0456024 蔡佩珊

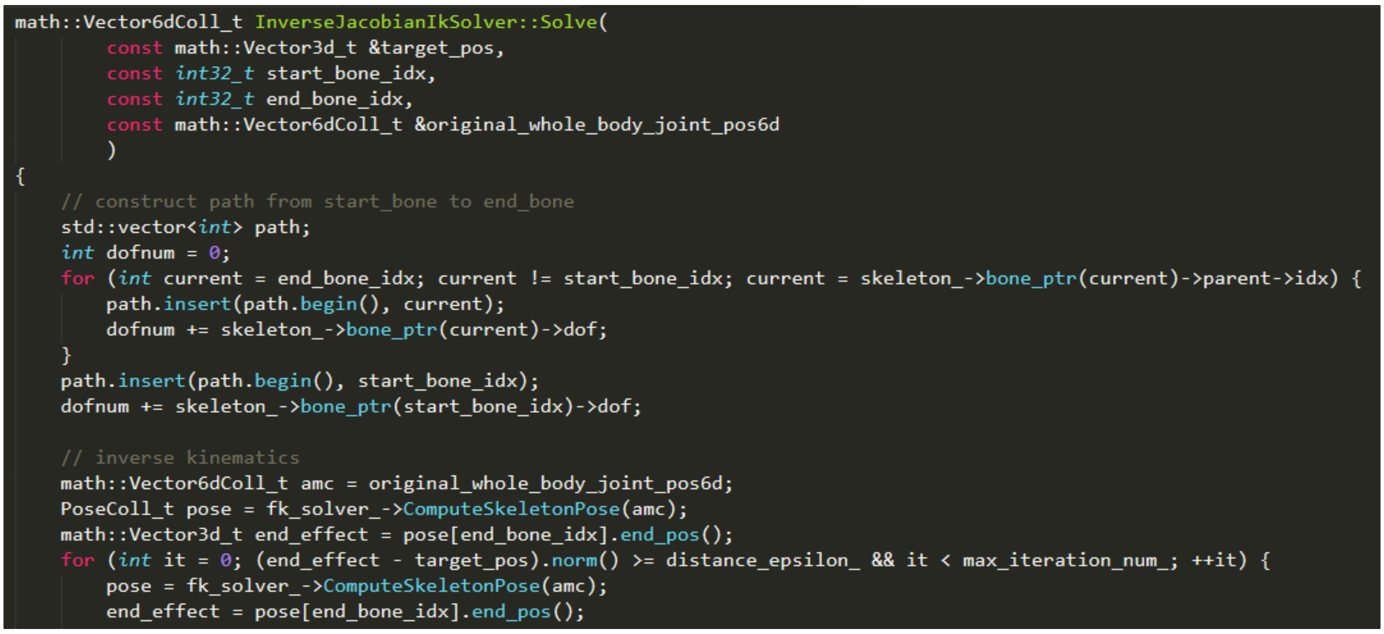
Computer Animation Project Assignment 3

♦ Topic

Inverse Kinematics

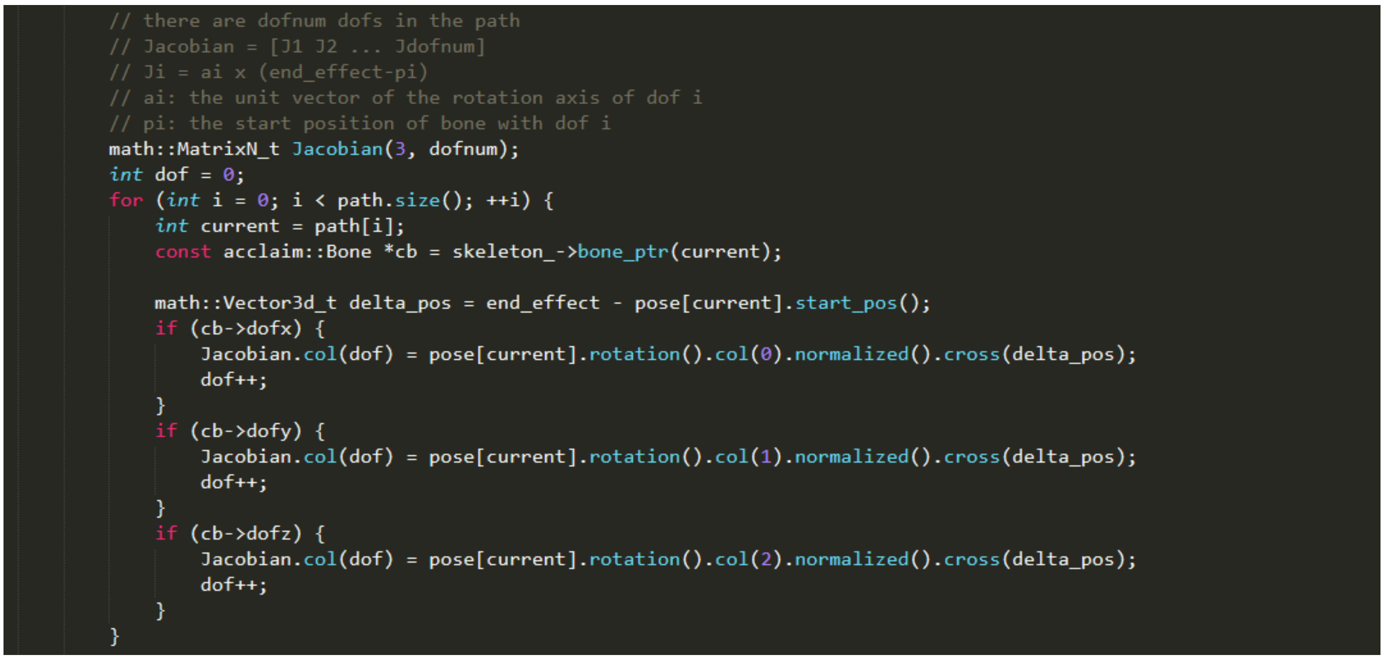
♦ Implementation (see commands in code)

InverseJacobianIkSolver::Solve()



