In [2]:
```
! pip install numpy matplotlib
import numpy as np
from numpy.linalg import eigh, norm
import matplotlib.pyplot as plt
rng = np.random.default_rng()
```

```
Requirement already satisfied: numpy in c:\users\codet\appdata\local\programs\p
ython\python313\lib\site-packages (2.2.4)
Requirement already satisfied: matplotlib in c:\users\codet\appdata\local\progr
ams\python\python313\lib\site-packages (3.10.7)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\codet\appdata\loca
l\programs\python\python313\lib\site-packages (from matplotlib) (1.3.3)
Requirement already satisfied: cycler>=0.10 in c:\users\codet\appdata\local\pro
grams\python\python313\lib\site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\codet\appdata\loca
l\programs\python\python313\lib\site-packages (from matplotlib) (4.61.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\codet\appdata\loca
l\programs\python\python313\lib\site-packages (from matplotlib) (1.4.9)
Requirement already satisfied: packaging>=20.0 in c:\users\codet\appdata\local\
programs\python\python313\lib\site-packages (from matplotlib) (25.0)
Requirement already satisfied: pillow>=8 in c:\users\codet\appdata\local\progra
ms\python\python313\lib\site-packages (from matplotlib) (12.0.0)
Requirement already satisfied: pyparsing>=3 in c:\users\codet\appdata\local\pro
grams\python\python313\lib\site-packages (from matplotlib) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\codet\appdata\r
oaming\python\python313\site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in c:\users\codet\appdata\roaming\pytho
n\python313\site-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
```

```
[notice] A new release of pip is available: 25.2 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
```

In [3]:
```
# Helper functions
def sqrtm_spd(A, eps=1e-12):
    """Matrix square root of a symmetric positive definite matrix A."""
    w, v = eigh(A)
    w_clipped = np.clip(w, eps, None)
    return (v * np.sqrt(w_clipped)) @ v.T

def inv_sqrtm_spd(A, eps=1e-12):
    """Inverse square root of a symmetric positive definite matrix A."""
    w, v = eigh(A)
    w_clipped = np.clip(w, eps, None)
    return (v * (1.0 / np.sqrt(w_clipped))) @ v.T
```

In [4]:
```
# Covariance matrix constructors
def make_sigma_identity(d):
    """Simple Σ = I."""
    return np.eye(d)

def make_sigma_log_spectrum(d, kappa=10.0):
    """
    Σ with log-spaced eigenvalues between 1 and kappa.
    """
```

```python
    # eigenvalues log-spaced in [1, kappa]
    eigvals = np.exp(np.linspace(0.0, np.log(kappa), d))
    Q, _ = np.linalg.qr(rng.normal(size=(d, d)))
    return Q @ np.diag(eigvals) @ Q.T

def make_sigma_random_spd(d, scale=1.0):
    """
    Σ = A A^T / d, random SPD with mild conditioning.
    """
    A = rng.normal(size=(d, d))
    return (A @ A.T) / (d * scale)
```

In [5]:
```python
# Single spike models

def make_T_spike(d, beta=3.0):
    """
    T = I + (beta - 1) u u^T  (single spike).
    beta >= 1 controls spike strength.
    """
    u = rng.normal(size=d)
    u /= norm(u)
    return np.eye(d) + (beta - 1.0) * np.outer(u, u), u

def make_T_diag(d, kappa=10.0):
    """
    T diagonal with log-spaced eigenvalues between 1 and kappa.
    """
    eigvals = np.exp(np.linspace(0.0, np.log(kappa), d))
    return np.diag(eigvals)

def make_H(Sigma, T):
    Sigma_sqrt = sqrtm_spd(Sigma)
    return Sigma_sqrt @ T @ Sigma_sqrt

def sample_G_hat(d, m, Sigma=None):
    """
    Sample empirical second-moment hat(G) = (1/m) sum g_i g_i^T
    with g_i ~ N(0, Sigma). If Sigma is None, use identity.
    """
    if Sigma is None:
        # Sigma = I: gradients are standard normal
        G_hat = np.zeros((d, d))
        for _ in range(m):
            g = rng.normal(size=d)
            G_hat += np.outer(g, g)
        G_hat /= m
        return G_hat
    else:
        # General Sigma via its sqrt
        Sigma_sqrt = sqrtm_spd(Sigma)
        G_hat = np.zeros((d, d))
        for _ in range(m):
            z = rng.normal(size=d)
            g = Sigma_sqrt @ z
            G_hat += np.outer(g, g)
        G_hat /= m
        return G_hat
```

In [ ]:
```
def sample_gradient(Sigma):
    """
    Sample g ~ N(0, Σ).
    """
    d = Sigma.shape[0]
    z = rng.normal(size=d)
    # Using Cholesky or sqrtm; for large d, we'd want a more efficient factori
    Sigma_sqrt = sqrtm_spd(Sigma)
    return Sigma_sqrt @ z

def build_ema_G(Sigma, beta=0.99, n_steps=5000, G0=None):
    """
    Build EMA preconditioner:
        G_t = beta * G_{t-1} + (1 - beta) * g_t g_t^T
    Returns the final G_t.
    """
    d = Sigma.shape[0]
    if G0 is None:
        G = np.zeros((d, d))
    else:
        G = G0.copy()

    for _ in range(n_steps):
        g = sample_gradient(Sigma)
        G = beta * G + (1.0 - beta) * np.outer(g, g)
    return G
```

In [7]:
```
# The curvature after preconditioning, and its spectrum

def preconditioned_curvature(H, G_hat):
    """
    Compute H' = G_hat^{-1/2} H G_hat^{-1/2}.
    """
    G_inv_sqrt = inv_sqrtm_spd(G_hat)
    return G_inv_sqrt @ H @ G_inv_sqrt

def spectrum(A):
    """
    Return sorted eigenvalues (ascending).
    Assumes A is symmetric.
    """
    w, _ = eigh(A)
    return np.sort(w)
```

