This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2021-15069C

Exceptional service in the national interest

Invariance, Equivariance, and Atomic Structure Representation

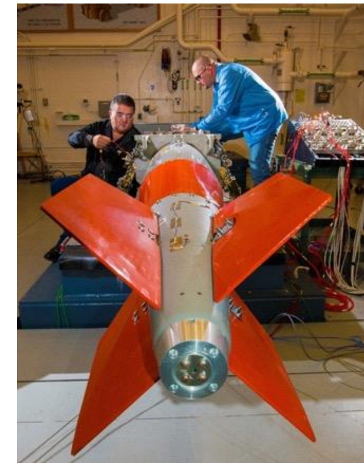
PRESENTED BY

Thomas J. Hardin

PRESIDENT HARRY S. TRUMAN FELLOW

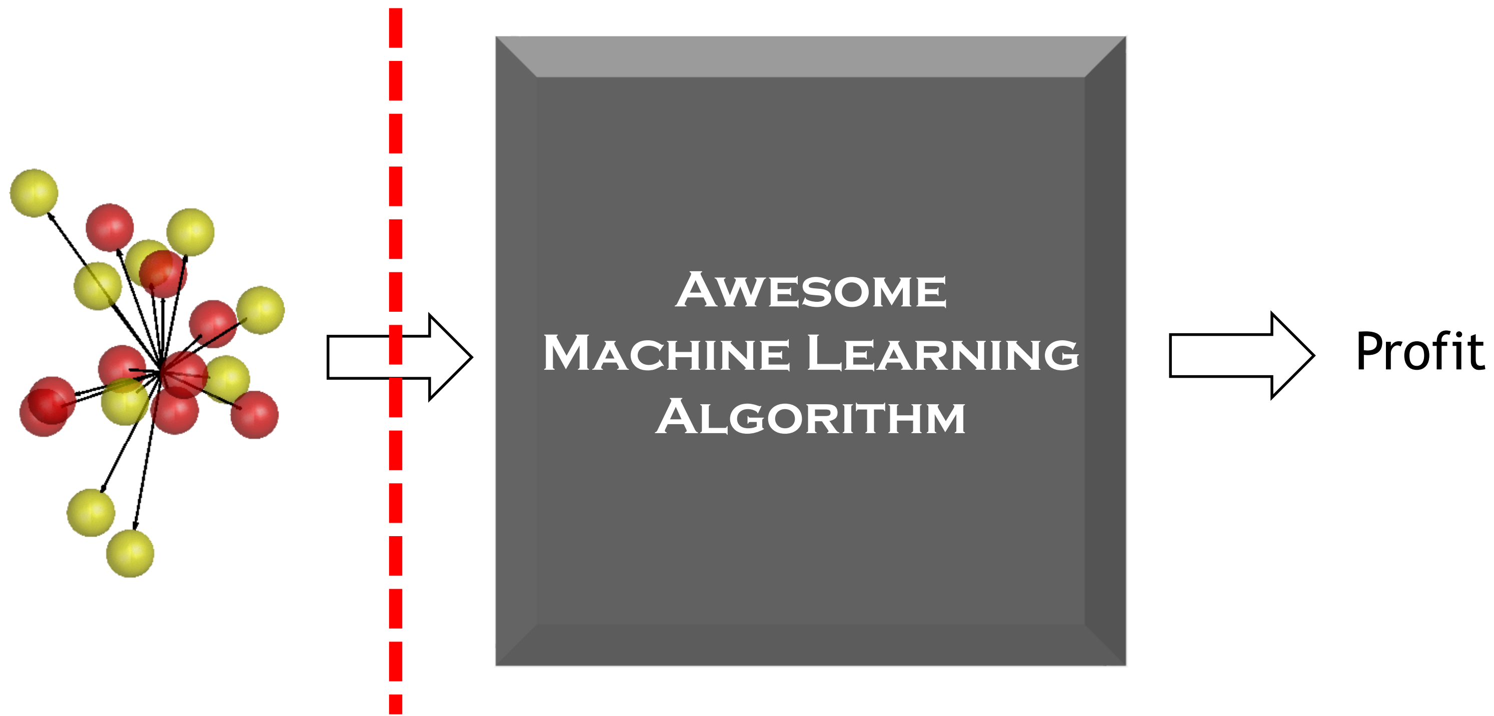
November 29, 2021

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



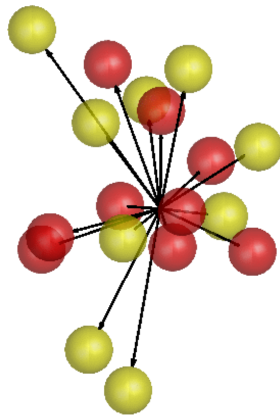
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The Big Problem: Awkwardness at the Interface



Naively Interfacing Machine Learning and Atomic Structure

- N atoms are described using (3N) real numbers for positions, and N labels for species.
- So, imagine describing an atomic structure as a matrix/vector, that we'll pass into the ML algorithm:



$$\mathbf{V} = \begin{bmatrix} x_1 & y_1 & z_1 & s_1 \\ x_2 & y_2 & z_2 & s_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_N & y_N & z_N & s_N \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \vdots & \vdots \\ \mathbf{X}_N & s_N \end{bmatrix}$$

$$f(\mathbf{V}) = \mathbf{U}$$

- Compact and straightforward

Naively Feeding Atomic Structure Into Machine Learning

- Problem 1a: Permutation Symmetry
  - Data augmentation: N! replicates
    - So, for 12 neighbors, 4.8*10^8 replicates

$$\mathbf{V}_a = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \\ \mathbf{X}_4 & s_4 \end{bmatrix} \quad \mathbf{V}_b = \begin{bmatrix} \mathbf{X}_2 & s_2 \\ \mathbf{X}_1 & s_1 \\ \mathbf{X}_4 & s_4 \\ \mathbf{X}_3 & s_3 \end{bmatrix} \quad f(\mathbf{V}_a) = f(\mathbf{V}_b)$$