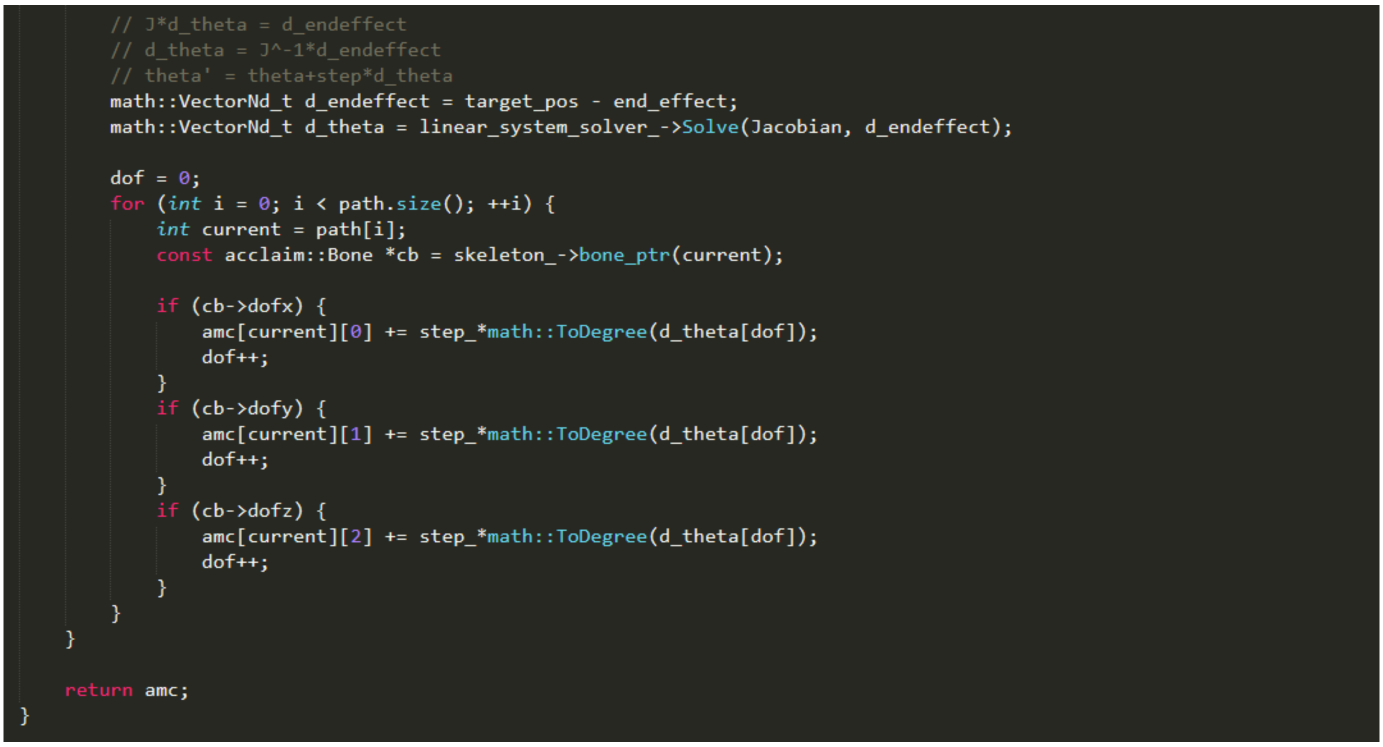
```
math::Vector6dColl_t InverseJacobianIkSolver::Solve(
        const math::Vector3d_t &target_pos,
        const int32_t start_bone_idx,
        const int32_t end_bone_idx,
        const math::Vector6dColl_t &original_whole_body_joint_pos6d
        )
{
    // construct path from start_bone to end_bone
    std::vector<int> path;
    int dofnum = 0;
    for (int current = end_bone_idx; current != start_bone_idx; current = skeleton_->bone_ptr(current)->parent->idx) {
        path.insert(path.begin(), current);
        dofnum += skeleton_->bone_ptr(current)->dof;
    }
    path.insert(path.begin(), start_bone_idx);
    dofnum += skeleton_->bone_ptr(start_bone_idx)->dof;

    // inverse kinematics
    math::Vector6dColl_t amc = original_whole_body_joint_pos6d;
    PoseColl_t pose = fk_solver_->ComputeSkeletonPose(amc);
    math::Vector3d_t end_effect = pose[end_bone_idx].end_pos();
    for (int it = 0; (end_effect - target_pos).norm() >= distance_epsilon_ && it < max_iteration_num_; ++it) {
        pose = fk_solver_->ComputeSkeletonPose(amc);
        end_effect = pose[end_bone_idx].end_pos();
```



```
        // there are dofnum dofs in the path
        // Jacobian = [J1 J2 ... Jdofnum]
        // Ji = ai x (end_effect-pi)
        // ai: the unit vector of the rotation axis of dof i
        // pi: the start position of bone with dof i
        math::MatrixN_t Jacobian(3, dofnum);
        int dof = 0;
        for (int i = 0; i < path.size(); ++i) {
            int current = path[i];
            const acclaim::Bone *cb = skeleton_->bone_ptr(current);

            math::Vector3d_t delta_pos = end_effect - pose[current].start_pos();
            if (cb->dofx) {
                Jacobian.col(dof) = pose[current].rotation().col(0).normalized().cross(delta_pos);
                dof++;
            }
            if (cb->dofy) {
                Jacobian.col(dof) = pose[current].rotation().col(1).normalized().cross(delta_pos);
                dof++;
            }
            if (cb->dofz) {
                Jacobian.col(dof) = pose[current].rotation().col(2).normalized().cross(delta_pos);
                dof++;
            }
        }
```



