Determining the Angle Between Two Vectors: Projection Approach

Patrick O'Neill
poneill8@gatech.edu

Abstract—When determining the angle between two vectors, the traditional method uses the dot product along with the magnitude of the two vectors. I propose a method that uses vector projection to determine the angle that two vectors form. This method performs roughly 28% faster than the traditional dot-product method when using CPython3 on modern computers. When dealing with pre-normalized vectors, the method performs 30.3% faster than the dot-product method when given one normalized vector, and 33.4% faster than the dot-product method when given two normalized vectors.

Further investigation reveals this is a quirk of the CPython compiler, and by using Numba (a python package) to compile the code before executing, we can eliminate any runtime advantage.

1 DOT PRODUCT METHOD

Given two n-dimensional vectors $a = (a_1, ..., a_n)$ and $b = (b_1, ..., b_n)$, we wish to calculate the angle formed between. The traditional method is to use the dot product using the following formula: $a \cdot b = |a|\,|b|\, cos(\theta)$.

1.1 Equation

$\theta = cos^{-1}\left(\frac{a \cdot b}{|a|\,|b|}\right)$ or $\theta = cos^{-1}\left(\frac{a}{|a|} \cdot \frac{b}{|b|}\right)$.

1.2 Implementation

Please see *Appendix 8.1: Dot Product Method Python Implementation (Normalize Version)* and *Appendix 8.2: Dot Product Method Python Implementation (Magnitude Version)* for implementations of both versions of the equation.

2 PROJECTION APPROACH

I propose a method that uses vector projection to find the proportionally correct adjacent side of a right triangle.

2.1 Method

First, we calculate the projection of vector b onto vector a.

$$b_{proj} = \frac{a \cdot b}{|a|^2} a = \frac{a \cdot b}{a \cdot a} a$$

The projected vector b was projected onto the subspace defined by a. Therefore the difference between the two vectors forms a right angle.

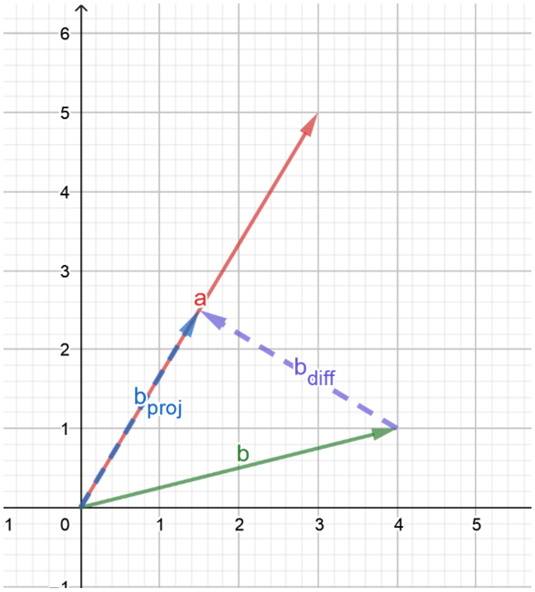

Figure 1—Visualization of projection process on a pair of 2d vectors.

Using this new side of a right angle (b_{proj}) in combination with the hypotenuse (b), we can determine the angle using cos^{-1}.

$$\theta = cos^{-1}\left(\frac{|b_{proj}|}{|b|}\right)$$

Finally, we need to determine if the two vectors form an angle that is greater than 90 degrees, as the angle returned will always be less than or equal to 90.

We use the dot product for this, as it will return a negative number if the two vectors form an angle greater than 90 degrees. If the angle is greater than 90, we must subtract the angle from 180 to get the true degree formed by the two vectors.

2.2 Implementation

Please see *Appendix 8.3: Projection Method Python Implementation*.

2.3 Performance Evaluation

Table 1—Runtime for each method to calculate the angle between 200,000 randomly generated vectors on an Intel® Core™ i7-8550U CPU.

