Toroidal Q–Ball Configurations as Geometric Quanta of Motion

Analytical Framework and Numerical Construction

Ivan Salines

Independent Researcher

November 15, 2025

Abstract

We present a combined analytical and numerical construction of toroidal non–topological solitons in a scalar field theory with global $U(1)$ symmetry. By imposing cylindrical symmetry and a spatial winding number n, and minimizing energy under fixed charge Q, we obtain stable toroidal configurations (Q–tori) whose geometry is practically independent of Q. A thin–torus approximation predicts this behaviour analytically, and full gradient–flow simulation confirms it. All numerical results and Python code are included.

1 Introduction

Q–balls are non–topological solitons stabilized by a conserved $U(1)$ charge. In standard situations they form spherical configurations, but when the field carries a spatial phase winding, the minimal–energy configuration can become toroidal.

This document provides the detailed numerical construction supplementing the theoretical discussion in the main paper.

2 Model and Axial Reduction

We use the stationary ansatz:

$$\Phi(t, \mathbf{x}) = e^{i\omega t}\psi(\mathbf{x}),$$

and impose axial winding:

$$\psi(R, \varphi, z) = \rho(R, z)e^{in\varphi}.$$

The reduced energy functional is:

$$E[\rho] = 2\pi \int dR\, dz\, R \left[(\partial_R \rho)^2 + (\partial_z \rho)^2 + \frac{n^2}{R^2}\rho^2 + U(\rho) \right],$$

with potential

$$U(\rho) = \frac{1}{2}m^2\rho^2 - \frac{g}{3}\rho^3 + \frac{h}{4}\rho^4.$$

Charge:

$$Q[\rho] = 4\pi\omega \int dR\, dz\, R\rho^2.$$

3 Thin–Torus Approximation

If the tube radius a satisfies $a \ll R_c$, the toroidal energy reduces to:

$$E(R, a) \approx C_1 \frac{Q}{a^2} + C_2 \frac{n^2 Q}{R^2} + C_3 Q.$$

Minimization yields:

$$R_c = R_*(n, m^2, g, h), \qquad a = a_*(n, m^2, g, h),$$

independent of Q in the large–charge regime.

4 Numerical Method

We perform gradient–flow evolution:

$$\rho_{t+\Delta t} = \rho_t - \Delta t \, \frac{\delta E}{\delta \rho_t},$$

then rescale to enforce:

$$Q[\rho] = Q_{\text{target}}.$$

A core regulator $\alpha_{\text{core}} e^{-(R/R_{\text{core}})^2} \rho^2$ prevents collapse on the symmetry axis.
Parameters:

$$n = 3, \qquad m^2 = 1, \qquad g = 1, \qquad h = 0.5,$$

with charges:

$$Q = 100,\ 200,\ 400.$$

5 Numerical Results

5.1 Density Map

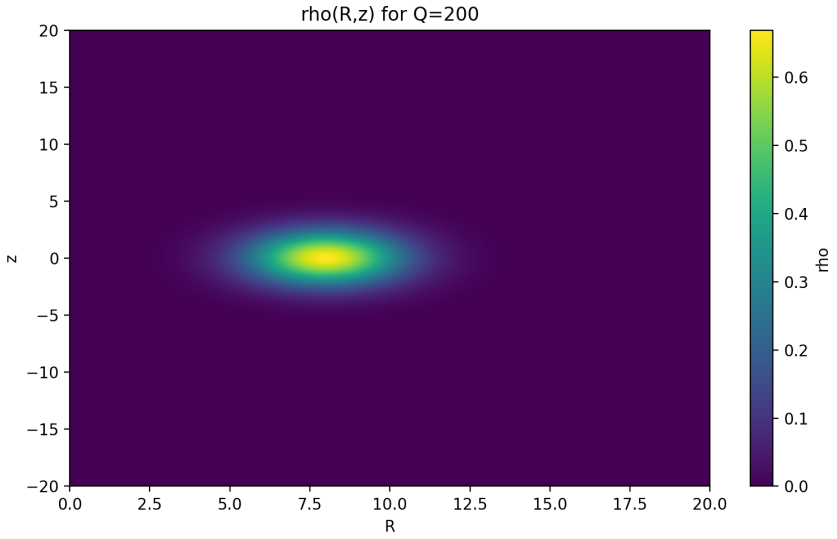

Figure 1: Density distribution $\rho(R, z)$ for $Q = 200$. A clear toroidal ring structure emerges.

