The first 20 minutes of this tutorial will be covering missing topics from Monday lectures. We will cover

- Inverse Kinematics
- Gradient Descent
- Line Search
- Gradient Descent for Inverse Kinematics See lecture slides for reference.

Forward Kinematics

`src/euler_angles_to_transform.cpp`

Transform Eulerian angles to the `4x4` matrix (Eigen::Affine3d).

To apply rotation sequentially, consider using Eigen::AngleAxisd.

Hint:

- if you need π, include `igl/PI.h`
- with `Eigen::AngleAxis`, you want the axis to be normalized. Since we are working with `x,y,z` consider using `Eigen::Vector3d::UnitX/Y/Z()`.

`src/forward_kinematics.cpp`

Since skeleton is literally a tree, if we want to get the transformation of a joint, we need to traverse the tree from the root to the current joint. The function `forward_kinematics` does this job.

See Forward Kinematics for how to get the transformation of a joint from its parent joint and current Eulerian angles.

Hint:

- gonna need `src/euler_angles_to_transform.cpp`.
- use DFS to accumulate the transformation matrix T from the root to the current joint.
- it returns a vector of Eigne::Affine3d

`src/transformed_tips.cpp`

Since you already did the hard lifting of computing the transformation of each joint in `src/forward_kinematics.cpp`, this function is trivial. It simply applies the transformation to the tips of the skeleton.

Animation

`src/catmull_rom_interpolation.cpp`

See lecture slides `34-47`.

`src/linear_blend_skinning.cpp`

Assign weights of each bone to each vertex. Then, the position of each vertex is a linear combination of the transformation of each bone.

$$\boldsymbol{v}_j = \sum_{i=1}^{nB} w_{ij} \boldsymbol{T}_i \hat{\boldsymbol{v}}_j$$

where nB is the number of bones, w_{ij} is the weight of bone i on vertex j, $\boldsymbol{T}_i$ is the transformation of bone i, $\boldsymbol{v}_j$ is the position of vertex j on mesh, and $\hat{\boldsymbol{v}}_j$ is the position of vertex j on the rest pose mesh.

Inverse Kinematics

`src/copy_skeleton_at.cpp`

Just copy the data and return a new `Skeleton` instance.

`src/end_effectors_objective_and_gradient.cpp`

Recall we want to minimize the objective function $f(\theta)$:

$$\theta = \operatorname{argmin} f(\theta) = \operatorname{argmin} ||\boldsymbol{x}(\theta) - \boldsymbol{q}||_2^2$$

where $||\cdot||_2^2$ is the squared 2-norm (in Eigen, it is just `squaredNorm()`).

This function returns *functions* that compute the objective function `f`, its gradient `grad_f`, and the projection function `proj_z` that project state z into an admisible state.

- `f`: compute the objective function $f(\theta)$, see above for definition
- `grad_f`: compute the gradient $\nabla_\theta f(\theta)$ (can be written as $\frac{df(\theta)}{d\theta}$), we do some chain rule here:

$$\begin{aligned}\frac{df(\theta)}{d\theta} &= \frac{d||\boldsymbol{x}(\theta) - \boldsymbol{q}||_2^2}{d\theta} \\ &= 2\frac{d\boldsymbol{x}(\theta)}{d\theta}(\boldsymbol{x}(\theta) - \boldsymbol{q})\end{aligned}$$

  wait, what about $\frac{d\boldsymbol{x}(\theta)}{d\theta}$? See `src/kinematics_jacobian.cpp` for details.

- `proj_z`: project state z into an admissible state

  Hint: clip values to be within the admissible range.

  ```
  // in Class Bone
  // max, min Euler Angles: joint limits
  Eigen::Vector3d xzx_max, xzx_min;
  ```

`src/kinematics_jacobian.cpp`

This function finds the Jacobian matrix $\frac{\partial \boldsymbol{x}(\theta)}{\partial \theta}$ of the forward kinematics function $\boldsymbol{x}(\theta)$.

Q: Why is $\frac{\partial \boldsymbol{x}(\theta)}{\partial \theta}$ a matrix? Jacobian?

See finite difference. Basically, for every i^{th} parameter θ_i, we can compute the partial derivative of $\boldsymbol{x}$ with respect to θ_i by finite difference:

$$\frac{\partial \boldsymbol{x}}{\partial \theta_i} \approx \frac{1}{h}(x(\theta + h) - x(\theta))$$

where h is a vector of all zeros except for the i^{th} element, which is a small number ϵ. (In implementation, just copy the current Skeleton and add ϵ to the i^{th} parameter θ_i.)

Q: why for every i^{th} parameter θ_i?

A: Because by doing finite difference, we are perturbing the i^{th} parameter θ_i and see how the output $\boldsymbol{x}$ changes. The change of $\boldsymbol{x}$ with respect to the small perturbation of θ_i is the partial derivative of $\boldsymbol{x}$ with respect to θ_i.

`src/projected_gradient_descent.cpp`

For every step of gradient descent, we first compute $\nabla_z f(z)$, then find the step size α with line search, and finally update z with $z \leftarrow z - \alpha \nabla_z f(z)$.

For projected gradient descent, we need to project z back to the constraint set after each update.

The algorithm is as follows:

```
def projected_gradient_descent(f, grad_f, proj_z, max_iters, z)
    # proj_z is a function that projects z onto the admissible
value
    # f is the value function
    # grad_f is the gradient function of f with respect to z
    for i in range(max_iters):
        dz = grad_f(z) # df(z)/dz
        alpha = line_search(f, proj_z, z, dz)   # addmissible step
size alpha
        z -= alpha * dz # descent along dz direction weighted by
alpha
```

```
        z = proj_z(z) # project z back to the constraint set
    return z
```

`src/line_search.cpp`

"Line search" means search for a good step fraction α along the line search direction $\nabla_z f(z)$, such that $f(z - \alpha \nabla_z f(z))$ is smaller than $f(z)$.

```
def line_search(f, proj_z, z, dz, max_step, threshold=1e-4):
    # proj_z is a function that projects z onto the admissible
value
    # f is the value function (think loss function in deep
learning)
    alpha = max_step
    f0 = f(z)   # evaluate f(z) before take a step along dz
direciton
    while alpha > threshold:
        z_ls = z - alpha * dz # compute z_ls along the line search
direction by step fraction alpha
        proj_z(z_ls)    # project z_ls onto the admissible value
        f_ls = f(z_ls)  # evaluate f(z_ls) after take a step along
dz direction by step fraction alpha
        if f_ls < f0:   # if f(z_ls) is smaller than f(z), then
current step fraction alpha is good
            return alpha
        alpha *= 0.5    # otherwise, reduce the step fraction by
half, take smaller step!
    return alpha
```

Q: Line search ganrantees that we can take any descent direction and $f(z)$ will never increase (in theory), why?

Hint: in theory, α can be reduced to...