Dimensions	**Dot Product - Norm Method Runtime (Sec)**	**Dot Product - Magnitude Method Runtime (Sec)**	**Projection Method Runtime (Sec)**
2	10.031	7.692	5.329
3	9.664	7.706	5.858
4	9.521	7.726	5.304
5	9.940	7.699	5.575
10	9.769	7.923	5.643
20	10.72	7.783	5.841
30	9.626	7.890	5.693
40	9.836	7.909	5.748
100	9.992	8.669	6.135

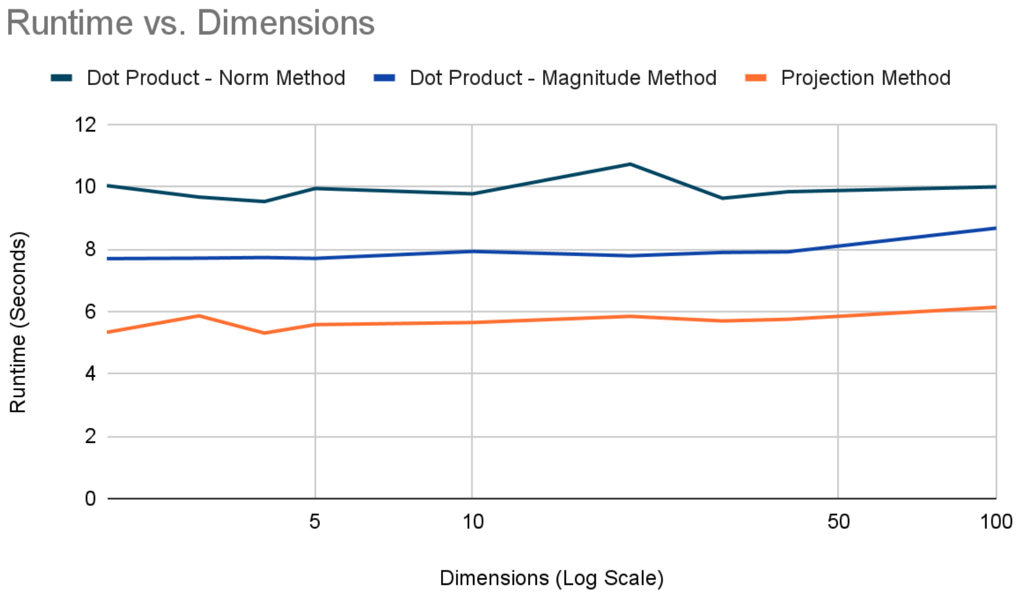

Figure 2—Runtime of each method in seconds compared to the number of dimensionality of the input.

Table 2—The average runtime for each method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Projection Method Runtime (Sec)
9.90	7.89	5.68

Table 3—The percent decrease from each method to the Projection Method.

Dot Product - Norm Method Runtime	Dot Product - Magnitude Method Runtime	Projection Method Runtime
42.63%	28.01%	0.0%

As seen in Table 3, the standard dot-product runtime improved by a minimum of 28.01%.

2.4 Performance Speculation

When comparing the two methods, it is helpful to break down the methods into their component operations relative to the dimensionality of the input vector.

Table 4—Runtime for each method to calculate the angle between 100,000 randomly generated vectors. n is the number of dimensions in the input vector

Operation	Dot-Norm Method	Dot-Mag Method	Projection Method
Multiplication	2n (normalize) + n (dot)	n (dot) + 2n (magnitude) + 1	3n (projection) + 2n (magnitude)
Division	2n (normalize)	1	2
Square Root	0	2	1
Arccos	1	1	1

When comparing between the dot-norm and projection method, the multiplication method is roughly 2x as frequent in the projection method, however the division method is O(1) as compared to dot's O(n) (relative to the dimensions of the vector). This is an advantageous trade off because division operations are roughly 3x-6x slower than multiplication operations (Boytsov, L., 2016).

