3D Printing the Cosmic Microwave Background Radiation Anisotropy

Setup

1. Set up a Rust environment, e.g., https://rustup.rs/
2. Install Jupyter notebook with your system package manager, e.g., `brew install jupyterlab`
3. Install the Jupyter evcxr kernel with `cargo install evcxr_jupyter && evcxr_jupyter --install`
4. Download data file from Planck Legacy Archives → Maps → CMB Maps → Uncheck "Only legacy products" → SEVEM → Click to download the single row (full mission). It's 1.2 GB.

The `:dep` directives below should download Rust library dependencies as needed.

Code flow

1. Load data
2. Downsample
3. Remove dipole (we are moving relative to CMBR rest frame)
4. Map temperature to radius
5. Compute Euclidean distances
6. Compute triangular mesh
7. Sanity checks
8. Output STL for import into CAD program

In [2]:

```
// Merged https://github.com/simonrw/rust-fitsio/pull/330 in 0.21.5
:dep fitsio = "^0.21.5"
:dep color-eyre = "0.6.3"

use color_eyre::eyre::Result;
use fitsio::FitsFile;

fn get_planck_temperature_data(fname: &str) -> Result<Vec<f32>> {
    let mut fits = FitsFile::open(fname).unwrap();
    let table = fits.hdu(1).unwrap();
    dbg!(&table);
```

```
    let temperature: Vec<f32> = table.read_col(&mut fits,
"TEMPERATURE")?;
    Ok(temperature)
}

let temperature =
        get_planck_temperature_data(&"COM_CMB_IQU-
commander_4096_R4.00_full.fits")?;
```

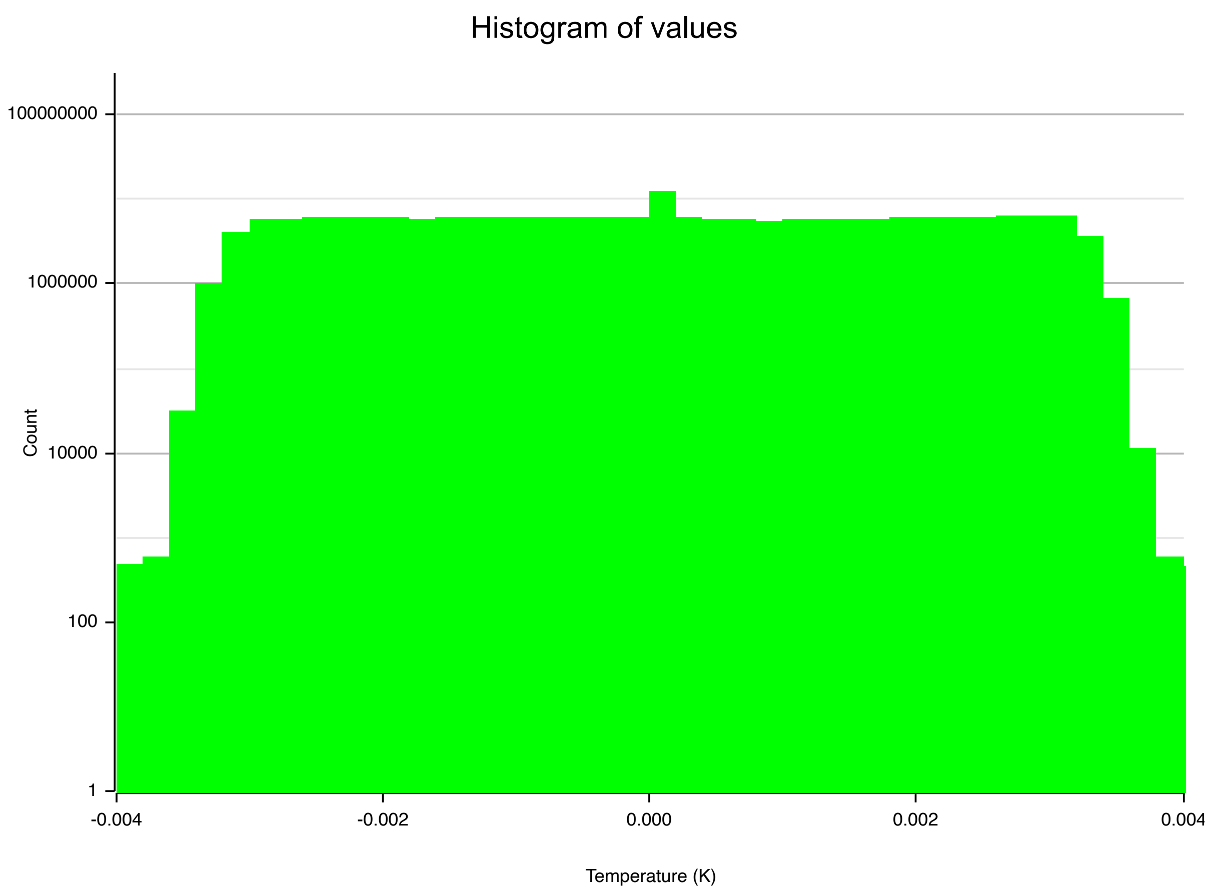
```
[src/lib.rs:7:5] &table = FitsHdu {
    info: TableInfo {
        column_descriptions: [
            ConcreteColumnDescription {
                name: "TEMPERATURE",
                data_type: ColumnDataDescription {
                    repeat: 201326592,
                    width: 1,
                    typ: Float,
                },
            },
        ],
        num_rows: 1,
    },
    number: 1,
}
```