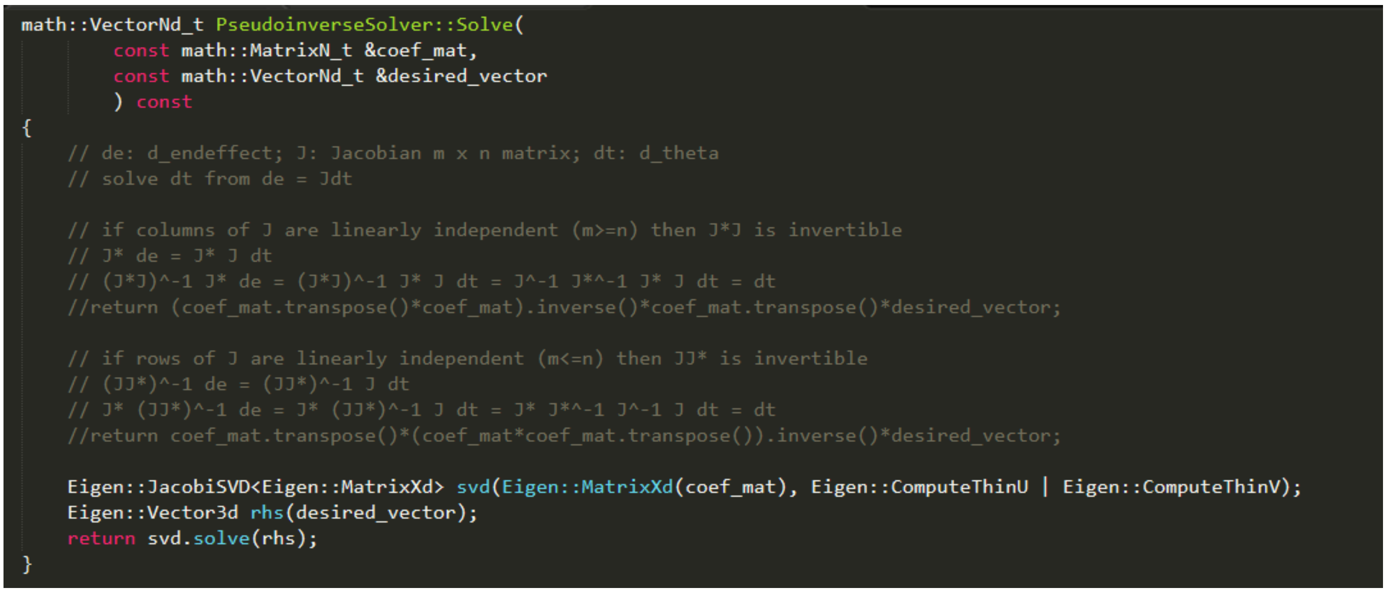
```
        // J*d_theta = d_endeffect
        // d_theta = J^-1*d_endeffect
        // theta' = theta+step*d_theta
        math::VectorNd_t d_endeffect = target_pos - end_effect;
        math::VectorNd_t d_theta = linear_system_solver_->Solve(Jacobian, d_endeffect);

        dof = 0;
        for (int i = 0; i < path.size(); ++i) {
            int current = path[i];
            const acclaim::Bone *cb = skeleton_->bone_ptr(current);

            if (cb->dofx) {
                amc[current][0] += step_*math::ToDegree(d_theta[dof]);
                dof++;
            }
            if (cb->dofy) {
                amc[current][1] += step_*math::ToDegree(d_theta[dof]);
                dof++;
            }
            if (cb->dofz) {
                amc[current][2] += step_*math::ToDegree(d_theta[dof]);
                dof++;
            }
        }
    }

    return amc;
}
```

PseudoinverseSolver::Solve()



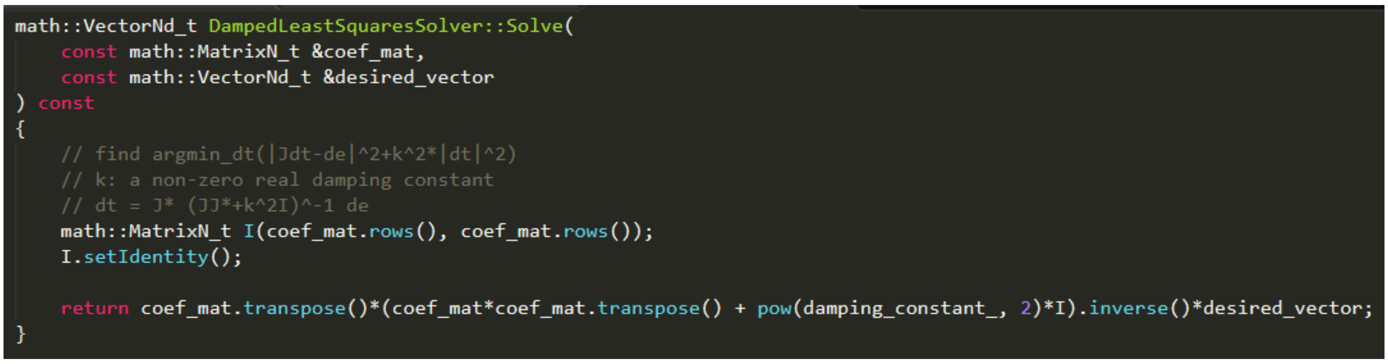