Experiment 1 for Claim 1

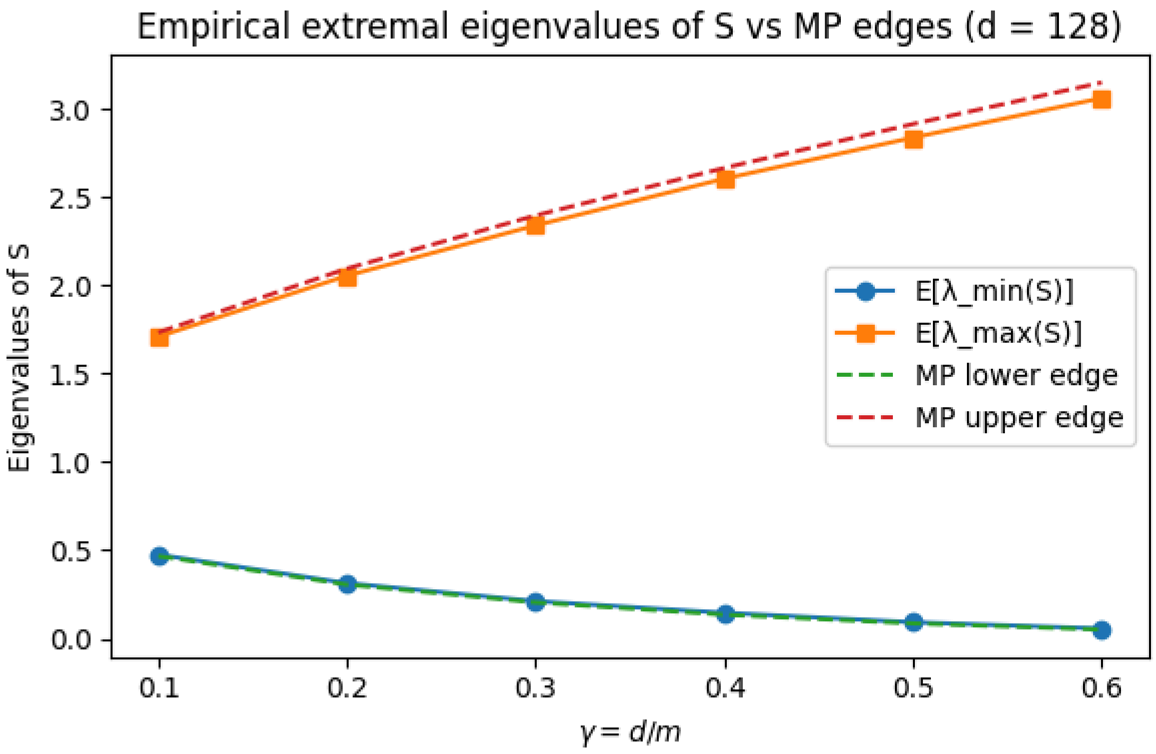
In [45]:
```
d = 128
```

```
gamma_list = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
kappa_T = 10.0
n_trials = 50  # feel free to increase if it's fast

# Build T and H (Sigma = I)
T = make_T_diag(d, kappa=kappa_T)
H = T.copy()

results = []

for gamma in gamma_list:
    m = int(d / gamma)  # m ≈ d / gamma
    lambda_min_S = []
    lambda_max_S = []
    lambda_min_Hp = []
    lambda_max_Hp = []
    kappa_Hp = []

    for _ in range(n_trials):
        # Sample G_hat
        G_hat = sample_G_hat(d, m, Sigma=None)  # Sigma = I

        # S = Sigma^{-1/2} G_hat Sigma^{-1/2} = G_hat when Sigma = I
        S = G_hat

        # Preconditioned curvature H' = G_hat^{-1/2} H G_hat^{-1/2}
        G_inv_sqrt = inv_sqrtm_spd(G_hat, eps=1e-10)
        H_prime = G_inv_sqrt @ H @ G_inv_sqrt

        # Spectra
        eig_S = spectrum(S)
        eig_Hp = spectrum(H_prime)

        lambda_min_S.append(eig_S[0])
        lambda_max_S.append(eig_S[-1])

        lambda_min_Hp.append(eig_Hp[0])
        lambda_max_Hp.append(eig_Hp[-1])
        kappa_Hp.append(eig_Hp[-1] / eig_Hp[0])

    # Aggregate
    lambda_min_S = np.array(lambda_min_S)
    lambda_max_S = np.array(lambda_max_S)
    lambda_min_Hp = np.array(lambda_min_Hp)
    lambda_max_Hp = np.array(lambda_max_Hp)
    kappa_Hp = np.array(kappa_Hp)

    # Theoretical MP edges
    lam_minus = (1 - np.sqrt(gamma))**2
    lam_plus  = (1 + np.sqrt(gamma))**2

    # Theoretical H' bounds
    eig_T = spectrum(T)
    lam_min_T = eig_T[0]
    lam_max_T = eig_T[-1]
    kappa_T_emp = lam_max_T / lam_min_T

    lb_Hp = lam_min_T / (1 + np.sqrt(gamma))**2
    ub_Hp = lam_max_T / (1 - np.sqrt(gamma))**2
    kappa_bound = kappa_T_emp * (1 + np.sqrt(gamma))**2 / (1 - np.sqrt(gamma))

    results.append(
        dict(
            gamma=gamma,
            m=m,
            lam_minus=lam_minus,
            lam_plus=lam_plus,
            lam_min_S_mean=lambda_min_S.mean(),
            lam_max_S_mean=lambda_max_S.mean(),
            lam_min_S_q5=np.quantile(lambda_min_S, 0.05),
            lam_min_S_q95=np.quantile(lambda_min_S, 0.95),
            lam_max_S_q5=np.quantile(lambda_max_S, 0.05),
            lam_max_S_q95=np.quantile(lambda_max_S, 0.95),
            lam_min_Hp_mean=lambda_min_Hp.mean(),
            lam_max_Hp_mean=lambda_max_Hp.mean(),
            lam_min_Hp_q5=np.quantile(lambda_min_Hp, 0.05),
            lam_min_Hp_q95=np.quantile(lambda_min_Hp, 0.95),
            lam_max_Hp_q5=np.quantile(lambda_max_Hp, 0.05),
            lam_max_Hp_q95=np.quantile(lambda_max_Hp, 0.95),
            kappa_Hp_mean=kappa_Hp.mean(),
            kappa_Hp_q5=np.quantile(kappa_Hp, 0.05),
            kappa_Hp_q95=np.quantile(kappa_Hp, 0.95),
            lb_Hp=lb_Hp,
            ub_Hp=ub_Hp,
            kappa_bound=kappa_bound,
            kappa_T_emp=kappa_T_emp,
        )
    )

results
```

```
Out[45]: [{'gamma': 0.1,
           'm': 1280,
           'lam_minus': np.float64(0.46754446796632404),
           'lam_plus': np.float64(1.732455532033676),
           'lam_min_S_mean': np.float64(0.47567972347083676),
           'lam_max_S_mean': np.float64(1.7083739957904873),
           'lam_min_S_q5': np.float64(0.4583880267764845),
           'lam_min_S_q95': np.float64(0.49084031892908353),
           'lam_max_S_q5': np.float64(1.6770236422795908),
           'lam_max_S_q95': np.float64(1.7457920865265182),
           'lam_min_Hp_mean': np.float64(0.8645339221305055),
           'lam_max_Hp_mean': np.float64(13.37785777806388),
           'lam_min_Hp_q5': np.float64(0.833018864416662),
           'lam_min_Hp_q95': np.float64(0.8890356136463019),
           'lam_max_Hp_q5': np.float64(12.982164727975446),
           'lam_max_Hp_q95': np.float64(14.050533456737025),
           'kappa_Hp_mean': np.float64(15.479285706561628),
           'kappa_Hp_q5': np.float64(14.882292033580582),
           'kappa_Hp_q95': np.float64(16.180656026479838),
           'lb_Hp': np.float64(0.5772153925510174),
           'ub_Hp': np.float64(21.388339901650326),
           'kappa_bound': np.float64(37.05434778363072),
           'kappa_T_emp': np.float64(10.000000000000002)},
          {'gamma': 0.2,
           'm': 640,
           'lam_minus': np.float64(0.3055728090000842),
           'lam_plus': np.float64(2.0944271909999155),
           'lam_min_S_mean': np.float64(0.31411862457827117),
           'lam_max_S_mean': np.float64(2.053711572649849),
           'lam_min_S_q5': np.float64(0.2967888779714568),
           'lam_min_S_q95': np.float64(0.33359687968741963),
           'lam_max_S_q5': np.float64(2.0057720315564236),
           'lam_max_S_q95': np.float64(2.1318959486114935),
           'lam_min_Hp_mean': np.float64(0.7867513374363487),
           'lam_max_Hp_mean': np.float64(17.4704944132441),
           'lam_min_Hp_q5': np.float64(0.7422786131157324),
           'lam_min_Hp_q95': np.float64(0.822594785162641),
           'lam_max_Hp_q5': np.float64(16.61362964594663),
           'lam_max_Hp_q95': np.float64(18.460465082832844),
           'kappa_Hp_mean': np.float64(22.229578336308123),
           'kappa_Hp_q5': np.float64(20.68295486323115),
           'kappa_Hp_q95': np.float64(24.02572524837783),
           'lb_Hp': np.float64(0.47745751406263154),
           'ub_Hp': np.float64(32.72542485937368),
           'kappa_bound': np.float64(68.54101966249682),
           'kappa_T_emp': np.float64(10.000000000000002)},
          {'gamma': 0.3,
           'm': 426,
           'lam_minus': np.float64(0.2045548849896678),
           'lam_plus': np.float64(2.395445115010332),
           'lam_min_S_mean': np.float64(0.21202982562882258),
           'lam_max_S_mean': np.float64(2.3391565839676685),
           'lam_min_S_q5': np.float64(0.1946617061194701),
           'lam_min_S_q95': np.float64(0.23131156505225503),
           'lam_max_S_q5': np.float64(2.2745717662168583),
           'lam_max_S_q95': np.float64(2.3903822194686435),
           'lam_min_Hp_mean': np.float64(0.7242266959789154),
           'lam_max_Hp_mean': np.float64(23.93842516243709),
           'lam_min_Hp_q5': np.float64(0.6913149832223281),
           'lam_min_Hp_q95': np.float64(0.7673839132253416),
           'lam_max_Hp_q5': np.float64(22.206015321133584),
           'lam_max_Hp_q95': np.float64(26.113946024458993),
           'kappa_Hp_mean': np.float64(33.071935892957825),
           'kappa_Hp_q5': np.float64(30.5294287964557),
           'kappa_Hp_q95': np.float64(35.8810051426708),
           'lb_Hp': np.float64(0.4174589489585057),
           'ub_Hp': np.float64(48.88663500021086),
           'kappa_bound': np.float64(117.10525100054822),
           'kappa_T_emp': np.float64(10.000000000000002)},
          {'gamma': 0.4,
           'm': 320,
           'lam_minus': np.float64(0.13508893593264826),
           'lam_plus': np.float64(2.6649110640673523),
           'lam_min_S_mean': np.float64(0.14605102472646952),
           'lam_max_S_mean': np.float64(2.6040527473897948),
           'lam_min_S_q5': np.float64(0.13177081731098864),
           'lam_min_S_q95': np.float64(0.1589569489402688),
           'lam_max_S_q5': np.float64(2.514264193648095),
           'lam_max_S_q95': np.float64(2.732242973028586),
           'lam_min_Hp_mean': np.float64(0.6724689580154358),
           'lam_max_Hp_mean': np.float64(32.98248060058157),
           'lam_min_Hp_q5': np.float64(0.6337516971332305),
           'lam_min_Hp_q95': np.float64(0.7067111642878571),
           'lam_max_Hp_q5': np.float64(30.128208200036674),
           'lam_max_Hp_q95': np.float64(37.2555051134557),
           'kappa_Hp_mean': np.float64(49.067747195437775),
           'kappa_Hp_q5': np.float64(44.8891044737266),
           'kappa_Hp_q95': np.float64(54.897778619273446),
           'lb_Hp': np.float64(0.3752470442573562),
           'ub_Hp': np.float64(74.02530733520423),
           'kappa_bound': np.float64(197.2708605385719),
           'kappa_T_emp': np.float64(10.000000000000002)},
          {'gamma': 0.5,
           'm': 256,
           'lam_minus': np.float64(0.08578643762690492),
           'lam_plus': np.float64(2.914213562373095),
           'lam_min_S_mean': np.float64(0.09265274579756026),
           'lam_max_S_mean': np.float64(2.8368486210580057),
           'lam_min_S_q5': np.float64(0.0814639841112012),
           'lam_min_S_q95': np.float64(0.10466134580306287),
           'lam_max_S_q5': np.float64(2.725259366300481),
           'lam_max_S_q95': np.float64(2.9702184046245774),
           'lam_min_Hp_mean': np.float64(0.6349691828465392),
           'lam_max_Hp_mean': np.float64(49.66950485949501),
           'lam_min_Hp_q5': np.float64(0.6013556104784524),
           'lam_min_Hp_q95': np.float64(0.6613310818239413),
           'lam_max_Hp_q5': np.float64(42.144661487596515),
           'lam_max_Hp_q95': np.float64(56.758480296030456),
           'kappa_Hp_mean': np.float64(78.33595405541182),
           'kappa_Hp_q5': np.float64(64.65930080919202),
           'kappa_Hp_q95': np.float64(89.46464916778602),
           'lb_Hp': np.float64(0.3431457505076198),
           'ub_Hp': np.float64(116.56854249492386),
           'kappa_bound': np.float64(339.7056274847716),
           'kappa_T_emp': np.float64(10.000000000000002)},
          {'gamma': 0.6,
           'm': 213,
           'lam_minus': np.float64(0.05080666151703323),
           'lam_plus': np.float64(3.149193338482967),
           'lam_min_S_mean': np.float64(0.05683182915239237),
           'lam_max_S_mean': np.float64(3.060956267524454),
           'lam_min_S_q5': np.float64(0.04889944837276513),
           'lam_min_S_q95': np.float64(0.06333814104811804),
           'lam_max_S_q5': np.float64(2.9249335740957303),
           'lam_max_S_q95': np.float64(3.215334145139818),
           'lam_min_Hp_mean': np.float64(0.5996501654557244),
           'lam_max_Hp_mean': np.float64(77.6898436440345),
           'lam_min_Hp_q5': np.float64(0.5528523099831465),
           'lam_min_Hp_q95': np.float64(0.6291174220502082),
           'lam_max_Hp_q5': np.float64(68.62620383552587),
           'lam_max_Hp_q95': np.float64(88.07805093207591),
           'kappa_Hp_mean': np.float64(129.7191084849282),
           'kappa_Hp_q5': np.float64(115.15830436132504),
           'kappa_Hp_q95': np.float64(151.43317609089237),
           'lb_Hp': np.float64(0.3175416344814578),
           'ub_Hp': np.float64(196.8245836551855),
           'kappa_bound': np.float64(619.8386676965937),
           'kappa_T_emp': np.float64(10.000000000000002)}]
```