In [3]:

```
// Histogram temperature values just to orient ourselves.
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }
use plotters::prelude::*;

const SCALE: f32 = 2e-4;

evcxr_figure((640, 480), |root| {
    let root = root.titled("Histogram of values", ("Arial",
20).into_font())?;
    root.fill(&WHITE)?;
    let mut chart = ChartBuilder::on(&root)
        .margin(10)
        .set_left_and_bottom_label_area_size(50)
        .build_cartesian_2d(-20..20i32,
(1..300_000_000i32).log_scale())  // (integerized bin, count)
        ?;
    chart.configure_mesh()
```

```
        .disable_x_mesh()
        .x_labels(5)
        .y_labels(5)
        .x_label_formatter(&|x| format!("{:.3}", *x as f32 * SCALE))
        .y_desc("Count")
        .x_desc("Temperature (K)")
        .draw()?;
    let hist = Histogram::vertical(&chart)
        .margin(0)
        .data(temperature.iter().map(|x| ((x / SCALE) as i32, 1)));
    chart.draw_series(hist)?;
    Ok(())
})
```

Out[3]:



In [4]:

```
// Average down to k=4 (depth=4; 3072 pixels). Use the property of
the HEALPix
// coordinate system that each pixel at lower depth ("resolution")
contains
// its 4^k children, which in the NESTED ordering, are stored
contiguously.
```

```
:dep rayon = { version = "1.10" }
use rayon::prelude::*;

const DEPTH: u8 = 4;
const TARGET_SIZE: usize = 3072;
let orig_size = temperature.len();
let chunk_size = orig_size / TARGET_SIZE;
assert_eq!(chunk_size * TARGET_SIZE, orig_size);  // no remainder
fn mean(arr: &[f32]) -> f32 {
    arr.iter().fold(0f32, |a, x| a + x) / arr.len() as f32
}
let ds_temp: Vec<f32> = temperature
    .par_chunks_exact(chunk_size)
    .map(|chunk| mean(chunk))
    .collect();
println!("Temperature: min, max = {}, {}",
    ds_temp.iter().fold(f32::INFINITY, |a, x| a.min(*x)),
    ds_temp.iter().fold(-f32::INFINITY, |a, x| a.max(*x))
);
```

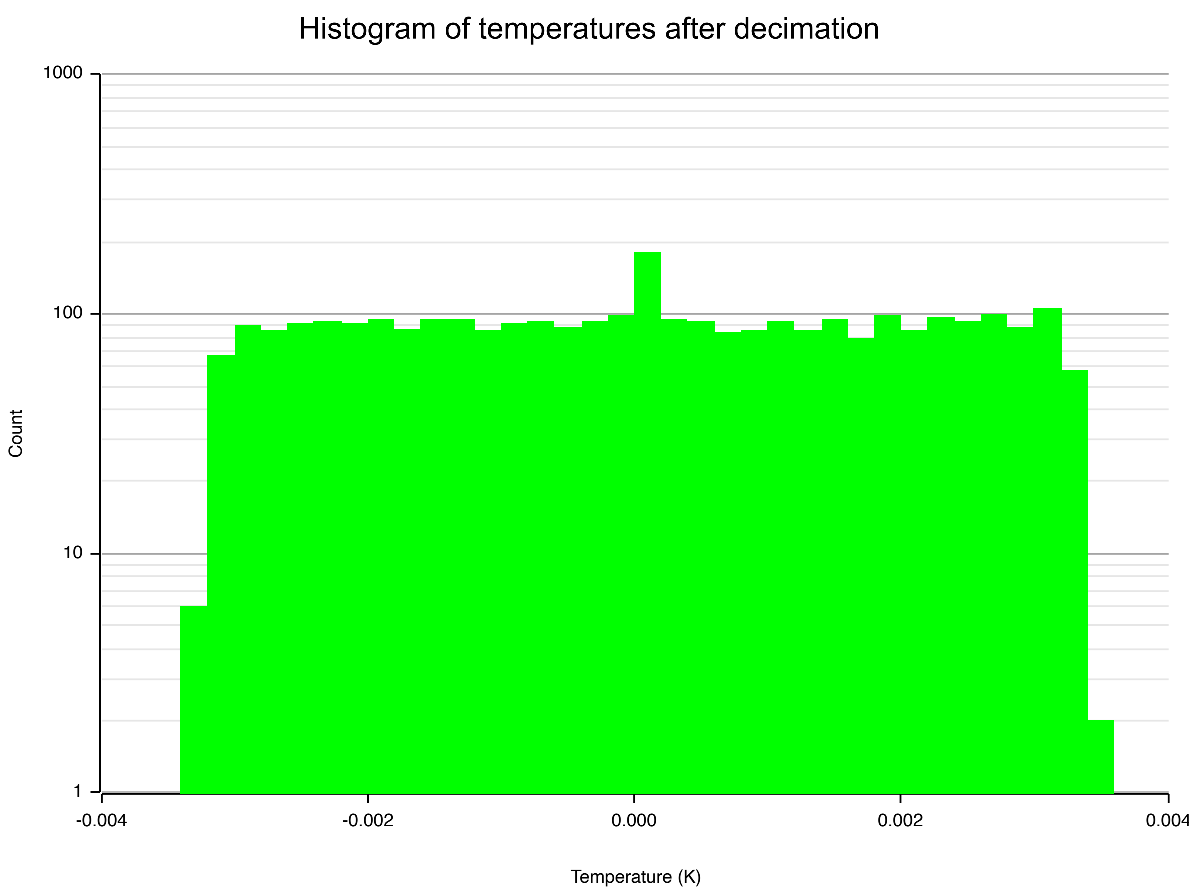
```
Temperature: min, max = -0.0034422572, 0.003420747
```