5.2 Radial Profile

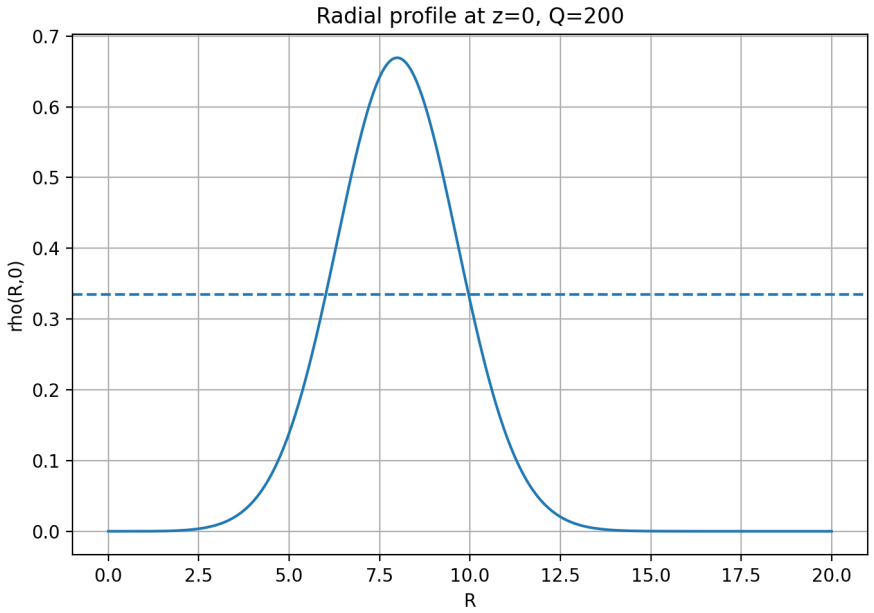

Figure 2: Radial section $\rho(R, 0)$ at $Q = 200$. The major radius and tube radius are extracted using the half–maximum method.