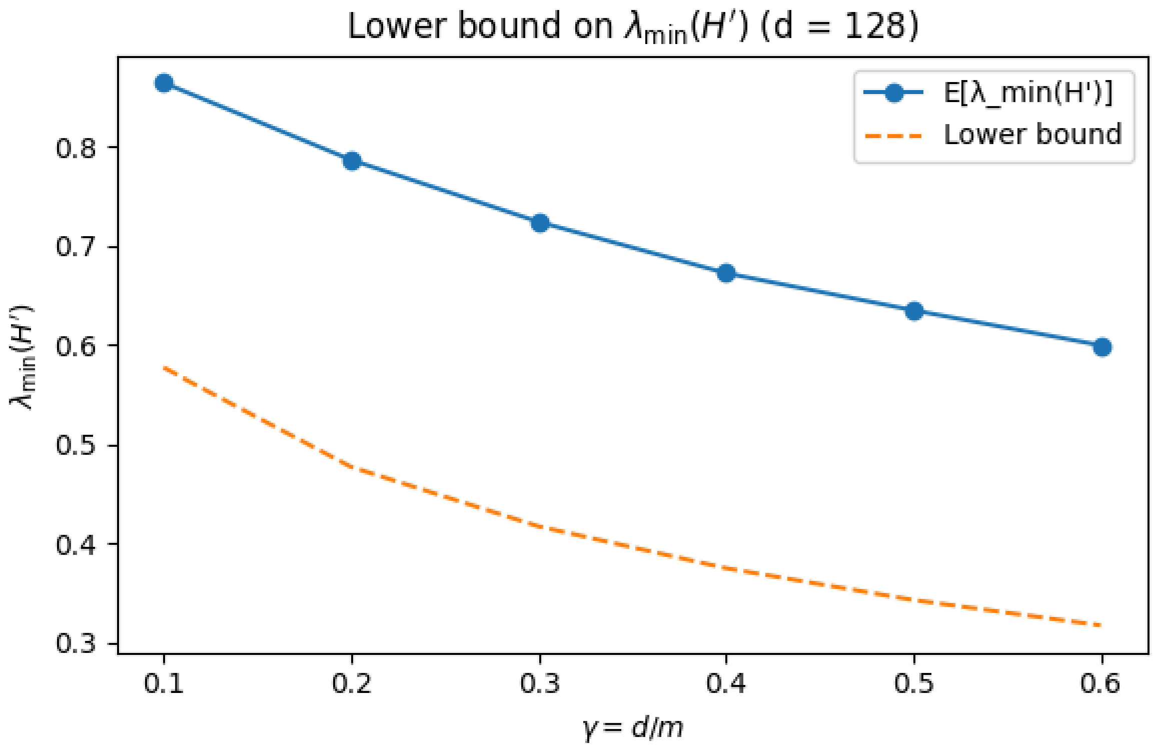
Plot 1: Show that MP law holds for our setup