In [5]:

```
// Replot to make sure we haven't messed anything up.
// We expect the range to compress somewhat since we've averaged out
the outliers.
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }

use plotters::prelude::*;

const SCALE: f32 = 2e-4;

evcxr_figure((640, 480), |root| {
    let root = root.titled("Histogram of temperatures after
decimation", ("Arial", 20).into_font())?;
    root.fill(&WHITE)?;
    let mut chart = ChartBuilder::on(&root)
        .margin(10)
        .set_left_and_bottom_label_area_size(50)
        .build_cartesian_2d(-20..20i32, (1..1000i32).log_scale())
```

```
// (integerized bin, count)
        ?;
    chart.configure_mesh()
        .disable_x_mesh()
        .x_labels(5)
        .y_labels(5)
        .x_label_formatter(&|x| format!("{:.3}", *x as f32 * SCALE))
        .y_desc("Count")
        .x_desc("Temperature (K)")
        .draw()?;
    let hist = Histogram::vertical(&chart)
        .margin(0)
        .data(ds_temp.iter().map(|x| ((x / SCALE) as i32, 1)));
    chart.draw_series(hist)?;
    Ok(())
})
```

Out[5]:



In [6]:

```
// Map Temperature to radius
const R_MAX: f32 = 1.0;
const R_MIN: f32 = 0.5;
```

```
const T_MAX: f32 = 0.0034;  // symmetric about 0

fn t2r(t: f32) -> f32 {
    (t - (-T_MAX)) * (R_MAX - R_MIN) / (2.0 * T_MAX) + R_MIN
}

let radius: Vec<f32> = ds_temp.iter().copied().map(t2r).collect();

println!("Radius: min, max = {}, {}",
    radius.iter().fold(f32::INFINITY, |a, x| a.min(*x)),
    radius.iter().fold(-f32::INFINITY, |a, x| a.max(*x))
);
```

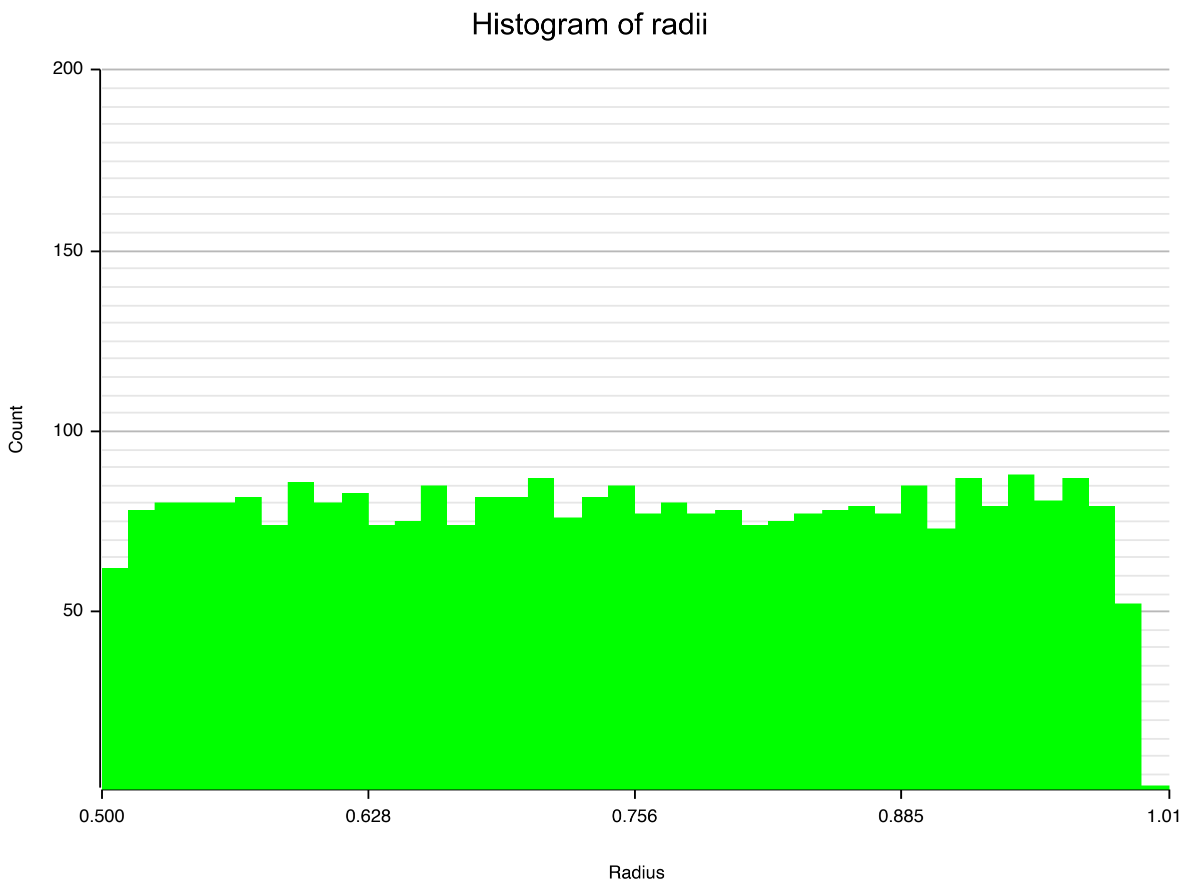
```
Radius: min, max = 0.49689287, 1.0015254
```