- Problem 1b: Variable Numbers of Atoms

$$\mathbf{V}_c = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \end{bmatrix} \quad \mathbf{V}_d = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \\ \mathbf{X}_4 & s_4 \end{bmatrix}$$

- Problem 2: Frame Invariance/Equivariance
  - Data augmentation: 10^4 replicates in 3d

$$\mathbf{RV} = \begin{bmatrix} \mathbf{RX}_1 & s_1 \\ \vdots & \vdots \\ \mathbf{RX}_N & s_N \end{bmatrix}$$

$$f(\mathbf{V}) = \mathbf{U} \implies f(\mathbf{RV}) = \mathbf{U}$$

$$f(\mathbf{V}) = \mathbf{U} \implies f(\mathbf{RV}) = \mathbf{RU}$$

The Trouble with Canonical Orderings / Orientations

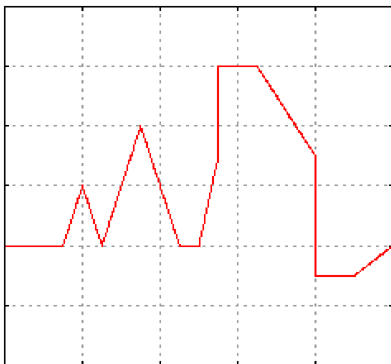
- I do not know of a strategy for assigning a “canonical” ordering or orientation to **arbitrary** local atomic environments that is stable with respect to atomic perturbation/thermal fluctuation.
- Example 1: resolving permutation symmetry by ordering atoms by distance from central atom.
  - Issue: if more than one atom is “the same distance” from the central atom, this is unstable.
  - This case occurs frequently, since atoms form more-or-less rough shells around central atoms.
- Example 2: resolving rotational symmetry by aligning local atomic environments along principal axes
  - Issue: if the mass of the local atomic environment is approximately spherically or cylindrically symmetrically arranged relative to the central atom, this is unstable
  - Again, this case occurs frequently when taking spheres cut out of bulk as the local atomic environment
- Bonus: Instability associated with atoms near the cutoff boundary vibrating in or out of the environment.

Basis Function Expansion

Fourier Series

Domain: periodic “rectangle” on R^n

$$s_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left(a_n \cos\left(\tfrac{2\pi}{P} nx\right) + b_n \sin\left(\tfrac{2\pi}{P} nx\right) \right).$$



Spherical Harmonics

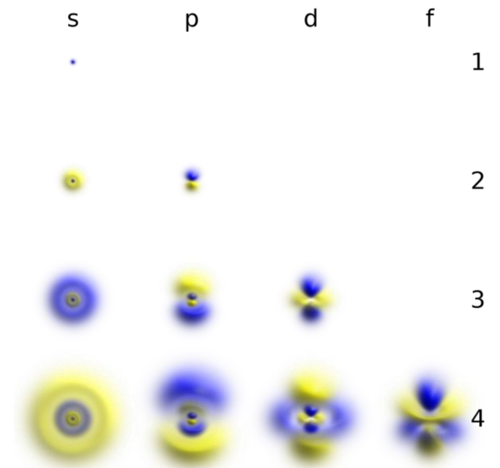
Domain: unit sphere in R^3

l:		$P_\ell^m(\cos\theta)\ \cos(m\varphi)$							$P_\ell^{\lvert m\rvert}(\cos\theta)\ \sin(\lvert m\rvert\varphi)$					
0	s													
1	p													
2	d													
3	f													
4	g													
5	h													
6	i													
	m:	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6

z
x y

Atomic Orbitals

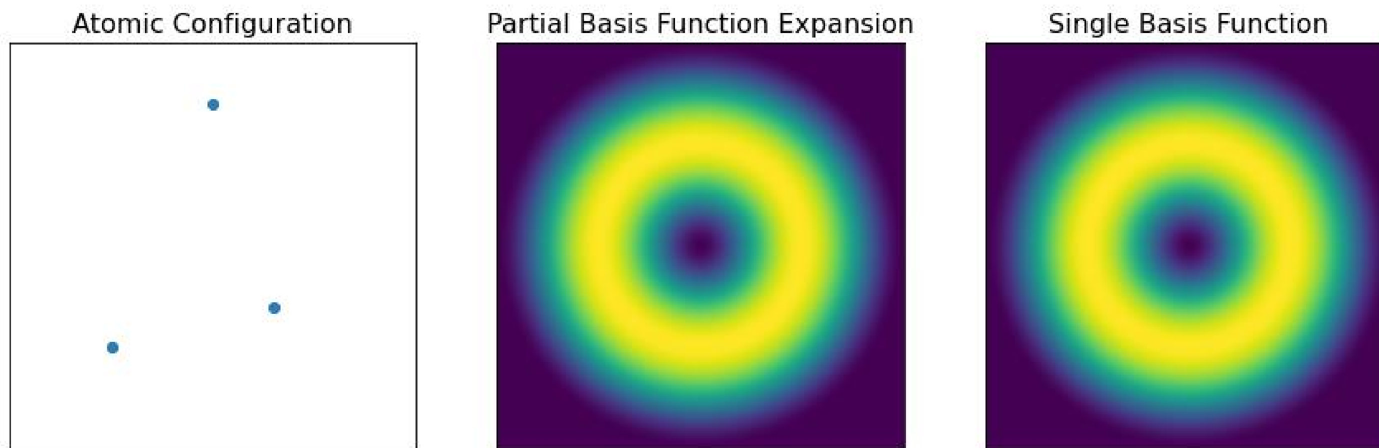
Domain: R^3 w/center



A way of approximating arbitrary functions, continuous or discontinuous, as a finite vector of real (or complex) coefficients, easy for a computer to store and use.