However, the dot-mag method is harder to justify. The multiplications remain roughly ½ as frequent as the projection method, but the divisions are also less. It is not clear what the distinguishing factor is that makes the projection method faster than the dot-mag method, other than the need for two square root operations.

3 SPECIAL CASE ANALYSIS

We are interested in evaluating the runtime of each of these methods when we know the input is already normalized. For example, in many gaming engines, vectors are normalized to speed up frequent calculations.

I defined four new implementations, two for each method that could use the knowledge of a unit vector to avoid/improve computations.

3.1 Performance Evaluation

3.1.1 Case 1: One Normalized Vector

Table 5—Runtime for each method to calculate the angle between 200,000 randomly generated vectors on an Intel® Core™ i7-8550U CPU. Only one of the vectors given is normalized.

Dimensions	Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
2	9.65	7.31	7.77	5.58	5.13
3	9.61	7.33	8.70	5.73	5.44
4	10.08	7.79	7.80	5.74	5.03
5	9.74	7.42	7.68	5.90	5.10
10	9.52	7.46	7.81	5.68	5.12
20	9.61	8.11	7.87	5.95	5.16
30	9.62	7.55	8.06	5.52	5.21
40	10.23	7.65	8.20	5.55	5.40
100	7.75	7.75	8.59	5.83	6.09

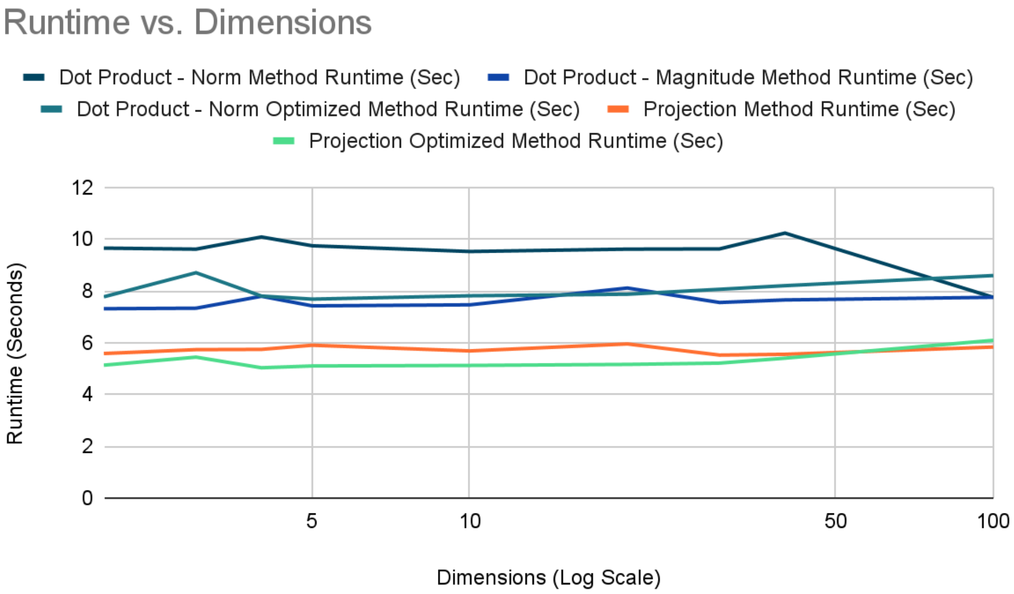

Figure 3—Runtime of each method in seconds compared to the number of dimensionality of the input.

As seen in Figure 3, the runtime of the projection method continues to outperform all other methods.

Table 6—The average runtime for each method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
9.53	7.60	8.05	5.72	5.30

Table 7—The percent decrease from each method to the Projection Optimized Method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
44.4%	30.3%	34.2%	7.3%	0.0%

As seen in Table 7, the standard dot-product runtime improved by a minimum of 30.3%. Additionally, the optimization decreased the projection method's runtime by 7.3%.