In [7]:

```
// Histogram radii
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }

use plotters::prelude::*;

const NUM_BINS: i32 = 40;
const SCALE: f32 = (R_MAX - R_MIN) / (NUM_BINS - 1) as f32;

evcxr_figure((640, 480), |root| {
    let root = root.titled("Histogram of radii", ("Arial",
20).into_font())?;
    root.fill(&WHITE)?;
    let mut chart = ChartBuilder::on(&root)
        .margin(10)
        .set_left_and_bottom_label_area_size(50)
        .build_cartesian_2d(0..NUM_BINS, 1..200i32)  // (integerized
bin, count)
        ?;
    chart.configure_mesh()
        .disable_x_mesh()
        .x_labels(5)
        .y_labels(5)
        .x_label_formatter(&|x| format!("{:.3}", *x as f32 * SCALE +
R_MIN))
```

```
        .y_desc("Count")
        .x_desc("Radius")
        .draw()?;
    let hist = Histogram::vertical(&chart)
        .margin(0)
        .data(radius.iter().map(|x| (((x - R_MIN) / SCALE) as i32,
1)));
    chart.draw_series(hist)?;
    Ok(())
})
```

Out[7]:



In [8]:

```
// Compute point cloud in Euclidean space coordinates (x, y, z)
// The whole magic of the HEALPix coordinate system is hidden in
nested::center().
:dep cdshealpix = { version = "^0.6" }

use std::f32::consts::FRAC_PI_2;
use cdshealpix::nested;

type XYZ = (f32, f32, f32);  // real Euclidean coordinates
```

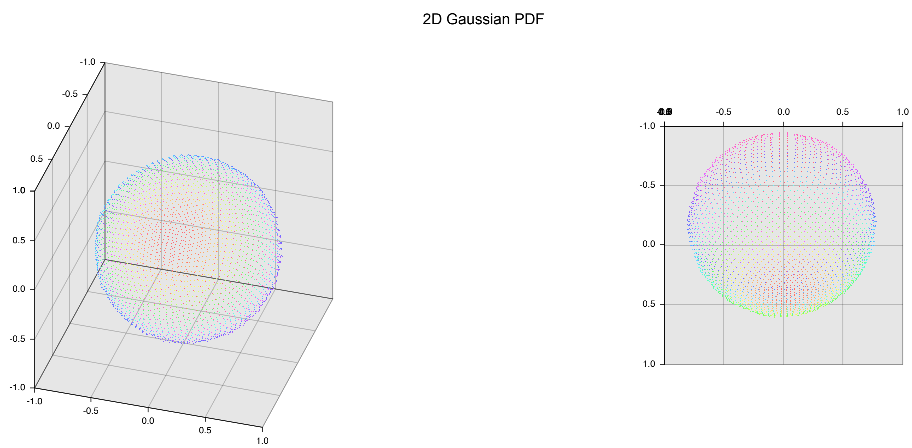
```
type IJK = (u16, u16, u16);  // vertex indices

fn pix_location(depth: u8, ind: u64, r: f32) -> XYZ {
    let (lon, lat) = nested::center(depth, ind);
    let phi = lon as f32;
    let theta = lat as f32 + FRAC_PI_2;
    (r * theta.sin() * phi.cos(),
     r * theta.sin() * phi.sin(),
     r * theta.cos())
}

let points: Vec<XYZ> = radius.iter()
    .copied()
    .zip(0..)
    .map(|(r, i)| pix_location(DEPTH, i, r))
    .collect();
```

In [10]:

```
// Plot point cloud in 3D
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }

use plotters::prelude::*;

evcxr_figure((640 * 2, 480), |root| {

    let root = root.titled("2D Gaussian PDF", ("Arial",
20).into_font())?;
    root.fill(&WHITE).unwrap();
    let (left, right) = root.split_horizontally(640);

    for (pitch, yaw, area) in vec![(0.6, 0.3, left), (1.5707, 0.0,
right)] {
        let mut chart = ChartBuilder::on(&area)
            .build_cartesian_3d(-1.0..1.0, -1.0..1.0, -1.0..1.0)?;
        chart.with_projection(|mut p| {
            p.pitch = pitch;
            p.yaw = yaw;
            p.scale = 0.7;
            p.into_matrix() // build the projection matrix
```

```
        });

        chart.configure_axes().draw()?;

        let series = points.iter().copied().zip(&radius).map(|(p,
r)| {
            let color = &HSLColor(((*r - R_MIN) / (R_MAX - R_MIN))
as f64,1.0,0.7);  // r < 1.0
            Pixel::new([p.0 as f64, p.1 as f64, p.2 as f64], color)
        });

        chart.draw_series(series)?;
    }

    Ok(())
})
```

