Spherical Gradient Lighting Sequence by Example

This page explores spherical gradient lighting, as trialled by Ma 2007 with USC ICT LS3-5 Lightstage by Debevec, et al.

The script loads a frame (the icosahedron 3v dome, with vertex numbering specified for the Aber Lightstage) orders the vertices by their x-axis position and applies a "spherical gradient" (a `gradient value`) to each light, corresponding to a smooth illumination of the target.

The `gradient value` can be used as a relative light wattage output value, within the constraints of control bit values per light's hardware interface. These spherical gradient values are proportional to their x-axis positions to the sphere, such that the target's illumination remains balanced.

The plots below help to illustrate how each gradient value is applied to each light, pre-ordered by it's x-axis position, and through a series of rotations culminating in a lighting sequence.

Imports & Declarations

To start this isolated test of the spherical gradient sequence code, we make the relevant imports and define a plot to visualise the output.

In [1]:

```
default_path = "../../src/"
import sys
sys.path.insert(0, default_path)
%matplotlib notebook
from spherical_gradient import GradientSequence_IntervalSpecified
from data_3d import WaveFront
import matplotlib.pyplot as plt


def barplot(x):
    fig, ax = plt.subplots()
    plt.bar(range(len(x)), x)
```

Configuring the Lighting Sequence Algorithm

Next we import the lightstage frame, specifying where each LED is located `[(x,y,z),..]`.

For the `hardcoded_frame()` case, we assume every frame vertex is going to have an LED. But it needn't be that way.

Note: the icosahedron frame's vertices' x-axis positions are not uniformly distributed around the x-axis of the sphere. We could apply a uniformly distributed set as light positions. You can try each out below to see the effect on gradient smoothness.

In [2]:

```
leds = WaveFront.get_hardcoded_frame( scale=8 )
# leds = zip( range(-8,9,1), range(-8,9,1), range(-8,9,1) )
print(len(leds))
```

```
92
```

Next we specify each light's `baseline_intensity`, that is the light's default wattage output value; which can be specified as `1.0` for no effect. Later, this will be used as a multiplier to the gradient value.

It can also be defined via a CSV file as a tuned set of values. See for example, (Brightness Control Tuning) [https://github.com/LightStage-Aber/LightStage-Repo#optimise-evenness-of-any-light-position-set--m3 (https://github.com/LightStage-Aber/LightStage-Repo#optimise-evenness-of-any-light-position-set--m3)]. With this tool, we can minimise the effects of imperfectly balanced positions of the lights, thus further improving the lighting balance.

In [3]:

```
baseline_intensity = [1.0]*len(leds)
```

Next, we specify the `scaled_range` of 0.5-1.0 to each light. That is the `min/max` range of gradient value to be applied to each light position.

The `quantity_of_intervals` let us control the granularity of rotations. Each plot sequence below is an output of from the current rotation. The sphere of light positions is going to appear to rotate, with a number of granular stops. That's what this value defines.

Below, there are 5 granular rotations, defined by `quantity_of_intervals`.

Then we instantiate the `GradientSequence` object.

In [4]:

```
scaled_range = [0.5, 1.0]
quantity_of_intervals = 5
gs = GradientSequence_IntervalSpecified( leds, baseline_intensity, scaled_range,
quantity_of_intervals )
```

Getting Each Rotation's Gradients of the Sequence:

To request the first (and next) rotation's lighting configuration, we call `gs.get_next_sequence()`.

We can also track the sequence progress, by requesting the rotation number of the current loop, and the loop number with `gs.get_sequence_number()`.