3.1.2 Case 2: Both Vectors Normalized

Table 8—Runtime for each method to calculate the angle between 200,000 randomly generated vectors on an Intel® Core™ i7-8550U CPU. Both of the vectors given are normalized.

Dimensions	Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
2	9.38	7.04	5.55	5.22	3.58
3	9.14	6.98	5.75	5.41	3.60
4	9.18	6.91	5.56	5.21	3.78
5	9.55	7.01	5.99	5.18	3.76
10	9.29	6.91	5.51	5.23	3.64
20	9.33	7.01	5.51	5.34	4.12
30	9.37	6.92	5.70	5.35	3.87
40	9.69	7.19	5.50	5.28	3.67
100	9.77	7.06	5.62	5.35	3.71

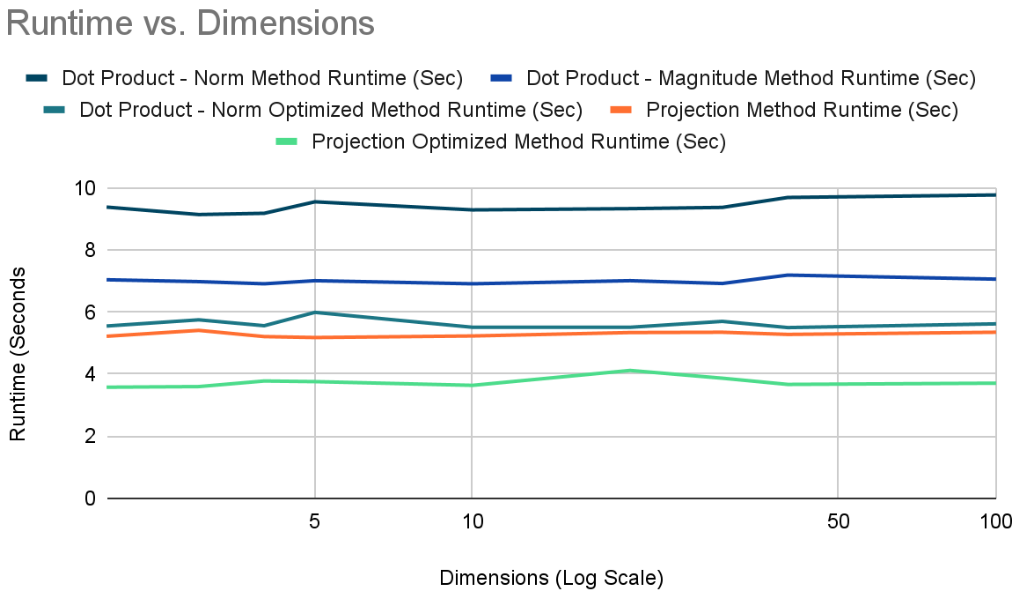

Figure 4—Runtime of each method in seconds compared to the number of dimensionality of the input.

As seen in Figure 4, the runtime of the projection method continues to outperform all other methods, even when both inputs are normalized.

Table 9—The average runtime for each method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
9.41	7.00	5.63	5.29	3.75

Table 10—The percent decrease from each method to the Projection Optimized Method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Dot Product - Norm Optimized Method Runtime (Sec)	Projection Method Runtime (Sec)	Projection Optimized Method Runtime (Sec)
60.1%	46.4%	33.4%	29.1%	0.0%

As seen in table 10, the standard dot-product runtime improved by a minimum of 33.4%. Additionally, the optimization decreased the projection method's runtime by 29.1%.

3.2 Implementation

Please see *Appendix 8.4: Optimized Dot-Norm Product Method Python Implementation (One Pre-Normalized)* and *Appendix 8.5: Optimized Dot-Norm Product Method Python Implementation (Both Pre-Normalized)* for implementations of the optimized dot-norm method.