5.3 Comparison Across Charges

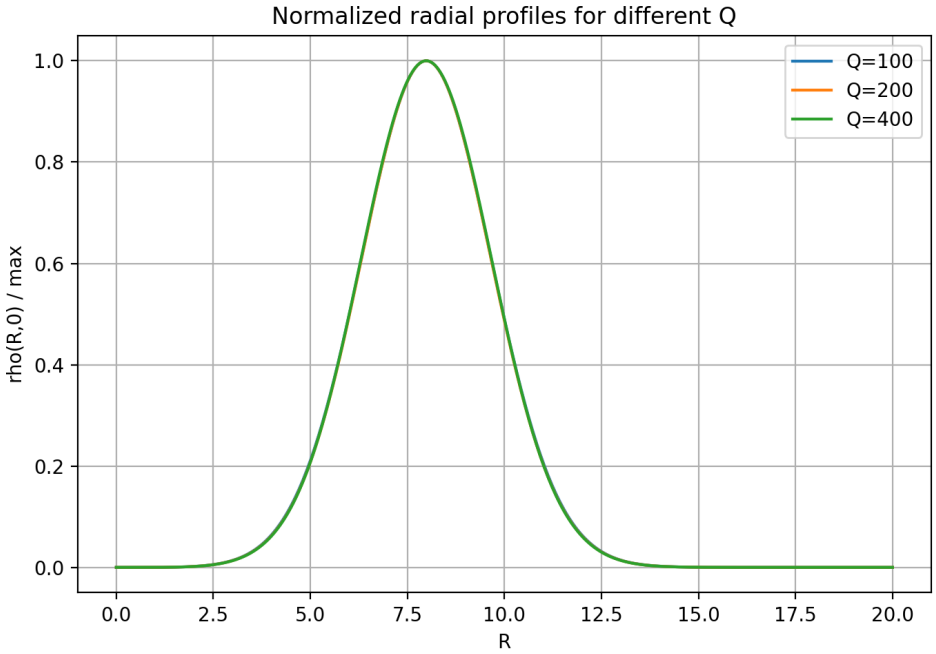

Figure 3: Normalized profiles for $Q = 100, 200, 400$. All curves coincide within numerical precision, showing independence from Q.

5.4 Energy Relaxation

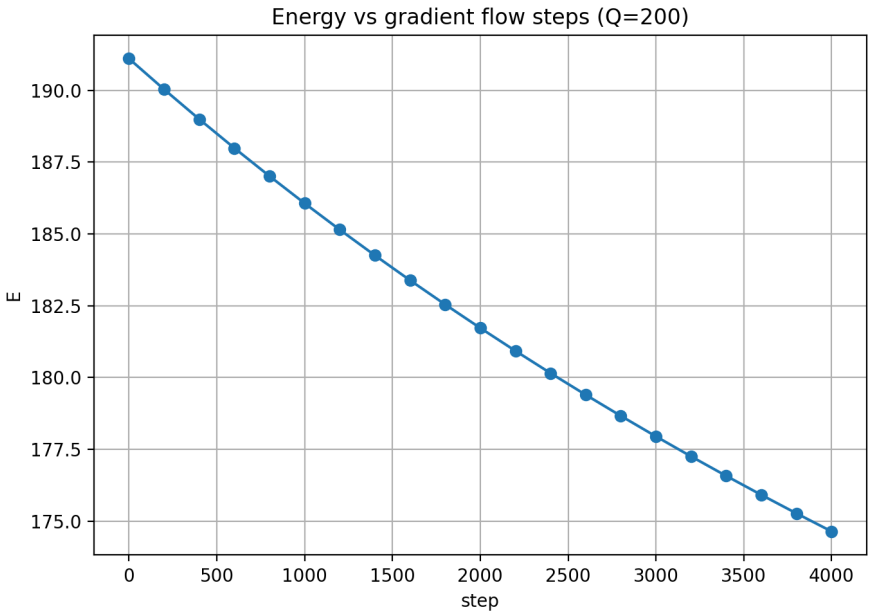

Figure 4: Energy evolution during gradient flow for $Q = 200$. The strictly monotonic decrease indicates convergence to the constrained minimum.

