1. Математические аспекты

Постановка задания

Даны два отрезка $s_{1,}s_2$ в $\mathbb{R}^3$.

Найти минимальное расстояние между точками двух отрезков:

$d = \inf\left\|\vec{\mathbf{r}}_1 - \vec{\mathbf{r}}_2\right\|, \vec{\mathbf{r}}_1 \in s_1, \vec{\mathbf{r}}_2 \in s_2$

Обозначения

Длину отрезка s обозначаем $|s|$. Радиус-векторы концов отрезка s_n обозначаем $\left[\vec{\mathbf{p}}_n^1, \vec{\mathbf{p}}_n^2\right] = \left[\left(x_n^1, y_n^1, z_n^1\right), \left(x_n^2, y_n^2, z_n^2\right)\right]$. Параллельный перенос отрезка на вектор $\vec{\mathbf{r}}$ обозначаем $s + \vec{\mathbf{r}}$. В математических формулах для удобства будем допускать операторы из C++, например $s -= \vec{\mathbf{r}}$ будет означать параллельный перенос вектора s с изменением его значения для последующих действий. Матричное умножение для отрезка $s = \mathbf{M}s = \left[\mathbf{Mr}^1, \mathbf{Mr}^2\right]$ означает умножение матрицы на радиус-векторы его концов.

Коллинеарными называем только отрезки, лежащие на одной прямой.

Прочие называем параллельными.

Алгоритм

1. Проверяем вырожденный случай. Если $|s_1| = |s_2| = 0$, результат равен $\left\|\vec{\mathbf{p}}_1^1 - \vec{\mathbf{p}}_2^1\right\|$ (расстояние между началом одного и второго отрезка). Следующие шаги игнорируются;

2. Ненулевые отрезки. Считаем, что $|s_1| \geq |s_2|$. При необходимости меняем отрезки местами ($\mathrm{std::swap}\left(s_1, s_2\right)$).

3. Смещаем оба вектора так, чтобы в результате было $\vec{\mathbf{p}}_1^1 = 0$:

$s_2 -= \vec{\mathbf{p}}_1^1;$
$s_1 -= \vec{\mathbf{p}}_1^1.$

4. Вычисляем матрицу поворота $\mathbf{M}$ такую, что выполняются два условия:

$$\mathbf{M}\vec{\mathbf{p}}_1^2 = \left(\left\|\vec{\mathbf{p}}_1^2\right\|, 0, 0\right) \quad (1)$$

$$\mathbf{M}\left(\vec{\mathbf{p}}_2^1 - \vec{\mathbf{p}}_2^2\right) = \left(x, y, 0\right) \quad (2)$$

где x,y – соответствующие компоненты вектора s_2 после поворота.

а) Если $s_{1,}s_2$ коллинеарны (лежат на одной прямой), вычисляем $\mathbf{M}$ как матрицу поворота s_1 к оси *x*.

б) Если $s_{1,}s_2$ параллельны, но лежат на разных прямых, находим вектор, ортогональный к общей плоскости, содержащей s_1, s_2, $\vec{\mathbf{v}}$ как векторное произведение:

$$\vec{\mathbf{v}} = \vec{\mathbf{p}}_1^2 \times \vec{\mathbf{p}}_2^2$$

в) Иначе находим вектор, $\vec{\mathbf{v}}$ ортогональный векторам $\vec{\mathbf{p}}_1^1, \left(\vec{\mathbf{p}}_2^2 - \vec{\mathbf{p}}_2^1\right)$:

$$\vec{\mathbf{v}} = \vec{\mathbf{p}}_1^1 \times \left(\vec{\mathbf{p}}_2^2 - \vec{\mathbf{p}}_2^1\right)$$

В случаях *б* и *в*ычисляем матрицу $\mathbf{M}$ по условию:

$$\mathbf{M}\vec{\mathbf{v}} = \left(\left\|\vec{\mathbf{v}}\right\|, 0, 0\right)$$

5. Вычисляем $s_1 = \mathbf{M} \times s_1, s_2 = \mathbf{M} \times s_2$

Матрица поворачивает пару отрезков так, что s_1 лежит на оси *x*, а s_2 параллелен плоскости *xy*.

6. После этого решение сводится к нахождению расстояния d_{xy} между отрезками $\left[x_1^1, x_1^2\right]$ и $\left[x_2^1, x_2^2\right]$ в плоскости *xy*. Это тривиальная задача. Решение «в лоб» (неоптимальное) сводится к определению *а*) пересекаются ли отрезки (d_{xy} =0); *б)* если нет, считаются 4 расстояния от каждого из 4 концов до другого отрезка, выбирается минимальное.

7. Искомое решение $d = \sqrt{d_{xy}^2 + z_2^2}$

2. Общие замечания по реализации алгоритма

1. В постановке задания было предложено создать классы Point3D, Vector3D, Section3D. На мой взгляд, класс Vector3D избыточен. Если рассматривать его как вектор с началом в нуле, то он становится тождествен классу Point3D. Если же считать его произвольным вектором в пространстве, то он неотличим от класса, описывающего отрезок. Можно было бы акцентировать разницу «отрезок» и «вектор – направленный отрезок». Но в программной реализации отрезок может быть только направленным, поскольку при машинном представлении его какой-то конец

будет «первым», а какой-то «вторым». Поэтому из трех предложенных классов реализованы только два: Point3D, Section3D.

2. Поскольку алгоритм использует повороты в пространстве, к заявленным добавлен класс Matrix3D, реализующий структуру и основные операции матричной алгебры.