Please see *Appendix 8.6: Optimized Projection Method Python Implementation (One Pre-Normalized)* and *Appendix 8.7: Optimized Projection Method Python Implementation (Both Pre-Normalized)* for implementations of the optimized projection method.

4 TESTING METHODOLOGY

4.1 Vector Generation

Vectors are generated using Python's random library. Vectors are generated to have random integer components with values between 100,000 and -100,000. Additionally, generation uses a seed value in order to guarantee that the same vectors are generated every testing session.

4.2 Implementation

For implementation details, please refer to *Appendix 8.8: Random Vector Pair Generation*.

5 STATIC COMPILATION

While the improvement is significant and useful when compiling Python using the CPython3 compiler, it is not clear why the improvement is so drastic.

For example, in Section 3.1.2, it is not clear why the optimized projection method (*Appendix 8.7*) is faster than the dot product (*Appendix 8.5*). The projection method uses all the operations used in the dot product method, but it also performs additional operations like calculating the projected vector and the magnitude of the projected vector squared. Therefore, this suggested something was wrong with my testing methodology, as these methods should perform worse comparatively.

This compilation quirk can be examined by running the code using Numba's static compilation to get highly performant code that does not use Python's interpreter.

> “Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. The most common way to use Numba is through its collection of decorators that can be applied to your functions to instruct Numba to compile them. When a call is made to a Numba-decorated function it is compiled to machine code “just-in-time” for execution and all or part of your code can subsequently run at native machine code speed!” (Anaconda, Inc., 2021)

5.1 Converting to Numba

Each method was compiled to static code using Numba’s @njit annotation (Reference link).

> “Numba reads the Python bytecode for a decorated function and combines this with information about the types of the input arguments to the function. It analyzes and optimizes your code, and finally uses the LLVM compiler library to generate a machine code version of your function, tailored to your CPU capabilities. This compiled version is then used every time your function is called.” (Anaconda, Inc., 2021)

Prior to testing, each method was called once to allow for compilation to occur without impacting runtime performance.

5.2 Runtime Evaluation

Numba offers a significant speedup in the runtime of all methods.

Table 11—Runtime for each method to calculate the angle between 200,000 randomly generated vectors on an Intel® Core™ i7-8550U CPU.

Dimensions	Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Projection Method Runtime (Sec)
2	0.342	0.142	0.209
3	0.331	0.142	0.212
4	0.352	0.142	0.217
5	0.335	0.143	0.218
10	0.341	0.145	0.219

20	0.340	0.150	0.243
30	0.461	0.164	0.270
40	0.377	0.160	0.266
100	0.443	0.182	0.323