Out[10]:



In [11]:

```
// I don't like what looks like a dipole moment. Fit and subtract.
// Ref:
https://healpix.sourceforge.io/doc/html/sub_remove_dipole.htm

// Step 1: Construct known matrices
#[allow(non_snake_case)]
let mut A = vec![0f32; 10]; // 4x4, but symmetric; represent upper-
triangular part
let mut b = vec![0f32; 4];

for (t, i) in ds_temp.iter().zip(0u64..) {
    let t = *t;
```

```
    let (lon, lat) = nested::center(DEPTH, i);
    let phi = lon as f32;
    let theta = lat as f32 + FRAC_PI_2;
    let x = theta.sin() * phi.cos();
    let y = theta.sin() * phi.sin();
    let z = theta.cos();

    A[0] += 1.0;
    A[1] += x;
    A[2] += y;
    A[3] += z;
    A[4] += x * x;
    A[5] += x * y;
    A[6] += x * z;
    A[7] += y * y;
    A[8] += y * z;
    A[9] += z * z;
    b[1] += t;
    b[1] += t * x;
    b[2] += t * y;
    b[3] += t * z;
}

dbg!(&A);
dbg!(&b);
dbg!(ds_temp.len() / 3);

// Step 2: Solve for unknown matrix of dipole coefficients, f.
// Take the mega-shortcut that A is effectively diagonal and
// we can just read off the answer.
let norm = 3.0 / ds_temp.len() as f32;
let mut f = vec![norm / 3.0 * b[0], norm * b[1], norm * b[2], norm *
b[3]];

// Step 3: Subtract from map
let temp: Vec<f32> = ds_temp.iter().zip(0u64..).map(|(t, i)| {
    let t = *t;
    let (lon, lat) = nested::center(DEPTH, i);
    let phi = lon as f32;
```

```
    let theta = lat as f32 + FRAC_PI_2;
    let x = theta.sin() * phi.cos();
    let y = theta.sin() * phi.sin();
    let z = theta.cos();

    t - f[0] - f[1] * x - f[2] * y - f[3] * z
}).collect();

println!("Temperature: min, max = {}, {}",
    temp.iter().fold(f32::INFINITY, |a, x| a.min(*x)),
    temp.iter().fold(-f32::INFINITY, |a, x| a.max(*x))
);
```

```
[src/lib.rs:195:1] &A = [
    3072.0,
    -1.8954277e-5,
    -1.2040138e-5,
    1.4193356e-6,
    1024.185,
    -1.3291836e-5,
    -1.0870397e-5,
    1024.185,
    0.00011607446,
    1023.62366,
]
[src/lib.rs:196:1] &b = [
    0.0,
    -0.230782,
    -2.2820942,
    -2.5703108,
]
[src/lib.rs:197:1] ds_temp.len() / 3 = 1024
Temperature: min, max = -0.00020696866, 0.00018331292
```

In [12]:

```
// Remap radius and recompute point cloud
const R_MAX: f32 = 1.0;
const R_MIN: f32 = 0.8;
const T_MAX: f32 = 0.00021;  // symmetric about 0

fn t2r(t: f32) -> f32 {
    (t - (-T_MAX)) * (R_MAX - R_MIN) / (2.0 * T_MAX) + R_MIN
}

let radius: Vec<f32> = temp.iter().copied().map(t2r).collect();
```

```
println!("Radius: min, max = {}, {}",
    radius.iter().fold(f32::INFINITY, |a, x| a.min(*x)),
    radius.iter().fold(-f32::INFINITY, |a, x| a.max(*x))
);

let points: Vec<XYZ> = radius.iter()
    .copied()
    .zip(0..)
    .map(|(r, i)| pix_location(DEPTH, i, r))
    .collect();
```

```
Radius: min, max = 0.8014435, 0.9872919
```

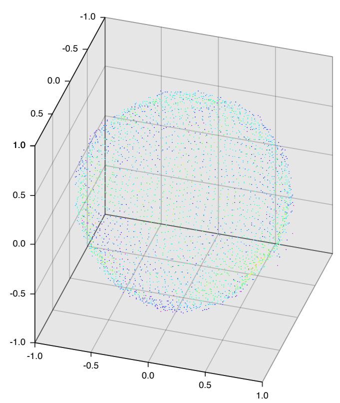
In [13]:

