```
SafeguardsCodec(
    codec=codec,  # any numcodecs codec
    safeguards=[
        # guarantee that the global minimum and maximum are preserved,
        #  which simplifies the rescaling in the quantity of interest
        dict(kind="sign", offset="$x_min"),
        dict(kind="sign", offset="$x_max"),
        dict(
            kind="qoi_eb_stencil",
            qoi="""
                # dssim constants
                V["sigma"] = 1.5;
                V["C1"] = 1e-8;
                V["C2"] = 1e-8;

                # we guarantee that
                #  min(data) = min(corrected) and
                #  max(data) = max(corrected)
                # with the sign safeguards above
                # dSSIM assumes that there are no infinite values
                V["smin"] = c["$x_min"];
                V["smax"] = c["$x_max"];
                V["r"] = V["smax"] - V["smin"];

                # re-scale to [0-1] and quantise to 256 bins
                V["sc_a1"] = round_ties_even(
                    ((C["$X"] - V["smin"]) / V["r"]) * 255
                ) / 255;
                V["sc_a2"] = round_ties_even(
                    ((X - V["smin"]) / V["r"]) * 255
                ) / 255;

                # create a 2D 3x3 Gaussian kernel
                V["i"] = [-1, 0, 1];
                V["k"] = 1/(sqrt(2*pi)*V["sigma"]) * exp(
                    (-square(V["i"])) / (2*square(V["sigma"]))
                );
                V["kern"] = matmul([V["k"]].T, [V["k"]]);

                # handle sparse NaN values by excluding them from the
                # convolution, then renormalise the kernel
                V["kern_no_NaN1"] = where(isnan(V["sc_a1"]), 0, V["kern"]);
                V["kernel1"] = V["kern_no_NaN1"] / sum(V["kern_no_NaN1"]);
                V["kern_no_NaN2"] = where(isnan(V["sc_a2"]), 0, V["kern"]);
                V["kernel2"] = V["kern_no_NaN2"] / sum(V["kern_no_NaN2"]);
                V["kern_no_NaN12"] = where(
                    isnan(V["sc_a1"] * V["sc_a2"]), 0, V["kern"]
                );
                V["kernel12"] = (
                    V["kern_no_NaN12"] / sum(V["kern_no_NaN12"])
                );

                # apply the Gaussian mean filter via convolution
                V["a1_mu_noNaN"] = sum(V["sc_a1"] * V["kernel1"]);
                V["a2_mu_noNaN"] = sum(V["sc_a2"] * V["kernel2"]);
                V["a1a1_noNaN"] = sum(square(V["sc_a1"]) * V["kernel1"]);
                V["a2a2_noNaN"] = sum(square(V["sc_a2"]) * V["kernel2"]);
                V["a1a2_noNaN"] = sum(
                    V["sc_a1"] * V["sc_a2"] * V["kernel12"]
                );

                # keep NaN for elements that are originally NaN
                V["a1_mu"] = where(
                    isnan(V["sc_a1"])[I], NaN, V["a1_mu_noNaN"]
                );
                V["a2_mu"] = where(
                    isnan(V["sc_a2"])[I], NaN, V["a2_mu_noNaN"]
                );
                V["a1a1"] = where(
                    isnan(V["sc_a1"])[I], NaN, V["a1a1_noNaN"]
                );
                V["a2a2"] = where(
                    isnan(V["sc_a2"])[I], NaN, V["a2a2_noNaN"]
                );
                V["a1a2"] = where(
                    isnan(V["sc_a1"] * V["sc_a2"])[I], NaN, V["a1a2_noNaN"]
                );

                # compute the variances and covariance
                V["var_a1"] = V["a1a1"] - square(V["a1_mu"]);
                V["var_a2"] = V["a2a2"] - square(V["a2_mu"]);
                V["cov_a1a2"] = V["a1a2"] - V["a1_mu"] * V["a2_mu"];

                # compute the SSIM components
                V["ssim_t1"] = 2 * V["a1_mu"] * V["a2_mu"] + V["C1"];
                V["ssim_t2"] = 2 * V["cov_a1a2"] + V["C2"];
                V["ssim_b1"] = (
                    square(V["a1_mu"]) + square(V["a2_mu"]) + V["C1"]
                );
                V["ssim_b2"] = V["var_a1"] + V["var_a2"] + V["C2"];

                V["ssim_1"] = V["ssim_t1"] / V["ssim_b1"];
                V["ssim_2"] = V["ssim_t2"] / V["ssim_b2"];

                # compute the pointwise dSSIM
                return V["ssim_1"] * V["ssim_2"];
            """,
            type="abs",
            # guarantee dSSIM >= 0.995
            eb=1 - 0.995,
            # 3x3 neighbourhood
            neighbourhood=[
                # latitude
                dict(axis=0, before=1, after=1, boundary="valid"),
                # longitude
                dict(axis=1, before=1, after=1, boundary="valid"),
            ],
        ),
    ],
)
```