Forward Kinematics

I. Introduction/Motivation

正向運動學(**Forward Kinematics**)，將空間中初始位置進行特定運算得出末位置，線性代數(**Linear Algebra**)的觀點即是

線性轉換(**Linear Transformation**)

$$L : R^n \rightarrow R^m \ n, m \in N$$

從線性代數(**Linear Algebra**)得證，任一線性轉換，都可以用矩陣表示，直觀上，線性轉換的問題都用矩陣處理。現考慮線性轉換為

$$L : R^3 \rightarrow R^3 \ , \mathrm{v} \in R^3$$

也就是三維空間的轉換過程，包含平移(**Translation**)與旋轉(**Rotation**)。對於 **v** 表示方式有很多種，包含 **Cartesian form , Euler Angle, Axis Angle, Quaternion**，四元數即是本次作業的重點

前部分從 **ASF** 得到的 **skeleton** 各個節點(**joint**)關係，**AMC** 得到各個 **frame** 的改變量(**delta translation and delta rotation**)，利用不斷利用利用矩陣或四元數進行跌代更新，

即 $_{i}v = \ skeleton_{transform} * \ _{i}\Delta frame_{transform} * {}_{i-1}v$

後部分則是插值問題(**interpolation**)，考慮使用者能在給定 **motion sequence** 播放的影片，將某時間點的 **frame** 進行拉動到其他時間的 **frame**，則整個 **motion sequence** 也會隨之改變，必須運用插值方式近似出最適合的 **motion sequence**。插值問題在動畫呈現上非常重要，卻也非常困難，過於高頻或低頻的訊號都容易造成動畫不流暢。不適當的插值方式都容易發生動作不連續，動作抖動的情形。

II. Fundamental

- **Forward Kinematics**

Quaternion

$$\mathrm{q} = \ q_0 + q_1 i + q_2 j + q_3 k$$

假設 **Uint axis-angle: n**

$$\mathrm{n} = \ \theta * [n_x, n_y, n_z]^T$$

則對應四元數為

$$\mathrm{q} = [\cos\frac{\theta}{2}, n_x \sin\frac{\theta}{2}, n_y \sin\frac{\theta}{2}, n_z \sin\frac{\theta}{2}]^T$$

- 四則運算

$$q_a \pm q_b = [s_a \pm s_b, v_a \pm v_b]$$

$$\begin{aligned} q_a q_b = & \ s_a s_b - x_a x_b - y_a y_b - z_a z_b \\ & +(s_a x_b - x_a s_b - y_a z_b - z_a y_b) i \\ & +(s_a y_b - x_a z_b - y_a s_b - z_a x_b) j \\ & +(s_a z_b - x_a y_b - y_a x_b - z_a s_b) k \end{aligned}$$

- 四元數旋轉

假設一個空間三維點$\mathbf{p} = [\mathbf{x}, \mathbf{y}, \mathbf{z}] \in \boldsymbol{R}^3$

利用一虛四元數描述

$$\mathbf{p} = [\mathbf{0}, \mathbf{x}, \mathbf{y}, \mathbf{z}] = [\mathbf{0}, \mathbf{v}]$$

並利用 q 代表旋轉

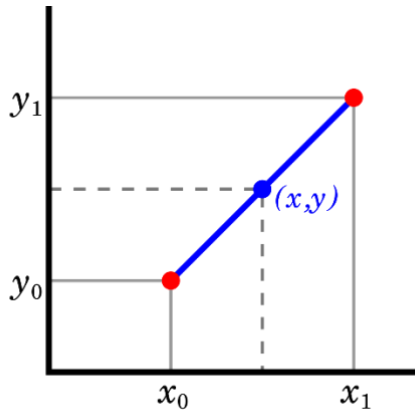
$$\mathbf{q} = [\cos\frac{\theta}{2}, \boldsymbol{n}\sin\frac{\theta}{2}]$$

則旋轉後的$\mathbf{p}'$點

$$\mathbf{p}' = \mathrm{Im}(\mathbf{qp}q^{-1})$$

- **Interpolation**

本次作業採用 **Linear Interpolation**



III. Implementation

- **Forward Kinematics**

Recursive

$$_{i}T = {}^{i-1}_{0}R_{i-1}V + {}_{i-1}T$$

Pseudo Code:

For (int bone_idx = 0; bone_idx < skeleton_bone_size(); bone_idx++) {

dof = get_amc_data();

$R_{local} = R_{local_{transform}}(\mathrm{idx}) * R_{dof}$;

$R_{skeleton} = {}_{global}R(idx - 1)$;

${}_{global}R = R_{skeleton} * R_{local}$;

}

- **Time warping**

Linear Interpolation

考慮**Quaternion motion sequence** $\mathbf{S} = \{q_n\}$

$$q_{n_*} = q_{n1} + (n_* - n_1) * \frac{1}{(n_2 - n_1)} * (q_{n2} - q_{n1}), n_* \in (\mathbf{n1}, \mathbf{n2})$$