By carefully picking the basis functions, the basis function expansion can acquire special properties.

Atomic Representation via Basis Function Expansion



[2.077]

Naively Feeding Atomic Structure Into Machine Learning

- Problem 1a: Permutation Symmetry
  - Data augmentation: N! replicates

$$\mathbf{V}_a = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \\ \mathbf{X}_4 & s_4 \end{bmatrix} \quad \mathbf{V}_b = \begin{bmatrix} \mathbf{X}_2 & s_2 \\ \mathbf{X}_1 & s_1 \\ \mathbf{X}_4 & s_4 \\ \mathbf{X}_3 & s_3 \end{bmatrix} \quad f(\mathbf{V}_a) = f(\mathbf{V}_b)$$

- Problem 1b: Variable Numbers of Atoms

$$\mathbf{V}_c = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \end{bmatrix} \quad \mathbf{V}_d = \begin{bmatrix} \mathbf{X}_1 & s_1 \\ \mathbf{X}_2 & s_2 \\ \mathbf{X}_3 & s_3 \\ \mathbf{X}_4 & s_4 \end{bmatrix}$$

- Problem 2: Frame Invariance/Equivariance
  - Data augmentation: 10^4 replicates in 3d

$$\mathbf{RV} = \begin{bmatrix} \mathbf{RX}_1 & s_1 \\ \vdots & \vdots \\ \mathbf{RX}_N & s_N \end{bmatrix}$$

$$f(\mathbf{V}) = \mathbf{U} \implies f(\mathbf{RV}) = \mathbf{U} \quad \text{(invariance)}$$

$$f(\mathbf{V}) = \mathbf{U} \implies f(\mathbf{RV}) = \mathbf{RU} \quad \text{(equivariance)}$$

Equivariant Neural Network Architecture

- Basic Multilayer Perceptron

$$f_1\left(\boldsymbol{b}_1 + \boldsymbol{A}_1 \cdot f_2\left(\boldsymbol{b}_2 + \boldsymbol{A}_2 \cdot f_3\left(\boldsymbol{b}_3 + \boldsymbol{A}_3 \cdot f_4(\boldsymbol{b}_4 + \boldsymbol{A}_4 \cdot \boldsymbol{x})\right)\right)\right) = \widehat{\boldsymbol{y}} \approx \boldsymbol{y}$$

- Generalized neural network

$$u_{\boldsymbol{A}_u}\left(v_{\boldsymbol{A}_v}\left(w_{\boldsymbol{A}_w}(\boldsymbol{x})\right)\right) = \widehat{\boldsymbol{y}} \approx \boldsymbol{y}$$

- Equivariance

$$f(\boldsymbol{R}\boldsymbol{x}) = \boldsymbol{R}f(\boldsymbol{x})$$

- Equivariant Neural Network

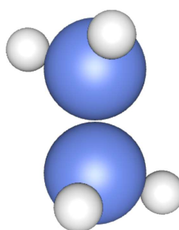
$$u_{\boldsymbol{A}_u}\left(v_{\boldsymbol{A}_v}\left(w_{\boldsymbol{A}_w}(\boldsymbol{R}\boldsymbol{x})\right)\right) = u_{\boldsymbol{A}_u}\left(v_{\boldsymbol{A}_v}\left(\boldsymbol{R}w_{\boldsymbol{A}_w}(\boldsymbol{x})\right)\right) = u_{\boldsymbol{A}_u}\left(\boldsymbol{R}v_{\boldsymbol{A}_v}\left(w_{\boldsymbol{A}_w}(\boldsymbol{x})\right)\right) = \boldsymbol{R}u_{\boldsymbol{A}_u}\left(v_{\boldsymbol{A}_v}\left(w_{\boldsymbol{A}_w}(\boldsymbol{x})\right)\right) = \boldsymbol{R}\widehat{\boldsymbol{y}} \approx \boldsymbol{R}\boldsymbol{y}$$

In [1]:
```
import matplotlib.pyplot as plt
import torch
import torch.autograd.functional
torch.set_default_dtype(torch.float32)

from tutorial.orthonormal_radial_basis import (
    FixedCosineRadialModel,
    CosineFunctions,
    FadeAtCutoff,
    OrthonormalRadialFunctions
    )
from tutorial.radial_spherical_tensor import *

import ase
from ase.io import read
from ase.visualize import view

import numpy as np

import plotly.graph_objects as go
```

We'll use a small molecule as an examplar. We'll store and visualize it using the ASE (Atomic Simulation Environment) library (J. Phys.: Condens. Matter Vol. 29 273002, 2017).

In [2]:
```
from ase.build import molecule
atoms = molecule('N2H4')
view(atoms, viewer='x3d')
```

Out[2]:



Set up a radial-spherical tensor definition, using the Gram-Schmidt orthornormalization of some truncated and shifted cosines for the radial basis.