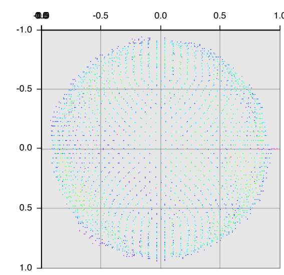
```
// Plot 3D point cloud again
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }

use plotters::prelude::*;

evcxr_figure((640 * 2, 480), |root| {

    let root = root.titled("2D Gaussian PDF", ("Arial",
20).into_font())?;
    root.fill(&WHITE).unwrap();
    let (left, right) = root.split_horizontally(640);

    for (pitch, yaw, area) in vec![(0.6, 0.3, left), (1.5707, 0.0,
right)] {
        let mut chart = ChartBuilder::on(&area)
            .build_cartesian_3d(-1.0..1.0, -1.0..1.0, -1.0..1.0)?;
        chart.with_projection(|mut p| {
            p.pitch = pitch;
            p.yaw = yaw;
            p.scale = 0.7;
            p.into_matrix() // build the projection matrix
        });

        chart.configure_axes().draw()?;

        let series = points.iter().copied().zip(&radius).map(|(p,
```

```
r)| {
            let color = &HSLColor(((*r - R_MIN) / (R_MAX - R_MIN))
as f64,1.0,0.7);  // r < 1.0
            Pixel::new([p.0 as f64, p.1 as f64, p.2 as f64], color)
        });

        chart.draw_series(series)?;
    }

    Ok(())
})
```

Out[13]:

2D Gaussian PDF





In [14]:

```
// Construct a mesh
use std::collections::HashSet;
use std::f64::consts::PI;
use cdshealpix::compass_point::MainWind::{
    S,
    SE,
    E,
    SW,
    NE,
    W,
    NW,
    N};

let mut triangles: HashSet<IJK> = HashSet::with_capacity(4 *
temp.len());

// Important properties to uphold:
```

```
// 1. Triangle vertices need to be stored in counter-clockwise order
for
//    normal vectors to point outward (STL requirement).
// 2. Keep a convention that the lowest index goes first so that the
hash
//    set will prevent duplicates.

fn rotate_to_min(a: u64, b: u64, c: u64) -> IJK {
    if a < b && a < c {
        (a as u16, b as u16, c as u16)
    } else if b < a && b < c {
        (b as u16, c as u16, a as u16)
    } else {
        (c as u16, a as u16, b as u16)
    }
}

// Set threshold at √2× the pixel edge length (approximating pixel
as a square)
let dist_threshold: f64 = (2. * 4. * PI / temp.len() as f64).sqrt();

// Haversine formula works well everywhere on the sphere
fn dist(a: u64, b: u64) -> f64 {
    let p1 = nested::sph_coo(DEPTH, a, 0.5, 0.5);
    let p2 = nested::sph_coo(DEPTH, b, 0.5, 0.5);
    let sindlon = f64::sin(0.5 * (p2.0 - p1.0));
    let sindlat = f64::sin(0.5 * (p2.1 - p1.1));
    2f64 * f64::asin(f64::sqrt(sindlat * sindlat + p1.1.cos() *
p2.1.cos() * sindlon * sindlon))
}

let mut too_far_counter = 0u16;
for i in 0..(temp.len() as u64) {
    let neighbors = nested::neighbours(DEPTH, i as u64, false);
    let mut valid_neighbors: Vec<u64> = Vec::with_capacity(8);
    for dir in [S, SE, E, NE, N, NW, W, SW] {
        if let Some(neighbor) = neighbors.get(dir) {
            if dist(i, *neighbor) < dist_threshold {
                valid_neighbors.push(*neighbor);
```

```
            } else {
                too_far_counter += 1;
            }
        }
    }
    if valid_neighbors.len() >= 2 {
        for (a, b) in
valid_neighbors.iter().zip(&valid_neighbors[1..]) {
            triangles.insert(rotate_to_min(i, *a, *b));
        }
        triangles.insert(rotate_to_min(
            i,
            valid_neighbors[valid_neighbors.len() - 1],
            valid_neighbors[0]
        ));  // SW to S
    }
}
dbg!(triangles.len());
dbg!(too_far_counter);

// Make a fixed iteration order we can index into it stably
let triangles: Vec<IJK> = triangles.iter().cloned().collect();
```

```
[src/lib.rs:241:1] triangles.len() = 6384
[src/lib.rs:242:1] too_far_counter = 6376
```

In [15]:

```
// Check mesh quality: orientation and area uniformity

// Orientation: compute normals and dot with radial normal to check
sign.
// Area: Area is half the magnitude of the normal.
fn normalize(xyz: XYZ) -> (f32, XYZ) {
    let (x, y, z) = xyz;
    let norm = (x * x + y * y + z * z).sqrt();
    (norm, (x / norm, y / norm, z / norm))
}

fn dot(a: XYZ, b: XYZ) -> f32 {
    a.0 * b.0 + a.1 * b.1 + a.2 * b.2
}
```

```rust
let mut num_bad_orientations = 0;
let mut normals: Vec<XYZ> = Vec::with_capacity(triangles.len());
let mut areas: Vec<f32> = Vec::with_capacity(triangles.len());
for (a_ind, b_ind, c_ind) in &triangles {
    let dx: XYZ = (
        points[*b_ind as usize].0 - points[*a_ind as usize].0,
        points[*b_ind as usize].1 - points[*a_ind as usize].1,
        points[*b_ind as usize].2 - points[*a_ind as usize].2,
    );
    let dy: XYZ = (
        points[*c_ind as usize].0 - points[*b_ind as usize].0,
        points[*c_ind as usize].1 - points[*b_ind as usize].1,
        points[*c_ind as usize].2 - points[*b_ind as usize].2,
    );
    let mut normal: XYZ = (  // n = dy × dx
        dy.1 * dx.2 - dy.2 * dx.1,
        -dy.0 * dx.2 + dy.2 * dx.0,
        dy.0 * dx.1 - dy.1 * dx.0,
    );
    let (norm, normal) = normalize(normal);
    let area = norm / 2.;
    let orientation = dot(normal, points[*a_ind as usize]);
    if orientation < 0.0 {
        num_bad_orientations += 1;
    }
    normals.push(normal);
    areas.push(area);
}
dbg!(num_bad_orientations);
```

```
[src/lib.rs:251:1] num_bad_orientations = 0
```