In [5]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
```

```
1
0
```

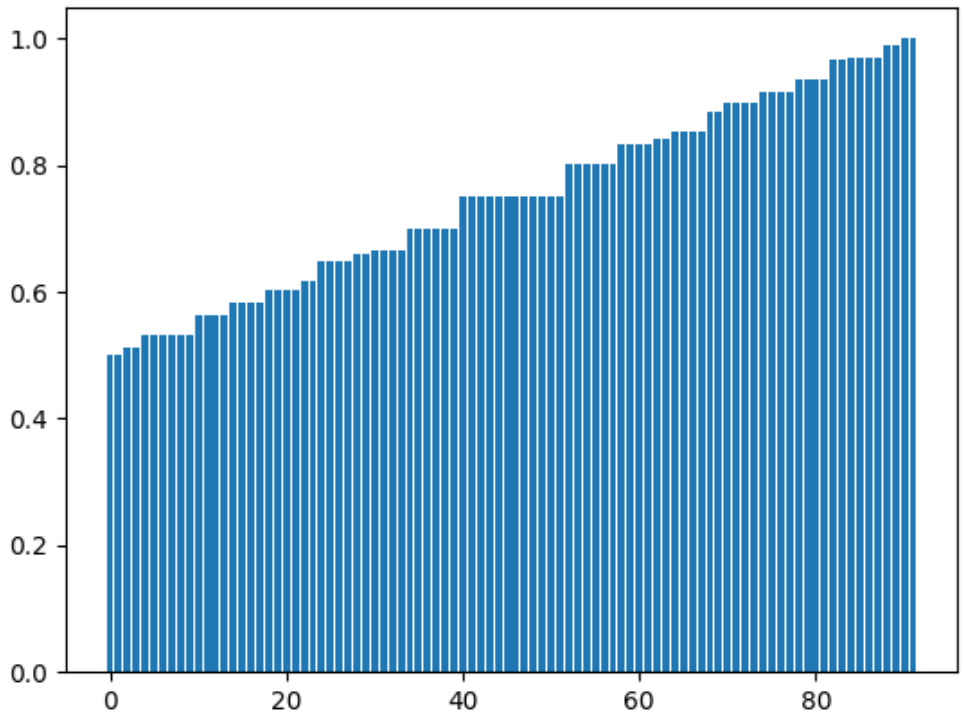
From the result, we extract data from the `BaseSequenceContainer` class objects, including `c.get_index()` for the vertex index and `c.get_intensity()` for the light's (relative) wattage output value.

Each light's `c.get_intensity()` wattage output value is multiplied by a wattage output value (`baseline_intensity`) that can be specified as `1.0` for no effect.

In [6]:

```
gradient_intensities = [ c.get_intensity() for c in l ]
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
# print(gradient_intensities)
barplot(gradient_intensities)
```

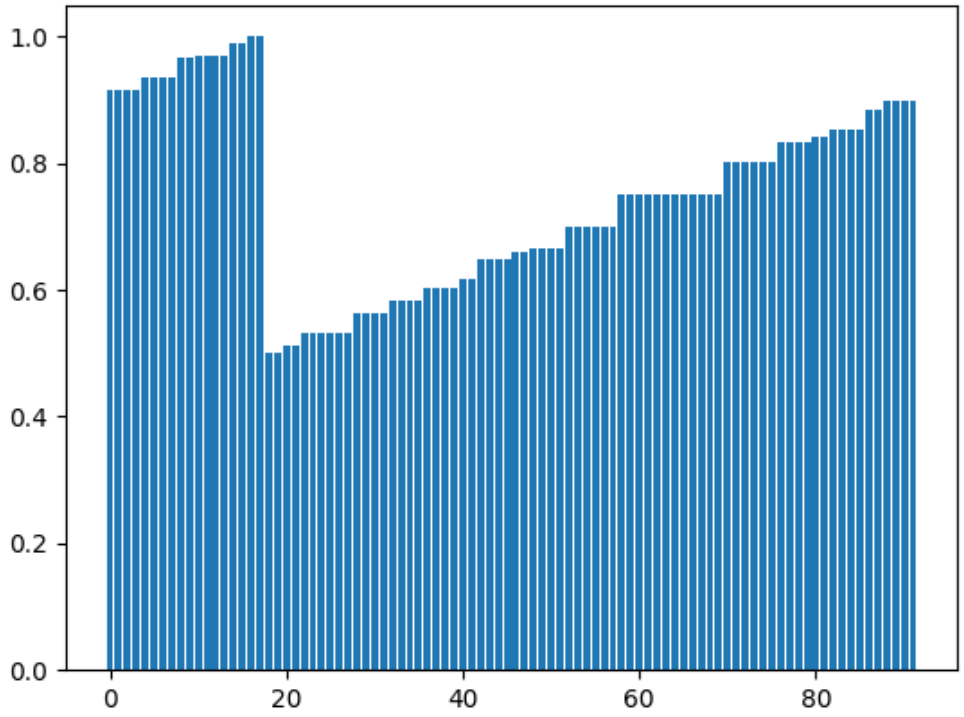
```
1
0
```



In [7]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

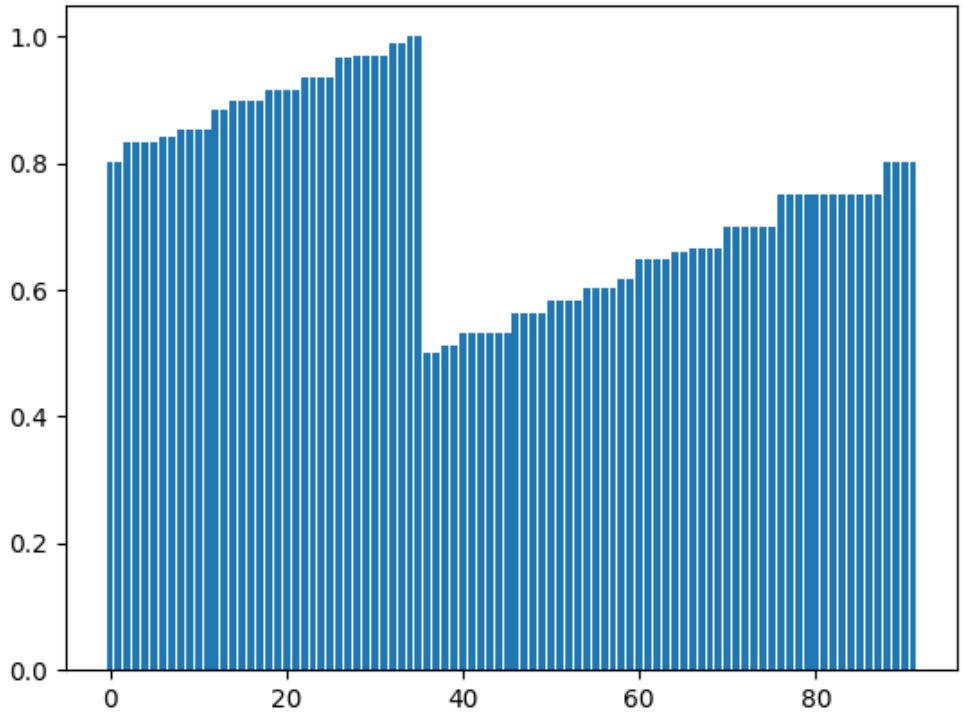
```
2
0
```



In [8]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

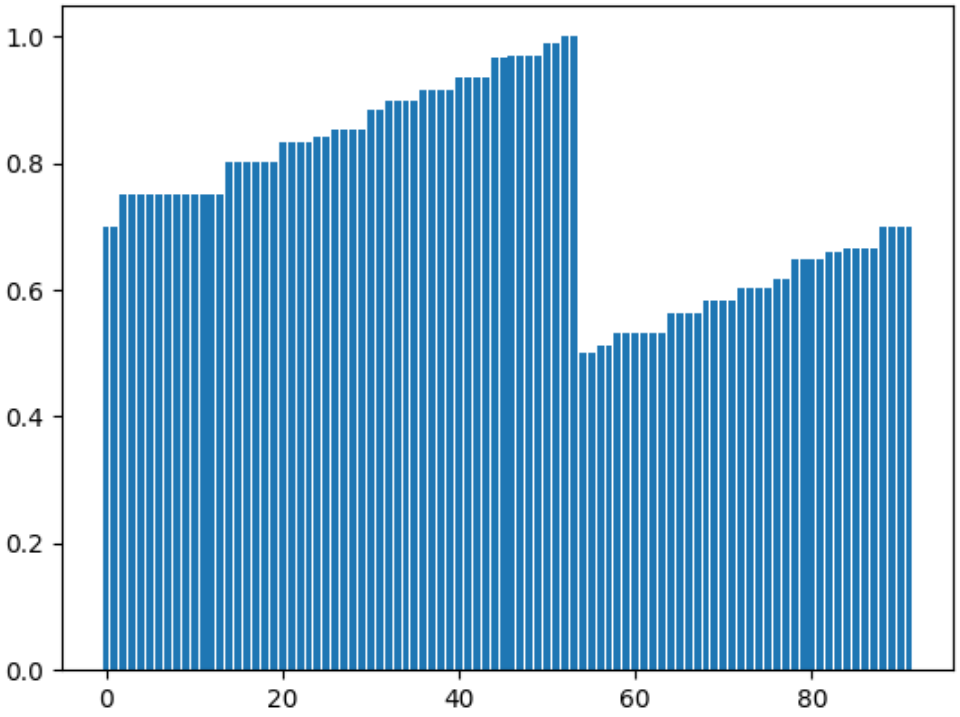
```
3
0
```