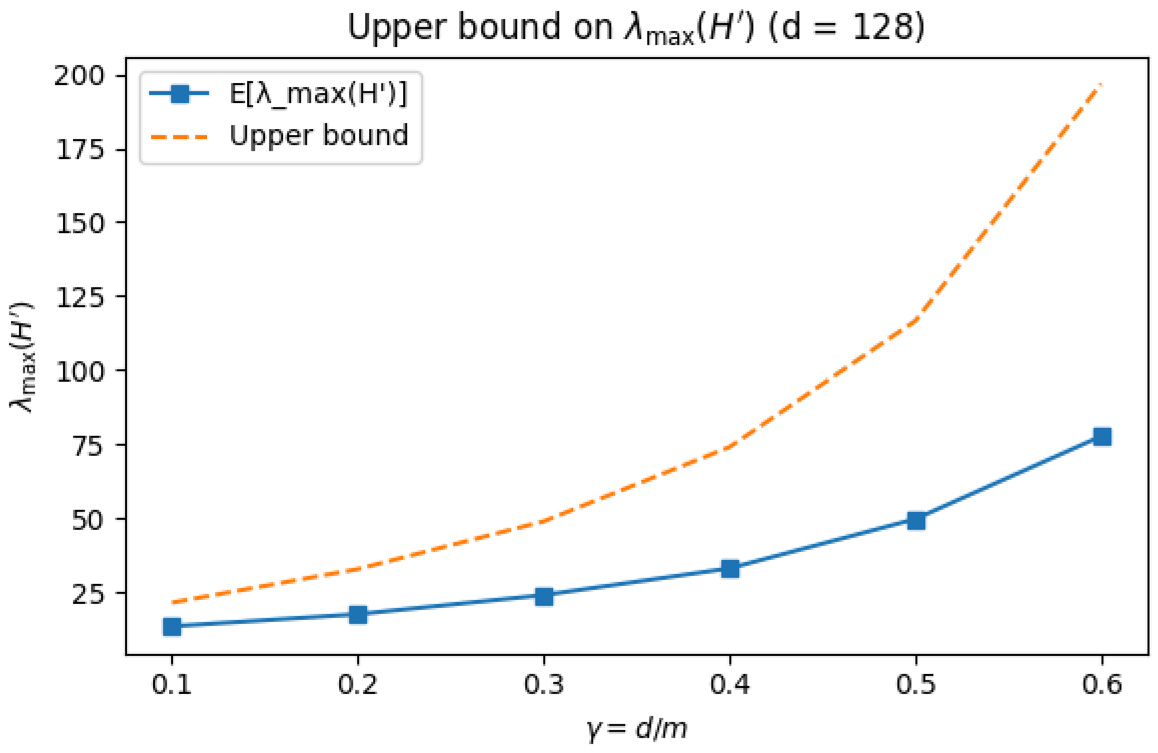
In [46]:
```
# Extract arrays for plotting
gammas = np.array([r["gamma"] for r in results])
lam_minus = np.array([r["lam_minus"] for r in results])
lam_plus  = np.array([r["lam_plus"]  for r in results])
lam_min_S_mean = np.array([r["lam_min_S_mean"] for r in results])
lam_max_S_mean = np.array([r["lam_max_S_mean"] for r in results])

plt.figure(figsize=(6, 4))

# empirical means
plt.plot(gammas, lam_min_S_mean, marker='o', label="E[λ_min(S)]")
plt.plot(gammas, lam_max_S_mean, marker='s', label="E[λ_max(S)]")

# theoretical MP edges
plt.plot(gammas, lam_minus, linestyle='--', label="MP lower edge")
plt.plot(gammas, lam_plus,  linestyle='--', label="MP upper edge")

plt.xlabel(r"$\gamma = d/m$")
plt.ylabel("Eigenvalues of S")
plt.title("Empirical extremal eigenvalues of S vs MP edges (d = 128)")
plt.legend()
plt.tight_layout()
plt.show()
```



Plot 2:

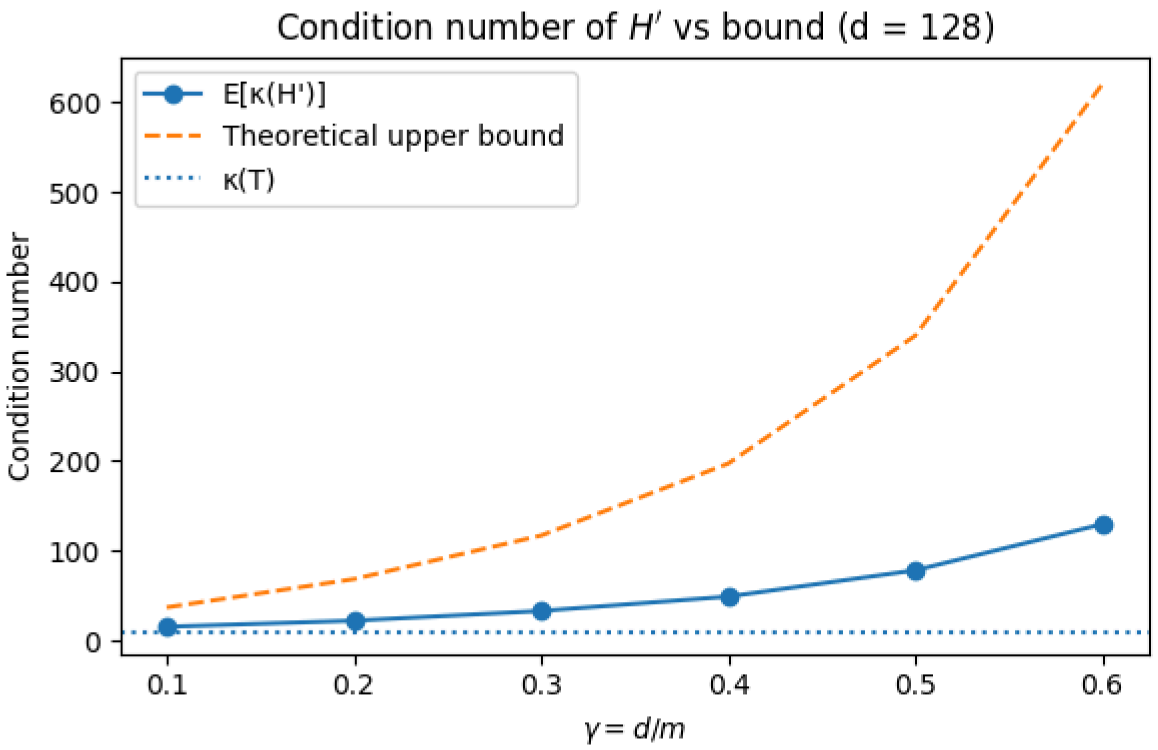
In [47]:
```
lam_min_Hp_mean = np.array([r["lam_min_Hp_mean"] for r in results])
lam_max_Hp_mean = np.array([r["lam_max_Hp_mean"] for r in results])
lb_Hp = np.array([r["lb_Hp"] for r in results])
ub_Hp = np.array([r["ub_Hp"] for r in results])

plt.figure(figsize=(6, 4))

# λ_min(H')
plt.plot(gammas, lam_min_Hp_mean, marker='o', label=r"E[λ_min(H')]")
plt.plot(gammas, lb_Hp, linestyle='--', label="Lower bound")

plt.xlabel(r"$\gamma = d/m$")
plt.ylabel(r"$\lambda_{\min}(H')$")
plt.title(r"Lower bound on $\lambda_{\min}(H')$ (d = 128)")
plt.legend()
plt.tight_layout()
plt.show()

plt.figure(figsize=(6, 4))

# λ_max(H')
plt.plot(gammas, lam_max_Hp_mean, marker='s', label=r"E[λ_max(H')]")
plt.plot(gammas, ub_Hp, linestyle='--', label="Upper bound")
plt.xlabel(r"$\gamma = d/m$")
plt.ylabel(r"$\lambda_{\max}(H')$")
plt.title(r"Upper bound on $\lambda_{\max}(H')$ (d = 128)")
plt.legend()
plt.tight_layout()
plt.show()
```





In [48]:

```python
kappa_Hp_mean = np.array([r["kappa_Hp_mean"] for r in results])
kappa_bound = np.array([r["kappa_bound"] for r in results])
kappa_T_emp = results[0]["kappa_T_emp"]  # same for all gammas

plt.figure(figsize=(6, 4))

plt.plot(gammas, kappa_Hp_mean, marker='o', label=r"E[κ(H')]")
plt.plot(gammas, kappa_bound, linestyle='--', label="Theoretical upper bound")
plt.axhline(kappa_T_emp, linestyle=':', label=r"κ(T)")

plt.xlabel(r"$\gamma = d/m$")
plt.ylabel(r"Condition number")
plt.title(r"Condition number of $H'$ vs bound (d = 128)")
plt.legend()
plt.tight_layout()
plt.show()
```



Theorem 2

In [1]:

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

rng = np.random.default_rng(0)

def make_spike_T(d, beta):
    """
    One-spike covariance:
      T = I + (beta - 1) u u^T
    with u random unit vector.
    Returns (T, u).
    """
    u = rng.normal(size=d)
    u /= np.linalg.norm(u)
    T = np.eye(d) + (beta - 1.0) * np.outer(u, u)
    return T, u

def sample_spiked_G_hat(T, m):
    """
    Sample G_hat = (1/m) sum g_i g_i^T with g_i ~ N(0, T).
    Uses T^{1/2} * z, z ~ N(0, I).
    """
    d = T.shape[0]
    T_sqrt = sqrtm_spd(T)
    G_hat = np.zeros((d, d))
    for _ in range(m):
        z = rng.normal(size=d)
        g = T_sqrt @ z
        G_hat += np.outer(g, g)
    G_hat /= m
    return G_hat

def spectrum(A):
    w = np.linalg.eigvalsh(A)
    return np.sort(w)
```

Diagram showing the BBP phase transitions detectability

In [18]:
```
# Config
d = 128
gamma = 0.5
m = int(d / gamma)
# n_trials = 50
n_trials = 20

beta_vals = np.linspace(0.5, 20.0, 30)  # sweep across threshold
beta_c = 1.0 + np.sqrt(gamma)
print("BBP threshold beta_c =", beta_c)

lam_max_means = []
lam_max_std = []

def bbp_outlier(beta, gamma):
    return beta * (1.0 + gamma / (beta - 1.0))

mp_upper = (1.0 + np.sqrt(gamma))**2

for beta in beta_vals:
    lam_max_list = []
    for _ in range(n_trials):
        T, u = make_spike_T(d, beta)
        G_hat = sample_spiked_G_hat(T, m)
        eig_G = spectrum(G_hat)
        lam_max_list.append(eig_G[-1])
    lam_max_list = np.array(lam_max_list)
    lam_max_means.append(lam_max_list.mean())
    lam_max_std.append(lam_max_list.std())

lam_max_means = np.array(lam_max_means)
lam_max_std = np.array(lam_max_std)

