Downlink MIMO Vector Perturbation Precoding to QUBO

Karthik Mudakalli

MIMO basics

- MIMO stands for Multiple Input Multiple Output
- Inputs are transmit antennas at the base station
- Outputs are receive antennas at user devices
- Let N_t be the number of transmit antennas
- Let N_r be the number of users
- The channel is a complex matrix $H \in \mathbb{C}^{N_r \times N_t}$
- Entry h_{ij} is a complex number describing how the signal from transmit antenna j arrives at user i with amplitude scaling and phase shift
- If the base station sends a complex vector $x \in \mathbb{C}^{N_t}$ then the users receive $y \in \mathbb{C}^{N_r}$
- The linear model is $y = Hx + n$ where n is additive noise
- Real and imaginary parts of n are commonly modeled as independent zero mean Gaussian variables with equal variance
- This is called complex additive white Gaussian noise

Why complex entries?

- Wireless signals have both amplitude and phase
- A complex number $ae^{j\theta} = a\cos(\theta) + ja\sin(\theta)$ captures both in one value
- Channel coefficients h_{ij} are complex since they describe amplitude scaling and phase shift from antenna j to user i
- Transmit vector $x \in \mathbb{C}^{N_t}$ has complex entries because antennas send complex-valued modulation symbols (e.g. QPSK, QAM)
- Receive vector $y \in \mathbb{C}^{N_r}$ is also complex because received signals still carry amplitude and phase
- Restricting to real numbers would lose phase information, which is essential in wireless communication

Downlink and interference

Downlink means the base station sends signals to users. The challenge is that one transmit vector must carry the data for all users at once. This creates multiuser interference since each user receives a mixture of all streams unless we pre process the signal. We handle this by precoding at the transmitter.

Symbols and constellations

Each user wants to receive a complex symbol that represents bits. The set of allowed symbols is a constellation. A common constellation is QPSK which stands for Quadrature Phase Shift Keying. The QPSK set is $\{1+j, 1-j, -1+j, -1-j\}$. These four points lie at plus or minus one on the real axis and plus or minus one on the imaginary axis. Since there are four symbols each symbol carries two bits. The vector of user symbols is $u \in \mathbb{C}^{N_r}$ where the ith entry is the symbol for user i.

QPSK in more words

QPSK stands for Quadrature Phase Shift Keying. It uses four points on the complex plane at coordinates $\pm 1 \pm j$. Each point can be labeled by two bits. For example one mapping is 00 to $1 + j$, 01 to $1 - j$, 10 to $-1 + j$, and 11 to $-1 - j$. The mapping is fixed by the system design. The key property is constant energy and equal distance between symbols along both axes.

What is precoding and zero forcing

Without processing if we transmit u each user receives a mixture of all symbols weighted by channel coefficients. Precoding shapes the transmit vector x so that users do not interfere. We focus on zero forcing precoding. The zero forcing precoder is

$$P = H^H \left(HH^H\right)^{-1}$$

when HH^H is invertible. The goal is that HP is close to the identity so that if we send $x = Pd$ then $y = HPd \approx d$. Each user receives mostly its own symbol.

Why vector perturbation

Zero forcing can result in large transmit power when H is poorly conditioned. Vector perturbation precoding reduces power by adding an integer shift before precoding. The transmit vector is

$$x = P\,(u + \tau v)$$

where $v \in \mathbb{Z}[j]^{N_r}$ is a **Gaussian integer** perturbation vector (each element has integer real and imaginary parts) and $\tau > 0$ is a lattice spacing parameter determined by the modulation constellation. By nudging u by integer multiples of τ, the pre-coded vector $P(u + \tau v)$ can have a smaller Euclidean norm, which means lower transmit power.

VPP objective

The base station chooses v to minimize transmit energy

$$v^\star = \arg\min_{v \in \mathbb{Z}^{N_r}} \; \left\| P \left(u + \tau v \right) \right\|_2^2 .$$

This is a closest vector problem in the lattice generated by the columns of P. Our goal is to reformulate this integer optimization as a QUBO so that it can be handled by solvers that accept quadratic objectives over binary variables.

Why convert to real

Optimization over binary variables is most convenient in the real domain. We map complex vectors and matrices to real ones. If $z = a + jb \in \mathbb{C}^n$ define

$$\phi(z) = \begin{bmatrix} a \\ b \end{bmatrix} \in \mathbb{R}^{2n}.$$

If $P = A + jB \in \mathbb{C}^{m \times n}$ define

$$\Phi(P) = \begin{bmatrix} A & -B \\ B & A \end{bmatrix} \in \mathbb{R}^{2m \times 2n}.$$

This mapping preserves matrix vector multiplication in the sense that $\phi(Pz) = \Phi(P)\phi(z)$ and preserves norms so that $\|Pz\|_2^2 = \|\Phi(P)\phi(z)\|_2^2$.

