$$

where $i \neq j \neq k$ here. Each element in the diagonal and upper triangle is evaluated again using SciPy "nquad" to perform the integration. However, the integrand is defined with the explicit coordinate values replaced with their deviation from the center of mass, $\Delta x_{i,j,k}$, except for in the density function, in effect performing a coordinate transformation that orients the inertia tensor about the center of mass rather than the origin of the coordinate system. Exploiting the symmetry of the tensor, the elements of the bottom triangle are equated to the corresponding element in the upper triangle. Each element is loaded into its corresponding location in a 3x3 NumPy array, which is printed as the "Raw Inertia Tensor" for the user with elements rounded to two or more decimal places depending on the initial inputs. The eigenvalues and normalized eigenvectors are found using the "eig" routine from NumPy's linear algebra package, and these too are printed for the user, where the eigenvectors are printed as the eigenvector matrix with the eigenvectors in columns[4]. Finally, the tensor is diagonalized according to $I_D = T^{-1} I_R T$ where I_D is the diagonalized inertia tensor, T is the transformation matrix obtained from the eigenvector matrix, and I_R is the raw inertia tensor. The "dot" and "inv" routines from NumPy's linear algebra package are used to multiply and invert the matrices[4]. The eigenvalues, eigenvectors, and diagonalized inertia tensor are printed for the user and labelled as "Eigenvalue, the Moment of Inertia", "Eigenvectors, the Principal Axes of Inertia", and "Diagonalized Inertia Tensor", respectively. The stability of each axis and any intermediate axis are determined later on when the axes are plotted in the 3D figure.

2.2 **3-D Plotting**

The remaining methods includes those used to create the three-dimensional display of the object and its principal axes of rotation, along with a density distribution, if the user so chooses. The figure is created by Matplotlib's "figure" function, and a subplot with a 3-D axis projection is subsequently loaded into it. Though the order in which the objects are plotted is largely insignificant, the principal axes are plotted first, followed by the surface of the object, then the density distribution, if selected, last. See Results and Discussions for the generated figures.

2.2.1 Principal Axes of Rotation

Firstly, the principal axes are plotted as a series of points collected from a parametric representation of the axes centered on the center of mass of the object. The parameter is defined by creating a NumPy array using the "linspace" routine containing 100 linearly spaced values ranging from 105% to -105% of the largest dimension of the prism inputted by the user[4]. This range will allow the axes to extend satisfactorily in the plot such that each axis may be seen clearly from most viewing angles. A NumPy array is created for each axis to contain the 100 3-D points. Nested for loops are used to calculate each point on the axis according to

$$\vec{P} = \vec{r}_{cm} + \vec{\lambda} t$$

where $\vec{P}$ is a point on the principal axis, $\vec{r}_{cm}$ is the center of mass, $\vec{\lambda}$ is the eigenvector corresponding to the axis, and t is the parameter. Each coordinate point is simultaneously assigned to each coordinate value into the corresponding location in the axis array.

At this point, the moments of inertia corresponding to each axis are compared two at a time using the "isclose" function from the Math module to determine stability of rotation. If any two combinations of moments are within a factor of $\pm 1 \times 10^{-9}$ are each other, the "isclose" function returns true, i.e. those moments are essentially equivalent. In this case, there is no intermediate moment of inertia nor intermediate axis[6]. Therefore, rotation is stable around each axis, and a statement is printed for the user indicating such. The axes are then plotted onto the subplot using dash markers to create continuous lines and a cool color, blue or green, to represent stability. Furthermore, a statement is printed indicating that any orientation of axes with the same moment of inertia, which are plotted with the same color, is an equally valid set of principal axes. If there are two distinct moments of inertia, any axes with the greater moment are depicted in blue; else, the axes are depicted in green.

In the case that no combination of moments returns a "true" from the "isclose" function, the three moments of inertia are distinct and an intermediate axis exists around which rotation is unstable. The greatest and least moments are determined using NumPy "amax" and "amin" respectively on the array containing the eigenvalues of the inertia tensor. The corresponding principal axes of rotation are plotted onto the subplot using dashed markers as before and in blue and green colors, respectively. The remaining axis is the intermediate axis and is plotted in red to indicate instability of rotation around that axis. Finally, a message is printed to the user indicating the uniqueness of the moments of inertia, the implication of axis color, and stability of rotation around each axis.

2.2.2 Prism Surface

In this program, the object is displayed by plotting each face individually using 3D processes from Matplotlib. Each face is created using the Poly3DCollection toolkit, which requires vertices as inputs and creates a polygon in 3-D space connecting the vertices in the order they appear in the argument[2]. The toolkit requires the vertices argument to be a list of vertex coordinates zipped together, so the coordinates for the vertices of each face are defined into lists for each coordinate with coordinates corresponding to vertices in a clockwise order and are zipped together into a single list containing the vertices for face[1]. A polygon for each face is created and made with alpha (opacity as in RGBA color convention) values of 0.1, such that the object can be seen through, but with closed edges such that the edges of the prism are solid and easily distinguishable. Each face is then added to the figure subplot via Matplotlib's "add_collection3d" operation.

2.2.3 Density Distribution

At this point, the user is given the option to display an approximate density distribution in the figure. A short message indicating such and explaining the display method is printed for the user. It is indicated that a grid of points of varying opacity, where the opacity corresponds to the relative local density, is plotted throughout the interior of the prism, across the surface of the prism, or both, as the user so chooses. A series of while loops and conditionals are used to prompt and collect the user's choice as to whether or not to include a density distribution and, if so, which

method(s) to employ. Two functions are defined corresponding to both display methods, and these functions are called according to the user's choice.

To begin the density display, in both methods, the user is prompted a choice of the number of points along each dimension in the corresponding region. An array for each length of the prism is created using NumPy "linspace" spanning the entire length with a size corresponding the number of points desired by the user. These arrays function to provide all possible coordinate values for the points in the density grid, thus in the interior display method, the arrays are shifted, by using a for loop to reassign each element to the preceding index, and truncated, using NumPy "delete" such as to eliminate the endpoints, which would reside on the surface of the prism rather than on the interior.

In the surface method, six more arrays are created each to hold the coordinate values for all the points to be displayed on a given face of the prism. Thus, these arrays have dimensions $3 \times n^2$ where the first index indicates the coordinate, i.e. x, y, or z, and the second is used to enumerate each point on the grid of the corresponding face (n is the number of points per grid axis to display and is selected by the user). In the interior method, three arrays are created each to hold one of the three spatial coordinates of the grid points. Thus, they are of size n^3. Nested for loops are used to fill these arrays by simultaneously iterating through the possible coordinate values for each spatial dimension and assigning each possible combination of values to the corresponding location, row by row, within these region arrays. Thus, each specific point in the density display grid is evaluated and contained within these arrays.