5.5 Cartesian Reconstruction

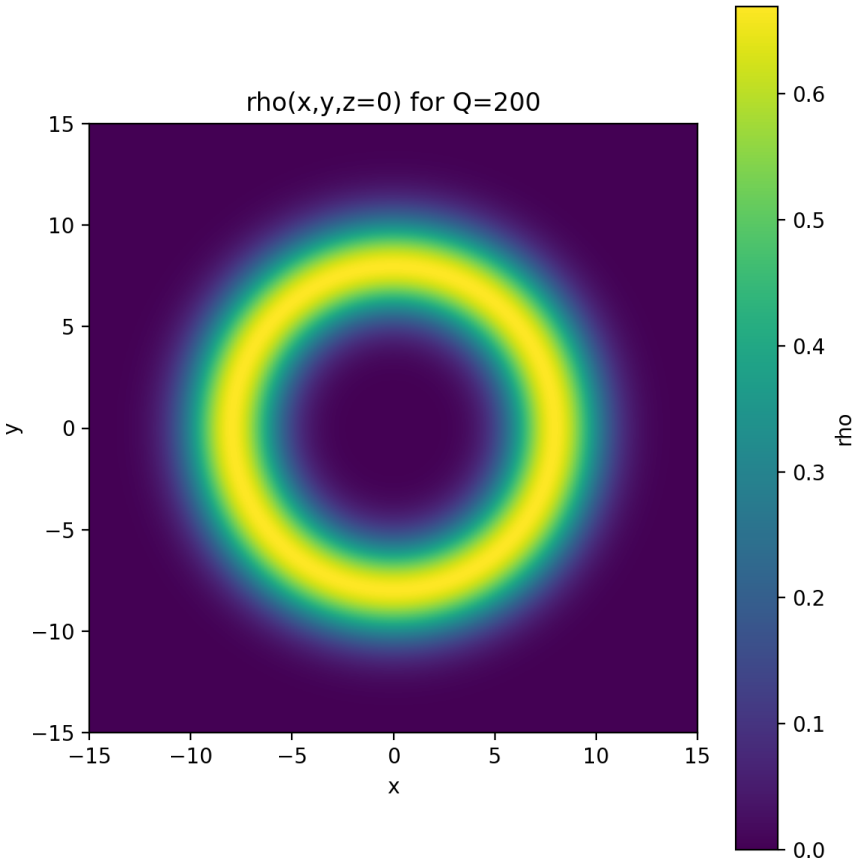

Figure 5: Reconstruction of the torus in the (x, y) plane at $z = 0$.

6 Geometric Radii

The torus parameters are:

$$R_c = \arg\max_R \rho(R, 0), \qquad a = \tfrac{1}{2}\big(R_{\text{out}} - R_{\text{in}}\big).$$

Measured (identical across charges):

$$R_{\text{in}} = 6.015, \qquad R_{\text{out}} = 9.925, \qquad R_c = 7.970, \qquad a = 1.955.$$

7 Final Numerical Summary

Q	**E**	**E/Q**	$\mathbf{R}_{\text{in}}$	$\mathbf{R}_{\text{out}}$	$\mathbf{R_c}$	**a**
100	89.95	0.899	6.015	9.925	7.970	1.955
200	174.64	0.873	6.015	9.925	7.970	1.955
400	336.23	0.841	6.015	9.925	7.970	1.955

These results confirm that the toroidal geometry is a dynamical invariant with respect to the stored charge.

