```
function R_des = desired_attitude_pointing(r_rel, p)

    r = norm(r_rel);

    % if near collision
    if r < 1e-9

        R_des = eye(3);

        return;
    end

    % Compute line-of-sight

    los = -r_rel / r;
    b1 = los;
    zref = p.lvlh_z_ref / norm(p.lvlh_z_ref);
    b3 = cross(b1, zref);

    % Handle case when b1 is parallel to zref
    if norm(b3) < 1e-6

        yref = [0; 1; 0];
        b3 = cross(b1, yref);

    end

    b3 = b3 / norm(b3);
    b2 = cross(b3, b1);
    R_point = [b1, b2, b3];

    % Blend between nadir-pointing and target-pointing↙
based on distance
    blend_start = 150;
    blend_end   = 80;
```

```
    if r >= blend_start

        R_des = eye(3);

    elseif r <= blend_end

        R_des = R_point;

    else

        % Smooth blend using quaternion SLERP
        weight = (blend_start - r) / (blend_start -↙
blend_end);
        q_nadir = dcm_to_quat(eye(3));
        q_point = dcm_to_quat(R_point);
        q_des = slerp(q_nadir, q_point, weight);
        R_des = quat_to_dcm(q_des);

    end
end

function q_out = slerp(q1, q2, t)

    % Spherical linear interpolation between quaternions

    if t <= 0

        q_out = q1 / norm(q1);

        return;

    elseif t >= 1

        q_out = q2 / norm(q2);

        return;
```

```
    end

    % Ensure shortest path on quaternion hypersphere
    if dot(q1, q2) < 0

        q2 = -q2;

    end

    cos_theta = dot(q1, q2);

    % Use linear interpolation when quaternions are very↙
close

    if cos_theta > 0.9995

        q_out = (1 - t) * q1 + t * q2;

    else

        theta = acos(cos_theta);
        q_out = (sin((1 - t) * theta) * q1 + sin(t *↙
theta) * q2) / sin(theta);

    end

    q_out = q_out / norm(q_out);
end
```