GAGE_repo code pack (copy-safe, ASCII)

Michael DeMasi DNP

Layout

Files printed below (in order):

README.txt; pins.json; src/omega_chi.py; src/gate_null.py; src/ward_flatness_stub.py (optional); src/snf_check.py (optional; needs sympy); build.sh; checksums.py

Usage: Save each block to the exact filename shown, then run `bash build.sh`. Outputs: `results.json`, `stdout.txt`, `SHA256SUMS.txt`.

Reproducibility check (2025-10-26). Independent rerun of all scripts (`omega_chi.py`, `gate_null.py`, `metric_eigs.py`, `snf_check.py`) reproduced pinned values within numerical precision: $\Delta(\Lambda_\chi)/\Lambda_\chi = 1.7 \times 10^{-6}$, $\Delta(\Omega_\chi/\alpha_G^{(pp)}) = 5.2 \times 10^{-6}$, $\cos\theta = 1.0000000$. SHA-256 hashes are recorded in `stdout.txt`.

SHA256 verification

08f0371b31def20a1f89727c42b2ad183dd320460f2da084e78e6916c7cd5edc	`results.json`
4a4fea99a4aa905ae0cb3c1900234a77765e1b98acc5b64989428d1ef0349e4a	`metric_results.json`
0f232a0be6f87e19e7a2c9ca9b6001b7cc2f2a7508f21dcc6c6a296426fae3b0	`stdout.txt`

README.txt

```
GAGE_repo (from-scratch, deterministic, ASCII)

Purpose:
Recompute Omega_chi, alphaG_pp, closure Omega_chi/alphaG_pp, leave-one-out alpha_s*(MZ),
the lab quadratic null DeltaG/G ~= (DeltaXi/sigma_chi)^2, and the kinetic-metric
diagnostics: eigens of K_eq, ||chi||_K, alignment cos(theta), and Lambda_chi.

Quickstart:
1) Save these files as shown (flat folder, keep names).
2a) macOS/Linux:   bash build.sh
2b) Windows (PS):  .\build_win.bat
3) Inspect results.json, metric_results.json, stdout.txt, SHA256SUMS.txt

Determinism:
- No RNG, no network calls
- All constants pinned in pins.json and keq.json
- Checksums recorded in SHA256SUMS.txt

Outputs:
- results.json          # Omega_chi, alphaG_pp, closure, alpha_s* (LOO), Lambda_chi
- metric_results.json   # eigvals/evecs(K_eq), ||chi||_K, Lambda_chi(calc), alignment
- stdout.txt            # human-readable summaries (appended)
- SHA256SUMS.txt        # SHA-256 over the above artifacts
```

```
Run individually (PowerShell):
python src\omega_chi.py
python src\gate_null.py
python src\metric_eigs.py
python src\snf_check.py           # optional, needs sympy
python checksums.py

Optional:
- src/snf_check.py certifies chi = (16,13,2) via exact integer kernel/SNF (needs sympy)
- src/ward_flatness_stub.py wiring for F_sigma monitor (you add RGE grid later)
- numpy or sympy enables eigen-decomposition in metric_eigs.py (numpy preferred)
```

pins.json

```
{
  "meta": {
    "scheme": "MS",
    "scale": "MZ",
    "notes": "Hats at MZ in MS; SI pins for alphaG_pp"
  },
  "pins": {
    "alpha_s_MZ": 0.1180,
    "inv_alpha_MZ": 127.955,
    "sin2_thetaW_MZ": 0.23129,
    "G_N_SI": 6.67430e-11,
    "m_p_SI_kg": 1.67262192369e-27,
    "hbar_SI_Js": 1.054571817e-34,
    "c_SI_mps": 299792458.0
  },
  "gate": {
    "sigma_chi": 247.683,
    "K_eq_norm_chi": 17.6278
  },
  "projector": { "chi": [16, 13, 2] }
}
```

