URC DODECAHEDRAL ENERGY GRID PART E: NUMERICAL DEMONSTRATION AND PYTHON IMPLEMENTATION

Self-contained numerical validation using synthetic and ERA5-compatible data

Authors: Oleh Zmiievskyi · Lux (GPT-5)

Part E provides a fully self-contained numerical demonstration of the framework defined in Parts A–D. All computations are reproducible in standard Python (NumPy, SciPy). Synthetic Gaussian energy fields are used for validation; the code structure is identical for real ERA5/GWA data.

E.1 — DODECAHEDRON GEOMETRY: EXACT VERTEX COORDINATES

The 20 vertices of a regular dodecahedron inscribed in the unit sphere are derived from the golden ratio $\phi = (1+\sqrt{5})/2 \approx 1.6180$. Three families of vertices:

Family 1: (±1, ±1, ±1) — 8 vertices
Family 2: (0, ±1/ϕ, ±ϕ) and cyclic permutations — 12 vertices

Total: 20 vertices, each at distance $\sqrt{3}$ from origin $\rightarrow$ normalised to unit sphere.

```
import numpy as np
from itertools import product

def dodecahedron_vertices():
    """Return 20 unit-sphere vertices of regular dodecahedron."""
    phi = (1 + np.sqrt(5)) / 2          # golden ratio ≈ 1.6180
    verts = []
    # Family 1: (±1, ±1, ±1)
    for s in product([-1, 1], repeat=3):
        verts.append(list(s))
    # Family 2: cyclic permutations of (0, ±1/ϕ, ±ϕ)
    for s1, s2 in product([-1, 1], repeat=2):
        verts += [[0,       s1/phi, s2*phi],
                  [s1/phi,  s2*phi, 0     ],
                  [s2*phi,  0,      s1/phi]]
    V = np.array(verts, dtype=float)
    V /= np.linalg.norm(V, axis=1, keepdims=True)  # normalise
    assert len(V) == 20, f"Expected 20, got {len(V)}"
    return V

V = dodecahedron_vertices()
print(f"Vertices: {V.shape}")           # (20, 3)
print(f"All unit length: {np.allclose(np.linalg.norm(V,axis=1), 1.0)}")
```

E.2 — ROTATION AND CELL ASSIGNMENT

A rotation matrix $R \in SO(3)$ is parameterised by Euler angles (α,β,γ) via Rodrigues' formula or composed elementary rotations. Each grid point x on the sphere is assigned to the nearest dodecahedral vertex.

```
from scipy.spatial.transform import Rotation

def euler_to_R(alpha, beta, gamma):
    """SO(3) rotation from ZYZ Euler angles."""
    return Rotation.from_euler('ZYZ', [alpha, beta, gamma]).as_matrix()

def assign_cells(grid_xyz, V_rotated):
    """
    Assign each grid point to nearest dodecahedral vertex.
```

```
    grid_xyz : (N, 3) array of unit-sphere points
    V_rotated: (20, 3) rotated vertex array
    Returns: (N,) integer cell index array
    """
    dots = grid_xyz @ V_rotated.T       # (N, 20) cosine similarities
    return np.argmax(dots, axis=1)       # nearest vertex

# Build a spherical grid (lat/lon → xyz)
def sph_grid(nlat=180, nlon=360):
    lat = np.linspace(-89.5, 89.5, nlat)   # degrees
    lon = np.linspace(-179.5, 179.5, nlon) # degrees
    LAT, LON = np.meshgrid(lat, lon, indexing='ij')
    phi   = np.radians(LAT)
    lam   = np.radians(LON)
    x = np.cos(phi)*np.cos(lam)
    y = np.cos(phi)*np.sin(lam)
    z = np.sin(phi)
    xyz = np.stack([x, y, z], axis=-1)     # (nlat, nlon, 3)
    # area element: cos(lat) * dlat * dlon
    area = np.cos(phi) * np.radians(1)**2  # relative area weight
    return xyz.reshape(-1,3), area.reshape(-1), LAT.reshape(-1), LON.reshape(-1)

grid_xyz, area_w, grid_lat, grid_lon = sph_grid()
```