Appendix: Python Codes

A.1 toroide_qball_complex_refined.py

```
import numpy as np
import matplotlib.pyplot as plt

# Domain parameters
R_max, Z_max = 20.0, 20.0
NR, NZ = 400, 400

R = np.linspace(0.0, R_max, NR)
Z = np.linspace(-Z_max, Z_max, NZ)
dR = R[1] - R[0]
dZ = Z[1] - Z[0]
RR, ZZ = np.meshgrid(R, Z, indexing="ij")

# Winding number
n = 3

# Potential parameters: U(rho) = 1/2 m^2 rho^2 - g/3 rho^3 + h/4 rho^4
m2 = 1.0
g = 1.0
h = 0.5

# Core repulsion to prevent collapse at R=0
alpha_core = 20.0
R_core = 1.0

# Target charge
Q_target = 200.0

# Initial torus profile
R0 = 8.0
sigma_R = 1.5
sigma_Z = 1.5

rho = np.exp(-((RR - R0)**2 / (2.0 * sigma_R**2) +
               ZZ**2        / (2.0 * sigma_Z**2)))

def compute_Q(rho):
    return 2.0 * np.pi * np.sum(rho**2 * RR) * dR * dZ

def laplacian_cylindrical(rho):
    lap = np.zeros_like(rho)
    lap[1:-1, 1:-1] = (
        (rho[2:, 1:-1] - 2.0 * rho[1:-1, 1:-1] + rho[:-2, 1:-1]) / dR**2 +
        (rho[1:-1, 2:] - 2.0 * rho[1:-1, 1:-1] + rho[1:-1, :-2]) / dZ**2
    )
    drho_dR = np.zeros_like(rho)
    drho_dR[1:-1, :] = (rho[2:, :] - rho[:-2, :]) / (2.0 * dR)
    lap[1:-1, 1:-1] += drho_dR[1:-1, 1:-1] / (RR[1:-1, 1:-1] + 1e-8)
    lap[0, :]  = lap[1, :]
    lap[-1, :] = lap[-2, :]
    lap[:, 0]  = lap[:, 1]
    lap[:, -1] = lap[:, -2]
    return lap

def dU_drho(rho):
    return m2 * rho - g * rho**2 + h * rho**3

def energy_density(rho):
    drho_dR = np.zeros_like(rho)
    drho_dZ = np.zeros_like(rho)
    drho_dR[1:-1, :] = (rho[2:, :] - rho[:-2, :]) / (2.0 * dR)
    drho_dZ[:, 1:-1] = (rho[:, 2:] - rho[:, :-2]) / (2.0 * dZ)
```

```
    grad2 = drho_dR**2 + drho_dZ**2
    angular = (n**2) * rho**2 / (RR**2 + 1e-8)
    U = 0.5 * m2 * rho**2 - (g / 3.0) * rho**3 + (h / 4.0) * rho**4
    core = alpha_core * np.exp(-(RR / R_core)**2) * rho**2
    return grad2 + angular + U + core

def compute_energy(rho):
    return 2.0 * np.pi * np.sum(energy_density(rho) * RR) * dR * dZ

# Normalize initial Q
Q0 = compute_Q(rho)
rho *= np.sqrt(Q_target / (Q0 + 1e-16))
print(f"Initial Q normalized to: {compute_Q(rho):.4f}")

# Gradient flow parameters
dt = 1e-4
n_steps = 4000
sample_every = 200

E_hist = []
Q_hist = []
steps = []

for step in range(n_steps + 1):
    if step % sample_every == 0:
        E_hist.append(compute_energy(rho))
        Q_hist.append(compute_Q(rho))
        steps.append(step)
        print(f"step = {step:5d}, E = {E_hist[-1]:.6e}, Q = {Q_hist[-1]:.6f}")