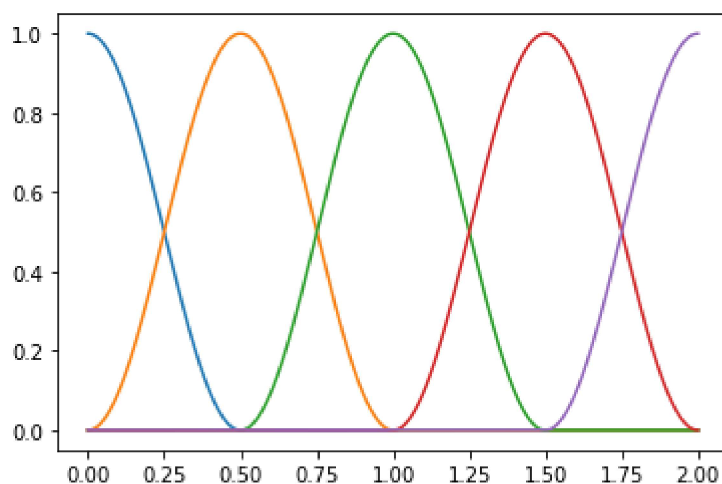
In [3]:
```
radialCutoff = 2
lmax = 4
p_val = 1
p_arg = 1
```

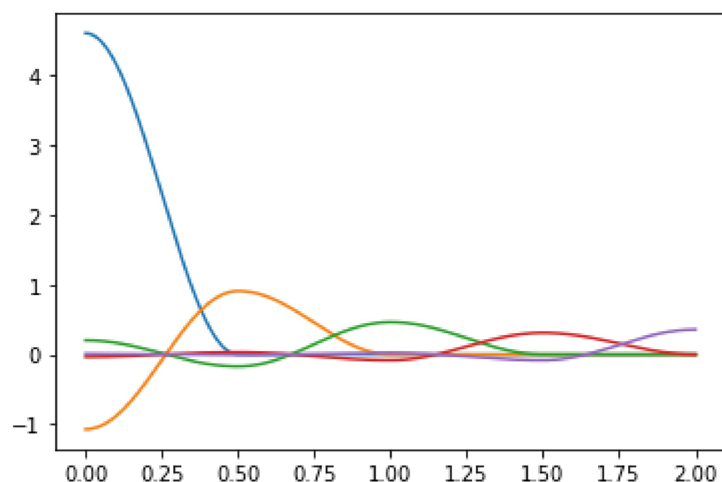
```
nRadialFunctions = (lmax+1)

fixedCosineRadialModel = FixedCosineRadialModel(radialCutoff, nRadialFunctions)
onRadialModel = OrthonormalRadialFunctions(nRadialFunctions, fixedCosineRadialModel, ra

rst = RadialSphericalTensor(nRadialFunctions, onRadialModel, lmax, p_val, p_arg)
```

We'll visualize the radial basis for your enjoyment. The first plot here is non-orthonormal, and the second is the Gram-Schmidt orthonormalization of the first.

In [4]:

```
r = torch.linspace(0,radialCutoff,1000)
y = fixedCosineRadialModel(r)
fig = plt.figure()
p = plt.plot(r,y)
fig.show()

r = torch.linspace(0,radialCutoff,1000)
y = onRadialModel(r)
fig = plt.figure()
p = plt.plot(r,y)
fig.show()
```





Convert the exemplar molecule to a signal using the radial-spherical tensor we defined in the previous cell. We'll place peaks of height 1 on heavy atoms, and peaks of height -1 on hydrogen atoms.

In [5]:
```
atomValues = torch.zeros((len(atoms),))
atomValues[atoms.get_atomic_numbers()==1]=-1
atomValues[atoms.get_atomic_numbers()>1]=1
originalSignal = rst.with_peaks_at(torch.tensor(atoms.get_positions(), dtype=atomValues
print(originalSignal)
```

```
tensor([-1.4319e-14,  1.4594e-01, -4.8736e-02, -5.3230e-01,  1.4249e-01,
         0.0000e+00, -4.2352e-22, -6.4970e-15,  0.0000e+00,  0.0000e+00,
        -2.7156e-02,  0.0000e+00,  0.0000e+00, -4.0020e-02,  0.0000e+00,
         1.4901e-08, -1.8268e-01,  0.0000e+00,  0.0000e+00,  4.8900e-02,
         0.0000e+00,  1.0996e-14, -7.3567e-15, -2.1176e-22,  6.1559e-16,
        -6.6174e-24, -3.8542e-16,  3.2069e-01,  0.0000e+00,  3.2617e-03,
        -9.3132e-10,  3.4941e-02, -2.4301e-03,  1.8626e-09, -5.6074e-03,
         0.0000e+00,  3.5846e-01, -4.0760e-01, -7.4506e-09,  2.4946e-02,
         0.0000e+00, -9.5954e-02,  1.0911e-01, -1.8626e-09, -6.6776e-03,
         0.0000e+00, -1.4811e-14,  0.0000e+00,  0.0000e+00, -1.4740e-14,
         0.0000e+00, -8.0173e-15, -6.6174e-24,  5.1913e-16,  0.0000e+00,
         0.0000e+00, -1.0014e-01,  0.0000e+00, -3.7848e-04,  4.6566e-10,
        -6.3709e-02,  0.0000e+00,  0.0000e+00, -9.0588e-02,  0.0000e+00,
        -3.4317e-02,  0.0000e+00, -4.6873e-01,  0.0000e+00,  0.0000e+00,
        -3.9661e-01,  0.0000e+00, -2.5369e-01, -9.3132e-10,  1.2547e-01,
         0.0000e+00,  0.0000e+00,  1.0617e-01,  0.0000e+00,  6.7908e-02,
         0.0000e+00, -1.4405e-14,  0.0000e+00,  1.8050e-15,  2.4006e-14,
         1.0588e-22, -2.1709e-15,  4.2352e-22, -3.0590e-15,  0.0000e+00,
         5.0489e-16, -1.3235e-23, -6.3265e-17,  4.1281e-01,  0.0000e+00,
         1.6720e-02, -1.3235e-23,  4.3126e-05,  0.0000e+00, -6.3026e-02,
         0.0000e+00,  2.2740e-03,  1.4574e-01, -4.6566e-10, -2.9480e-02,
         0.0000e+00, -1.3623e-02,  0.0000e+00, -4.5496e-01,  0.0000e+00,
         6.1771e-02,  5.3058e-01, -5.5879e-09, -5.9467e-02,  0.0000e+00,
        -9.6433e-02,  0.0000e+00,  1.2179e-01,  0.0000e+00, -1.6535e-02,
        -1.4203e-01, -1.3970e-09,  1.5918e-02,  3.7253e-09,  2.5814e-02])
```