In [9]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

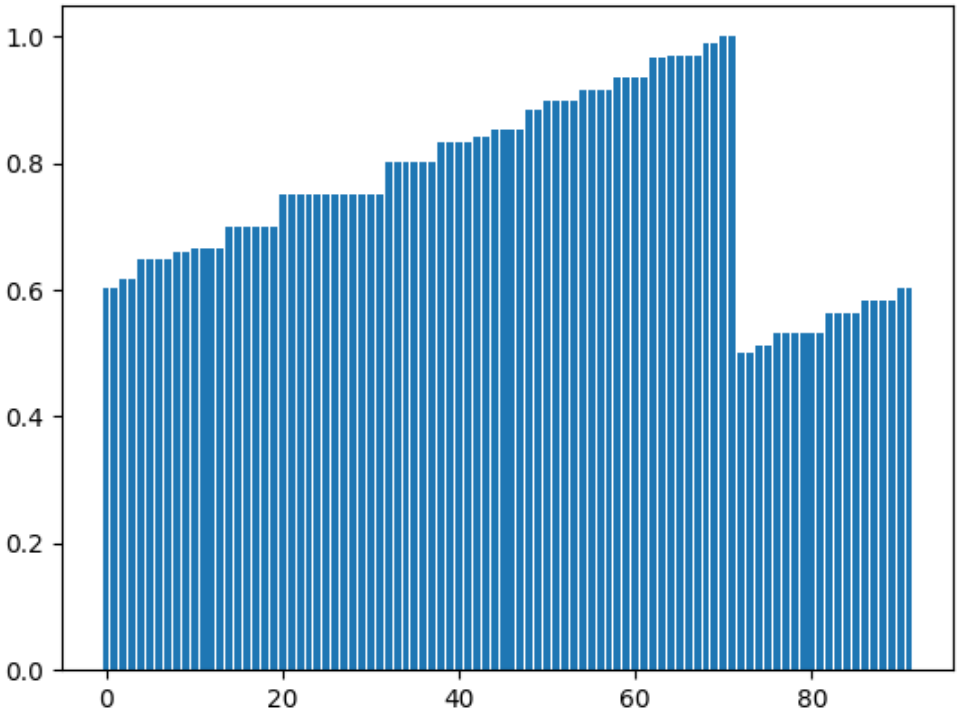
```
4
0
```



In [10]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

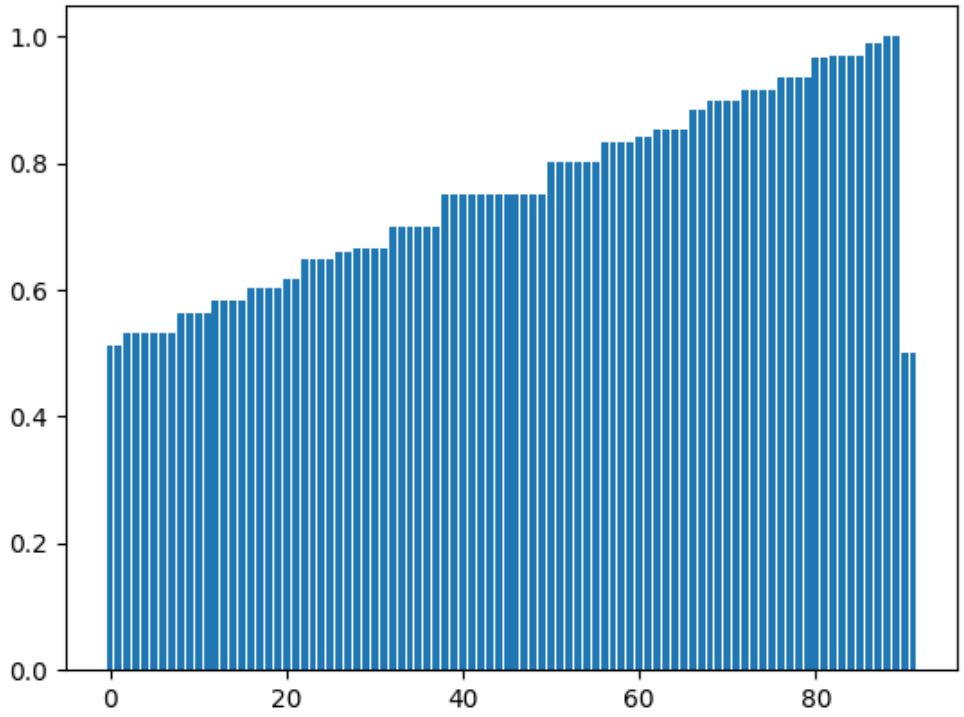
```
5
0
```



In [11]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

```
1
1
```



In [12]:

```
l = gs.get_next_sequence()
num, loop_num = gs.get_sequence_number()
print(num)
print(loop_num)
gradient_intensities = [ c.get_intensity() for c in l ]
barplot(gradient_intensities)
```

```
2
1
```