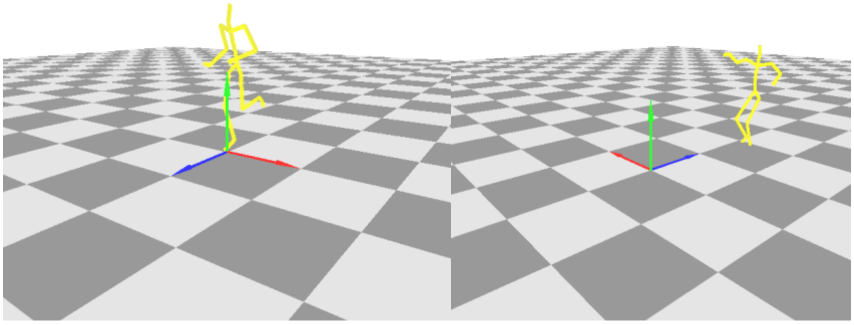
Pseudo Code:

```
For (int frame = 0; frame < total_frame; frame++) {
    For (int bone_idx = 0; bone_idx < skeleton_bone_size(); bone_idx++) {
        change_frame = total_frame/play_second; // default = 150
        time_step =    hard_constraint / change_frame;

        // offset, for convenience
        int _frame;
        if(frame > change_frame) _frame = frame – change_frame;
        else _frame = frame;

        double n1_frame = (_frame) * time_step;
        double n2_frame = (_frame+1) * time_step;
        int number_of_interpolation = int(n2_frame) – int(n1_frame);
        /*smaller hardconstraint (140)
           frame <= change_frame : sequence will become smaller
           frame > change_frame : sequence will become bigger

           bigger hardconstraint (160)
           reverse order of (140 case)
        */
        防呆機制();
        /*
        1.  /when number_of_interpolation == 0
        2.  if 4 coefficient multiply are all negative
              coef(quaternion(n1)) * coef(quaternion(n2)) < 0
            change sign of one term
            p.s q = -q , if q is quaternion, then it represent the same direction in
            3d space
        For (int interpolation = 0; interpolation < number_of_interpolation;
        interpolation++) {
            int n* = n*_frame_compute();
            /*
              Case 1: smaller interpolation:
                    n* = int(x1_frame);
              Case 2: bigger interpolation:
                    n* = std::ceil(x0) + number_of_interpolation;
              */
```

```
            q1 = qn1;
            q2 = qn2;
            qn* = qn1 + (n*-n1)*(1.0/n2-n1)(qn2-qn1);

        if (bone_idx == 0) //root
            same idea to deal with translation
        }
    }
}
```

IV. Result & Discussion

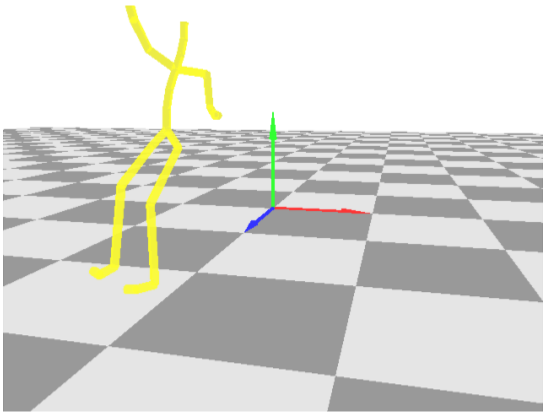
- **Forward Kinematics**



實作成果跟 **helper_function** 一致，沒有明顯擾動或震動情形發生

- **Disscussion**

  1. **Root pose**

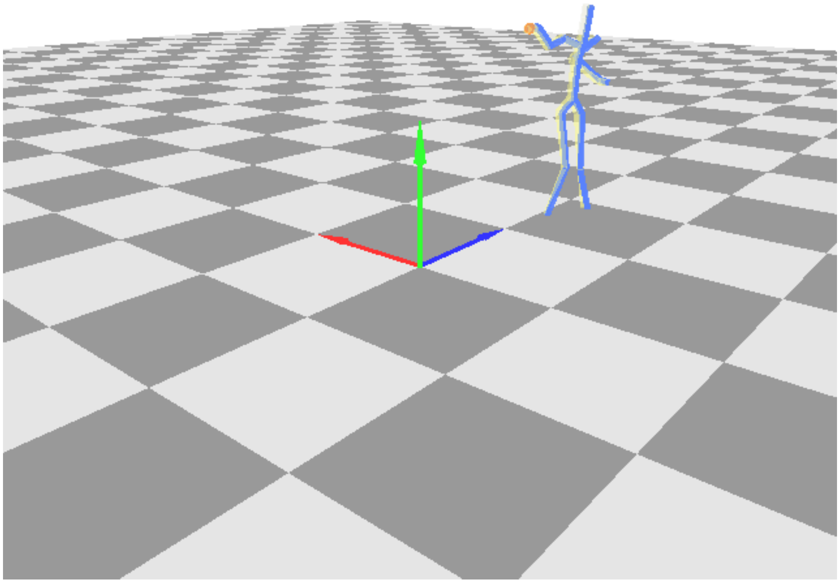


Root 雖然只是一個點，沒有 **direction_vector**
但因為 **AMC motion_sequence** 中 **root** 會有 **delta pose**，**root** 那點會有朝向，等於長出去的 **tree**，**skeleton tree** 的朝向會不同
原本 **punch_kick** 朝向應面對原點
但發生完全轉向的狀況