To represent the local density, each point is assigned an alpha value corresponding to its relative density as compared to the maximum density within the prism. Since the density distribution is specifically linear with respect to each variable, this maximum density must occur at a corner. Thus, the density at each corner is calculated, and the greatest of which is collected as the maximum density. The alpha values are then calculated according to the ratio of local to maximum density as well as other parameters implemented to aid such that a visually appropriate opacity is observed for a wide variety of inputs. For the interior display method, alpha values, α, are calculated as

$$\alpha = \frac{2}{3n} \cdot \frac{\rho(\vec{r})}{\rho_{max}}$$

where ρ_{max} is the maximum density. The proportionality coefficient $\frac{2}{3n}$ is included to ensure an appealing degree of opacity and, via the inverse proportionality to n, to somewhat counteract the additional opacity created in the figure when many points are either in high concentration or superimposed one behind the other. In the surface display method, the alpha values are calculated as

$$\alpha = \sigma \cdot \frac{\rho(\vec{r})}{\rho_{max}} \cdot \frac{A}{l_{max}^2}$$

where σ is a darkness coefficient, A is the area of the face where the point in question resides, and l_{max} is the largest dimension of the prism. The $\frac{A}{l_{max}^2}$ term functions to counteract the effect of the concentration of points on a face in creating extra apparent opacity. The darkness coefficient σ is determined by n as

$$\sigma = \begin{cases} \frac{10}{n}, n \geq 50 \\ \frac{15}{n}, 50 > n \geq 20 \\ 1, n < 20 \end{cases}$$

and again is generally inversely proportional to n to counteract effects of the concentration in the grid.

Once the alpha values are calculated, a function is defined to take the grid point coordinates and alpha values and plot them into the subplot of the figure display as a scatter plot via Matplotlib's "scatter" function. The alpha values are incorporated into a list of RGB values for black, alike the prism color, to create a list for each individual grid point to be plotted with its corresponding RGBA values. The points are plotted with minimal size 1 to allow for the display to appear continuous when a higher number of points per dimension is chosen by the user. However, in the interior method, if $n < 10$, then the points are displayed with a larger size 5 to aid visibility. Finally, the functions are called using the arrays containing the points for each display region and the alpha values calculated previously, thus plotting the density display onto the figure.

2.2.4 Formatting

Once everything is plotted onto the subplot of the figure, the figure is given a title, axis labels, and a range centered about the center of mass. The title is also set to include the dimensions of the prism and the density distribution equation, i.e. all of the prism characteristics input by the user. The range of each dimension is set by adding or subtracting the center of mass coordinate in that coordinate axis by the greatest dimension of the prism, thus creating a range that certainly contains the prism with ample space to see the principal axes of rotation.

Early trials of the program revealed that the z-axis of the subplot was compressed by a factor of $\frac{3}{4}$ by default. This compression is corrected by adjusting the projection of the subplot. The initial matrix projection is obtained calling the "Axes3D.get_proj" function on the subplot. The function returns a 4x4 matrix containing the projection information for the plot from the default viewing position used by the figure renderer to create the display[5]. This matrix is multiplied by a diagonal matrix with unity elements except for the third, which corresponds to the z-axis and has a value of $\frac{4}{3}$. The resulting matrix is one akin to the initial but with all values pertaining to the z-axis scaled by the $\frac{4}{3}$ factor. The resulting matrix, which has elements corresponding to the z-axis is reassigned to the projection of the subplot, thus correcting the compression and preparing the plot to be displayed for the user.

Finally, the figure is displayed with the sublot, title, labels, and a legend enumerating the axes of rotation. Once the figure is closed, a goodbye message is printed.

3. RESULTS AND DISCUSSIONS

This program proved to be highly effective and efficient in its practical function. Once all the inputs are successfully collected and every quantity of interest is calculated and printed within a fraction of a second thereafter. Furthermore, the integration performed by SciPy "nquad" returns

an error that is consistently about a factor of 10^{-14} lower than the value of the integral itself, and thus certainly negligible in comparison. However, early trials found that when inertia values deviate significantly from 1, i.e. by a factor of about 10^4, that errors, even near the order of 10^{-9} where they would be physically negligible, are not neglected by the NumPy "linalg.eig" function, thus leading often to inaccurate results for the principal axes and moments of inertia. Thus, to help correct this issue, the elements of the inertia tensor are rounded to the nearest 10^{-7}, when elements are generally large, or 10^{-19}, when elements are generally small. For low to somewhat large input values, this rounding corrected any integration errors, setting these infinitesimal values to zero, and allowing the "linalg.eig" function to operate appropriately. However, for significantly large input values, some errors survive the rounding, leading still to incorrect behaviors of the inertia tensor. It is aforementioned that the script does not incorporate units into any of the physical quantities. While this is a limitation on the ease of application for the user, it allows for a universal application, provided that the user normalizes inputs and tracks units appropriately, both of which are highly recommended. Normalization is especially recommended as it serves also to prevent any significantly large or small inertia value and associated errors.

3.1 **Test Cases**

When the normalization is done, the program is shown to be highly accurate and effective by comparison to various test cases. Each quantity is accurately calculated and corroborated by literature results for multiple simple prisms. For a first test case, a cube of uniform density centered about the origin of a coordinate system has an inertia tensor known to be diagonal with elements equal to $\frac{1}{6}Ml^2$, where M is the total mass of the cube and l is the side length, with principal axes perpendicular to the faces of the cube[3]. This cube can be input in this program by inputting a prism with dimensions $1 \times 1 \times 1$, the unit cube, and density distribution with $\rho = 1$ ($A = B = C = 0$, $D = 1$). As expected, the program returns total mass of 1 and center of mass $\vec{r}_{cm} = (0.5, 0.5, 0.5)$, the center of the unit cube. The raw inertia tensor is calculated to be

$$I = \begin{bmatrix} 0.1667 & 0 & 0 \\ 0 & 0.1667 & 0 \\ 0 & 0 & 0.1667 \end{bmatrix}$$

where 0.1667 equates to $\frac{1}{6}Ml^2$ for this unit cube. Furthermore, the program returns the eigenvector matrix as the identity matrix, i.e. the principal axes of inertia run parallel to the standard basis and perpendicular to the faces of the cube, again as expected. Further testing can be done with a $2 \times 2 \times 2$ cube with $\rho = 0.125$ ($M = 1$), which returns moments of inertia of 0.667, a factor of 4 above that of the unit cube, corroborating the proportionality to l^2. Similarly, a unit cube with $\rho = 2$ ($M = 2$) returns moments of 0.333, a factor of 2 above the first unit cube, corroborating the proportionality to M. Thus, these results are exactly accurate to the literature values for this test case.

Figure 1 below shows the 3D display of the unit cube generated by the prism without any density distribution display. The dimensions and mass density equation are included clearly in the title, as intended. Within the subplot, the cube is shown clearly as a translucent gray prism with solid black edges. The principal axes of the prism appear as distinct continuous lines passing through the center of the cube and perpendicularly through the faces of the cube, directly reflecting the calculated results. The axes are also labelled in the legend according to the order in which the

corresponding eigenvectors appear in the eigenvector matrix. Thus, the 3D display is a clear and accurate portrayal of the quantitative results calculated prior for the test case of the unit cube. For the $2 \times 2 \times 2$ cube and the mass 2 cube, the 3D displays are virtually unchanged, with the exception of numerical values adjusted accordingly. Their displays are included in *Figure 2* below. For all of these cubes, since all the moments of inertia are equal, the axes are plotted in green, giving the axes a sense of stability reflecting the nature of rotation around each axis.