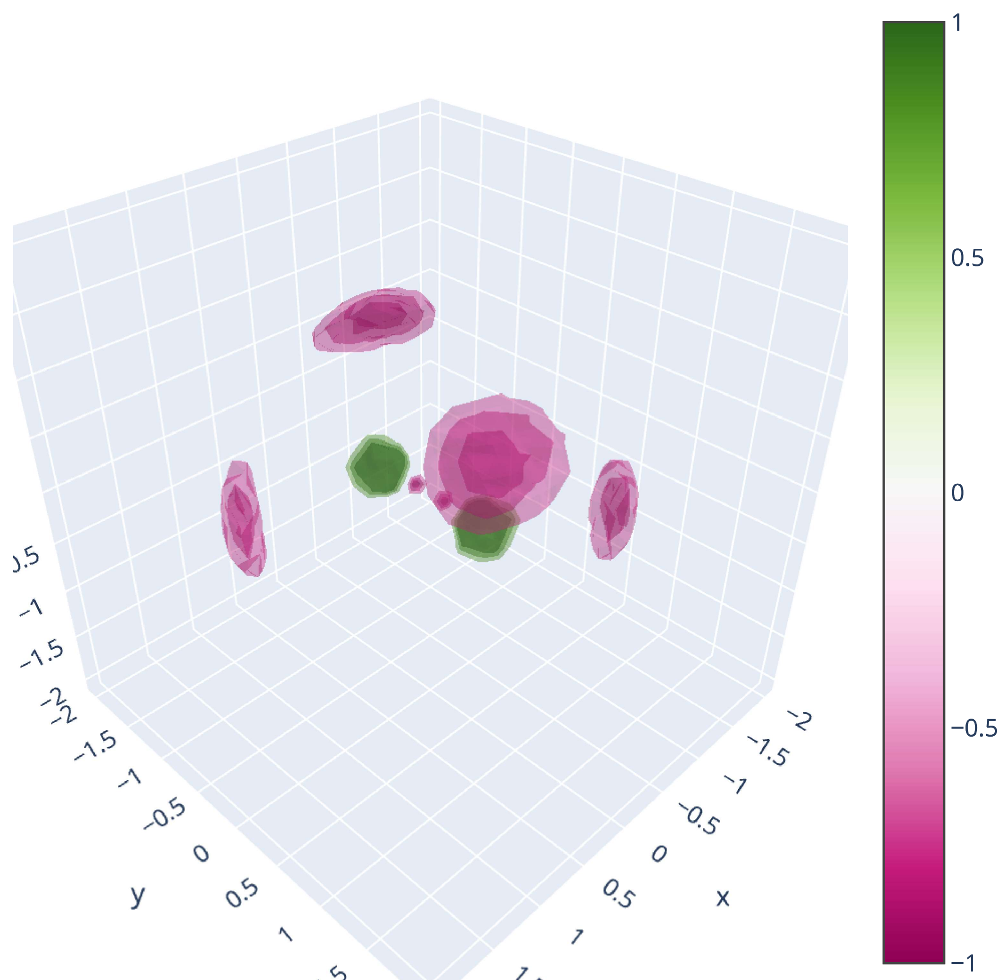
Now expand that signal into real-space. Compare to the molecule above. You should see positive peaks on heavy atoms, and negative peaks on hydrogen atoms.

In [6]:
```
samplePointsLinear, signalOnGrid = rst.signal_on_grid(originalSignal, radialCutoff, 25,
```

Plot the signal in real space.

In [7]:
```
X, Y, Z = np.meshgrid(samplePointsLinear, samplePointsLinear, samplePointsLinear, index
layout = go.Layout(width=500, height=500,
                   margin=dict(l=0, r=0, t=10, b=0),)

traceVolumePositive = go.Volume(
    x=X.flatten(),
    y=Y.flatten(),
    z=Z.flatten(),
    value=signalOnGrid.flatten(),
    isomin=.75,
    isomax=1.0,
    opacity=0.25,
    surface_count=3,
    caps= dict(x_show=False, y_show=False, z_show=False),
    cmin=-1,
    cmax=1,
    )

traceVolumeNegative = go.Volume(
```

```
    x=X.flatten(),
    y=Y.flatten(),
    z=Z.flatten(),
    value=signalOnGrid.flatten(),
    isomin=-1.0,
    isomax=-.75,
    opacity=0.25,
    surface_count=3,
    caps= dict(x_show=False, y_show=False, z_show=False),
    cmin=-1,
    cmax=1,
    )

fig = go.Figure(data=[traceVolumePositive, traceVolumeNegative], layout=layout)
fig.show()
```



OK, so we've taken a molecule and converted it into a signal. That signal can be expanded into real space.