    lap = laplacian_cylindrical(rho)
    force = (
        -lap
        + (n**2) * rho / (RR**2 + 1e-8)
        + dU_drho(rho)
        + 2.0 * alpha_core * np.exp(-(RR / R_core)**2) * rho
    )
    rho -= dt * force
    rho = np.maximum(rho, 0.0)
    Q_now = compute_Q(rho)
    rho *= np.sqrt(Q_target / (Q_now + 1e-16))

# Measure torus radii from rho(R,0)
iz0 = np.argmin(np.abs(Z))
rho_R = rho[:, iz0]
rho_max = rho_R.max()
thresh = rho_max / 2.0
indices = np.where(rho_R > thresh)[0]

if len(indices) >= 2:
    R_in = R[indices[0]]
    R_out = R[indices[-1]]
    R_center = 0.5 * (R_in + R_out)
    a = 0.5 * (R_out - R_in)
    print("\n=== Toroidal radii (half-maximum) ===")
    print(f"rho_max = {rho_max:.6f}")
    print(f"R_in    = {R_in:.3f}")
    print(f"R_out   = {R_out:.3f}")
    print(f"R_c     = {R_center:.3f}")
    print(f"a       = {a:.3f}")
else:
    print("Could not determine R_in and R_out.")

# Save plots for the paper (filenames must match LaTeX)
# 1) rho(R,z) heatmap
plt.figure(figsize=(8, 5))
```

```
plt.imshow(
    rho.T,
    origin="lower",
    extent=[0, R_max, -Z_max, Z_max],
    aspect="auto",
    cmap="viridis"
)
plt.colorbar(label="rho")
plt.xlabel("R")
plt.ylabel("z")
plt.title("rho(R,z) for Q=200")
plt.tight_layout()
plt.savefig("figures/rho_RZ_Q200.png", dpi=200)

# 2) radial profile rho(R,0)
plt.figure(figsize=(7, 5))
plt.plot(R, rho_R)
plt.axhline(thresh, linestyle="--")
plt.xlabel("R")
plt.ylabel("rho(R,0)")
plt.title("Radial profile at z=0, Q=200")
plt.grid(True)
plt.tight_layout()
plt.savefig("figures/rho_profile_Q200.png", dpi=200)

# 3) reconstruction in x-y plane at z=0
Lxy = 15.0
Nx = Ny = 400
x = np.linspace(-Lxy, Lxy, Nx)
y = np.linspace(-Lxy, Lxy, Ny)
XX, YY = np.meshgrid(x, y, indexing="ij")
R_xy = np.sqrt(XX**2 + YY**2)
rho_xy = np.interp(R_xy, R, rho_R, left=0.0, right=0.0)

plt.figure(figsize=(6, 6))
plt.imshow(
    rho_xy.T,
    origin="lower",
    extent=[-Lxy, Lxy, -Lxy, Lxy],
    aspect="equal",
    cmap="viridis"
)
plt.colorbar(label="rho")
plt.xlabel("x")
plt.ylabel("y")
plt.title("rho(x,y,z=0) for Q=200")
plt.tight_layout()
plt.savefig("figures/rho_xy_Q200.png", dpi=200)

# 4) energy vs steps
plt.figure(figsize=(7, 5))
plt.plot(steps, E_hist, marker="o")
plt.xlabel("step")
plt.ylabel("E")
plt.title("Energy vs gradient flow steps (Q=200)")
plt.grid(True)
plt.tight_layout()
plt.savefig("figures/energy_flow_Q200.png", dpi=200)

plt.close("all")
```