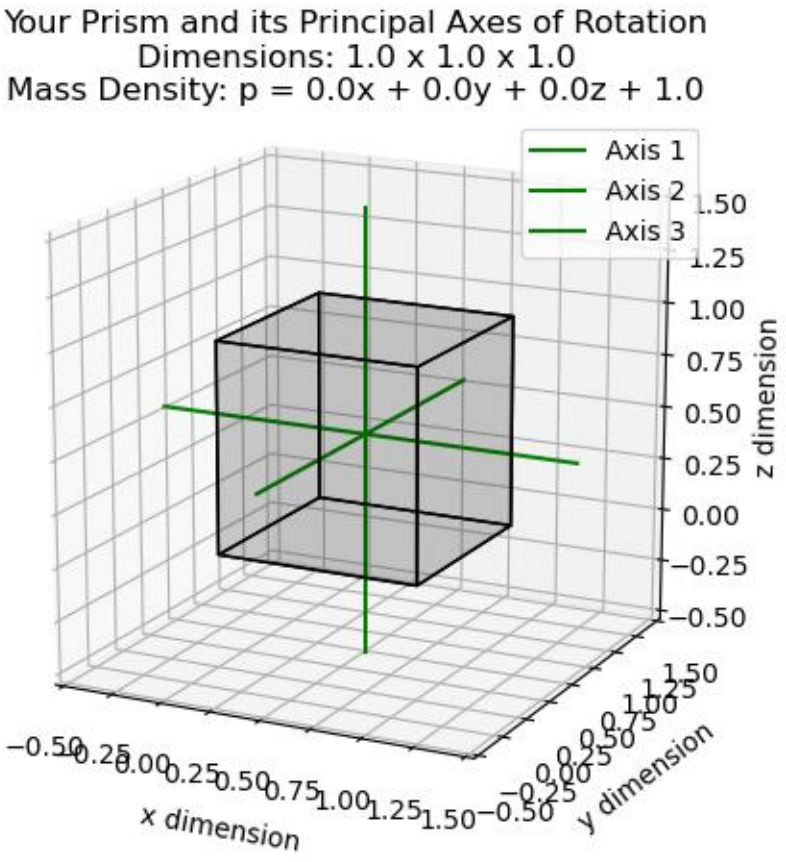
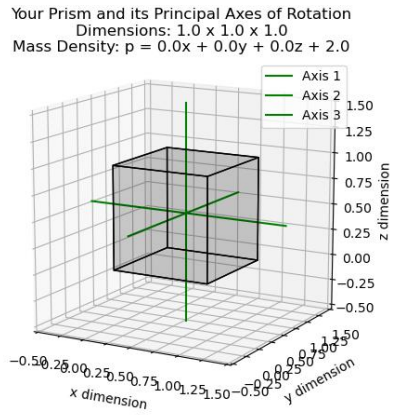
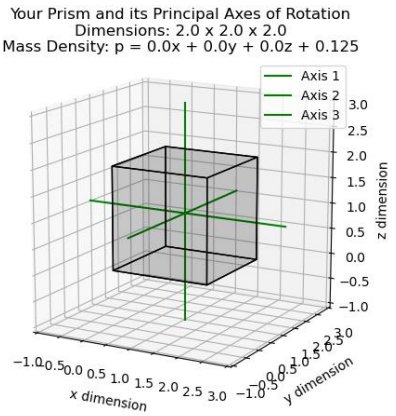


Figure 1: *The display generated by the program for a uniform unit cube with unit uniform density.*



a.



b.

Figures 2a and b: *The display for the* $M = 2$ *unit cube (left) and the* $2 \times 2 \times 2$ *cube (right, 2b) generated by the program. It should be noted that these figures are portrayed with slightly different viewing angles from other figures. This is because the 3D projection in Matplotlib allows for figures to be rotated while displayed. Therefore, a suitable angle is chosen such that each significant element can be clearly seen, but not necessarily one that is exactly the same as another figure.*

Next, any uniform prism can also be studied as a more general test case. By the symmetry of such an object, the principal axes of rotation about the center of mass would pass perpendicularly through each face, as in the case of a uniform cube. The inertia tensor is still expected to be diagonal but with moments of inertia $I_{ii} = \frac{1}{12}M(l_j^2 + l_k^2)$, where l_j and l_k are the lengths of the sides perpendicular to the x_i direction, which reduces to $\frac{1}{6}Ml^2$ for the uniform cube[3]. For an example test case, a $1 \times 2 \times 3$ prism with $\rho = 1$ ($M = 6$) returns the inertia tensor

$$I = \begin{bmatrix} 6.5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 2.5 \end{bmatrix}$$

in which each diagonal element equates exactly to $I_{ii} = \frac{1}{12}M(l_j^2 + l_k^2)$, as expected. Since this inertia tensor is already diagonalized, the principal axes are again parallel to the standard basis and pass perpendicularly through the faces of the prism. However, now each eigenvalue is distinct, so we have an intermediate moment of inertia, I_{yy}, and a corresponding intermediate axis around which rotation is unstable. *Figure 3* below depicts this prism and its axes, with the intermediate highlighted red.

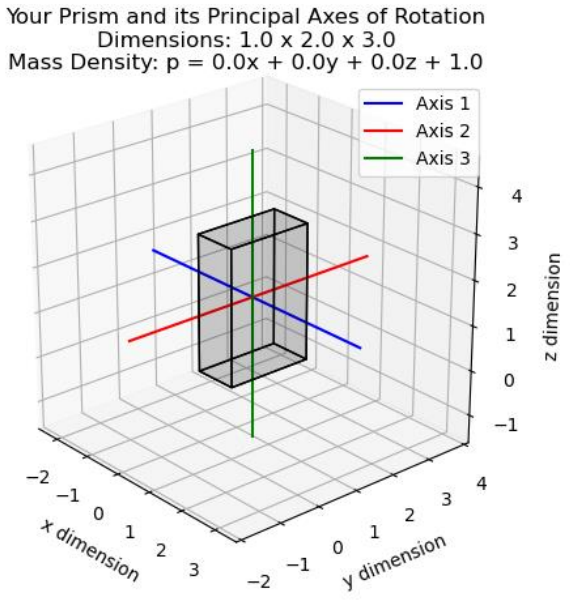


Figure 3: *Display generated for a uniform 1x2x3 prism.*

The next test cases branch from the uniform densities of the previous objects to include cubes of side length l with linear density by $\rho = x$ and $\rho = x + y + z$ (The coefficient in front of each coordinate variable is understood to have units of $\frac{mass}{length^4}$ which retains appropriate units for subsequent quantities). The inertial properties of these objects are not as commonly known; however, theoretical values can be obtained via analytic evaluation of the integral definitions of each quantity. For the $\rho = x$ cube analytic approaches yield a total mass $M = \frac{l^4}{2}$, center of mass $\vec{r}_{cm} = (\frac{2}{3}l,\ \frac{1}{2}l,\ \frac{1}{2}l\ )$, and raw inertia tensor

$$I = \begin{bmatrix} \frac{1}{12}l^6 & 0 & 0 \\ 0 & \frac{5}{72}l^6 & 0 \\ 0 & 0 & \frac{5}{72}l^6 \end{bmatrix}.$$

Evidently, the inertia tensor is already diagonal, so the principal axes are again parallel to the standard basis, and the principal moments are the diagonal elements where no intermediate moment is present, so rotation around each axis is stable, but two distinct moments do exist.