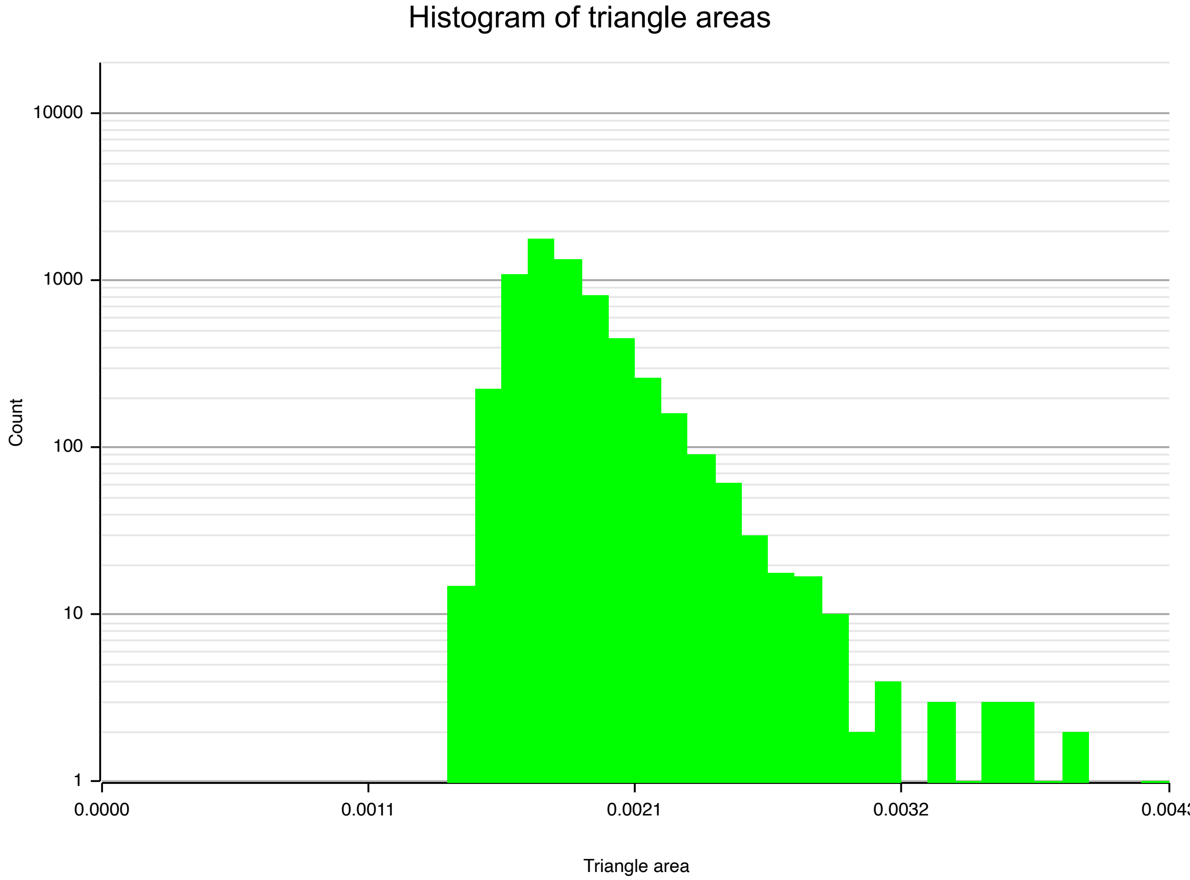
In [16]:

```rust
:dep plotters = { version = "^0.3.0", default_features = false,
features = ["evcxr", "all_series", "all_elements"] }

use plotters::prelude::*;

let a_min: f32 = areas.iter().fold(f32::INFINITY, |a, x| a.min(*x));
let a_max: f32 = areas.iter().fold(-f32::INFINITY, |a, x|
```

```
a.max(*x));
const NUM_BINS: i32 = 40;

println!("For intuition: Expect two triangles per pix, so if
constant radius, so area ~4π / #pixels / 2 = 0.002.");
dbg!(4. * PI / temp.len() as f64);
dbg!(a_min);
dbg!(a_max);

let a_min = 0.0;  // clamp to zero for a sense of scale
let scale: f32 = (a_max - a_min) / (NUM_BINS - 1) as f32;

evcxr_figure((640, 480), |root| {
    let root = root.titled("Histogram of triangle areas", ("Arial",
20).into_font())?;
    root.fill(&WHITE)?;
    let mut chart = ChartBuilder::on(&root)
        .margin(10)
        .set_left_and_bottom_label_area_size(50)
        .build_cartesian_2d(0..NUM_BINS, (1..20000i32).log_scale())
// (integerized bin, count)
        ?;
    chart.configure_mesh()
        .disable_x_mesh()
        .x_labels(5)
        .y_labels(5)
        .x_label_formatter(&|x| format!("{:.4}", *x as f32 * scale +
a_min))
        .y_desc("Count")
        .x_desc("Triangle area")
        .draw()?;
    let hist = Histogram::vertical(&chart)
        .margin(0)
        .data(areas.iter().map(|x| (((x - a_min) / scale) as i32,
1)));
    chart.draw_series(hist)?;
    Ok(())
})
```

```
For intuition: Expect two triangles per pix, so if constant radius, so area ~4
π / #pixels / 2 = 0.002.
```