Now let's define a function that takes a signal from our RadialSphericalTensor and turns it into a rotationally invariant power spectrum. In e3nn terms, that conversion is a tensor product.

In [8]:
```
signalToPowerSpectrum_tensorProduct = o3.ReducedTensorProducts(
    'ij=ji', i=rst,#o3.Irreps.spherical_harmonics(lmax),
    filter_ir_out=list(o3.Irrep.iterator(lmax=0)),
    filter_ir_mid=list(o3.Irrep.iterator(lmax=0)))
```

```
print(signalToPowerSpectrum_tensorProduct.irreps_out)
signalToPowerSpectrum = lambda x: signalToPowerSpectrum_tensorProduct(x, x)
```

```
75x0e
```

So the tensor product produces many scalar values, which we know are rotationally invariant.

Now let's evaluate the power spectrum for the specific signal associated with our molecule above.

In [9]:

```
originalPowerSpectrum = signalToPowerSpectrum(originalSignal)
originalPowerSpectrum = originalPowerSpectrum.detach().clone()
print(originalPowerSpectrum)
```

```
tensor([-2.6759e-02,  1.7311e-02, -3.1964e-02,  6.4652e-02,  1.5716e-02,
         1.6748e-02,  7.1911e-02,  3.7412e-02,  1.4191e-01,  4.8044e-02,
        -8.7590e-02,  4.0647e-02,  4.4059e-02, -6.7369e-02,  2.4805e-02,
         6.1585e-03, -4.4548e-02, -1.3550e-02,  3.1770e-03,  7.1823e-02,
         1.6164e-02, -2.6508e-02,  2.8364e-02, -1.6222e-02, -6.5237e-03,
        -5.6491e-03, -4.0209e-02, -1.5336e-02, -3.5219e-02,  2.8106e-02,
         7.7943e-03,  3.7046e-02, -3.5656e-02, -1.8536e-02,  5.0588e-03,
        -4.1698e-02,  3.7289e-02,  5.8041e-02, -5.8190e-02, -1.4725e-03,
        -8.9924e-03,  2.3940e-03,  4.5927e-03, -1.9954e-02, -2.0033e-03,
         8.2758e-03,  9.6907e-16,  1.4661e-02, -1.8457e-03, -6.4157e-03,
         1.4406e-16,  3.5692e-03, -8.5387e-02,  4.4846e-02,  3.4593e-02,
        -1.0726e-01, -3.8197e-02,  2.0379e-02, -5.4389e-02,  2.4841e-01,
        -9.8208e-03,  3.6688e-02, -5.6981e-02,  7.9939e-03,  4.2692e-02,
         6.0731e-03, -1.0986e-01, -1.0059e-02, -1.8626e-01,  2.8494e-02,
        -2.8853e-15,  1.0779e-14,  9.8688e-16, -2.9552e-15,  2.0502e-28])
```

Now, let's generate a perturbed configuration of the molecule and see if we can recover it using optimization.

In [10]:

```
originalPositions = torch.tensor(atoms.get_positions(), dtype=torch.float32)
perturbation = 0.25 * (2*torch.rand(originalPositions.shape) - 1.0)
perturbedPositions = originalPositions + perturbation
```

Set up an optimization problem. We're going to try to drive from perturbedPositions back to originalPositions by minimizing the difference in power spectra.

In [11]:

```
candidatePositions = perturbedPositions.clone().detach()
candidatePositions.requires_grad = True
loss_fn = torch.nn.MSELoss()
optimiser = torch.optim.Adam([candidatePositions], lr=.01)

def objectiveFunction(candidatePos):
    candidateSignal = rst.with_peaks_at(candidatePos, atomValues)
    candidatePowerSpectrum = signalToPowerSpectrum(candidateSignal)
    loss = loss_fn(originalPowerSpectrum, candidatePowerSpectrum)
    return loss
```

Aaaaaand optimize away.