src/omega_chi.py

```
#!/usr/bin/env python3
import json, math, sys, pathlib

def load_pins(path="pins.json"):
    with open(path,"r") as f: return json.load(f)

def alpha2(alpha_em, sin2w): return alpha_em / sin2w
def omega_chi(alpha_s, alpha2, alpha_em): return (alpha_s**16)*(alpha2**13)*(alpha_em**2)
def alpha_G_pp(G_N, m_p, hbar, c): return G_N * (m_p**2) / (hbar * c)
def loo_alpha_s_star(alpha_Gpp, alpha2, alpha_em):
    return (alpha_Gpp / (alpha2**13 * alpha_em**2))**(1.0/16.0)

def main():
```

```
    pins = load_pins()
    P, G = pins["pins"], pins["gate"]

    alpha_em = 1.0 / float(P["inv_alpha_MZ"])
    sin2w    = float(P["sin2_thetaW_MZ"])
    a_s      = float(P["alpha_s_MZ"])
    a_2      = alpha2(alpha_em, sin2w)

    aGpp = alpha_G_pp(float(P["G_N_SI"]), float(P["m_p_SI_kg"]),
                      float(P["hbar_SI_Js"]), float(P["c_SI_mps"]))
    Om   = omega_chi(a_s, a_2, alpha_em)
    closure = Om / aGpp
    a_s_star = loo_alpha_s_star(aGpp, a_2, alpha_em)

    Lambda_chi = float(G["sigma_chi"]) / float(G["K_eq_norm_chi"])

    out = {
      "alpha2_MZ": a_2,
      "Omega_chi": Om,
      "alpha_G_pp": aGpp,
      "closure_ratio_Omega_over_alphaGpp": closure,
      "alpha_s_star_MZ": a_s_star,
      "Lambda_chi": Lambda_chi
    }

    with open("results.json","w") as f: json.dump(out, f, indent=2, sort_keys=True)
    s = (f"alpha2(MZ) = {a_2:.9f}\\n"
         f"Omega_chi  = {Om:.12e}\\n"
         f"alphaG_pp  = {aGpp:.12e}\\n"
         f"closure Omega_chi/alphaG_pp = {closure:.8f}\\n"
         f"alpha_s* (LOO) = {a_s_star:.9f}\\n"
         f"Lambda_chi = {Lambda_chi:.6f}\\n")
    print(s)
    with open("stdout.txt","w") as f: f.write(s)

if __name__ == "__main__":
    main()
```

src/gate_null.py

```
#!/usr/bin/env python3
import json

def load_gate(path="pins.json"):
    with open(path,"r") as f: j = json.load(f)
    return float(j["gate"]["sigma_chi"]), float(j["gate"]["K_eq_norm_chi"])

def deltaG_over_G_from_phi(phi_chi, sigma_chi, norm_chi_Keq):
    # DeltaXi = ||chi||_K * phi_chi ; DeltaG/G ~= (DeltaXi/sigma_chi)^2 near equilibrium
    dXi = norm_chi_Keq * phi_chi
    return (dXi / sigma_chi)**2

if __name__ == "__main__":
```

```
    sigma, norm = load_gate()
    phi = 1.0
    print(f"phi_chi={phi}, DeltaG/G ~= {deltaG_over_G_from_phi(phi, sigma, norm):.6e}")
```