Define F, y, and G

Let $F = \Phi(P) \in \mathbb{R}^{2N_t \times 2N_r}$ and let $y = \phi(u) \in \mathbb{R}^{2N_r}$ denote the real-valued representations of the complex precoder and user symbol vector, respectively. After stacking real and imaginary parts, the perturbation vector becomes a real integer vector $v \in \mathbb{Z}^{2N_r}$ derived from the complex Gaussian integer vector $v \in \mathbb{Z}[j]^{N_r}$.
The transmit energy becomes

$$\|P\,(u + \tau v)\|_2^2 = \|F\,(y + \tau v)\|_2^2 \,.$$

Define the Gram matrix $G = F^T F \in \mathbb{R}^{2N_r \times 2N_r}$. The energy is

$$E(v) = (y + \tau v)^T \, G \, (y + \tau v) \,,$$

which is a real quadratic function of the integer vector v.

What is a QUBO

A QUBO is a quadratic unconstrained binary optimization. The unknown is a binary vector $q \in \{0,1\}^M$. The objective is a quadratic form

$$E(q) = q^T Q q + \text{offset}$$

where Q is a real symmetric matrix. Our task is to represent the integer vector v with a binary vector q so that the original energy becomes a quadratic function of q.

Encode integers with binary variables

Each integer component of the realified perturbation vector is encoded in binary. We use two's complement with t magnitude bits so each integer v_i uses $t + 1$ bits. The weights are $[-2^t, 2^0, 2^1, \ldots, 2^{t-1}]$. For example, if $t = 2$, then the weights are $[-4, 1, 2]$ and the represented integers range from -4 to 3.

The parameter t controls the **bit depth and dynamic range** of each integer variable: increasing t expands the search space for the perturbation values.

We build a block-diagonal encoding matrix $C \in \mathbb{R}^{(2N_r)\times M}$ with one weight block per integer so that

$$v = Cq, \qquad q \in \{0,1\}^M, \qquad M = (t+1)\cdot 2N_r.$$

We also keep a decode function $q \mapsto v = Cq$ to interpret solutions.

Derive the QUBO

Start from the real energy

$$E(v) = (y + \tau v)^T \, G \, (y + \tau v) \, .$$

Substitute $v = Cq$ to get

$$E(q) = (y + \tau Cq)^T \, G \, (y + \tau Cq) \, .$$

Use $(a+b)^T G(a+b) = a^T Ga + 2a^T Gb + b^T Gb$ with $a = y$ and $b = \tau Cq$:

$$E(q) = y^T Gy + 2\tau \, y^T GC \, q + \tau^2 \, q^T C^T GC \, q.$$

Define $A = \tau^2 C^T GC$ and $b^T = 2\tau \, y^T GC$. Make A symmetric by $Q \leftarrow \frac{1}{2}(A + A^T)$. Since $q_i^2 = q_i$ for binary q_i we fold the linear term into the diagonal by updating $Q_{ii} \leftarrow Q_{ii} + b_i$. The constant offset is $y^T Gy$. The final QUBO is $E(q) = q^T Qq + \text{offset}$.

Algorithm pipeline

Step one. Input H and u. Step two. Compute $P = H^H(HH^H)^{-1}$. Step three. Realify P and u to F and y. Step four. Compute $G = F^T F$. Step five. Choose t and build C. Step six. Compute $A = \tau^2 C^T GC$, compute $b = 2\tau\, C^T Gy$, and the constant $y^T Gy$. Step seven. Symmetrize A to Q and add b to the diagonal. The QUBO instance is (Q, offset) together with the decoder $v = Cq$.

Code files

- `encoding.py` Implements complex to real mapping and integer encoding.
- `vpp.py` Implements the VPP to QUBO pipeline and a verification routine.
- `vpp_qubo_example.py` Creates random toy instances, builds the QUBO, runs brute force checks, and prints the result.

encoding.py

Function complex_to_real maps complex vectors to stacked real and imaginary parts and maps complex matrices to the standard block form with A and B. This preserves matrix vector multiplication and norms. Function build_integer_encoding takes the number of integer variables and t and returns the block diagonal matrix C with weights $[-2^t, 1, 2, \ldots, 2^{t-1}]$. It also returns a decode function that applies C to a binary vector to recover the integer vector.

encoding.py explained

```
import numpy as np  # Import NumPy for arrays and linear algebra

def complex_to_real(M):
    M = np.asarray(M)  # Ensure M is a NumPy array to safely access .real and .imag

    # vector case
    if M.ndim == 1:  # If M is a 1D complex vector
        # complex vector u = a + j b
        # map to real vector: [Re(u); Im(u)]
        a = M.real   # Extract real part
        b = M.imag   # Extract imaginary part
        return np.concatenate([a, b], axis=0).astype(np.float64)
        # Stack real above imaginary parts into one vector of doubles

    # matrix case
    elif M.ndim == 2:  # If M is a 2D complex matrix
        A = M.real     # Real part of the matrix
        B = M.imag     # Imaginary part of the matrix
        top = np.concatenate([A, -B], axis=1)  # Build block [A  -B]
        bot = np.concatenate([B,  A], axis=1)  # Build block [B   A]
        return np.concatenate([top, bot], axis=0).astype(np.float64)
        # Stack top and bottom blocks to form real-valued block matrix

    else:
        raise ValueError("M must be 1D or 2D")
        # Error if input is not a vector or matrix
```