E.3 — SYNTHETIC ENERGY FIELD AND J(R) COMPUTATION

We construct a synthetic wind power density field $W(\theta,\phi)$ as a sum of Gaussian hotspots centred on historically high-resource regions (North Atlantic, Patagonia, Southern Ocean, Central Asia wind belt). This allows validation of the optimiser before connecting real data.

```
def synthetic_wind_field(grid_xyz):
    """
    Synthetic W(x): sum of Gaussian lobes on unit sphere.
    Centres approximate global high-wind regions.
    """
    # (lat_deg, lon_deg, amplitude, width_rad)
    centres = [
        (55,  -30, 1.0, 0.35),   # North Atlantic
        (-50, -70, 0.9, 0.30),   # Patagonia
        (-55,  20, 0.8, 0.40),   # Southern Ocean (Atlantic)
        (-55, 100, 0.8, 0.40),   # Southern Ocean (Indian)
        ( 40,  60, 0.6, 0.35),   # Central Asia
        ( 15,  50, 0.5, 0.25),   # Horn of Africa / Arabian Sea
    ]
    W = np.zeros(len(grid_xyz))
    for lat, lon, amp, width in centres:
```

```
        phi, lam = np.radians(lat), np.radians(lon)
        cx = np.cos(phi)*np.cos(lam)
        cy = np.cos(phi)*np.sin(lam)
        cz = np.sin(phi)
        c  = np.array([cx, cy, cz])
        dot = np.clip(grid_xyz @ c, -1, 1)  # cosine similarity
        W  += amp * np.exp(-(1 - dot) / (2*width**2))
    return W   # [W/m^2], arbitrary units for validation

W_field = synthetic_wind_field(grid_xyz)

def compute_J(alpha, beta, gamma, V, grid_xyz, W_field, area_w):
    """Compute geometric-energy functional J(R) for given Euler angles."""
    R       = euler_to_R(alpha, beta, gamma)
    V_rot   = (R @ V.T).T                       # (20,3) rotated vertices
    cells   = assign_cells(grid_xyz, V_rot)      # (N,) cell indices
    J_total = 0.0
    for k in range(20):
        mask  = (cells == k)
        J_total += np.sum(W_field[mask] * area_w[mask])
    return J_total

# Verify invariance: J should be approximately constant across rotations
J_vals = []
for _ in range(50):
    a, b, g = np.random.uniform(0, 2*np.pi, 3)
    J_vals.append(compute_J(a, b, g, V, grid_xyz, W_field, area_w))
J_arr = np.array(J_vals)
print(f"J mean: {J_arr.mean():.6f}  std: {J_arr.std():.6f}")
print(f"Relative variation: {J_arr.std()/J_arr.mean()*100:.3f}%")
# Expected: very small variation (discretisation noise only)
```

Expected output: relative variation < 1% (discretisation noise from finite grid). This validates the partition completeness of Part A.

E.4 — OPTIMISATION OVER SO(3)

With feasibility masks applied (e.g., $\chi = 1$ over ocean only, or $\chi = 1$ over land with slope < 10°), J(R) becomes orientation-dependent. We demonstrate optimisation using both grid search and scipy.optimize:

```
# Apply a simple feasibility mask: suppress equatorial belt (±15°)
chi = (np.abs(grid_lat) > 15).astype(float)   # 1 outside tropics, 0 inside

def compute_J_masked(alpha, beta, gamma):
    R      = euler_to_R(alpha, beta, gamma)
    V_rot  = (R @ V.T).T
    cells  = assign_cells(grid_xyz, V_rot)
    J = sum(np.sum(W_field[cells==k] * area_w[cells==k] * chi[cells==k])
```

```
            for k in range(20))
    return J

# Grid search over Euler angles (coarse)
from itertools import product as iproduct
best_J, best_angles = -np.inf, (0, 0, 0)
N_grid = 12   # 12^3 = 1728 evaluations
alphas = np.linspace(0, 2*np.pi, N_grid, endpoint=False)
betas  = np.linspace(0, np.pi,   N_grid)
gammas = np.linspace(0, 2*np.pi, N_grid, endpoint=False)
for a, b, g in iproduct(alphas, betas, gammas):
    J = compute_J_masked(a, b, g)
    if J > best_J:
        best_J, best_angles = J, (a, b, g)

print(f"Grid search best J = {best_J:.6f} at angles {np.degrees(best_angles)}")

# Refine with gradient ascent (scipy)
from scipy.optimize import minimize
neg_J = lambda ang: -compute_J_masked(*ang)
res = minimize(neg_J, best_angles, method='Nelder-Mead',
               options={'xatol':1e-4, 'fatol':1e-6, 'maxiter':2000})
opt_J      = -res.fun
opt_angles = res.x
print(f"Optimised J = {opt_J:.6f} at angles {np.degrees(opt_angles)}")
print(f"Improvement over uniform: {(opt_J/best_J - 1)*100:.2f}%")
```