# Theoretical curve for outlier: only valid for beta > beta_c
bbp_curve = np.where(
    beta_vals > beta_c,
    bbp_outlier(beta_vals, gamma),
    mp_upper
)

plt.figure(figsize=(7, 4))
plt.errorbar(beta_vals, lam_max_means, yerr=lam_max_std, fmt='o', label="Empir
plt.axhline(mp_upper, color='gray', linestyle='--', label="MP upper edge")
plt.plot(beta_vals, bbp_curve, linestyle='--', color='C1', label="BBP predicti
plt.axvline(beta_c, color='red', linestyle=':', label=r"BBP threshold $\beta_c
plt.xlabel(r"Spike strength $\beta$")
plt.ylabel(r"Top eigenvalue $\lambda_{\max}(\hat G)$")
plt.title(f"BBP transition: d={d}, gamma={gamma}, m={m}")
plt.legend()
plt.tight_layout()
plt.show()
```

```
BBP threshold beta_c = 1.7071067811865475
```

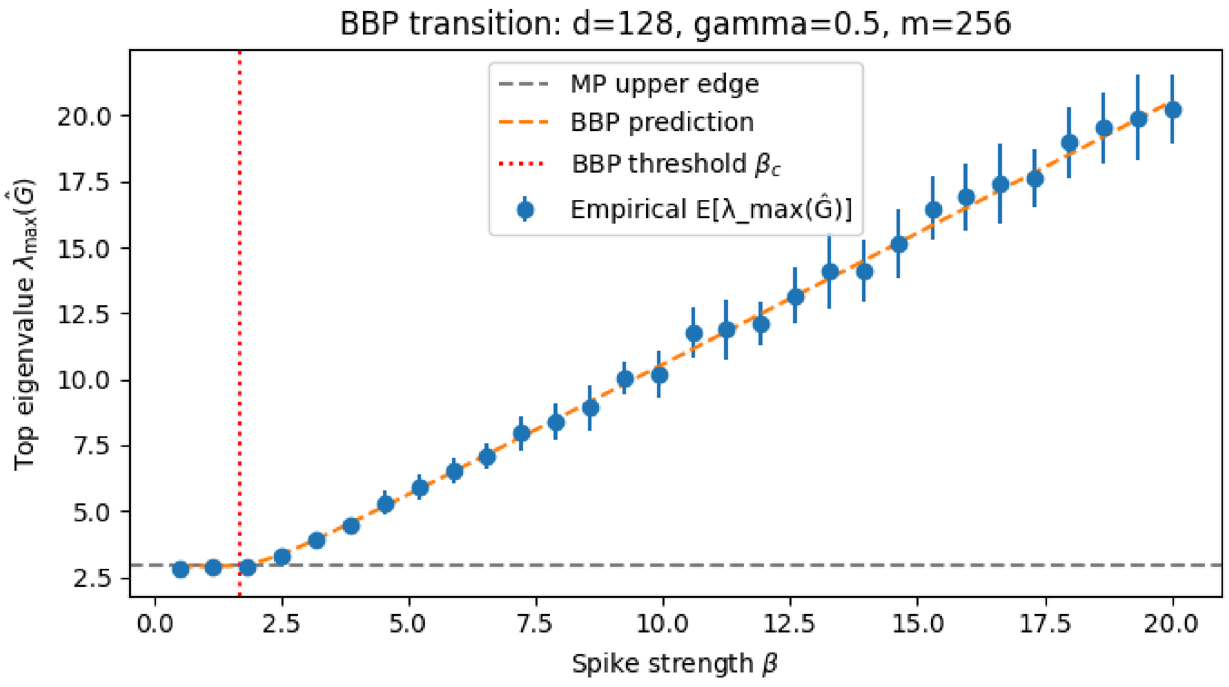


Diagram showing the eigenvector alignment

In [20]:
```
overlap_means = []
```

```
overlap_std = []

for beta in beta_vals:
    overlaps = []
    for _ in range(n_trials):
        T, u = make_spike_T(d, beta)
        G_hat = sample_spiked_G_hat(T, m)
        # top eigenvector of G_hat
        w, V = np.linalg.eigh(G_hat)
        v_top = V[:, -1]
        # squared overlap |<v_top, u>|^2
        overlaps.append((v_top @ u)**2)
    overlaps = np.array(overlaps)
    overlap_means.append(overlaps.mean())
    overlap_std.append(overlaps.std())

overlap_means = np.array(overlap_means)
overlap_std = np.array(overlap_std)