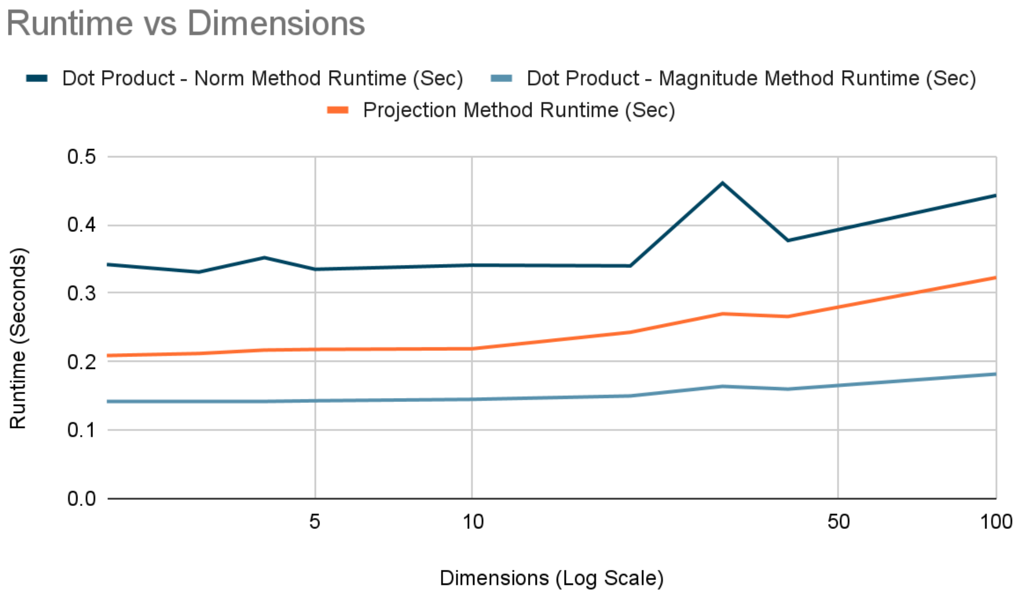

Figure 5—Runtime of each method in seconds compared to the number of dimensionality of the input.

As seen in Figure 5, while the projection method loses the speedup relative to the other methods, it still remains very competitive overall.

Table 12—The average runtime for each method.

Dot Product - Norm Method Runtime (Sec)	Dot Product - Magnitude Method Runtime (Sec)	Projection Method Runtime (Sec)
0.369	0.152	0.242

Table 13—The percent decrease for each method relative to the Dot Product - Magnitude Method.

Dot Product - Norm Method Runtime	Dot Product - Magnitude Method Runtime	Projection Method Runtime
58.81%	0.0%	37.19%

As seen in table 12, the best method is Dot product - magnitude, followed by Projection method, and Dot product - norm method.

6 CONCLUSION

Overall, the projection method reveals a quirk in CPython. While the projection approach is relatively performant, it is outshined by the standard dot product - magnitude method.

7 REFERENCES

1. Wolever, D. (2016, March 29). *Answer to 'Angles between two n-dimensional vectors in Python'*. Stack Overflow. Retrieved October 1, 2021, from https://stackoverflow.com/revisions/13849249/6.
2. Boytsov, L. (2013, November 18). *Is division slower than multiplication?* Srchvrs's Blog Retrieved October 24, 2021, from http://searchivarius.org/blog/division-slower-multiplication.
3. Anaconda, Inc. (2021, October 7). *Numba documentation*. Numba documentation. Retrieved October 27, 2021, from https://numba.readthedocs.io/en/stable/index.html.

8 APPENDIX

8.1 Dot Product Method Python Implementation (Normalize Version)

Adapted from David Wolever's code on StackOverflow (Wolever, 2016).

```python
def normalize(v: np.array):
    return v / np.linalg.norm(v)

def dot_product_norm(v1: np.array, v2: np.array) -> float:
    """
    Calculate the angle two vectors form, using the dot product.

    A*B = |A|*|B|*cos(theta)
    Using unit vectors:
    A*B = 1*1*cos(theta)
    theta = cos^-1(A*B)

    :param v1: Vector 1
    :param v2: Vector 2
    :return: The angle formed by the two vectors
    """
    v1_u = normalize(v1)
    v2_u = normalize(v2)
    rad = np.arccos(np.dot(v1_u, v2_u))
    return np.rad2deg(rad)
```

8.2 Dot Product Method Python Implementation (Magnitude Version)

```python
def dot_product_mag(v1: np.array, v2: np.array) -> float:
    """
    Calculate the angle two vectors form, using the dot product.

    A*B = |A|*|B|*cos(theta)
    Using magnitude:
    theta = cos^-1((A*B)/(|A||B|))

    :param v1: Vector 1
    :param v2: Vector 2
    :return: The angle formed by the two vectors
    """
    dot = np.dot(v1, v2)
    v1_mag = np.sqrt(np.dot(v1, v1))
    v2_mag = np.sqrt(np.dot(v2, v2))
    rad = np.arccos(dot/(v1_mag * v2_mag))
    return np.rad2deg(rad)
```