```
math::VectorNd_t PseudoinverseSolver::Solve(
        const math::MatrixN_t &coef_mat,
        const math::VectorNd_t &desired_vector
        ) const
{
    // de: d_endeffect; J: Jacobian m x n matrix; dt: d_theta
    // solve dt from de = Jdt

    // if columns of J are linearly independent (m>=n) then J*J is invertible
    // J* de = J* J dt
    // (J*J)^-1 J* de = (J*J)^-1 J* J dt = J^-1 J*^-1 J* J dt = dt
    //return (coef_mat.transpose()*coef_mat).inverse()*coef_mat.transpose()*desired_vector;

    // if rows of J are linearly independent (m<=n) then JJ* is invertible
    // (JJ*)^-1 de = (JJ*)^-1 J dt
    // J* (JJ*)^-1 de = J* (JJ*)^-1 J dt = J* J*^-1 J^-1 J dt = dt
    //return coef_mat.transpose()*(coef_mat*coef_mat.transpose()).inverse()*desired_vector;

    Eigen::JacobiSVD<Eigen::MatrixXd> svd(Eigen::MatrixXd(coef_mat), Eigen::ComputeThinU | Eigen::ComputeThinV);
    Eigen::Vector3d rhs(desired_vector);
    return svd.solve(rhs);
}
```