plt.figure(figsize=(7, 4))
plt.errorbar(beta_vals, overlap_means, yerr=overlap_std, fmt='o', label=r"Empi
plt.axvline(beta_c, color='red', linestyle=':', label=r"BBP threshold $\beta_c
plt.xlabel(r"Spike strength $\beta$")
plt.ylabel(r"Squared overlap $|\langle v_{\max}, u\rangle|^2$")
plt.title(f"Eigenvector alignment: d={d}, gamma={gamma}")
plt.legend()
plt.tight_layout()
plt.show()
```

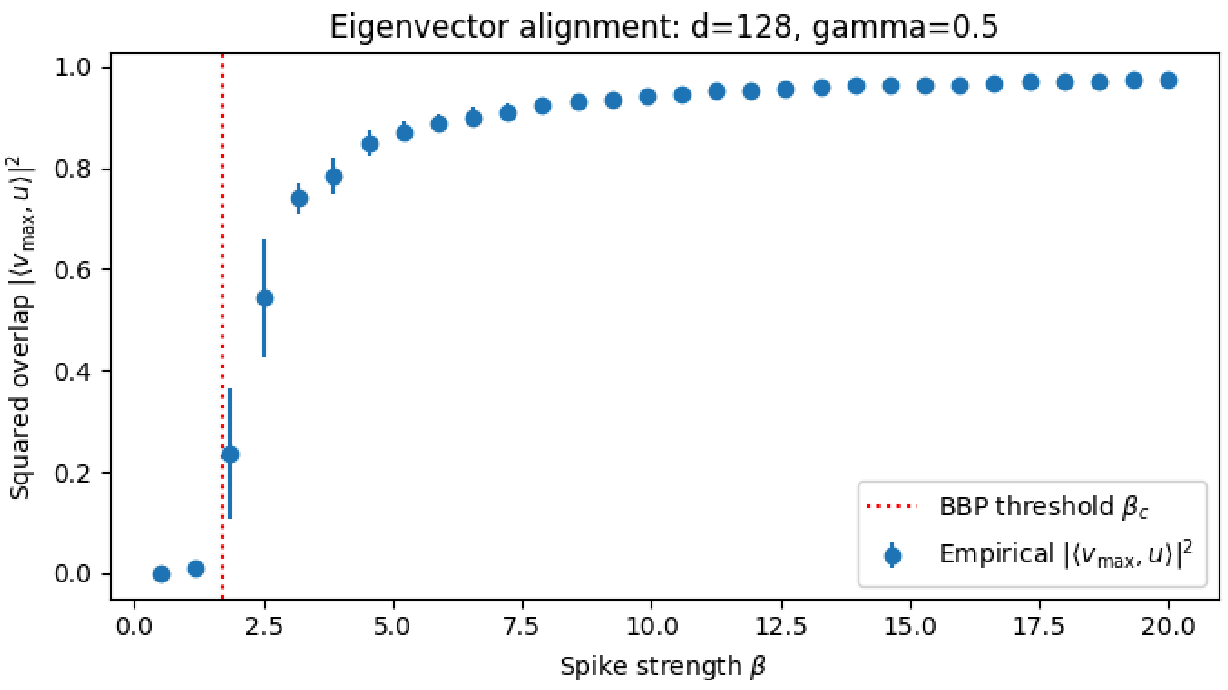
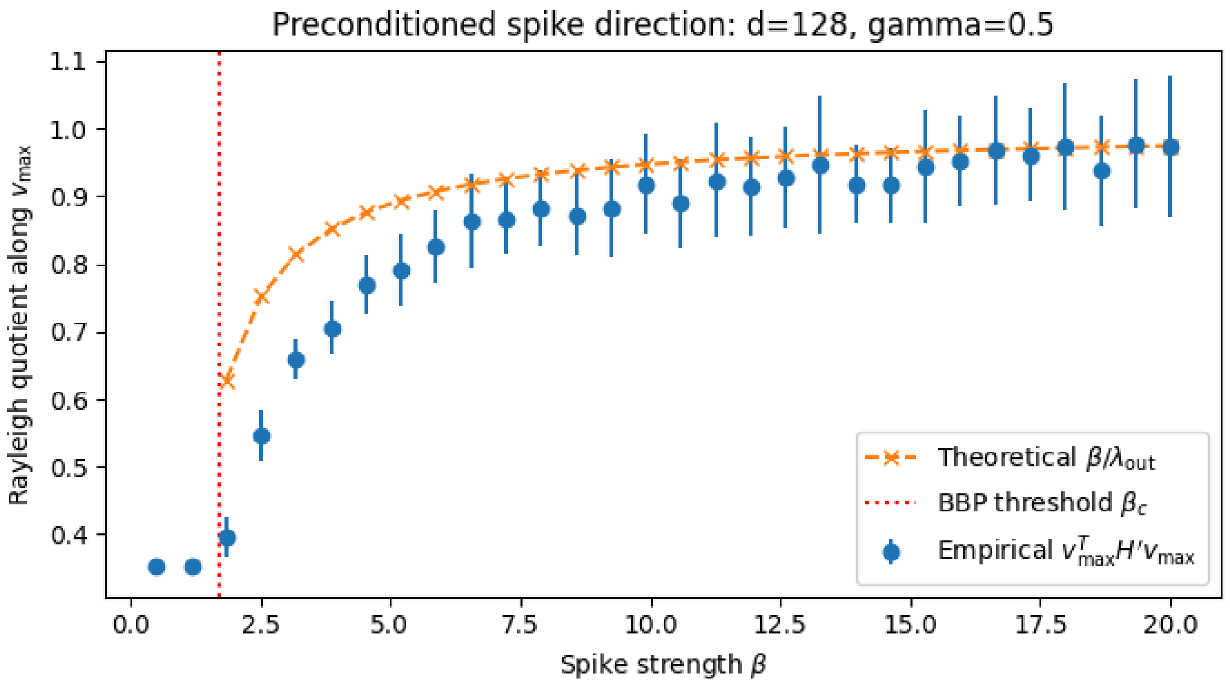
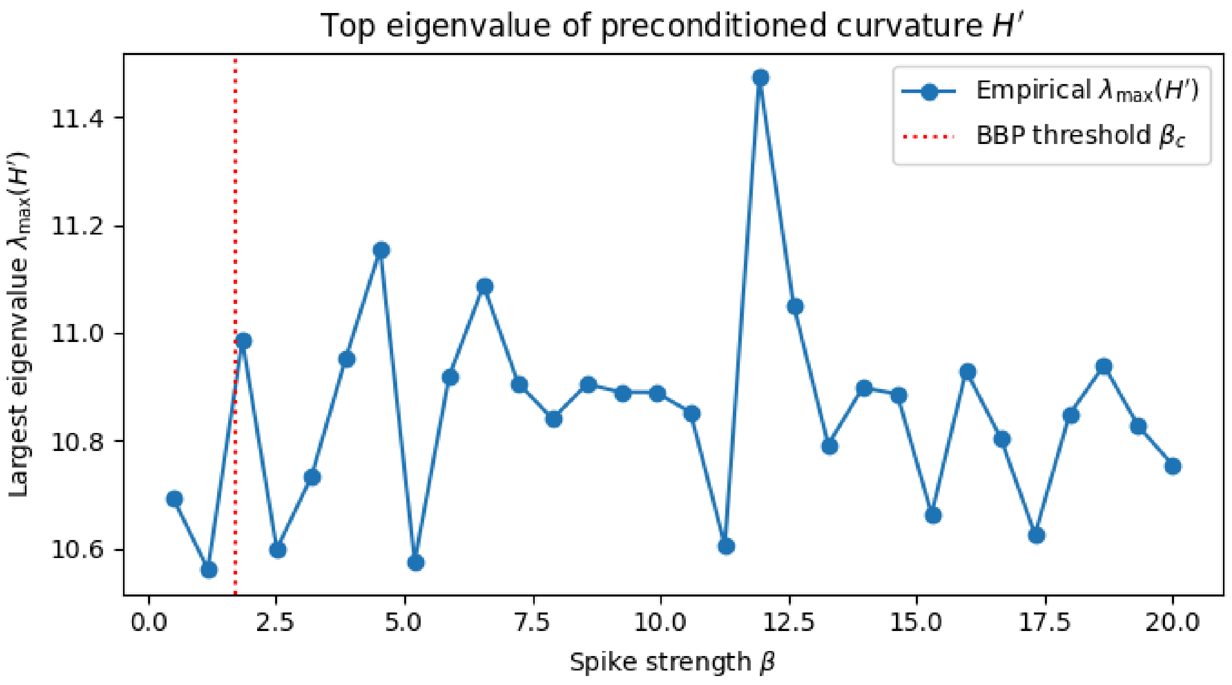


Diagram showing the effect on preconditioned curvature H'

```
In [19]: def preconditioned_curvature(T, G_hat):
             G_inv_sqrt = inv_sqrtm_spd(G_hat, eps=1e-10)
             return G_inv_sqrt @ T @ G_inv_sqrt

         rayleigh_means = []
         rayleigh_std = []
         lam_max_Hp_means = []

         for beta in beta_vals:
             rq_vals = []
             lam_max_vals = []
             for _ in range(n_trials):
                 T, u = make_spike_T(d, beta)
                 G_hat = sample_spiked_G_hat(T, m)
                 H_prime = preconditioned_curvature(T, G_hat)

                 # eigen-decomp of G_hat
                 w_G, V_G = np.linalg.eigh(G_hat)
                 v_top = V_G[:, -1]
                 # Rayleigh quotient along v_top
                 rq = float(v_top.T @ H_prime @ v_top)
                 rq_vals.append(rq)

                 # lambda_max(H')
                 eig_Hp = spectrum(H_prime)
                 lam_max_vals.append(eig_Hp[-1])

             rq_vals = np.array(rq_vals)
             lam_max_vals = np.array(lam_max_vals)
             rayleigh_means.append(rq_vals.mean())
             rayleigh_std.append(rq_vals.std())
             lam_max_Hp_means.append(lam_max_vals.mean())

         rayleigh_means = np.array(rayleigh_means)
         rayleigh_std = np.array(rayleigh_std)
         lam_max_Hp_means = np.array(lam_max_Hp_means)

         # Theoretical shrinkage beta / lambda_out (only meaningful above threshold)
         theoretical_rq = np.empty_like(beta_vals)
         for i, beta in enumerate(beta_vals):
             if beta > beta_c:
                 lambda_out = bbp_outlier(beta, gamma)
                 theoretical_rq[i] = beta / lambda_out
             else:
                 theoretical_rq[i] = np.nan  # undefined / not BBP regime

         plt.figure(figsize=(7, 4))
         plt.errorbar(beta_vals, rayleigh_means, yerr=rayleigh_std, fmt='o', label=r"Em
         plt.plot(beta_vals, theoretical_rq, 'x--', label=r"Theoretical $\beta / \lambd
         plt.axvline(beta_c, color='red', linestyle=':', label=r"BBP threshold $\beta_c
         plt.xlabel(r"Spike strength $\beta$")
         plt.ylabel(r"Rayleigh quotient along $v_{\max}$")
         plt.title(f"Preconditioned spike direction: d={d}, gamma={gamma}")
         plt.legend()
         plt.tight_layout()
         plt.show()

