Euclidean cross product in dimension $n \geq 3$ via the Claude cross product operator as applied to the matrix determinant

S. Halayka*

Monday 3rd February, 2025 15:39

Abstract

This paper contains a short introduction to the cross product in dimension $n \geq 3$. The main focus is on some C++ code.

1 Application: the matrix determinant

In this paper, we focus on a cross product operation in dimension $n \geq 3$.

The main goal is to acquaint the coder with the basic idea behind the Claude operator in n-D in the case that accepts $(n-1)$ n-vectors as input. For instance, where $n = 3$, the 3-D cross product accepts $(n-1) =$ two 3-vectors as input. Using C++ templates, the abstraction to any $n \geq 3$ is provided. This cross product is used to calculate the matrix determinant.

2 Code

Here we include the Eigen linear algebra library, as well as various parts of the standard library:

```
#include <Eigen/Dense>
using namespace Eigen;

#include <iostream>
#include <vector>
#include <numeric>
#include <string>
#include <sstream>
#include <algorithm>
#include <array>
using namespace std;
```

Here we define the vector class, where the data type is T (e.g., double), and N is the dimension:

```
template<class T, size_t N>
class Vector_nD
{
public:
  array<T, N> components;
```

*sjhalayka@gmail.com

```
// Helper function to get the sign of permutation
static signed char permutation_sign(const array<int, (N - 1)>& perm)
{
  bool sign = true;

  for (int i = 0; i < (N - 2); i++)
    for (int j = i + 1; j < (N - 1); j++)
      if (perm[i] > perm[j])
        sign = !sign;

  if (sign)
    return 1;
  else
    return -1;
}

Vector_nD(const array<T, N>& comps) : components(comps)
{
}

Vector_nD(void)
{
  components.fill(0.0);
}

T operator[](size_t index) const
{
  return components[index];
}
```

Here we make a static cross product function that takes in $(n-1)$ n-vectors. This function returns one n-vector:

```
// Claude cross product operator
static Vector_nD cross_product(const vector<Vector_nD<T, N>>& vectors)
{
  if (vectors.size() != (N - 1))
  {
    cout << "nD cross product requires (n - 1) input vectors" << endl;
    return Vector_nD<T, N>();
  }

  array<T, N> result;

  for (size_t i = 0; i < N; i++)
    result[i] = 0.0;

  // These are the indices we'll use for each component calculation
  array<int, (N - 1)> base_indices;

  for (int i = 0; i < (N - 1); i++)
    base_indices[i] = i;

  // Skip k in our calculations -
  // this is equivalent to removing the k-th column
```

```
// For each permutation of the remaining (N - 1) indices
for (int k = 0; k < N; k++)
{
  do
  {
    // Calculate sign of this term
    const signed char sign = permutation_sign(base_indices);

    // Calculate the product for this permutation
    T product = 1.0;
    ostringstream product_oss;

    for (int i = 0; i < (N - 1); i++)
    {
      const int col = base_indices[i];

      // Adjust column index if it's past k
      int actual_col = 0;

      if (col < k)
        actual_col = col;
      else
        actual_col = col + 1;

      product_oss << "v_{" << i << actual_col << "} ";

      product *= vectors[i][actual_col];
    }

    if (sign == 1)
      cout << "x_{" << k << "} += " << product_oss.str() << endl;
    else
      cout << "x_{" << k << "} -= " << product_oss.str() << endl;

    result[k] += sign * product;

  } while(next_permutation(
      base_indices.begin(),
      base_indices.end()));
}

// Flip handedness
for (size_t k = 0; k < N; k++)
  if (k % 2 == 1)
    result[k] = -result[k];

cout << endl;

for (int k = 0; k < N; k++)
  cout << "result[" << k << "] = " << result[k] << endl;

cout << endl;

if (N == 3)
{
  // Demonstrate the traditional cross product too
```

```
      double x = vectors[0][0];
      double y = vectors[0][1];
      double z = vectors[0][2];

      double rhs_x = vectors[1][0];
      double rhs_y = vectors[1][1];
      double rhs_z = vectors[1][2];

      double cross_x = y * rhs_z - rhs_y * z;
      double cross_y = z * rhs_x - rhs_z * x;
      double cross_z = x * rhs_y - rhs_x * y;

      cout << cross_x << " " << cross_y << " " << cross_z << endl << endl;
    }

    return Vector_nD(result);
  }
```