A.2 scan_Q_toroide.py

```
import numpy as np
```

```
import matplotlib.pyplot as plt

# Shared grid and parameters (same as main simulation)
R_max, Z_max = 20.0, 20.0
NR, NZ = 400, 400

R = np.linspace(0.0, R_max, NR)
Z = np.linspace(-Z_max, Z_max, NZ)
dR = R[1] - R[0]
dZ = Z[1] - Z[0]
RR, ZZ = np.meshgrid(R, Z, indexing="ij")

n = 3
m2 = 1.0
g = 1.0
h = 0.5
alpha_core = 20.0
R_core = 1.0

R0 = 8.0
sigma_R = 1.5
sigma_Z = 1.5

dt = 1e-4
n_steps = 4000
sample_every = 400

def init_rho_ring():
    return np.exp(-((RR - R0)**2 / (2.0 * sigma_R**2) +
                    ZZ**2        / (2.0 * sigma_Z**2)))

def compute_Q(rho):
    return 2.0 * np.pi * np.sum(rho**2 * RR) * dR * dZ

def laplacian_cylindrical(rho):
    lap = np.zeros_like(rho)
    lap[1:-1, 1:-1] = (
        (rho[2:, 1:-1] - 2.0 * rho[1:-1, 1:-1] + rho[:-2, 1:-1]) / dR**2 +
        (rho[1:-1, 2:] - 2.0 * rho[1:-1, 1:-1] + rho[1:-1, :-2]) / dZ**2
    )
    drho_dR = np.zeros_like(rho)
    drho_dR[1:-1, :] = (rho[2:, :] - rho[:-2, :]) / (2.0 * dR)
    lap[1:-1, 1:-1] += drho_dR[1:-1, 1:-1] / (RR[1:-1, 1:-1] + 1e-8)
    lap[0, :]  = lap[1, :]
    lap[-1, :] = lap[-2, :]
    lap[:, 0]  = lap[:, 1]
    lap[:, -1] = lap[:, -2]
    return lap

def dU_drho(rho):
    return m2 * rho - g * rho**2 + h * rho**3

def energy_density(rho):
    drho_dR = np.zeros_like(rho)
    drho_dZ = np.zeros_like(rho)
    drho_dR[1:-1, :] = (rho[2:, :] - rho[:-2, :]) / (2.0 * dR)
    drho_dZ[:, 1:-1] = (rho[:, 2:] - rho[:, :-2]) / (2.0 * dZ)
    grad2 = drho_dR**2 + drho_dZ**2
    angular = (n**2) * rho**2 / (RR**2 + 1e-8)
    U = 0.5 * m2 * rho**2 - (g / 3.0) * rho**3 + (h / 4.0) * rho**4
    core = alpha_core * np.exp(-(RR / R_core)**2) * rho**2
    return grad2 + angular + U + core

def compute_energy(rho):
    return 2.0 * np.pi * np.sum(energy_density(rho) * RR) * dR * dZ
```

```
def measure_torus(rho):
    iz0 = np.argmin(np.abs(Z))
    rho_R = rho[:, iz0]
    rho_max = rho_R.max()
    thresh = rho_max / 2.0
    indices = np.where(rho_R > thresh)[0]
    if len(indices) < 2:
        return {
            "rho_max": rho_max,
            "thresh": thresh,
            "R_in": np.nan,
            "R_out": np.nan,
            "R_c": np.nan,
            "a": np.nan,
        }
    R_in = R[indices[0]]
    R_out = R[indices[-1]]
    R_c = 0.5 * (R_in + R_out)
    a = 0.5 * (R_out - R_in)
    return {
        "rho_max": rho_max,
        "thresh": thresh,
        "R_in": R_in,
        "R_out": R_out,
        "R_c": R_c,
        "a": a,
    }

def build_torus_for_Q(Q_target):
    rho = init_rho_ring()
    Q0 = compute_Q(rho)
    rho *= np.sqrt(Q_target / (Q0 + 1e-16))
    print(f"\n=== Q_target = {Q_target} ===")
    print(f"Initial Q normalized: {compute_Q(rho):.6f}")
    for step in range(n_steps + 1):
        if step % sample_every == 0:
            E_now = compute_energy(rho)
            Q_now = compute_Q(rho)
            print(f" step = {step:5d}  E = {E_now:.6e}  Q = {Q_now:.6f}")
        lap = laplacian_cylindrical(rho)
        force = (
            -lap
            + (n**2) * rho / (RR**2 + 1e-8)
            + dU_drho(rho)
            + 2.0 * alpha_core * np.exp(-(RR / R_core)**2) * rho
        )
        rho -= dt * force
        rho = np.maximum(rho, 0.0)
        Q_now = compute_Q(rho)
        rho *= np.sqrt(Q_target / (Q_now + 1e-16))
    E_final = compute_energy(rho)
    meas = measure_torus(rho)
    print(" -> Final energy =", E_final)
    print(" -> Torus measure:", meas)
    return rho, E_final, meas

Q_list = [100.0, 200.0, 400.0]
results = []

for Q_target in Q_list:
    rho_final, E_final, meas = build_torus_for_Q(Q_target)
    results.append({"Q": Q_target, "E": E_final, **meas})

print("\n====================== FINAL RESULTS ======================")
print(" Q_target     E_final      E/Q       R_in      R_out      R_c        a")
print("-----------------------------------------------------------------------")
```

```
for res in results:
    EQ = res["E"] / res["Q"]
    print(f" {res['Q']:7.1f}  {res['E']:11.4e}  {EQ:7.3f}  "
          f"{res['R_in']:8.3f}  {res['R_out']:8.3f}  "
          f"{res['R_c']:8.3f}  {res['a']:8.3f}")

# Save normalized radial profiles for figure
plt.figure(figsize=(7, 5))
for res, Q_target in zip(results, Q_list):
    rho_final, _, _ = build_torus_for_Q(Q_target)
    iz0 = np.argmin(np.abs(Z))
    rho_R = rho_final[:, iz0]
    rho_R_norm = rho_R / rho_R.max()
    plt.plot(R, rho_R_norm, label=f"Q={Q_target:g}")
plt.xlabel("R")
plt.ylabel("rho(R,0) / max")
plt.title("Normalized radial profiles for different Q")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("figures/rho_profiles_Q100_200_400.png", dpi=200)
plt.close()
```