Out[16]:




```
[src/lib.rs:234:1] 4. * PI / temp.len() as f64 = 0.0040906154343617095
[src/lib.rs:235:1] a_min = 0.0013909262
[src/lib.rs:236:1] a_max = 0.0041482765
```

In [17]:

```
// Check mesh quality: watertightness
// Ref: https://davidstutz.de/a-formal-definition-of-watertight-meshes/, Def 2
//  A 2-manifold mesh is called watertight if each edge has exactly two
//  incident faces, i.e. no boundary edges exist.
use std::collections::HashMap;

// Store with lower index first.
let mut edge_count: HashMap<(u16, u16), u16> = HashMap::with_capacity(4 * temp.len());
for (a, b, c) in &triangles {
    *edge_count.entry((*a, *b)).or_insert(0) += 1;  // Vertices were stored with min index first
    *edge_count.entry((*a, *c)).or_insert(0) += 1;
    if b < c {
```

```
        *edge_count.entry((*b, *c)).or_insert(0) += 1;
    } else {
        *edge_count.entry((*c, *b)).or_insert(0) += 1;
    }
}

// How did we do?
dbg!(edge_count.len());
let mut count_hist = [0u16; 6];  // overflow in last bin
for count in edge_count.values() {
    if (*count as usize) < count_hist.len() - 1 {
        count_hist[*count as usize] += 1;
    } else {
        count_hist[count_hist.len() - 1] += 1;
    }
}
dbg!(count_hist);
```

```
[src/lib.rs:250:1] edge_count.len() = 9332
[src/lib.rs:259:1] count_hist = [
    0,
    0,
    8964,
    248,
    120,
    0,
]
```

In [18]:

```
// OK, a few hundred edges are used 3 or 4 times. Could exhaustively prune them or refine
// generation further. Or we can say good enough for now and let Fusion 360 fix it.
```

In [19]:

```
// Output to STL file
// Per https://en.wikipedia.org/wiki/STL_(file_format)#Binary_STL
// UINT8[80]    – Header                 -     80 bytes
// UINT32       – Number of triangles    -      4 bytes
// foreach triangle                      - 50 bytes:
//     REAL32[3] – Normal vector             - 12 bytes
//     REAL32[3] – Vertex 1                  - 12 bytes
//     REAL32[3] – Vertex 2                  - 12 bytes
//     REAL32[3] – Vertex 3                  - 12 bytes
```

```
//     UINT16    – Attribute byte count      –  2 bytes
// end
// NB: All in little-endian.

use std::fs::File;
use std::io::Write;

const NORMAL_VECTOR: [u8; 12] = [0u8; 12];
const ATTRIBUTE_BYTE_COUNT: [u8; 2] = [0u8; 2];

let num_bytes: usize = 84 + 50 * triangles.len();
let mut bytes: Vec<u8> = Vec::with_capacity(num_bytes);

// UINT8[80]    – Header                    –     80 bytes
let comment = "Nick Fotopoulos <nickolas.fotopoulos@gmail.com>";
assert!(comment.len() <= 80);
bytes.write(comment.as_bytes())?;
bytes.write(&[0u8].repeat(80 - comment.len()))?;

// UINT32       – Number of triangles    –      4 bytes
bytes.write(&(num_bytes as u32).to_le_bytes())?;

// foreach triangle                         - 50 bytes:
for ((a, b, c), normal) in triangles.iter().zip(normals) {
//     REAL32[3] – Normal vector             - 12 bytes
    let (x, y, z) = normal;
    bytes.write(&x.to_le_bytes())?;
    bytes.write(&y.to_le_bytes())?;
    bytes.write(&z.to_le_bytes())?;
//     REAL32[3] – Vertex 1                  - 12 bytes
    let (x, y, z) = points[*a as usize];
    bytes.write(&x.to_le_bytes())?;
    bytes.write(&y.to_le_bytes())?;
    bytes.write(&z.to_le_bytes())?;
//     REAL32[3] – Vertex 2                  - 12 bytes
    let (x, y, z) = points[*b as usize];
    bytes.write(&x.to_le_bytes())?;
    bytes.write(&y.to_le_bytes())?;
    bytes.write(&z.to_le_bytes())?;
```

```
//     REAL32[3] – Vertex 3                    - 12 bytes
    let (x, y, z) = points[*c as usize];
    bytes.write(&x.to_le_bytes())?;
    bytes.write(&y.to_le_bytes())?;
    bytes.write(&z.to_le_bytes())?;
//     UINT16    – Attribute byte count       -  2 bytes
    bytes.write(&ATTRIBUTE_BYTE_COUNT)?;
}
assert_eq!(bytes.len(), num_bytes);

// Write to disk
let mut file = File::create("cmbr.stl")?;
file.write(&bytes).unwrap();
file.sync_all()?;
```