Here we have the static dot product function:

```
  static T dot_product(const Vector_nD<T, N>& a, const Vector_nD<T, N>& b)
  {
    return inner_product(
      a.components.begin(),
      a.components.end(),
      b.components.begin(), 0.0);
  }
};
```

Finally, we calculate the determinant of a square matrix using the cross and dot products as defined above:

```
template <class T, typename size_t N>
T determinant_nxn(const MatrixX<T>& m)
{
  if (m.cols() != m.rows())
  {
    cout << "Matrix must be square" << endl;
    return 0;
  }

  // We will use this N-vector later, in the dot product operation
  Vector_nD<T, N> a_vector;

  for (size_t i = 0; i < N; i++)
    a_vector.components[i] = m(0, i);

  // We will use these (N - 1) N-vectors later,
  // in the cross product operation
  vector<Vector_nD<T, N>> input_vectors;

  for (size_t i = 1; i < N; i++)
  {
    Vector_nD<T, N> b_vector;

    for (size_t j = 0; j < N; j++)
```

```
      b_vector.components[j] = m(i, j);

    input_vectors.push_back(b_vector);
  }

  // Compute the cross product using (N - 1) N-vectors
  Vector_nD<T, N> result = Vector_nD<T, N>::cross_product(input_vectors);

  // Compute the dot product
  T det = Vector_nD<T, N>::dot_product(a_vector, result);

  // These numbers should match
  cout << "Determinant:       " << det << endl;
  cout << "Eigen Determinant: " << m.determinant() << endl << endl;

  return det;
}
```

This main function is for testing the above code:

```
int main(int argc, char** argv)
{
  srand(static_cast<unsigned int>(time(0)));

  const size_t N = 4; // Anything larger than 12 takes eons to solve for

  MatrixX<double> m(N, N);

  for (size_t i = 0; i < N; i++)
  {
    for (size_t j = 0; j < N; j++)
    {
      m(i, j) = rand() / static_cast<double>(RAND_MAX);

      if (rand() % 2 == 0)
        m(i, j) = -m(i, j);
    }
  }

  determinant_nxn<double, N>(m);

  return 0;
}
```

3 Examples of the Claude cross product operator

For $n = 3$:

$$x_0 = v_{01}v_{12} - v_{02}v_{11}, \tag{1}$$
$$x_1 = v_{00}v_{12} - v_{02}v_{10}, \tag{2}$$
$$x_2 = v_{00}v_{11} - v_{01}v_{10}. \tag{3}$$

For $n = 4$:

$$\begin{aligned}
x_0 &= v_{01}v_{12}v_{23} - v_{01}v_{13}v_{22} - v_{02}v_{11}v_{23} + v_{02}v_{13}v_{21} + v_{03}v_{11}v_{22} - v_{03}v_{12}v_{21}, && (4)\\
x_1 &= v_{00}v_{12}v_{23} - v_{00}v_{13}v_{22} - v_{02}v_{10}v_{23} + v_{02}v_{13}v_{20} + v_{03}v_{10}v_{22} - v_{03}v_{12}v_{20}, && (5)\\
x_2 &= v_{00}v_{11}v_{23} - v_{00}v_{13}v_{21} - v_{01}v_{10}v_{23} + v_{01}v_{13}v_{20} + v_{03}v_{10}v_{21} - v_{03}v_{11}v_{20}, && (6)\\
x_3 &= v_{00}v_{11}v_{22} - v_{00}v_{12}v_{21} - v_{01}v_{10}v_{22} + v_{01}v_{12}v_{20} + v_{02}v_{10}v_{21} - v_{02}v_{11}v_{20}. && (7)
\end{aligned}$$