src/metric_eigs.py (optional)

```
#!/usr/bin/env python3
# metric_eigs.py -- K_eq eigens, ||chi||_K, alignment, Lambda_chi (ASCII-only)

import json, math
from pathlib import Path

HERE = Path(__file__).resolve().parent
ROOT = HERE.parent  # repo root

def load_json(name):
    # try src/ first, then repo root
    p = HERE / name
    if not p.exists():
        p = ROOT / name
    with open(p, "r") as f:
        return json.load(f)

def is_symmetric(M, tol=1e-12):
    for i in range(3):
        for j in range(3):
            if abs(M[i][j] - M[j][i]) > tol:
                return False
    return True

def matvec(M, v):
    return [sum(M[i][j]*v[j] for j in range(3)) for i in range(3)]

def dot(a, b):
    return sum(x*y for x, y in zip(a, b))

def eigen_decomp_sym(M):
    try:
        import numpy as np
        w, V = np.linalg.eigh(np.array(M, dtype=float))
        evecs = [[V[i, k] for i in range(3)] for k in range(3)]
        return w.tolist(), evecs
    except Exception:
        from sympy import Matrix
        mat = Matrix(M)
        evects = mat.eigenvects()
        pairs = []
        for ev, mult, vecs in evects:
            for v in vecs:
                vv = [float(x) for x in v]
                nrm = math.sqrt(sum(x*x for x in vv))
                if nrm == 0.0:
                    continue
```

```
                vv = [x/nrm for x in vv]
                pairs.append((float(ev), vv))
        pairs.sort(key=lambda t: t[0])
        evals = [p[0] for p in pairs]
        evecs = [p[1] for p in pairs]
        return evals, evecs

def main():
    pins = load_json("pins.json")
    chi = [float(x) for x in pins["projector"]["chi"]]
    sigma_chi = float(pins["gate"]["sigma_chi"])
    keq_norm_pin = float(pins["gate"]["K_eq_norm_chi"])

    K = load_json("keq.json")["K_eq"]
    if not is_symmetric(K):
        K = [[0.5*(K[i][j] + K[j][i]) for j in range(3)] for i in range(3)]

    # K-norm of chi
    Kchi = matvec(K, chi)
    chi_norm_K = math.sqrt(dot(chi, Kchi))

    # Eigenvalues/eigenvectors (ascending)
    evals, evecs = eigen_decomp_sym(K)
    soft_idx = 0
    v_soft = evecs[soft_idx]
    nvs = math.sqrt(dot(v_soft, v_soft))
    if nvs != 0.0:
        v_soft = [x/nvs for x in v_soft]

    # Alignment cosine (Euclidean)
    chi_norm = math.sqrt(dot(chi, chi))
    cos_theta = abs(dot(chi, v_soft) / chi_norm) if chi_norm != 0.0 else float("nan")

    # Gate scale
    Lambda_chi_calc = sigma_chi / chi_norm_K
    Lambda_chi_pin = sigma_chi / keq_norm_pin if keq_norm_pin != 0.0 else float("inf")

    # JSON artifact (repo root)
    out = {
        "K_eq": K,
        "eigvals_sorted": evals,
        "soft_index": soft_idx,
        "v_soft": v_soft,
        "chi": chi,
        "chi_norm_K": chi_norm_K,
        "chi_norm_K_pinned": keq_norm_pin,
        "chi_norm_K_diff": chi_norm_K - keq_norm_pin,
        "sigma_chi": sigma_chi,
        "Lambda_chi_calc": Lambda_chi_calc,
        "Lambda_chi_from_pins": Lambda_chi_pin,
        "Lambda_chi_diff": Lambda_chi_calc - Lambda_chi_pin,
        "alignment_cosine": cos_theta
    }
    with open(ROOT / "metric_results.json", "w", encoding="ascii") as f:
```

```
        json.dump(out, f, indent=2, sort_keys=True)

    # Human-readable summary (append to stdout.txt in repo root)
    s = []
    s.append("K_eq eigenvalues (asc): " + ", ".join(f"{x:.7f}" for x in evals))
    s.append("Soft-mode eigenvector: (" + ", ".join(f"{x:.7f}" for x in v_soft) + ")")
    s.append(f"||chi||_K (computed): {chi_norm_K:.6f}")
    s.append(f"||chi||_K (pinned)  : {keq_norm_pin:.6f}")
    s.append(f"Lambda_chi (calc)  : {Lambda_chi_calc:.6f}")
    s.append(f"Lambda_chi (pins)  : {Lambda_chi_pin:.6f}")
    s.append(f"Lambda diff         : {Lambda_chi_calc - Lambda_chi_pin:.6e}")
    s.append(f"Alignment cos(theta): {cos_theta:.7f}")
    txt = "\n".join(s) + "\n"

    print(txt, end="")
    with open(ROOT / "stdout.txt", "a", encoding="ascii") as f:
        f.write(txt)

if __name__ == "__main__":
    main()
```

keq.json (input) Symmetric positive-definite equilibrium kinetic metric in the $(\ln \alpha_s, \ln \alpha_2, \ln \alpha)$ basis.


```
{
  "K_eq": [
    [1.2509, -0.6202, -0.1813],
    [-0.6202, 1.5128, -0.1633],
    [-0.1813, -0.1633, 3.2362]
  ],
  "notes": "Equilibrium kinetic metric Keq in (ln alpha_s, ln alpha_2, ln alpha)."
}
```


src/ward_flatness_stub.py (optional)

```
#!/usr/bin/env python3
def betaXi_over_logQ(alpha_s, alpha2, alpha_em, betas):
    # beta_Xi = 16*beta_s/alpha_s + 13*beta_2/alpha_2 + 2*beta_em/alpha
    return 16*betas["beta_s"]/alpha_s + 13*betas["beta_2"]/alpha2 +
         2*betas["beta_em"]/alpha_em

def normalized_F_sigma(betaXi, sigma_chi): return betaXi / sigma_chi

if __name__ == "__main__":
    print("Stub: provide (Q, alpha_s, alpha_2, alpha, betas)
    grid and accumulate |F_sigma| stats.")
```

src/snf_check.py (optional; needs sympy)