8.3 Projection Method Python Implementation

```python
def projection(v1: np.array, v2: np.array) -> float:
    """
    Calculate the angle two vectors form, using projection.

    1. project v2 onto v1
           ____
          |   /
      v2  |  / v2
    (proj)| /
          |/
    2. calculate the magnitudes squared of proj and v2
    3. get the angle of the two vertices by calculating cos^-1
of sqrt((mag proj^2)/(mag v2^2))
    4. calculate the dot product of v1 and v2. This will tell us
if the angle is less than or greater than 90.

    :param v1: Vector 1
    :param v2: Vector 2
    :return: The angle formed by the two vectors
    """
    # 1 - a*b / (b_mag) * b
    dot = np.dot(v1, v2)
    div = np.dot(v1, v1)
    scalar = dot / div
    projected = scalar * v1
    # 2
    hor_mag_sq = np.sum(projected ** 2)
    hyp_mag_sq = np.sum(v2 ** 2)
    # 3
    # arccos of proj over v2
    rad = np.arccos(np.sqrt(hor_mag_sq / hyp_mag_sq))
    deg = np.rad2deg(rad)
    # 4
    if dot < 0:
        deg = 180 - deg

    return deg
```

8.4 Optimized Dot-Norm Product Method Python Implementation (One Pre-Normalized)

```python
def dot_product_1_unit(v1_u: np.array, v2: np.array) -> float:
    v2_u = normalize(v2)
    rad = np.arccos(np.dot(v1_u, v2_u))
    return np.rad2deg(rad)
```

8.5 Optimized Dot-Norm Product Method Python Implementation (Both Pre-Normalized)

```python
def dot_product_2_unit(v1_u: np.array, v2_u: np.array) -> float:
    rad = np.arccos(np.dot(v1_u, v2_u))
    return np.rad2deg(rad)
```

8.6 Optimized Projection Method Python Implementation (One Pre-Normalized)

```python
def projection_1_unit(v1_u: np.array, v2: np.array) -> float:
    dot = np.dot(v1_u, v2)
    scalar = dot
    projected = scalar * v1_u

    hor_mag_sq = np.sum(projected ** 2)
    hyp_mag_sq = np.sum(v2 ** 2)

    rad = np.arccos(np.sqrt(hor_mag_sq / hyp_mag_sq))
    deg = np.rad2deg(rad)

    if dot < 0:
        deg = 180 - deg

    return deg
```

8.7 Optimized Projection Method Python Implementation (Both Pre-Normalized)

```python
def projection_2_unit(v1_u: np.array, v2_u: np.array) -> float:
    dot = np.dot(v1_u, v2_u)
    scalar = dot
    projected = scalar * v1_u

    hor_mag_sq = np.sum(projected ** 2)

    rad = np.arccos(np.sqrt(hor_mag_sq))
    deg = np.rad2deg(rad)

    if dot < 0:
        deg = 180 - deg

    return deg
```

8.8 Random Vector Pair Generation Python Implementation

```
def normalize(v: np.array):
   return v / np.linalg.norm(v)


def random_vector_pair(min_dim=1, normalize_elem=(False, False),
                       max_dim=100, seed=random.random()):
   random.seed(seed)
   seed2 = random.randint(-1000, -1)

   dim = random.randint(min_dim, max_dim)

   v1 = random_vector(dimension=dim, normalize_elem=normalize_elem[0],
seed=seed)
   v2 = random_vector(dimension=dim, normalize_elem=normalize_elem[1],
seed=seed2)


def random_vector(dimension=2, normalize_elem=False, max_pos=100000,
                  max_neg=-100000, seed=random.random()):
   random.seed(seed)
   vector = []
   for i in range(dimension):
       vector.append(random.randrange(max_neg, max_pos))

   vector = np.array(vector, dtype="int64")

   if normalize_elem:
       vector = normalize(vector)

   return vector
```