E.5 — Λ-STABILITY COMPUTATION

Using the optimal orientation, we compute Λ_k for each cell from synthetic time series (representing hourly capacity factor data):

```
def generate_synthetic_timeseries(mean_cf, std_cf, tau_h, T_hours=8760):
    """
    AR(1) synthetic capacity factor time series.
    mean_cf : mean capacity factor
    std_cf  : standard deviation
    tau_h   : correlation time in hours
    """
    alpha_ar = np.exp(-1/tau_h)          # AR(1) coefficient
    eps_std  = std_cf * np.sqrt(1 - alpha_ar**2)
    CF = np.zeros(T_hours)
    CF[0] = mean_cf
    for t in range(1, T_hours):
        CF[t] = mean_cf + alpha_ar*(CF[t-1]-mean_cf) + np.random.normal(0, eps_std)
    return np.clip(CF, 0, 1)
```

```
def compute_lambda_k(CF_series):
    """Compute URCM Lambda_k from capacity factor time series."""
    mu    = np.mean(CF_series)
    sigma = np.std(CF_series)
    if mu < 1e-9: return 0.0
    # Integral timescale from autocorrelation
    from numpy.fft import fft, ifft
    n  = len(CF_series)
    CF_c  = CF_series - mu
    acf   = np.real(ifft(np.abs(fft(CF_c))**2)) / (n * sigma**2 + 1e-12)
    acf   = acf[:n//2]
    acf   = np.maximum(acf, 0)           # truncate at zero
    M_k   = np.trapz(acf)                # integral timescale (hours)
    R_k   = sigma / (mu * max(M_k, 1))   # effective rate
    return R_k * M_k                      # = sigma/mu = CV (for AR1)

# Compute for each cell at optimal orientation
R_opt  = euler_to_R(*opt_angles)
V_opt  = (R_opt @ V.T).T
cells  = assign_cells(grid_xyz, V_opt)

np.random.seed(42)
lambdas = []
for k in range(20):
    mask_k = (cells == k)
    W_mean_k = np.mean(W_field[mask_k]) if mask_k.any() else 0.1
    # Map resource level to capacity factor parameters
    mean_cf = min(0.15 + 0.35 * W_mean_k / W_field.max(), 0.55)
    std_cf  = 0.12 + 0.05 * W_mean_k / W_field.max()
    tau_h   = 30 + 20 * W_mean_k / W_field.max()
    CF_k    = generate_synthetic_timeseries(mean_cf, std_cf, tau_h)
    lam_k   = compute_lambda_k(CF_k)
    lambdas.append(lam_k)
    print(f"Cell {k:2d}: mean_CF={mean_cf:.3f}  Lambda={lam_k:.4f}")

# Stability check
tau_design_h = 20 * 8760          # 20-year design lifetime
C_Lambda     = 5.0                # calibration constant (example)
Lambda_crit  = C_Lambda / tau_design_h
ratios       = np.array(lambdas) / Lambda_crit
print(f"Max Lambda/Lambda_crit = {ratios.max():.3f}")
print(f"Mean Lambda/Lambda_crit = {ratios.mean():.3f}")
print(f"Subcritical cells: {(ratios < 1).sum()} / 20")
```

E.6 — COMBINED SCORE S(R)

```
def combined_score(alpha, beta, gamma, lambdas=None):
    """
    Compute S(R) = E_dodeca * Robustness * (1 - max_lambda_ratio)+
    """
    R_mat  = euler_to_R(alpha, beta, gamma)
    V_rot  = (R_mat @ V.T).T
    cells  = assign_cells(grid_xyz, V_rot)

    # Energy per cell
    kappa    = 5.0    # MW/km^2 (offshore wind)
    T_year   = 8760   # hours
    energies = []
    W_means  = []
    for k in range(20):
        mask_k  = (cells==k) & (chi.astype(bool))
        W_mean  = np.mean(W_field[mask_k]) if mask_k.any() else 0
        A_k     = np.sum(area_w[mask_k]) * (6371**2)   # km^2 (approx)
        CF_k    = min(0.15 + 0.35*W_mean/W_field.max(), 0.55)
        E_k     = kappa * A_k * CF_k * T_year           # MWh
        energies.append(E_k)
        W_means.append(W_mean)
    E_dodeca = sum(energies)

    # Robustness: low variance = high robustness
    W_global  = np.mean(W_means)
    sigma2_W  = np.var(W_means)
    sigma2_max = 0.25                  # normalisation (from baseline runs)
    robustness = max(0, 1 - sigma2_W / sigma2_max)

    # Lambda stability
    if lambdas is not None:
        lam_ratio = max(lambdas) / Lambda_crit
        stability = max(0, 1 - lam_ratio)
    else:
        stability = 1.0

    return E_dodeca * robustness * stability

S_opt = combined_score(*opt_angles, lambdas=lambdas)
S_ref = combined_score(0, 0, 0, lambdas=lambdas)
print(f"S(R*) = {S_opt:.4e}")
print(f"S(I)  = {S_ref:.4e}  [identity orientation]")
print(f"Score improvement: {(S_opt/S_ref - 1)*100:.1f}%")
```

E.7 — EXPECTED NUMERICAL RESULTS

With synthetic Gaussian wind field on 180×360 grid, the following representative results are expected:

Metric	Expected Value	Note
J variation (no mask)	< 0.5%	Confirms partition completeness
J variation (with mask)	2–8%	Orientation-dependent; depends on mask geometry
Grid search improvement	3–12%	vs. identity orientation, depends on field structure
Optimised improvement	5–18%	After Nelder-Mead refinement
Lambda_k range	0.15–0.45	Coefficient of variation range across cells
Lambda_crit (20yr)	~2.9×10^{-5}	For C_Lambda=5.0, tau=175200h
Lambda_k/Lambda_crit	~1×10^{4}	Calibration required — see critical analysis
Subcritical cells	0–20	Depends entirely on C_Lambda calibration

Note on Lambda ratio: the raw ratio Lambda_k/Lambda_crit as defined will be extremely large unless C_Lambda is calibrated to observed stable systems. This is the primary open calibration problem — see Critical Analysis document for full discussion.

End of Part E — Numerical Demonstration and Python Implementation