The program agrees with these analytic results, returning, for a unit cube with the corresponding density distribution, a total mass of 0.5, center of mass $\boldsymbol{r_{cm}} = (0.667,\ 0.5,\ 0.5)$, and inertia tensor

$$I = \begin{bmatrix} 0.08333 & 0 & 0 \\ 0 & 0.06944 & 0 \\ 0 & 0 & 0.06944 \end{bmatrix}$$

which matches to the analytic result rounded to the third decimal place. A trial of a $l = 2$ cube returns values adjusted by their corresponding factor of l, again corroborating the agreement with

analytic results. *Figure 4* below contains the corresponding figure generated for the unit cube of this form. This figure serves as an example of the program displaying an offset center of mass indicated by the intersection of the principal axes and indicating relative moments of inertia via the color of the axes.

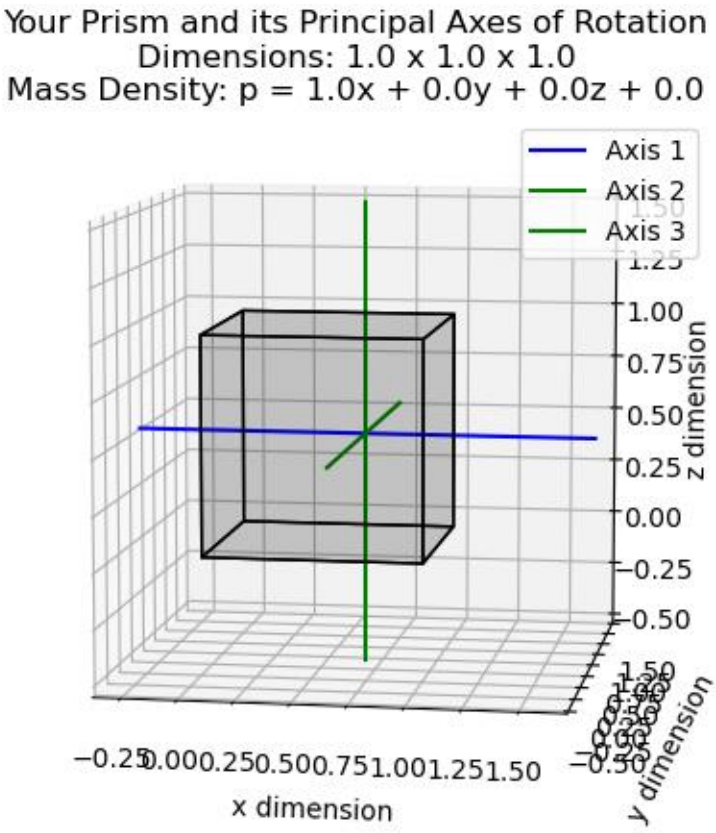


Figure 4: *The display generated for the unit cube with $\rho = x$. Note the offset of the center of mass from the center of the cube, corresponding to the calculated results above, and the blue color of Axis 1 parallel to the x direction, corresponding to the greater moment of inertia in comparison to the other two axes.*

The program also agrees with theoretical results for the $\rho = x + y + z$ cube. The analytic approach yields total mass $M = \frac{3}{2}l^4$, center of mass $\vec{r}_{cm} = (\frac{5}{9}l,\ \frac{5}{9}l,\ \frac{5}{9}l)$, and inertia tensor

$$I = \begin{bmatrix} \frac{13}{54} & \frac{1}{216} & \frac{1}{216} \\ \frac{1}{216} & \frac{13}{54} & \frac{1}{216} \\ \frac{1}{216} & \frac{1}{216} & \frac{13}{54} \end{bmatrix} l^6$$

which has eigenvalues $\lambda \approx 0.23611, 0.23611, \frac{1}{4}$ (by l^6) with the eigenvector for $\lambda = \frac{1}{4}$ given by $\vec{\lambda} = (1,1,1)$ along the gradient of the density distribution and through two opposite corners of the

cube where one is at the origin, as expected by the symmetry of the object. The program returns the following quantities for a unit cube with this density profile, all which agree to the theoretical results: mass $M = 1.5$, center of mass $\vec{r}_{cm} = (0.556, 0.556, 0.556)$, raw inertia tensor

$$I = \begin{bmatrix} 0.24074 & 0.00463 & 0.00463 \\ 0.00463 & 0.24074 & 0.00463 \\ 0.00463 & 0.00463 & 0.24074 \end{bmatrix},$$

moments of inertia $\lambda = 0.23611, 0.25, 0.23611$ with an axis parallel to (0.577, 0.577, 0.577) (normalized (1,1,1), the diagonal) corresponding to $\lambda = 0.25$. The other two axes given are (-0.816, 0.408, 0.408) and (-0.529, -0.274, 0.803), but these are arbitrary since their moments of inertia are the same. *Figure 5* below contains the display for this cube and serves to show the program's handling of principal axes that no longer lie along the standard basis. Again, given the symmetry of the cube along the diagonal axis and two similar moments of inertia, rotation around each axis is stable, as indicated in *Figure 5*.

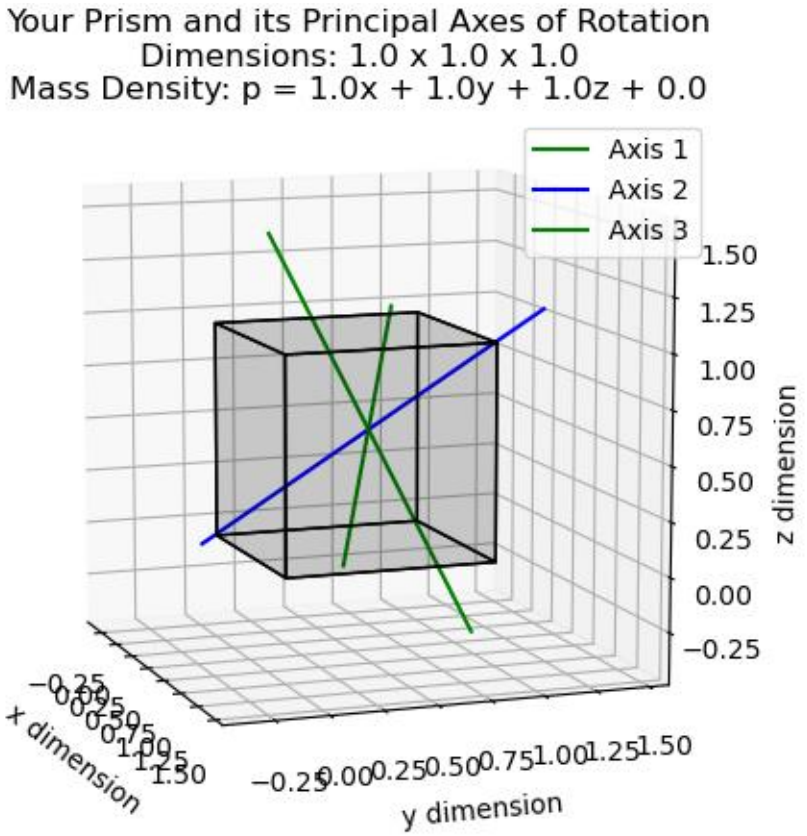


Figure 5: *The display generated for a unit cube with density profile $\rho = x + y + z$. Note especially Axis 2 passing through two opposite corners of the cube, as predicted.*

3.2 **Further Error Analysis**

The program thus far has proven to be very accurate in each of the test cases. However, each of these cases involved normalized or nearly normalized input values, such as to prevent

calculation errors as aforementioned. Here, similar prisms are compared (one normalized, one significantly larger, and two significantly smaller) to demonstrate the errors occurring when integral values, especially in the inertia tensor, become especially large.