In [12]:

```
for iteration in range(1000):
    def closure():
        optimiser.zero_grad()
        loss = objectiveFunction(candidatePositions)
```

```
        loss.backward()
        if iteration%10==0: print(loss)
        return loss

    optimiser.step(closure)
```

```
tensor(0.0013, grad_fn=<MseLossBackward>)
tensor(0.0006, grad_fn=<MseLossBackward>)
tensor(0.0004, grad_fn=<MseLossBackward>)
tensor(0.0003, grad_fn=<MseLossBackward>)
tensor(0.0003, grad_fn=<MseLossBackward>)
tensor(0.0003, grad_fn=<MseLossBackward>)
tensor(0.0002, grad_fn=<MseLossBackward>)
tensor(0.0002, grad_fn=<MseLossBackward>)
tensor(0.0002, grad_fn=<MseLossBackward>)
tensor(0.0002, grad_fn=<MseLossBackward>)
tensor(0.0002, grad_fn=<MseLossBackward>)
tensor(0.0001, grad_fn=<MseLossBackward>)
tensor(8.3522e-05, grad_fn=<MseLossBackward>)
tensor(5.4089e-05, grad_fn=<MseLossBackward>)
tensor(5.1126e-05, grad_fn=<MseLossBackward>)
tensor(4.1541e-05, grad_fn=<MseLossBackward>)
tensor(2.3973e-05, grad_fn=<MseLossBackward>)
tensor(8.7481e-06, grad_fn=<MseLossBackward>)
tensor(5.4754e-06, grad_fn=<MseLossBackward>)
tensor(3.1460e-06, grad_fn=<MseLossBackward>)
tensor(1.9989e-06, grad_fn=<MseLossBackward>)
tensor(1.5169e-06, grad_fn=<MseLossBackward>)
tensor(1.3040e-06, grad_fn=<MseLossBackward>)
tensor(1.1625e-06, grad_fn=<MseLossBackward>)
tensor(1.0595e-06, grad_fn=<MseLossBackward>)
tensor(9.8136e-07, grad_fn=<MseLossBackward>)
tensor(9.1590e-07, grad_fn=<MseLossBackward>)
tensor(8.5958e-07, grad_fn=<MseLossBackward>)
tensor(8.1056e-07, grad_fn=<MseLossBackward>)
tensor(7.6693e-07, grad_fn=<MseLossBackward>)
tensor(7.2743e-07, grad_fn=<MseLossBackward>)
tensor(6.9123e-07, grad_fn=<MseLossBackward>)
tensor(6.5776e-07, grad_fn=<MseLossBackward>)
tensor(6.2667e-07, grad_fn=<MseLossBackward>)
tensor(5.9768e-07, grad_fn=<MseLossBackward>)
tensor(5.7063e-07, grad_fn=<MseLossBackward>)
tensor(5.4539e-07, grad_fn=<MseLossBackward>)
tensor(5.2185e-07, grad_fn=<MseLossBackward>)
tensor(4.9988e-07, grad_fn=<MseLossBackward>)
tensor(4.7938e-07, grad_fn=<MseLossBackward>)
tensor(4.6028e-07, grad_fn=<MseLossBackward>)
tensor(4.4246e-07, grad_fn=<MseLossBackward>)
tensor(4.2586e-07, grad_fn=<MseLossBackward>)
tensor(4.1038e-07, grad_fn=<MseLossBackward>)
tensor(3.9593e-07, grad_fn=<MseLossBackward>)
tensor(3.8244e-07, grad_fn=<MseLossBackward>)
tensor(3.6982e-07, grad_fn=<MseLossBackward>)
tensor(3.5800e-07, grad_fn=<MseLossBackward>)
tensor(3.4692e-07, grad_fn=<MseLossBackward>)
tensor(3.3652e-07, grad_fn=<MseLossBackward>)
tensor(3.2674e-07, grad_fn=<MseLossBackward>)
tensor(3.1753e-07, grad_fn=<MseLossBackward>)
tensor(3.0884e-07, grad_fn=<MseLossBackward>)
tensor(3.0062e-07, grad_fn=<MseLossBackward>)
```

```
tensor(2.9283e-07, grad_fn=<MseLossBackward>)
tensor(2.8542e-07, grad_fn=<MseLossBackward>)
tensor(2.7838e-07, grad_fn=<MseLossBackward>)
tensor(2.7167e-07, grad_fn=<MseLossBackward>)
tensor(2.6526e-07, grad_fn=<MseLossBackward>)
tensor(2.5913e-07, grad_fn=<MseLossBackward>)
tensor(2.5326e-07, grad_fn=<MseLossBackward>)
tensor(2.4763e-07, grad_fn=<MseLossBackward>)
tensor(2.4221e-07, grad_fn=<MseLossBackward>)
tensor(2.3700e-07, grad_fn=<MseLossBackward>)
tensor(2.3197e-07, grad_fn=<MseLossBackward>)
tensor(2.2712e-07, grad_fn=<MseLossBackward>)
tensor(2.2244e-07, grad_fn=<MseLossBackward>)
tensor(2.1790e-07, grad_fn=<MseLossBackward>)
tensor(2.1352e-07, grad_fn=<MseLossBackward>)
tensor(2.0927e-07, grad_fn=<MseLossBackward>)
tensor(2.0515e-07, grad_fn=<MseLossBackward>)
tensor(2.0114e-07, grad_fn=<MseLossBackward>)
tensor(1.9726e-07, grad_fn=<MseLossBackward>)
tensor(1.9349e-07, grad_fn=<MseLossBackward>)
tensor(1.8981e-07, grad_fn=<MseLossBackward>)
tensor(1.8624e-07, grad_fn=<MseLossBackward>)
tensor(1.8276e-07, grad_fn=<MseLossBackward>)
tensor(1.7937e-07, grad_fn=<MseLossBackward>)
tensor(1.7607e-07, grad_fn=<MseLossBackward>)
tensor(1.7285e-07, grad_fn=<MseLossBackward>)
tensor(1.6971e-07, grad_fn=<MseLossBackward>)
tensor(1.6664e-07, grad_fn=<MseLossBackward>)
tensor(1.6366e-07, grad_fn=<MseLossBackward>)
tensor(1.6074e-07, grad_fn=<MseLossBackward>)
tensor(1.5788e-07, grad_fn=<MseLossBackward>)
tensor(1.5510e-07, grad_fn=<MseLossBackward>)
tensor(1.5238e-07, grad_fn=<MseLossBackward>)
tensor(1.4971e-07, grad_fn=<MseLossBackward>)
tensor(1.4711e-07, grad_fn=<MseLossBackward>)
tensor(1.4456e-07, grad_fn=<MseLossBackward>)
tensor(1.4206e-07, grad_fn=<MseLossBackward>)
tensor(1.3963e-07, grad_fn=<MseLossBackward>)
tensor(1.3723e-07, grad_fn=<MseLossBackward>)
tensor(1.3490e-07, grad_fn=<MseLossBackward>)
tensor(1.3260e-07, grad_fn=<MseLossBackward>)
tensor(1.3035e-07, grad_fn=<MseLossBackward>)
tensor(1.2815e-07, grad_fn=<MseLossBackward>)
tensor(1.2599e-07, grad_fn=<MseLossBackward>)
tensor(1.2388e-07, grad_fn=<MseLossBackward>)
tensor(1.2180e-07, grad_fn=<MseLossBackward>)
```

Well that seemed to go well. Let's rotate our candidatePositions in an attempt to line it up with the originalPositions.

