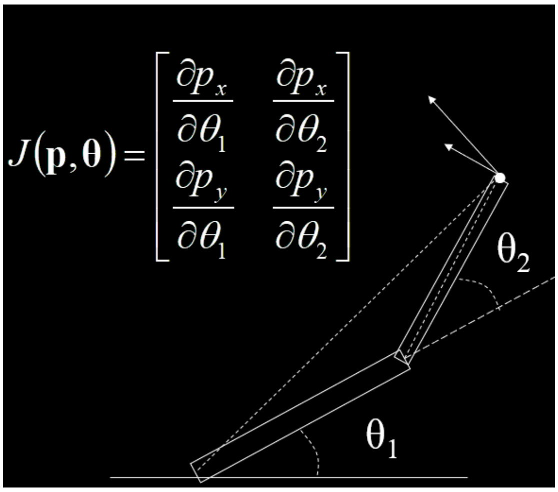
CASE HW3

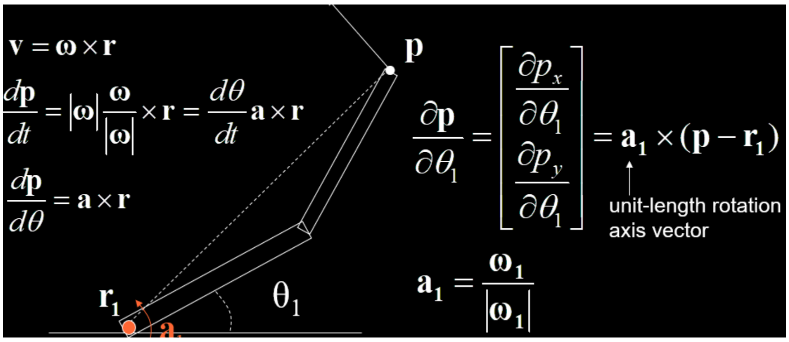
309553008

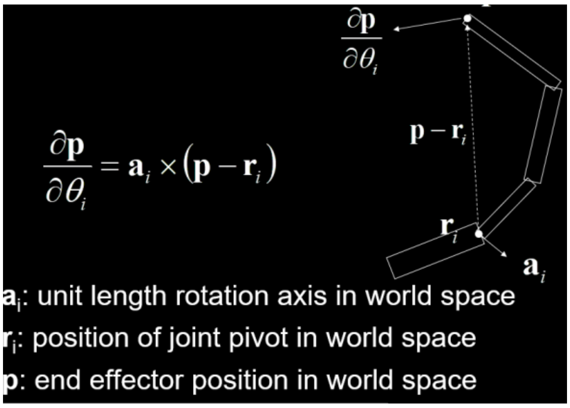
- Introduction
  1. 了解 Inverse Kinematics 是如何實現的。
  2. 了解 Jacobian Matrix 是如何求出來、如何使用的。
  3. 根據改變的 target position(終點)逆推回去所有 bone 的 position。
- Fundamentals

  從 target position 逆回去到 start bone 的，每一根 bone 的 rotation 都會算出一個 Jacobian Matrix 的 column(如下圖)



根據公式算出對每一個 rotation 之 x, y, z3 維的偏微分(如下圖)





- Implementation

  Jacobian Matrix 是一個 4 的 row(x, y, z, w)然後一個 bone 分別有 x, y, z 個 rotation theta，所以 column 數是 3*bone num。

```
Eigen::Matrix4Xd Jacobian(4, 3 * bone_num);
```

  bone num 就是從 end bone 一直往 parent 算，直到 start bone or root bone。

```
size_t bone_num = 1;
acclaim::Bone* curr_bone = end_bone;
while (curr_bone != start_bone && curr_bone != root_bone)
{
    bone_num += 1;
    curr_bone = curr_bone->parent;
}
```

  因為一個 rotation theta 會有 3 個方向(x, y, z)，所以先根據目前計算的 axis 提取該 rotation 的那一維。

```
Eigen::Matrix3d rot_mat = curr_bone->rotation.linear();
```

```
if (j == 0)
    a = (rot_mat* Eigen::Vector3d(1, 0, 0)).normalized();
else if(j == 1)
    a = (rot_mat * Eigen::Vector3d(0, 1, 0)).normalized();
else
    a = (rot_mat * Eigen::Vector3d(0, 0, 1)).normalized();
```

  然後依照公式，求得偏微分

$$\frac{\partial \mathbf{p}}{\partial \theta_i} = \mathbf{a}_i \times (\mathbf{p} - \mathbf{r}_i)$$

```
Eigen::Vector4d r = curr_bone->start_position;
Eigen::Vector4d p = end_bone->end_position;
Eigen::Vector4d delta_position = p - r;
```

```
radius << delta_position.head(3);
Eigen::Vector3d par_dif = a.cross(radius);
```

  存進 Jacobian Matrix

```
Jacobian.col(i * 3 + j) << par_dif, 1;
```

將求出來的 Jacobian Matrix 與最後一根 bone(應該 touch 到 ball 的那根)應該旋轉的方向做線性求解，再將求出來的解更新到下一個 frame 的 rotation。

```
Eigen::Vector4d desiredVector = target_pos - end_bone->end_position;
```

```
Eigen::VectorXd deltatheta = step * pseudoInverseLinearSolver(Jacobian, desiredVector);
// HINT:
// Change `posture.bone_rotation` based on deltatheta
curr_bone = end_bone;
for (size_t i = 0; i < bone_num; i += 1)
{
    for (int j = 0; j < 3; j += 1)
    {
        posture.bone_rotations[curr_bone->idx][j] += deltatheta[i * 3 + j];
    }
    curr_bone = curr_bone->parent;
}
```

因為 col 不是 linearly independent，所以跟下面 PPT/wiki 給出來的公式不一樣，我們 linearly independent 的是 row。

X 為 Jacobian Matrix，y 為 target vector，求 beta

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1p} \\ X_{21} & X_{22} & \cdots & X_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{n1} & X_{n2} & \cdots & X_{np} \end{bmatrix}, \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}.$$

然後根據 TA 給的公式算出移動，所以是 $X^T * (X * X^T)^{-1}$ * target vector

```
Eigen::VectorXd pseudoInverseLinearSolver(const Eigen::Matrix4Xd& Jacobian, const Eigen::Vector4d& target) {
    // TODO
    // You need to return the solution (x) of the linear least squares system:
    //     i.e. find x which min(| Jacobian * x - target |)
    Eigen::Matrix3Xd J = Jacobian.topRows(3);
    Eigen::Vector3d V = target.head<3>();
    Eigen::VectorXd sol = J.transpose() * (J * J.transpose()).inverse() * V;
    //Eigen::VectorXd sol = (Jacobian.transpose() * Jacobian).inverse() * Jacobian.transpose() * target;
    return sol;
    //return Eigen::VectorXd(Jacobian.cols());
}
```

- Discussion

-How different step and epsilon affect the result

Epsilon 決定 end bone 到 target ball 的距離，太大會看起來不像抓到球
Step 決定 deltatheta 的大小，大的話我看不太出來差異(太大直接爆掉)可是越小的話因為 deltatheta 也會小，如果 target ball 動太快會跟不上(很像行動遲緩)