         plt.figure(figsize=(7, 4))
         plt.plot(beta_vals, lam_max_Hp_means, 'o-', label=r"Empirical $\lambda_{\max}(
         plt.axvline(beta_c, color='red', linestyle=':', label=r"BBP threshold $\beta_c
         plt.xlabel(r"Spike strength $\beta$")
         plt.ylabel(r"Largest eigenvalue $\lambda_{\max}(H')$")
         plt.title(f"Top eigenvalue of preconditioned curvature $H'$")
         plt.legend()
         plt.tight_layout()
         plt.show()
```





Experiment 3

Experiment 1 for theorem 3

In [21]:
```
# Basic helpers
def spiked_T(d, beta, rng):
    """Construct T = I + (beta - 1) u u^T with ||u||=1."""
    u = rng.normal(size=d)
    u /= np.linalg.norm(u)
    T = np.eye(d) + (beta - 1.0) * np.outer(u, u)
    return T, u

def op_norm(A):
    """Operator norm (spectral norm) of a symmetric matrix."""
    # For symmetric A, spectral norm = max |eigenvalue|
    vals = np.linalg.eigvalsh(A)
    return np.max(np.abs(vals))

def principal_angle_1d(v, w):
    """Principal angle between 1D subspaces span{v}, span{w}."""
    v = v / np.linalg.norm(v)
    w = w / np.linalg.norm(w)
    c = np.abs(np.dot(v, w))
    c = np.clip(c, -1.0, 1.0)
    return np.arccos(c)  # in radians
```

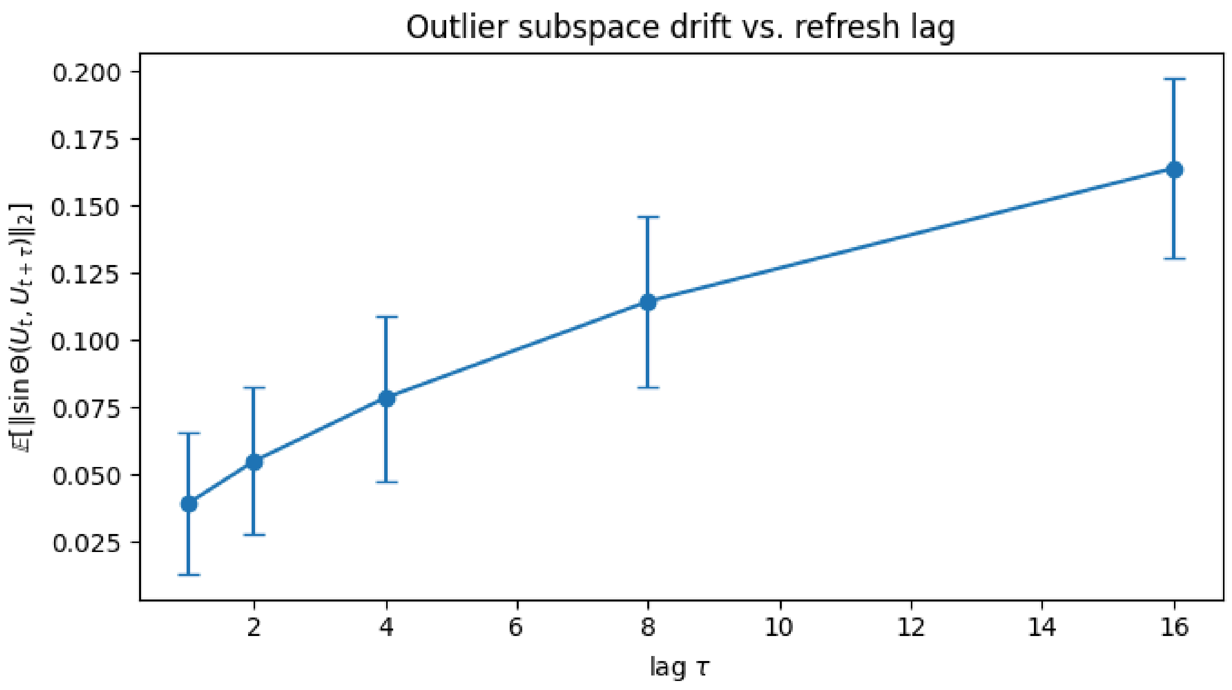
In [23]:
```
# Simulation parameters
d = 128
```

```
gamma = 0.5
m_eff = int(d / gamma)  # effective sample size ~ 256
rho = 1.0 / m_eff       # EMA rate

beta_spike = 3.0        # supercritical spike strength (> 1 + sqrt(gamma))
T, u = spiked_T(d, beta_spike, rng)
H = T                   # curvature proxy for these experiments

T_steps = 800           # total time steps
burn_in = 200           # discard early steps when analyzing

# Initialize G_0 reasonably close to T (short warm-up)
G = T.copy()
Gs = [G.copy()]

for t in range(1, T_steps + 1):
    g = rng.multivariate_normal(mean=np.zeros(d), cov=T)
    G = (1.0 - rho) * G + rho * np.outer(g, g)
    Gs.append(G.copy())

len(Gs), Gs[0].shape
```

Out[23]:
```
(801, (128, 128))
```

In [24]:
```
eigvals = []
eigvecs = []

for G in Gs:
    vals, vecs = np.linalg.eigh(G)
    # sort in descending order
    idx = vals.argsort()[::-1]
    vals = vals[idx]
    vecs = vecs[:, idx]
    eigvals.append(vals)
    eigvecs.append(vecs)

eigvals = np.array(eigvals)  # shape: (T_steps+1, d)
```

In [25]:
```
taus = [1, 2, 4, 8, 16]
max_tau = max(taus)

t0_indices = np.arange(burn_in, T_steps - max_tau, 5)  # sample base times eve

records_dk = []  # (tau, t0, delta_G, sin_theta, gap_t)

for tau in taus:
    for t0 in t0_indices:
        G_t = Gs[t0]
        G_ttau = Gs[t0 + tau]

        # Matrix drift
        delta_G = op_norm(G_ttau - G_t)
        # Outlier (leading eigenvector) at t and t+tau
        v0 = eigvecs[t0][:, 0]
        v_tau = eigvecs[t0 + tau][:, 0]

        theta = principal_angle_1d(v0, v_tau)
        sin_theta = np.sin(theta)

        # Empirical eigengap between outlier and bulk:
        gap_t = eigvals[t0][0] - eigvals[t0][1]

        records_dk.append((tau, t0, delta_G, sin_theta, gap_t))

len(records_dk)


records_dk = np.array(records_dk)
taus_rec = records_dk[:, 0]
t0_rec = records_dk[:, 1]
delta_G_rec = records_dk[:, 2]
sin_theta_rec = records_dk[:, 3]
gap_rec = records_dk[:, 4]

means = []
stds = []

for tau in taus:
    mask = (taus_rec == tau)
    means.append(np.mean(sin_theta_rec[mask]))
    stds.append(np.std(sin_theta_rec[mask]))

plt.figure(figsize=(7, 4))
plt.errorbar(taus, means, yerr=stds, marker="o", capsize=4)
plt.xlabel(r"lag $\tau$")
plt.ylabel(r"$\mathbb{E}[\|\sin\Theta(U_t, U_{t+\tau})\|_2]$")
plt.title("Outlier subspace drift vs. refresh lag")
plt.tight_layout()
plt.show()

print("Mean eigengap between λ1 and λ2 over t0 >= burn_in:",
      gap_rec.mean())
print("Min eigengap over sampled t0:",
      gap_rec.min())
```