3. Имеющаяся реализация алгоритма неоптимальна с точки зрения производительности. Это принципиальная особенность используемого метода разработки: сначала создать и протестировать функциональность, не думая о времени счета. И потом, как отдельную задачу, решать вопросы быстродействия. Эту вторую задачу я даже не пытался рассматривать.

3. Программные аспекты

1. template<class T, class CT, size_t N> class FieldObject

Этот класс будет общим базовым для всех последующих (Point3D, Section3D, Matrix3D).

Класс описывает объект, принадлежащий к некоторому полю (https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_(%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)), т.е., элемент линейного векторного пространства.

Аргументы шаблона:

T – тип числа-компоненты вектора. В последующих вычислениях используется T=double. При вызове конструктора от initializer_vector в тестах может промежуточно возникать объект с T=int;

CT (child_type) – тип наследника. Благодаря этому аргументу объект всегда «знает», в контексте какого унаследованного класс он сейчас используется;

N – размерность.

Определены действия линейной алгебры:

- сложение и вычитание с себе подобным (шаблонные);
- умножение на число;
- скалярное произведение с подобным классом;
- лебегова мера L^2;
- мера L^1, которая используется в операторах сравнения как более быстрая (обходится без *sqrt*);
- операторы доступа at() и operator[];
- конструкторы.

Данные хранятся в массиве T[N]. Другие данные как в самом классе, так и в его наследниках не допускаются. Попытки сделать подобное пресекаются

static_assert(sizeof(self)==sizeof(child_type).

Вне класса определен оператор вывода в ostream, используемый при отладке и анализе результатов.

2. template<class T> Point3D : public FieldObject<T,Point3D<T>,3>

Массив из 3 T. В дополнение к унаследованным свойствам определены функции поименного доступа к компонентам x(), y(), z(), которые работают через вызов at(0), at(1), at(2).

3. template<class T> Segment3D : public FieldObject<T, Segment3D <T>,2>

Массив из двух Point3D<T>. В дополнение к унаследованным свойствам определены функции поименного доступа к компонентам p1(), p2(), которые работают через вызов at(0), at(1).

Оператор сложения с себе подобным запрещен (убран в private).

Добавлен оператор сложения с Point3D<T> (параллельный перенос отрезка).

4. template<class T> Matrix3D : public FieldObject<T, Matrix3D <T>,3>

Массив из двух Point3D<T>. В дополнение к унаследованным свойствам определены функции поименного доступа к компонентам row(size_t), которые работают через вызов at().

Встроенная функция умножения слева на вектор-столбец Point3<T2>.

Встроенная функция умножения на другую матрицу.

Матричное умножение оформлено не в виде operator*, а в виде именованных функций во избежание путаницы с обычным умножением.

Определитель не писал, т.к. для текущей задачи он был не нужен. В общем случае надо бы!

5. Внешние функции

Умножение матрицы на Section3D (поворот системы координат для отрезка);

Вычисление матрицы поворота. По заданному вектору вычисляет матрицу поворота его к одной из осей *x,y,z*. Поскольку такой поворот определяется неоднозначно, используется последовательность: «Поворот в *xy* с обнулением *z*», «Поворот в *xz* с обнулением *y*», «поворот от *z* к *y* вокруг *x*»; «поворот от *z* к *x* вокруг *y*».

Функция double ComputeDistance(const s3_F64 &s1, const s3_F64 &s2), вычисляющая требуемое расстояние. Функция не оптимизирована по производительности, некоторые операции в ней заведомо избыточны. Это только «концепт»: в случае реального использования такие функции проходят тщательную доработку, но это выходит за рамки текущего теста.

Инструменты учета погрешностей: функции almost_zero(), almost_same(), o_small(). В результате прямых и обратных вычислений результат не обязательно совпадает из-за погрешности вычислений. В результате, например, выражение (log(exp(x))==x) может вернуть false. Во избежание подобных ситуаций сделаны функции, которые определяют равенство выражений с точностью до машинной погрешности almost_same(log(exp(x)), x) гарантированно вернет true.

6. Вспомогательные тестовые функции

Генератор случайных пар отрезков с наперед заданным расстоянием между ними.

Генерируются следующие частные случаи:

1. Пара коллинеарных отрезков (лежат на одной прямой, могут частично совпадать).

2. Пара параллельных отрезков в пространстве;

3. Пара отрезков, лежащих на скрещивающихся (skew) прямых. Ближайшие точки пары лежат внутри обоих отрезков. В качестве параметра передается кратчайшее расстояние между прямыми. При нулевом параметре отрезки пересекаются.

4. Пара отрезков, лежащих на скрещивающихся (skew) прямых. Ближайшая точки одного отрезков находится на его конце.

Для всех видов пар в качестве параметра задается расстояние, которое должно быть между отрезками. Тестирование состоит в сравнении предзаданного и измеренного расстояний.

Длины отрезков представляют собой случайные числа от 1 до 2. Отрезки генерируются в плоскостях, параллельных плоскости *xy,* после чего к ним применяется поворот в пространстве на случайный угол.

7. Тестирование

Запускается цикл, в котором генерируются пары всеми четырьмя заданными способами. Предзаданное расстояние меняется в некотором диапазоне с определенным шагом. Ошибкой считается ситуация, когда almost_same(preset_distance, computed_distance)==false. Ошибки регистрируются.

Проделано несколько прогонов по 400 000 тестов. Ошибок нет.