The prisms chosen have density profile $\rho = 4x + 3y + 2z$ and dimensions $1 \times 2 \times 3$ or $20 \times 40 \times 60$, nonsymmetric in order to ensure unique elements in the inertia tensor (otherwise, these errors may not be observed). For the $1 \times 2 \times 3$ prism, the calculated properties are as follows: total mass $M = 48.0$, center of mass $\vec{r}_{cm} = (0.542, 1.125, 1.687)$, raw inertia tensor

$$I = \begin{bmatrix} 49.5625 & 0.25 & 0.375 \\ 0.25 & 38.22917 & 1.125 \\ 0.375 & 1.125 & 19.16667 \end{bmatrix},$$

moments of inertia $\lambda = 49.573, 19.096, 38.289$ with respective principal axes of rotation

$$\vec{\lambda} = \begin{pmatrix} 0.9996 \\ 0.0233 \\ 0.0132 \end{pmatrix} \begin{pmatrix} 0.0118 \\ 0.0585 \\ -0.9982 \end{pmatrix} \begin{pmatrix} 0.0241 \\ -0.9980 \\ -0.0582 \end{pmatrix}$$

(these vectors printed are rounded slightly, so their magnitude may not equate exactly to 1), and the appropriate diagonalized inertia tensor corresponding to the moments of inertia.

In the test cases for various cubes, each quantity is shown analytically to vary by some power of l. Although these prisms are no longer cubes, the larger prism has each length increased by a factor of 20 from the smaller and it is expected that each physical quantity be adjusted by the corresponding power of 20 according to the cases with variable density (without constant term as including this term would complicate the conversion, i.e. different portions of a quantity would vary by a different power of l). Thus, the program returns the following quantities for the $20 \times 40 \times 60$ prism: total mass $M = 7680000.0$, center of mass $\vec{r}_{cm} = (10.833, 22.5, 33.75)$, raw inertia tensor

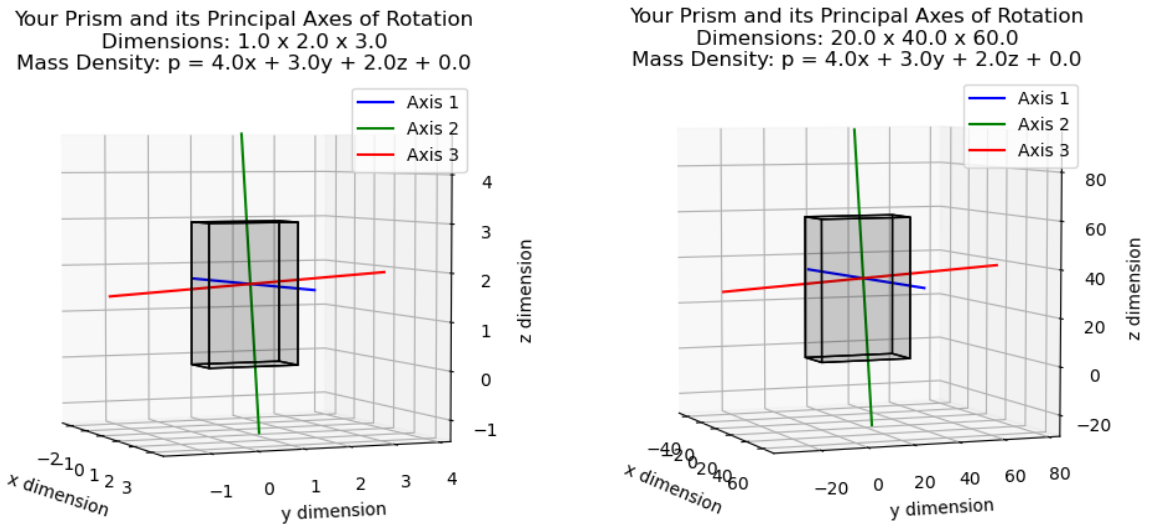
$$I = \begin{bmatrix} 3.172 \times 10^9 & 1.600 \times 10^7 & 2.400 \times 10^7 \\ 1.600 \times 10^7 & 2.447 \times 10^9 & 7.200 \times 10^7 \\ 2.400 \times 10^7 & 7.200 \times 10^7 & 1.227 \times 10^9 \end{bmatrix},$$

moments of inertia $\lambda = 3.173e9, 1.222e9, 2.450e9$, with the exact same eigenvectors. Each quantity is proportional to the corresponding quantity in the $1 \times 2 \times 3$ prism by the power of 20 predicted. Thus, the program is shown to actually maintain accuracy even in this extreme. However, the SciPy integration returns an "IntegrationWarning" due to the detection of roundoff error causing deviance from the default error tolerance. Similarly, the diagonalization of the inertia tensor is not exactly accurate; off-diagonal elements exist as in

$$I_D = \begin{bmatrix} 3.173 \times 10^9 & 6.107 \times 10^{-8} & 1.363 \times 10^{-8} \\ 6.107 \times 10^{-8} & 1.222 \times 10^9 & 4.244 \times 10^{-7} \\ 1.363 \times 10^{-8} & 4.244 \times 10^{-7} & 2.450 \times 10^9 \end{bmatrix}$$

suggesting some rounding discrepancy between the integration and eigenvector functions. Regardless, these off-diagonal elements are consistently about 17 orders of magnitude lower than the diagonal elements, indicating that they are physically negligible in comparison. Furthermore, this is somewhat less than the estimated integral errors at 14 orders of magnitude below the integral values. Therefore, since the effect of rounding errors in significantly large cases only occurs at values many orders of magnitude less than the values of interest and sits within the error margin, they can be neglected and the program retains its practical accuracy in this larger case, though it may be more theoretically accurate when normalized.

Figure 6 contains the displays generated for each of the prism. Note that, with the exception of the dimensions, the two displays are nearly indistinguishable, reflecting the program's accuracy and consistency in handling similar objects despite the rounding errors in the significantly large regime.



a. **b.**

Figure 6: *The displays generated for the prisms with density profile* $\rho = 4x + 3y + 2z$ *and dimensions* $1 \times 2 \times 3$ *(left, a) or* $20 \times 40 \times 60$ *(right, b).*

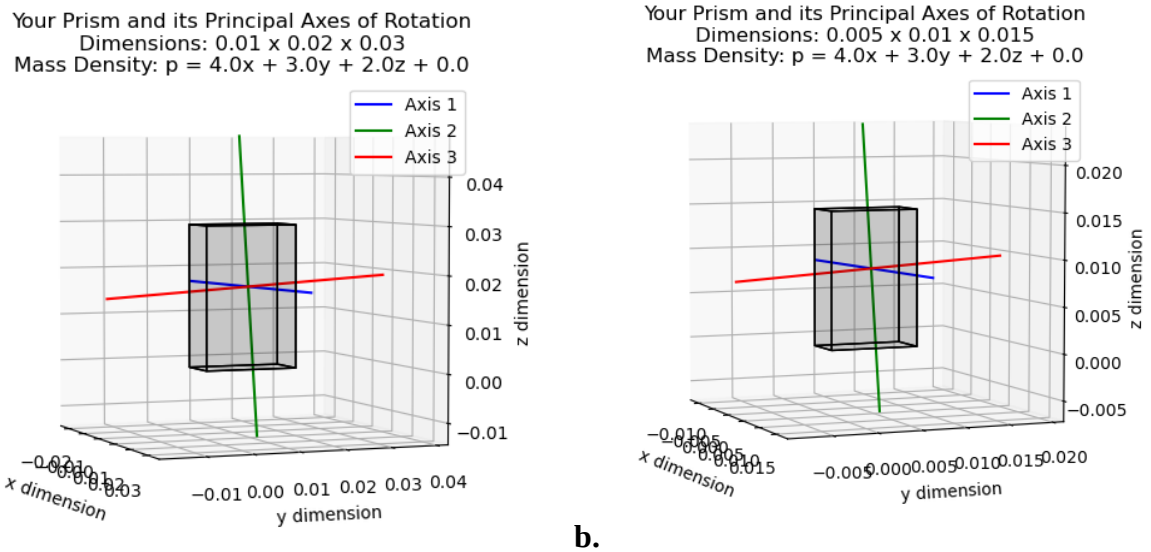
In the significantly small regime, however, rounding creates error, as the rounding done when printing the quantities eliminates significant elements in the raw inertia tensor. A prism with the same density profile as above but with dimensions $0.01 \times 0.02 \times 0.03$ (a factor of 100 reduced from the normalized prism above) returns the appropriate values for total mass, center of mass, and eigenvectors, but returns the raw inertia tensor

$$I = \begin{bmatrix} 0.00000000005 & 0 & 0 \\ 0 & 0.00000000004 & 0 \\ 0 & 0 & 0.00000000002 \end{bmatrix}$$

which results in the same tensor when diagonalized. This raw tensor appears diagonalized, but this is erroneous, as the matrix printing operation rounds the tensor values. Thus, the off-diagonal elements of the tensor, which would be exceptionally small (on order 10^{-13}), and much of the diagonal elements are lost to the rounding when printed. However, these elements are kept internally for calculation, hence why the appropriate eigenvectors are returned rather than the standard basis. Thus, the program itself retains some accuracy, but the rounded inertia values may be erroneous.

Furthermore, testing a yet smaller prism, $0.005 \times 0.01 \times 0.015$, leads to a more erroneous inertia tensor printed to the user, though all other physical quantities vary as expected. In the tensor, every element is eliminated by the print rounding, resulting in a zero matrix. Similarly, the eigenvalues are all printed as zero. However, the eigenvectors remaining the same reflects that the correct values are held within the program and that this error is due only to rounding when printing. Thus, either the user should normalize their input values or manually adjust the printing precision

within the source code to alleviate this error. Regardless, the values internal to the program remain accurate, as suggested by the displays generated for these small prisms, which match the displays of their larger counterparts and are included in *Figure 7*.



a. **b.**

Figure 7: *The displays generated for the prisms with density profile* $\rho = 4x + 3y + 2z$ *and dimensions* $0.01 \times 0.02 \times 0.03$ *(left, a) or* $0.005 \times 0.01 \times 0.015$ *(right, b). Note the correlation with Figure 6.*

3.3 Evaluation of Density Display Methods

Continuing with the $1 \times 2 \times 3$ prism with $\rho = 4x + 3y + 2z$, the two density display methods are qualitatively evaluated.

Figure 8 contains the display generated using the surface method with 70 points per axis from multiple viewing angles.

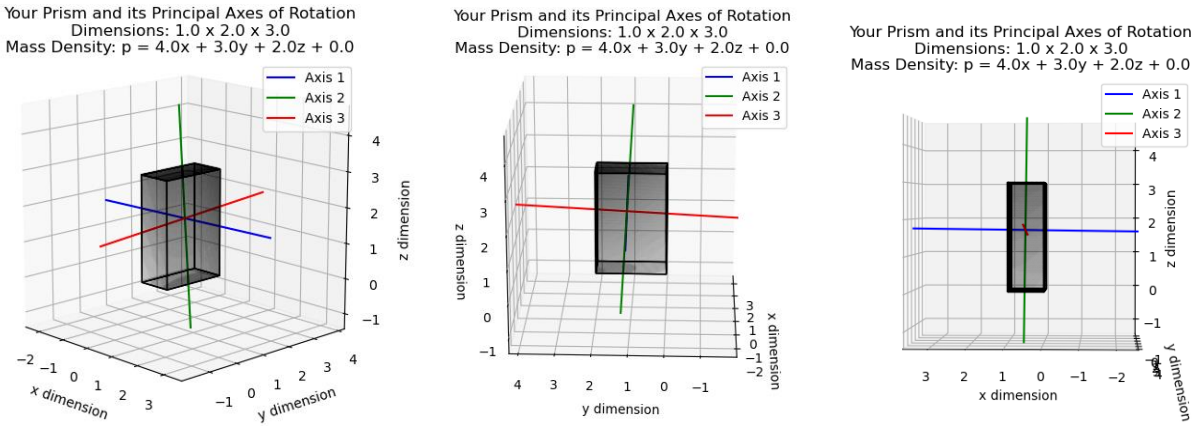


Figure 8: *Display of the* $1 \times 2 \times 3$ *prism with* $\rho = 4x + 3y + 2z$ *using the surface method with* $n = 70$ *as viewed from the default angle (left), the back face (center), and the right face (right), arbitrarily chosen.*

The choice of 70 points per axis is shown to be effective in producing a nearly continuous shading representing the continuous density of the object. Furthermore, a relatively uniform color gradient is observed on the visible faces of the prism corresponding to the gradient of the linear

density profile. Thus, this method is effective in providing a visualization of the variation in density along the surface of the object, which is sufficient for a linear density profile given there is no unique behavior within the interior.

However, it is also evident that the darkness of a face changes with viewing angle, likely due to points overlapping each other, increasing the opacity of a face, erroneously suggesting a greater density. Thus, only the foremost faces at a given viewing angle should be considered to intuit the density distribution most accurately, as faces at an angle appear denser than they actually are. Similarly, a close look at *Figure 8* will reveal diagonal lines where the darkness of the surface changes noticeably, sometimes inconsistently with the true density gradient. These lines were also observed to move and change with viewing angle; therefore, they are also believed to result from overlapping points, but from a front face to a rear face as well as within a given face, given that this phenomenon sometimes, but not always, changes when looking through different faces. Moreover, the notion that multiple faces can be seen at once in line with one another is unrealistic and may mislead the user by depicting multiple color gradients in one region. Lastly, the overall darkness of the prism is found to vary with the number of points per axis: selecting fewer points generally leads to a lighter display, selecting more produces the opposite effect. Although this phenomenon is not depicted here, the user should be aware when comparing displays of similar prisms.