  2. 實作過程在計算 **local_transform** 跟 **global_transform** 都是使用矩

陣跌代，最後整理成 **composite form** 的矩陣後，再利用 **math::ComputeQuaternionXyz** 計算對應四元數。原本想完全用四元數取代矩陣運算，但四元數乘法在某些情況下會近似 **0** 或趨近無窮大，此問題到此都還無法知悉原因

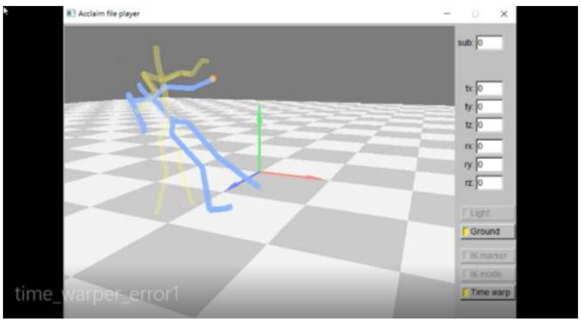
- **Time Warping**



實作成果跟助教影片幾乎一致，只有在快要結束整個 **450 frame** 時候(已經接完球後很久)，有唯一一個擾動，問題到此都還無法知悉原因，直接 **print** 出 **interpolate** 後的 **sequence** 前後值的變化上也沒有特別異常(多數情況若有動作震動，**sequence** 值都有很明顯變化，只是唯獨這個沒有明顯變化，呈現卻有震動)

- **Disscussion**

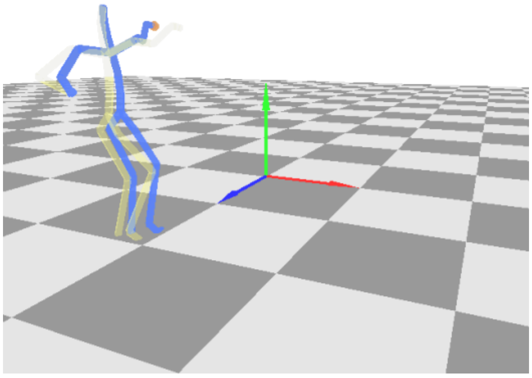
  1. **Overlap two frame**



假設 **frame** 間時間間距變大，當 **n2-n1** 越過 **2** 個整數值，會需要做兩次插值。這個問題原本我沒有想到，因為過去所遇過的插值問題都是在 **2** 點中插入 **1** 點，並以此點取代該 **2** 點。前面提到過動畫上，動畫不連續性是很容易被觀察出來的，原本當 **n2-n1** 越過兩個整數值，指

插入 1 點，新的 motion sequence 就會少 1 點，造成動作不連續，skeleton 會整個跑掉(新的 motion sequence 初始值是直接 assign *original_motion_sequence_)。

2. **q = -q**



四元數的正負號在三維空間表示上是一致，但由於 motion sequence 使用線性內插，正負號會直接影響最後結果。

e.g q1=q1; q2=-q1;

空間上 q1 跟 q2 是指向同方向，但若進行加權則可能會讓四元數 x,y,z,w 值更小或等於 0，導致空間指向完全錯誤

如圖，照理說接完球會，手臂收回不會有太多的左右分量，但圖中接完球後卻往直接左偏

3. ***original_motion_sequence 的覆寫**

   由於剛開始實作的時候，插值後的結果都直接放入 *original_motion_sequence，但由於 for loop 是針對原本初始未調整 frame 進行跌代，很多時候正在插值的新的 frame，跟正在跌代的原有 frame_idx 是不同，可能是小於或大於，若是大於，很可能會 frame_idx=135，但 n*=150，插值後就會修改到原本 150 時刻接球的動作，因此後來多設一變數 new_sequence 來儲存插值後的 sequence。

V. Conclusion

本次作業最有心得的是 Time Warper，跟整份程式的資料結構。Time Warper 部分由於是直接處理一連串時間點的 motion，很像在寫濾波程式，由於動畫不

連續性太容易看得出來，所以只要插值有問題的 **frame** 都會很明顯看出，但是除錯真的很困難，因為這是第一次處理四元數，像 **q=-q** 那個 **bug** 我真的思考非常久，後來 **print** 出值，才發現有問題。而關於資料結構的部分真的非常花時間，雖然 **pdf** 提供的數學看得很清楚，不過在 **local** 跟 **global** 的註記真的也是一個個 **print** 和推論才明白。

總結來說這次作業非常有挑戰性，**FK** 部分是只要錯一行空間轉換，由於 **skeleton** 彼此 **joint** 需要不斷乘 **transform**，就會直接錯掉，**Time Warper** 是 **bug** 除錯花很多時間，不過相對過去用過 **pcl(point cloud library)**的 **gui** 視窗，這次作業 **gui** 視窗還有每個 **frame** 截圖功能都很好拿來除錯用**(**不過我也是後來才發現有這些功能，如果早點發現或許能大幅減少除錯時間**)**。