DampedLeastSquaresSolver::Solve()



```
math::VectorNd_t DampedLeastSquaresSolver::Solve(
    const math::MatrixN_t &coef_mat,
    const math::VectorNd_t &desired_vector
) const
{
    // find argmin_dt(|Jdt-de|^2+k^2*|dt|^2)
    // k: a non-zero real damping constant
    // dt = J* (JJ*+k^2I)^-1 de
    math::MatrixN_t I(coef_mat.rows(), coef_mat.rows());
    I.setIdentity();

    return coef_mat.transpose()*(coef_mat*coef_mat.transpose() + pow(damping_constant_, 2)*I).inverse()*desired_vector;
}
```

♦ Result and Discussion

● effect of step

step 越大，每次 iteration 調整的角度越大，使 end effect 越快接近 target position，完成 IK 所需的 iteration 次數越少；step 越小，每次 iteration 調整的角度越小，使 end effect 越慢接近 target position，完成 IK 所需的 iteration 次數越多。

但是過大或過小的 step 可能會造成在 iteration 最大次數的限制下，無法求得接近 target position 的 end effect。

以下是在使用 pseudoinverse solver 完成第一圈動作，每一個 frame 進行 IK 所需的 iteration 次數：

step	1000	100	1	0.01	0.001
iteration	failed	<15	<150	(max) 2000	failed

● effect of distance epsilon

distance epsilon 越小，則 iterative IK 停止時的 end effect 越接近 target position，輸出之動作位置更精確，然而所需的 iteration 次數越多。

過大的 distance epsilon 會無法使 end effect 接近 target position。

● comparison between pseudoinverse solver and damped least squares solver

	pseudoinverse solver	damped least squares solver
accuracy	<	
convergence speed	>	

使用 damped least squares solver 可以避免 (near) singular Jacobian matrix 無法求得反矩陣的問題。

● why damped least squares method can avoid singularities

$$d\theta = J^T(JJ^T + \lambda^2 I)^{-1} de$$

因為不確定 J 是否是 row/column linearly independent，所以 JJ^T 仍可能會有 (near) singularity 無法求得反矩陣的問題。在後面加上 $\lambda^2 I$ 形同對原本矩陣加上雜訊，以確保 $JJ^T + \lambda^2 I$ 是 row/column linearly independent 可以求得反矩陣。

● effects of damping constant to the damped least squares method

$\lambda^2 I$ 在此扮演的角色是雜訊，因此 λ 不宜過大以避免 $JJ^T + \lambda^2 I$ 跟 JJ^T 有太大的差異而求得誤差過大的 $d\theta$，可能輸出不正確的動作位置；而過小的 λ 則會失去雜訊功能，無法避免 (near) singularity。

♦ Vedio

left: PseudoinverseSolver

right: DampedLeastSquaresSolver with damping constant 0.01

step = 1

distance epsilon = 0.001