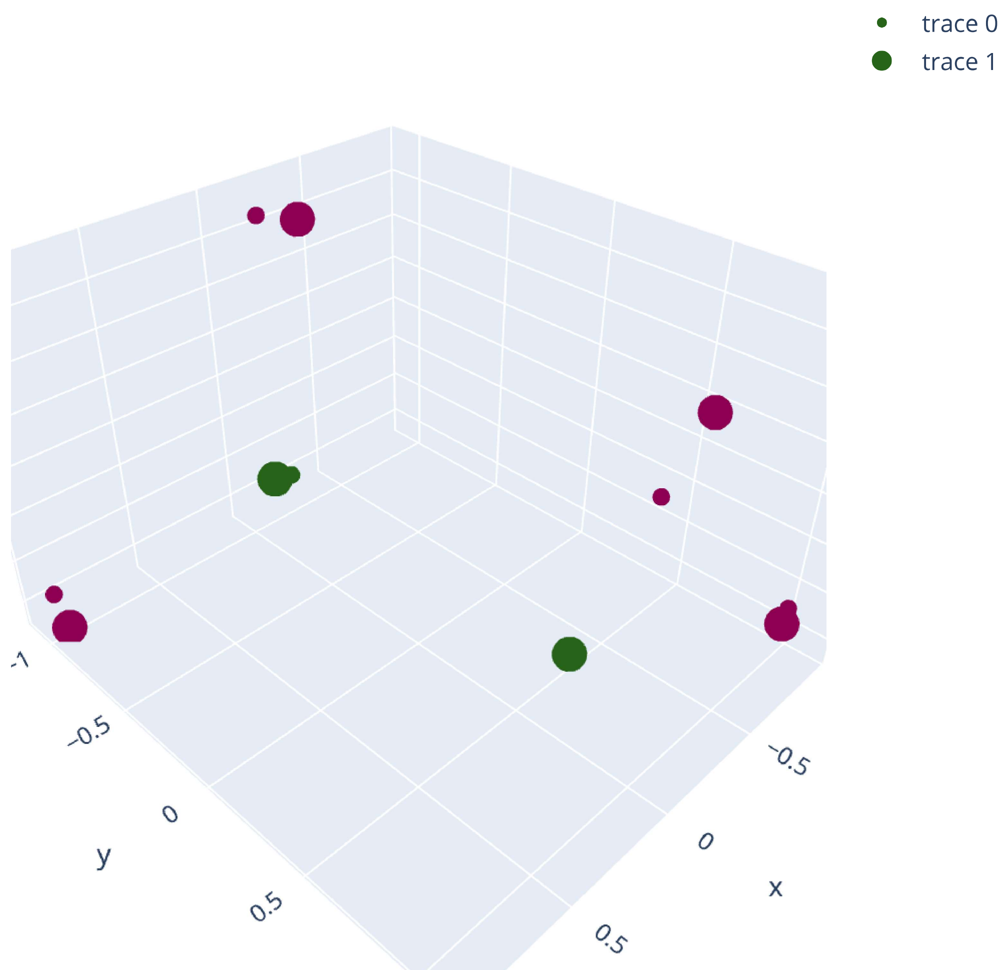
In [13]:

```
originalU,_,_ = torch.svd(torch.transpose(originalPositions,0,1))
_,candidateS,candidateV = torch.svd(torch.transpose(candidatePositions,0,1))
candidatePositionsRegistered = torch.transpose(torch.matmul(torch.matmul(originalU, tor
```

In [14]:

```
candidateSignal = rst.with_peaks_at(candidatePositions, atomValues)
candidatePowerSpectrum = signalToPowerSpectrum(candidateSignal)
```

In [15]:
```
layout = go.Layout(width=500, height=500,
                   margin=dict(l=0, r=0, t=10, b=0),)

traceOriginal = go.Scatter3d(
    x=originalPositions[:,0],
    y=originalPositions[:,1],
    z=originalPositions[:,2],
    mode='markers',
    marker=dict(size=5,color=atomValues),
    )

candidatePositions = candidatePositions.detach().clone()
traceCandidate = go.Scatter3d(
    x=candidatePositions[:,0],
    y=candidatePositions[:,1],
    z=candidatePositions[:,2],
    mode='markers',
    marker=dict(size=10,color=atomValues),
    )

fig = go.Figure(data=[traceOriginal, traceCandidate], layout=layout)
fig.show()
```



Interesting, these two configurations are not as close as we might have hoped. We seem to have recovered a separate configuration that has the same power spectrum as our original molecule.

Let's take a look at the Hessian of the objective function. Now, we know that at originalPositions, the

objective function and its gradient are zero, so the null space of the Hessian tells us about the "shape" of the minimum/minimal manifold. We expect the Hessian to have at least three null singular values, corresponding to orientational degrees of freedom; that is, because the power spectrum is rotationally invariant, there is a manifold of configurations related to each other by rotations that have the same power spectrum.

In [16]:

```
hessian = torch.autograd.functional.hessian(objectiveFunction, originalPositions)
hessianFlat = hessian.view(3*len(atoms),3*len(atoms))
_,S,_ = torch.svd(hessianFlat)
nullity = torch.sum((S/torch.max(S)) < 1e-6).item()
print(nullity)
```

```
10
```

Wow, that's a lot more than three. So there is a ten-dimensional manifold of configurations that have the same power spectrum as our molecule above. This is the reason why we don't often use the power spectrum as an invariant descriptor-- many configurations map to the same power spectrum!

In [ ]: