NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4255 - Robotic Vision

Matrix-free Transformations Using Projective Geometric Algebra (PGA)

Author:
Zahra Parvinashtiani

29.04.2024

Table of Contents

List of Figures

List of Tables

1 Abstract

This project outlines a project on matrix-free transformations using Projective Geometric Algebra (PGA) within the context of robotic vision and delves into PGA as a unifying framework for geometrical computation, enabling simplification of computational difficulties in domains where calculations depend heavily on geometrical properties. It particularly emphasizes the practicality of PGA in performing operations such as rotations, translations, and manipulations of geometric entities without the need for coordinate matrices. It also covers the theoretical foundations of PGA, describes the algebraic representation of geometric entities, and discusses various PGA operations, including rotors, translators, and motors. It also details an experiment involving a Python program designed to rotate and translate an image using PGA, discussing the results and the advantages of PGA over traditional methods.

2 Introduction

Projective Geometric Algebra is a new way of thinking about geometry and graphics that can solve many different problems. it was eveloped from the work of academia in the early 20th century, this algebraic structure has, over the years, been refined to adequately address the practical and theoretical challenges associated with geometrical computation. The algebra is an extension of the traditional Clifford Algebra with additional space from projective geometry hence the ability to offers a uniform approach to rotation, translation, and manipulation of geometrical objects without the requirement of Coordinate matrices. This algebra has been extremely helpful in simplification of computational difficulties in cases where computing depends on geometrical properties, and robotic vision is the most important domain. The utility of PGA to this field can never be exaggerated.

The algebra by using algebraic statements helps us to define geometrical entities and their properties to reflect the actual properties of these objects. More specifically, PGA achieves its goal with the use of multivectors to encapsulate the properties of points, lines, and planes in matrix leads to a single algebraic property and expression. In addition to simplifying the computational level usually associated with matrices in affine and Euclidean geometries, this approach also succeeded in expanding the level of algebraic expression. In short, PGA goes beyond the algebra of matrices and expression to increase the level of fluency in both levels without any form of its simplification. (n.d.)

From the following discussion, it is evident that PGA is a practical system with practical application in the development of various computations systems. More specifically, it serves as an essential algebra that enables mathematicians, engineers, and computer scientists to simplify a wide range of properties observed in algorithmic structures. The following discourse seeks to provide the underlying logical argument that demonstrates the utility in this remarkable algebraic structure.

3 Theoretical Foundations of Projective Geometric Algebra

Projective Geometric Algebra is built on a straightforward set of rules centered around multivectors and three foundational operations: the geometric product, its inverse, and the outer product. The latter is defined for a limited set of inputs, such as bivectors, with all operations on multivectors being linear.Keninck (March 14, 2022)

3.1 Representation of Geometric Entities

In projective geometry, multivectors in PGA represent various geometric entities. A multivector might encompass geometric primitives or constructs in two dimensions—points, lines, and planes. In three dimensions, basic geometric entities are points, lines, and planes, chosen not arbitrarily but for their convenience in naturally expressing geometric relationships.

- Plane: A plane p in PGA is represented by $p = n_x e_1 + n_y e_2 + n_z e_3 + d.e_0$, where n_x, n_y, n_z make the normal vector, and d is the scalar that represents the distance from the origin and e_0 is the element that represents the ideal plane at infinity. This representation integrates perfectly with the homogeneous coordinates of PGA.
- Line: In PGA, lines are typically represented as the meet of two planes, resulting in a bivector.
- Point: Points are represented as the intersection of three plane, resulting a trivector.

3.2 PGA Operations

In geometric algebra, the geometric products are the key operations that combine elements of the algebra to produce new geometric entities or transformations. The geometric products between two multivectors (which can be vectors, bivectors, trivectors, etc.) results in a new multivector.

3.2.1 Geometric Product

The geometric product between two vectors a and b is showed ab and can be expressed as the sum of an inner product $(a.b)$ and an outer producta^b.:

$ab = a.b + a^b$

The geometric product of a vector and a bivector or of two bivectors usually results in a mix of scalar, bivector, and possibly trivector components in PGA.

3.2.2 Inner Product

The inner product measures the extent that two vectors or other elements are parallel.

The inner product between two vectors results in a scalar and is equivalent to the dot product in traditional vector algebra where θ is the angle between two vectors and the $|a| \, and \, |b|$ are the magnitudes of two vectors:

$a.b = |a| \, |b| \cos \Theta$

The inner product of a plane and a line, for example, will determine if they are orthogonal (the inner product is zero) or will give the line's component that lies in the plane.

3.2.3 Outer Product

The outer product, or wedge product, is shown by ‘. Outer products is antisymmetric, meaning $a^b = -b^a$.

- For Points: The outer product of two points gives a line (bivector) passing through both points.
- For Lines: The outer product of two lines gives a plane (vector) if the lines are coplanar and non-parallel or a point (trivector) if the lines intersect.
- For Planes: The outer product of two planes gives their line of intersection (bivector).

3.2.4 Sandwich Product

This is not a distinct product but a common operation where a geometric entity is "sandwiched" between a product and its reverse. For example, in the case of rotations, a point is rotated by

sandwiching it between a rotor R and its reverse R^{-1} (or conjugate, depending on the convention). This applies the rotation defined by the rotor to the point.

3.3 Handling of Infinity

In Projective Geometric Algebra (PGA), the concept of infinity is nicely handled by the algebraic structure itself. In homogeneous coordinates, a point in projective space is represented by an n+1 dimensional vector, where the additional dimension allows for the representation of points at infinity. In PGA, the element e_0 is used to represent the ideal plane, or the plane at infinity. This allows PGA to handle transformations and intersections that involve infinity without any special cases or additional complexity. For example:

- Lines at Infinity: In 2D PGA, a line at infinity can be represented simply as $l_\infty = e_{12}$ where e_{12} is the bivector basis element for the ideal line.
- Points at Infinity: Similarly, a point at infinity in the direction of a vector $v = xe_1 + ye_2 + ze_3$ can be represented as v_0^e, highlighting the direction of v but indicating that it's a point at an infinite distance.

3.4 Projective Duality

Projective duality in PGA is a principle that allows for the interchange of concepts such as points and planes by using the notion of duality. This concept is rooted in projective geometry, where each theorem has a dual statement that can be derived by interchanging the roles of points and planes, lines and line segments, and so on.

This duality is often expressed algebraically by taking advantage of the dual space. For instance, if a point is represented as a trivector in PGA, its dual would be a vector representing a plane, and vice versa. The act of taking a dual in PGA can be as simple as multiplying by the pseudoscalar of the space, which is the product of all the basis vectors and represents the highest-dimensional element of the algebra.

For transformations, this means that operations on points can be translated into equivalent operations on planes using duality. This is powerful in geometric computing because it simplifies the construction and manipulation of geometric entities and transformations, allowing for a unified approach to solving problems.

3.5 Rotors

Rotors in PGA are used to represent rotations. They are the same as to complex numbers or quaternions used in traditional rotation representations, but are native to PGA and work directly with the geometric entities of the algebra.

In 3D, a rotor R for a rotation by an angle θ in the plane defined by the bivector B is given by the exponential of the bivector scaled by half the angle: $R = exp(-\frac{B\Theta}{2})$

where B is a bivector that is equivalent to the plane of rotation and exp denotes the exponential map.

In 2D, since there's only one plane of rotation, a rotor is simpler:

$R = exp(-\frac{\Theta}{2}e_{12})$

We can simplify that to

$R = \cos\frac{\Theta}{2} - e_{12}\sin\frac{\Theta}{2}$

where θ is the rotation angle and e_0 is the bivector representing the plane of rotation (in 2D, it's simply the unit bivector since there's only one plane).

3.6 Translators

Translators in PGA are used to represent translations. They are similar to the concept of rotors but meant to capture translational movements. A translator T that translates by a vector v is shown as:

$T = 1 + \frac{1}{2}v.e_0$

where v is the translation vector, and e_0 is the element representing the point at infinity in PGA.

In 2D, A translator T for a translation by a vector $t = (t_x, t_y)$ is given by:

$T = 1 + \frac{1}{2}t_x.e_{01} + \frac{1}{2}t_y.e_{02}$

where e_{01} and e_{02} are basis elements that, together with the unit pseudoscalar e_{12} , complete the algebra.

3.7 Motors

Motors are the unification of rotors and translators. They represent a rigid body transformation in space, including both rotation and translation. A motor M is constructed by combining a rotor R and a translator T in PGA, typically by their geometric product:

$M = TR$

This motor M can then be used to apply the combined rotation and translation to a point or other geometric entity using the sandwiching operation:

$p^{'} = MpM^{-1}$

where p is the original point, and $p^{'}$ is the transformed point.

4 Experiment

For better understanding the PGA I wrote a simple python program that rotates and translates an image. The code used for that is in appendices.

4.1 Basis Elements of PGA

defining the basis elements correctly for a 2D PGA implementation.

4.2 Geometric Product

The gp function implements the geometric product for the PGA, which is used to combine rotors and translators.

4.3 Reverse Multivector

The reverse function is used to reverse the elements of a multivector, which is necessary when applying transformations using the sandwich product.

4.4 Rotor Creation

The make_rotor function creates a rotor, which represents a rotation in PGA. The rotor is based on the angle theta.

4.5 Translator Creation

The make_translator function creates a translator for translations in PGA.

4.6 Transformation Points

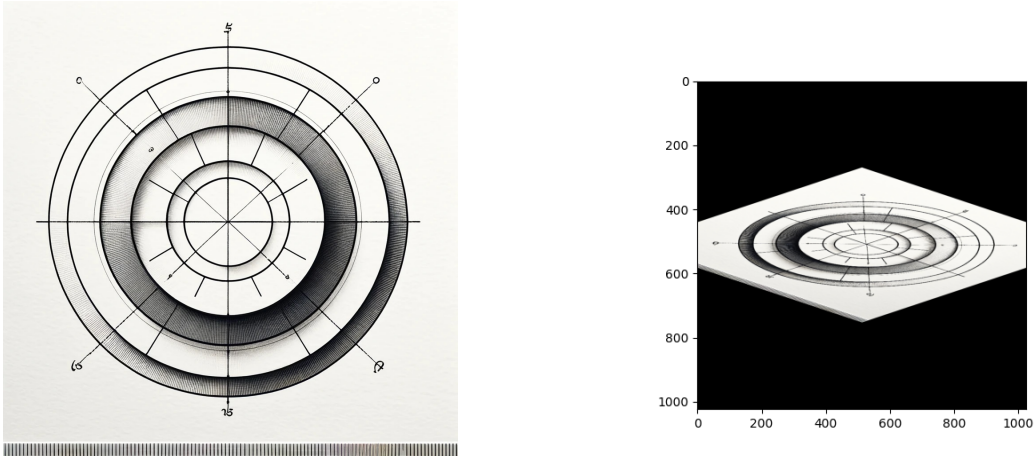
he transform_point function converts a point to its PGA representation, applies a transformation via the motor, and converts it back to the original space.

4.7 Applying the Transformation

- loading an image using PIL and converting it to a numpy array for manipulation.
- setting the rotation angle theta and translations tx and ty
- applying the motor to each pixel in the image to create a new transformed image.

4.8 Result

The result for $\theta = 45 and (t_x, t_y) = (2, 2)$ is shown below in figure 1:

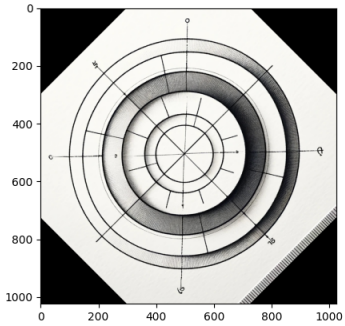


The orginal image

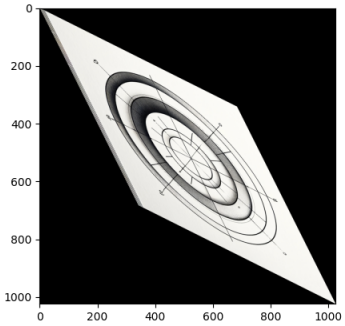
The image with 45 dgree rotation and 2,2 translation

Figure 1: The original image and transformed image

We can easily just do rotation or translation by just setting $\theta = 0 or (t_x, t_y) = (0, 0)$. The result for that are shown below in figure **??**:



The image with just 45 dgree rotation



The image with just 2,2 translation

5 Discussion

5.1 Advantages Over Traditional Methods

Compared to traditional methods that use matrices and quaternions for transformations, PGA offers several advantages:

- Fewer parameters are needed to describe the same transformations.
- PGA unifies different types of transformations into a single algebraic framework, which makes the mathematical handling and computation easier.

5.2 Obstacles

- Because of having for loops, the implementation was slow
- Transformations moved pixels outside the original image bounds and i had empty (black) regions in the transformed image.

6 Conclusion

The project's findings showcase the efficacy of PGA in simplifying geometric computations, providing a uniform approach to transformations that traditionally rely on matrices. The Python implementation of the experiment demonstrates the practical application of PGA for manipulating image data, resulting in successful rotation and translation transformations. While the PGA method offers fewer parameters and a unified framework for transformations, our implementation also encounters challenges, particularly with respect to performance due to the non-optimized nature of the loops in the Python code. Moreover, the handling of image boundaries posed a challenge, as transformed pixels moved outside the original image bounds, creating empty regions. Future improvements could include optimizing the code for performance, refining the handling of image edges, and implementing interpolation methods to enhance image quality post-transformation. Given the opportunity to restart the project, a focus on optimization and comprehensive error handling would be prioritized to ensure seamless transformation processes.

7 Use of AI

- ChatGPT:For what should I choose for my project and what's the difference between topics.

- Copilot: Help with code writing
- ChatGPT: For helping to write technical language and having well-structured sentences
- ChatGPT: Help with organizing the article.

Bibliography

(N.d.). URL: https://bivector.net/doc.html.

Keninck, Leo Dorst Steven De (March 14, 2022). *A Guided Tour to the Plane-Based Geometric Algebra PGA*. bivector.net.

Appendix

A Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Define the basis elements of PGA (2D)
e0 = np.array([1, 0, 0, 0])
e1 = np.array([0, 1, 0, 0])
e2 = np.array([0, 0, 1, 0])
e12 = np.array([0, 0, 0, 1])

# Geometric product for PGA 2D
def gp(a, b):
    result = np.zeros(4)
    result[0] = a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3]
    result[1] = a[0]*b[1] + a[1]*b[0] + a[2]*b[3] - a[3]*b[2]
    result[2] = a[0]*b[2] + a[2]*b[0] - a[1]*b[3] + a[3]*b[1]
    result[3] = a[0]*b[3] + a[3]*b[0] + a[1]*b[2] - a[2]*b[1]
    return result

# Reverse the multivector
def reverse(a):
    return np.array([a[0], a[1], a[2], -a[3]])

# Create a rotor for rotation by theta radians around the origin
def make_rotor(theta):
    cos_half_theta = np.cos(theta / 2.0)
    sin_half_theta = np.sin(theta / 2.0)
    return cos_half_theta * e0 + sin_half_theta * e12

# Create a translator for translation by (tx, ty)
def make_translator(tx, ty):
    return e0 + 0.5 * (tx * e1 + ty * e2)



# Transform a point using a motor (translator * rotor)
def transform_point(p, motor, center):
    p_centered = p - center
    p_mv = p_centered[0] * e1 + p_centered[1] * e2 + e12
    p_transformed = gp(gp(motor, p_mv), reverse(motor))
    p_transformed_centered = np.array([p_transformed[1], p_transformed[2]]) +
    ↪  center
    return p_transformed_centered

# Load an image
image_path = '01.jpg'
image = Image.open(image_path)
data = np.array(image)
print(image.size)

# Rotation and translation parameters
theta = np.pi / 4  # Rotation angle
tx = 2  # Translation along x-axis
```

```
ty = 2  # Translation along y-axis


rotor = make_rotor(theta)
translator = make_translator(tx, ty)
motor = gp(translator, rotor)

# Apply the transformation to the image
height, width, channels = data.shape
new_data = np.zeros(data.shape, dtype=data.dtype)
cx, cy = width // 2, height // 2

for y in range(height):
    for x in range(width):
        transformed_pos = transform_point(np.array([x, y]), motor, np.array([cx,
        ↪  cy]))
        xi, yi = int(transformed_pos[0]), int(transformed_pos[1])
        if 0 <= xi < width and 0 <= yi < height:
            new_data[y, x, :] = data[yi, xi, :]

transformed_image = Image.fromarray(new_data)
plt.imshow(transformed_image)
plt.show()
```