Figure 9 contains the displays generated for the same prism using the interior grid method but with 50 points per axis.

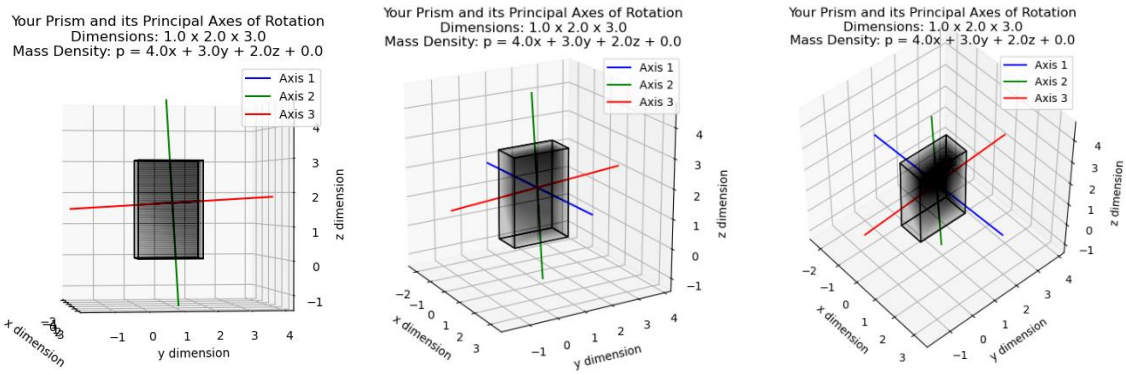


Figure 9: *Interior grid method with $n = 50$ as viewed from the front face (left), above and left of the front face (center), and higher above the front face (right).*

The density gradient is most clearly seen in the left plot of *Figure 9* where the object is viewed head on, demonstrating that the grid accurately represents local mass densities. However, when the prism is viewed at an angle, the outer regions of the object appear lighter with a gradient progressing into the object and the inner region appears darker depending on the angle, erroneously suggesting that the object is less dense around the outside and that the density inside varies with angle. This effect, though, is simply a result of the variable number of points along the line of sight from a given viewing position. Thus, since each point represents local mass, here the darkness of a region correllates more with total mass along line of sight rather than with local density, which is not the desired effect, but is included should a user desire this effect. Therefore, the interior method is only effective in depicting local density when the prism is viewed head on to a face.

Lastly, *Figure 10* contains the diplays generated using both the surface and interior methods.

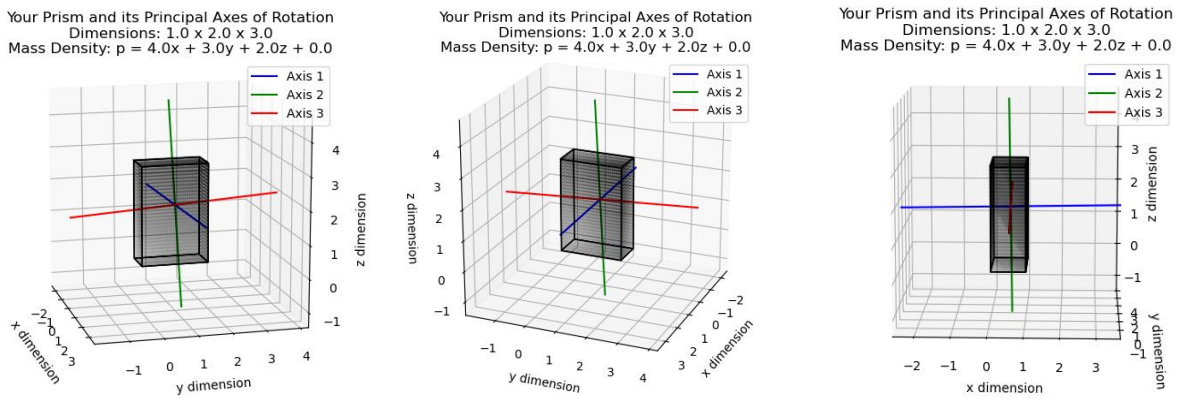


Figure 10: *Both density display methods using $n = 30$ as viewed from near head on to the front face (left), from above and right of the front face (center), and from near head on to the left face (right).*

This display option results in the most complete image and clearest representation of the density profile of the object. The density gradient is clear and consistent throughout the entirety of the prism from nearly every viewing angle. The only exception to this behavior being the occasional appearance of lines with drastic opacity change akin to those observed in the surface display method as observed in the right plot of *Figure 10*. Thus, the choice to plot points both within the interior and along the exterior results in the most physically accurate representation of the object's density.

However, because of the large number of total points plotted, the figure is rendered much more slowly when using both methods as compared to using either individually. Similarly, the interior method takes longer than the surface method, since the number of points goes by $6n^2$ for the surface method, n^3 for the interior method, and of course the sum of the two for both. The rendering speed, thus, decreases drastically with larger n, though each of the figures included in this paper were each generated within a few seconds on a Dell laptop with an Intel® Core™ i7 10th generation processor running at 2.6 GHz. It is expected that rendering times on other devices remain within a similar timeframe.

Lastly, it should be noted that the overall darkness of the prism in a given density distribution method, too, varies with n; i.e. the display usually appears darker with greater n. One may note that the display in *Figure 10* is lighter than what one would expect from superimposing the displays from *Figure 8* and *Figure 9*, thus indicating that the individual displays for the two methods with $n = 30$ are lighter than the corresponding displays with $n = 50$ or $n = 70$. *Figure 11* contains these displays and demonstrates this behavior when compared with *Figures 8* and *9*. While the general gradient is still observed, it may be harder to notice, and a user should be aware when attempting to intuit relative densities.