```
Mean eigengap between λ1 and λ2 over t0 >= burn_in: 1.2036517649726104
Min eigengap over sampled t0: 0.7777202366814442
```

Experiment 2 for theorem 3

In [ ]:
```
H_op_norm = np.max(np.linalg.eigvalsh(H))
H_op_norm
```

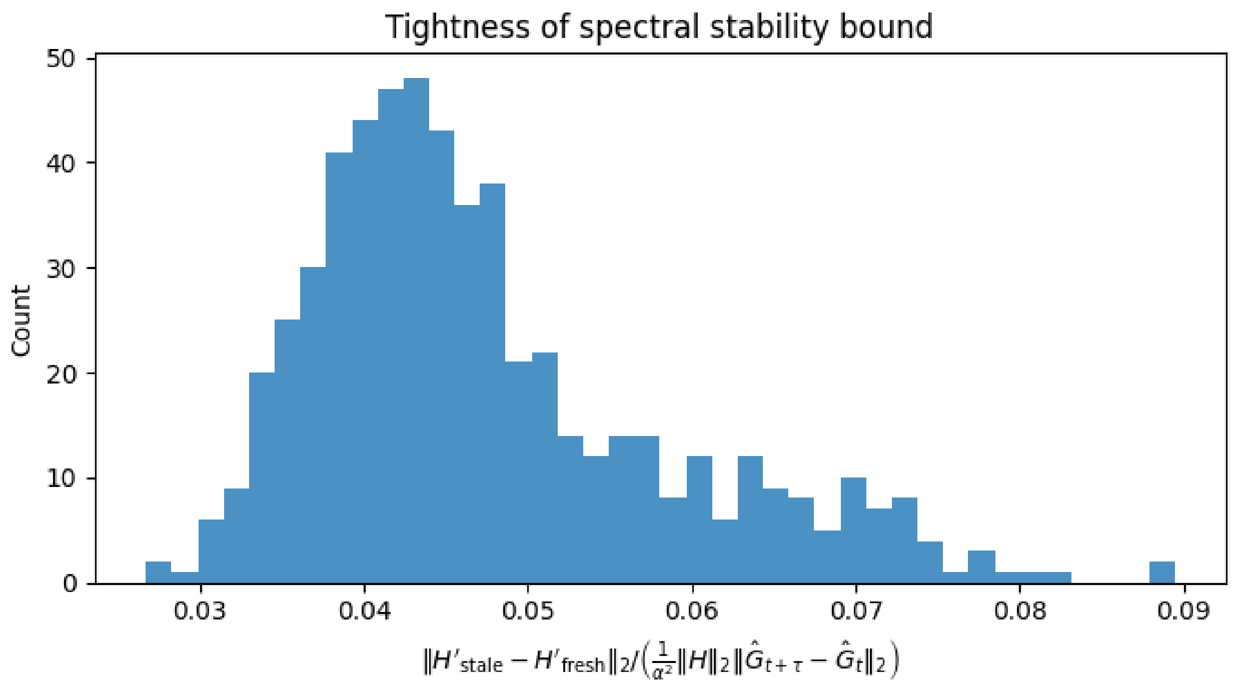
Out[ ]:
```
np.float64(3.0)
```

In [27]:
```
taus2 = [1, 2, 4, 8, 16]
max_tau2 = max(taus2)
t0_indices2 = np.arange(burn_in, T_steps - max_tau2, 5)

records_stale = []  # (tau, t0, ΔG, ΔH, bound, |Δλ_max|, |Δλ_min|)

for tau in taus2:
    for t0 in t0_indices2:
        G_t = Gs[t0]
        G_ttau = Gs[t0 + tau]

        # Matrix drift
        delta_G = op_norm(G_ttau - G_t)

        # Minimum eigenvalue (for alpha)
        lam_min_t = eigvals[t0][-1]
        lam_min_ttau = eigvals[t0 + tau][-1]
        alpha = min(lam_min_t, lam_min_ttau)

        # Preconditioners
        G_t_inv_sqrt = inv_sqrt_psd(G_t)
        G_ttau_inv_sqrt = inv_sqrt_psd(G_ttau)

        # Fresh vs stale H'
        H_fresh = G_ttau_inv_sqrt @ H @ G_ttau_inv_sqrt
        H_stale = G_t_inv_sqrt @ H @ G_t_inv_sqrt

        delta_H = op_norm(H_stale - H_fresh)

        # Eigenvalues
        lam_fresh = np.linalg.eigvalsh(H_fresh)
        lam_stale = np.linalg.eigvalsh(H_stale)

        delta_lam_max = np.abs(lam_stale[-1] - lam_fresh[-1])
        delta_lam_min = np.abs(lam_stale[0] - lam_fresh[0])

        # Theoretical bound
        bound = (H_op_norm / (alpha ** 2)) * delta_G

        records_stale.append(
            (tau, t0, delta_G, delta_H, bound, delta_lam_max, delta_lam_min)
        )

records_stale = np.array(records_stale)
records_stale.shape
```

Out[27]:
```
(585, 7)
```

In [28]:
```
taus_s = records_stale[:, 0]
t0_s = records_stale[:, 1]
delta_G_s = records_stale[:, 2]
delta_H_s = records_stale[:, 3]
bound_s = records_stale[:, 4]
delta_lam_max_s = records_stale[:, 5]
delta_lam_min_s = records_stale[:, 6]

ratio_s = delta_H_s / bound_s  # how tight is the bound?
```

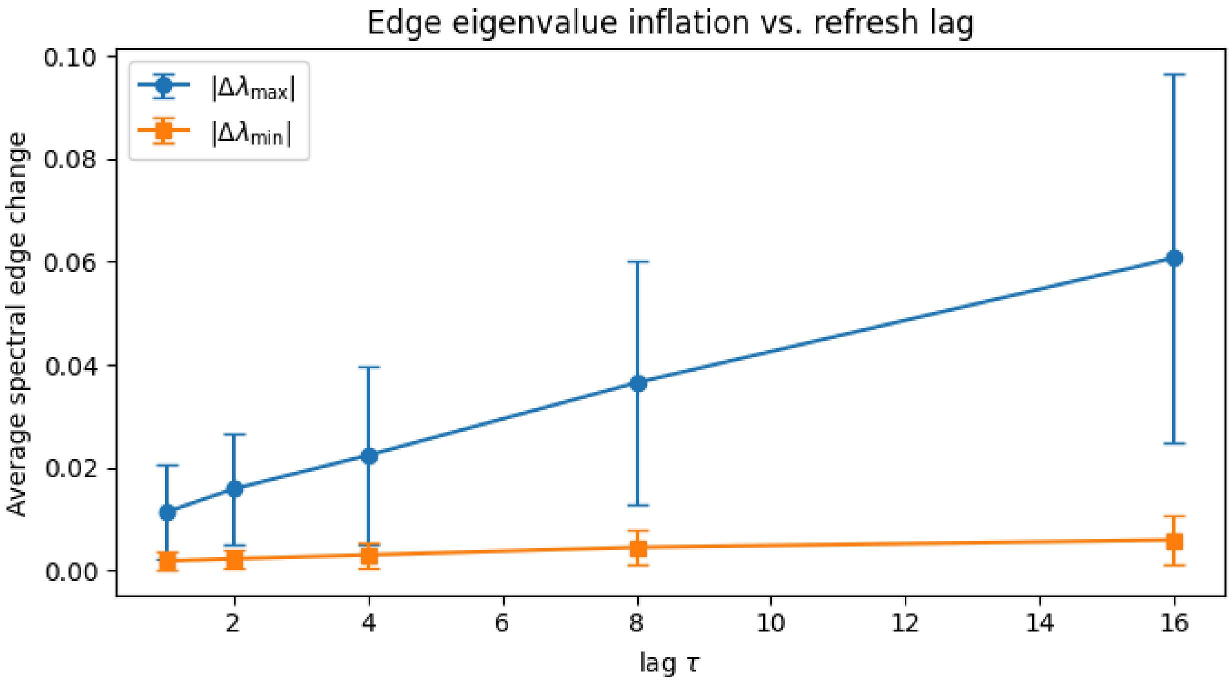
In [29]:
```
plt.figure(figsize=(7, 4))
plt.hist(ratio_s, bins=40, alpha=0.8)
plt.xlabel(r"$\|H'_{\mathrm{stale}} - H'_{\mathrm{fresh}}\|_2 / \left(\frac{1}
plt.ylabel("Count")
plt.title("Tightness of spectral stability bound")
plt.tight_layout()
plt.show()

print("Mean ratio:", np.mean(ratio_s))
print("Max ratio:", np.max(ratio_s))
```



```
Mean ratio: 0.047420530891105955
Max ratio: 0.08943701977392546
```

In [30]:

```python
means_max = []
stds_max = []
means_min = []
stds_min = []

for tau in taus2:
    mask = (taus_s == tau)
    means_max.append(np.mean(delta_lam_max_s[mask]))
    stds_max.append(np.std(delta_lam_max_s[mask]))
    means_min.append(np.mean(delta_lam_min_s[mask]))
    stds_min.append(np.std(delta_lam_min_s[mask]))

plt.figure(figsize=(7, 4))
plt.errorbar(taus2, means_max, yerr=stds_max, marker="o", capsize=4,
             label=r"$|\Delta\lambda_{\max}|$")
plt.errorbar(taus2, means_min, yerr=stds_min, marker="s", capsize=4,
             label=r"$|\Delta\lambda_{\min}|$")
plt.xlabel(r"lag $\tau$")
plt.ylabel("Average spectral edge change")
plt.title("Edge eigenvalue inflation vs. refresh lag")
plt.legend()
plt.tight_layout()
plt.show()
```


Edge eigenvalue inflation vs. refresh lag
$|\Delta\lambda_{\max}|$
$|\Delta\lambda_{\min}|$
Average spectral edge change
0.00
0.02
0.04
0.06
0.08
0.10
lag τ
2
4
6
8
10
12
14
16