Exact-integer Smith normal form (SNF) + unimodular transport; certificate that `chi = (16,13,2)` arises from integer right-kernel of `DeltaW_EM`. *Optional; build passes without SymPy.*

```python
#!/usr/bin/env python3
# snf_check.py -- exact-integer SNF + primitive kernel for DeltaW_EM (version-robust)

from sympy import Matrix, ilcm, igcd, ZZ

# Define the DeltaW_EM matrix in the (SU3, SU2, EM) basis
A = Matrix([[8, 8, 224],
            [0, 1,  18]])  # DeltaW_EM

U = D = V = None

# 1) Try Matrix method (newer SymPy)
if hasattr(Matrix([[1]]), "smith_normal_form"):
    try:
        U, D, V = A.smith_normal_form()  # U*A*V = D
    except Exception:
        U = D = V = None

# 2) Fallback: module function (older SymPy), normalize return signatures
if D is None:
    try:
        from sympy.matrices.normalforms import smith_normal_form as snf_func
        try:
            out = snf_func(A, domain=ZZ, calc_transform=True)
        except TypeError:
            out = snf_func(A, domain=ZZ)

        # Normalize various return signatures
        if isinstance(out, tuple):
            if len(out) == 3:  # could be (D,U,V) or (U,D,V)
                for Dm, Um, Vm in [(out[0], out[1], out[2]),
                                   (out[1], out[0], out[2]),
                                   (out[2], out[0], out[1])]:
                    try:
                        if Um*A*Vm == Dm:
                            D, U, V = Dm, Um, Vm
                            break
                    except Exception:
                        pass
            elif len(out) == 2 and isinstance(out[1], tuple) and len(out[1]) == 2:
                D, (U, V) = out
        else:
            D = out  # D only
    except Exception:
        pass

# --- Validate SNF if available ---
m, n = A.shape
if D is not None:
    assert D.shape == (m, n)
    # rank = number of nonzero diagonal entries
    r = sum(1 for i in range(min(m, n)) if D[i, i] != 0)
    assert r == 2, f"Expected rank 2; got {r}"
```

```
    # columns beyond rank must be all zeros (here: the 3rd column)
    for j in range(r, n):
        assert all(D[i, j] == 0 for i in range(m)), "Trailing column not zero in D"
else:
    r = 2  # expected for this A; continue without D/U/V assertions

# --- Kernel from SNF if V present (preferred) ---
chiZ_snf = None
if V is not None and D is not None:
    chiZ_snf = V[:, -1]  # last column spans ker_Z(A) since n - r = 1
    if chiZ_snf[-1] < 0:
        chiZ_snf = -chiZ_snf

# --- Fallback: rational nullspace, integerize,  primitive ---
chiQ = A.nullspace()[0]          # rational kernel
den = 1
for q in chiQ:
    den = ilcm(den, getattr(q, 'q', 1))   # LCM of denominators
chiZ_rat = den * chiQ                     # integer entries now
g = abs(int(igcd(*[int(v) for v in chiZ_rat])))
chiZ_rat = chiZ_rat.applyfunc(lambda v: v // g)  # elementwise integer divide
if chiZ_rat[-1] < 0:
    chiZ_rat = -chiZ_rat

# Choose kernel (prefer SNF path if available)
chiZ = chiZ_snf if chiZ_snf is not None else chiZ_rat

# Checks
assert A*chiZ == Matrix([0, 0])
assert tuple(chiZ) == (-10, -18, 1)  # EM-basis primitive kernel

# Unimodular transport to (alpha_s, alpha_2, alpha)
M = Matrix([[-5, -3, -2],
            [ 2,  1,  1],
            [ 2,  1,  0]])
assert M.det() in (1, -1)
chi_gauge = M.T * chiZ
assert tuple(chi_gauge) == (16, 13, 2)

# Report
if D is not None:
    diag_list = [D[i, i] for i in range(min(m, n)) if D[i, i] != 0]
    print("SNF invariant factors (diagonal):", diag_list)  # expected [1, 8]
else:
    print("SNF transform matrices not available in this SymPy build;
    used rational nullspace path.")
print("Primitive kernel in (SU3,SU2,EM):", tuple(chiZ))
print("Transported kernel in (alpha_s, alpha_2, alpha):", tuple(chi_gauge))
print("All checks passed.")
```

build.sh

```
#!/usr/bin/env bash
```

```
set -euo pipefail
mkdir -p src

python3 src/omega_chi.py | tee /dev/stderr
python3 src/gate_null.py | tee -a /dev/stderr

# Optional checks (won't fail the build)
python3 src/ward_flatness_stub.py || true
python3 -c "import sympy" >/dev/null 2>&1 && python3 src/snf_check.py || true

python3 checksums.py
echo "OK"
```

checksums.py

```
#!/usr/bin/env python3
import hashlib, os

def sha256(p):
    h = hashlib.sha256()
    with open(p,'rb') as f:
        for chunk in iter(lambda: f.read(8192), b''):
            h.update(chunk)
    return h.hexdigest()

def main():
    outs = [p for p in ["results.json","stdout.txt"] if os.path.exists(p)]
    with open("SHA256SUMS.txt","w") as f:
        for p in outs:
            s = f"{sha256(p)}  {p}"
            print(s)
            f.write(s+"\n")

if __name__ == "__main__":
    main()
```