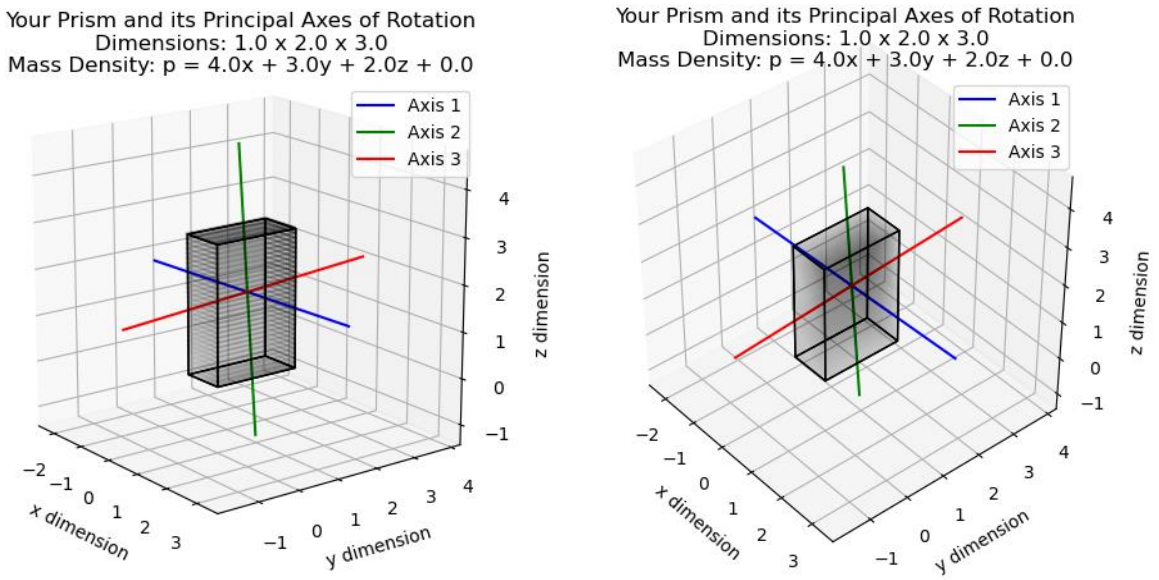


Figure 11: *The displays generated for* $n = 30$ *with the surface method (left) and the interior method (right).*

4. CONCLUSIONS

This program proved to be highly accurate in each test case, matching literature or analytic results for each inertial property of a normalized object exactly within the rounding parameters, with negligible estimated integration errors (of order 10^{-14} from the corresponding quantity), and with very quick speed. The 3D display generated also consistently depicts exactly what is expected for each test case, clearly displaying the prism with its principal axes of rotation intersecting at the center of mass and color-coded according to relative principal moment of inertia and stability of rotation around the axis. The density display methods also prove to be fairly effective, depicting the general behavior of the density profile by displaying a nearly continuous color gradient corresponding to the gradient of the density profile. Displaying grids both across the surface and within the interior of the prism proved to be the most physically accurate depiction, although it is the slowest to render. The surface method renders the quickest and provides a reasonable depiction for the linear density profile, although it contains some confusing behaviors due to the overlap of grids on different faces. The interior method proved to be more accurate in depicting total mass along line of sight than local density behavior. Each can be used to effectively visualize the density profile, at the very least when viewed from an appropriate angle or when ignoring inconsistencies in favor of general trends, though the equations governing the darkness of an individual require further alteration to ensure consistency across display parameters. Regardless, the inertial properties of the user-input prism are quickly and effectively calculated and displayed, meeting the primary goals of the project.

However, when inertia values become excessively large or small (a factor of $10^{\pm 9}$ from unity), some notable errors appear in the inertia tensor. In the large regime, the diagonalized inertia tensor picks up off-diagonal elements, though they are small and negligible relative to the diagonal elements. In the small regime, some tensor elements are rounded to zero or few significant digits when printed for the user, withholding lots of information from the user. The eigenvectors and 3D

display remain consistent, indicating that accurate values are held internally, but it is necessary for the user to alter the printing options within the source code in order to extract accurate inertia values.

Alternatively, the user can normalize their input values to avoid these limitations. Given that this program does not incorporate units, it is already necessary for the user to track units themself. Similarly, the user may attribute some factor of the dimensions or density profile to the physical unit, shifting their input values from small or large quantities closer to unity before inputting them into the program and avoiding these extremes where the program behaves inaccurately. The user would then reincorporate these factors along with their physical units to obtain the appropriate quantities for their physical object. Thus, this normalization practice is highly recommended as it ensures optimal accuracy of the program for any object and allows the program to be applicable for objects on any scale.

Given the accuracy and applicability of this program, it serves as an effective model program to develop programs to calculate and visualize the inertial properties of other non-uniform objects. Such a program for a given object could follow the same structure as this program, but with a few alterations to adapt the program to fit the appropriate object. For example, the simplest adaptation would be to a rectangular prism with a density profile of a different functional form. One would simply need to adjust the functional form of the density definition in the inputs, printed messages, and callable calculation function. It would be necessary to reevaluate estimated errors returned by the SciPy "nquad" function. The Gaussian quadrature method used fits polynomials best, so other functional forms may return larger errors than are returned here[9]. Moreover, the surface density display method may no longer be appropriate alone given the potentially nonlinear behavior of the density within the prism. With these alterations in mind, the program would be appropriately adapted for rectangular prisms with nonlinear density distributions.

Adaptations for other shapes would be more complex, but may still follow the same structure as this model. For calculations, the SciPy "nquad" function allows for callable functions to be given as bounds for integration, so one may define functions to represent the surface of an object. Alternatively, one may introduce a Jacobian term in each integrand and change coordinate systems entirely. The calculation of each inertial quantity would follow the same structure, but the 3D plot of the object might involve plotting 3-dimensional surfaces rather than polygons. Similarly, the density display grids would have to be redefined according to the surface functions of the object. These adaptations are merely suggestions, though, and further study is needed to produce or determine the feasibility of producing such a program. Furthermore, further study is needed for the implementation of the parallel axis theorem to adapt the program for rotation about any location, rather than strictly the center of mass. Therefore, assuming these adaptations are feasible, this program serves as an effective model to develop programs to calculate and visualize these inertial properties for any object of any density profile for rotation about any point.

5. ACKNOWLEDGEMENTS

I would like to sincerely thank Dr. Massimiliano Bonamente for introducing me to Python methods in physics and for advising this project and guiding its development.

References

[1]"API Reference". *Matplotlib*. 2002. https://matplotlib.org/stable/api/index.html

[2]Khan, Shahbaz. "3D plotting in Python using matplotlib". *LikeGeeks.com*. August 27, 2022. https://likegeeks.com/3d-plotting-in-python/#Plotting_a_3D_Polygon

[3]Moore, Jacob, et. al. "Center of Mass and Mass Moments of Inertia for Homogeneous Bodies." *Mechanics Map: Open Textbook Project*. Pennsylvania State University, 2022. http://mechanicsmap.psu.edu/websites/centroidtables/centroids3D/centroids3D.html

[4]"NumPy Reference". *Numpy*. December 18, 2022. https://numpy.org/doc/stable/reference/index.html

[5]Pronobis, Andrzej. "matplotlib (mplot3d) - how to increase the size of an axis (stretch) in a 3D Plot?". *Stack Overflow*. May 24, 2015. https://stackoverflow.com/questions/30223161/matplotlib-mplot3d-how-to-increase-the-size-of-an-axis-stretch-in-a-3d-plo/30419243#30419243

[6]"Python math.isclose() Method". *W3 Schools*. 1999. https://www.w3schools.com/python/ref_math_isclose.asp

[7]"SciPy API". *Scipy*. https://docs.scipy.org/doc/scipy/reference/index.html

[8]Thorton, S., Marion, J. *Classical Dynamics of Particles and Systems*. Fifth Ed. 2004, Brooks/Cole–Thomson Learning. ISBN 0-534-40896-6

[9]Weisstein, Eric W. "Gaussian Quadrature." From *MathWorld*--A Wolfram Web Resource. https://mathworld.wolfram.com/GaussianQuadrature.html