encoding.py explained continued

```
# builds binary encoding matrix C for n integers
def build_integer_encoding(n_vars, t=1):
    cols_per = t + 1  # Bits per integer (t magnitude bits + 1 sign bit)
    C = np.zeros((n_vars, n_vars * cols_per), dtype=int)
    # Allocate encoding matrix of size (n_vars x n_vars*cols_per)

    # 2's complement weights: [ -2^t, 1, 2, ..., 2^(t-1) ]
    weights = np.array([-(2**t)] + [2**i for i in range(t)], dtype=int)
    # This defines how binary bits map to signed integer values

    # Fill block diagonal where each integer gets its own set of weights
    for k in range(n_vars):
        C[k, k*cols_per:(k+1)*cols_per] = weights

    def decode(q):
        q = np.asarray(q).reshape(-1)  # Flatten binary vector q
        return (C @ q).astype(int)     # Multiply by C to recover integers

    return C, decode  # Return both the encoding matrix and the decoder
```

vpp.py main steps

Function zf_precoder returns $P = H^H(HH^H)^{-1}$. Function build_basis_map returns $F = \Phi(P)$. Function gram_of_map returns $G = F^T F$. Function form_qubo_core takes G, y, τ, and C and builds Q and the constant offset by computing A, b, symmetrizing A, and adding b to the diagonal. Function qubo_from_vpp ties the steps together. It returns Q, offset, C, and the decode function.

vpp.py verification

Function brute_force_check is a verifier for small toy problems. It does two exhaustive searches. The first is a direct integer search over a small set of integer vectors v that are consistent with the chosen t. It computes $E = (y + \tau v)^T G(y + \tau v)$ and tracks the minimum. The second is a binary search over all q of length M where $M = (t + 1) \cdot 2N_r$. It computes $E\_\text{qubo} = q^T Qq + \text{offset}$ and tracks the minimum then decodes the best q into v. It compares the best energies and the best perturbations. It skips if M is too large since 2^M grows very fast. Agreement confirms that the QUBO encodes the original objective.

vpp.py explained line by line

```
import numpy as np                      # Numerical computations
from itertools import product           # Used for brute-force search
from .encoding import complex_to_real  # Helper to convert complex to real form

# Zero-Forcing (ZF) precoder from 2102.12540v1.pdf
# Formula: P = H^H * (H H^H)^(-1)
def zf_precoder(H):
    H = np.asarray(H, dtype=np.complex128)   # Ensure H is complex128 array
    return H.conj().T @ np.linalg.pinv(H @ H.conj().T)

# Gram matrix G = F^T F
# F = real-valued basis matrix from precoder P
# G encodes inner products of basis vectors; needed to make cost quadratic in v
def gram_of_map(F):
    return F.T @ F

# Convert complex precoder P into real matrix representation
# So optimization is expressed over real numbers (not complex)
def build_basis_map(P):
    return complex_to_real(P)
```

vpp.py explained line by line

```
# Integer-to-binary encoding: creates block-diagonal matrix C
# v = C q  where v is integer vector, q is binary vector
def build_integer_encoding(n, t=1):
    bits_per_int = t + 1                # Each integer encoded with t+1 bits (2’s complement)
    m = n * bits_per_int                # Total number of binary variables
    weights = np.array([-(2**t)] + [2**i for i in range(t)], dtype=int)
    # Example: t=2 → weights = [-4, 1, 2]
    C = np.zeros((n, m), dtype=int)
    for i in range(n):
        C[i, i*bits_per_int:(i+1)*bits_per_int] = weights
    def decode(q):                      # Helper: recover integer vector from q
        q = np.asarray(q).reshape(-1)
        return (C @ q).astype(int)
    return C, decode
```

vpp.py explained line by line

```
# Build QUBO cost matrix
def form_qubo_core(G, y_vec, tau, C, const_offset=0.0):
    y = y_vec.reshape(-1, 1)                        # Column vector
    A = tau**2 * (C.T @ G @ C)                      # Quadratic term
    b = 2.0 * tau * (y.T @ G @ C).reshape(-1)   # Linear term
    const = float(y.T @ G @ y) + float(const_offset)  # Constant term
    Q = 0.5 * (A + A.T)                             # Symmetrize Q
    for i in range(Q.shape[0]):                     # Add linear term to diagonal
        Q[i, i] += b[i]
    Q = 0.5 * (Q + Q.T)                             # Final symmetrization
    return Q, const                                 # Return QUBO form

# Full QUBO construction pipeline from VPP
def qubo_from_vpp(H, u, tau, t=1):
    H = np.asarray(H, dtype=np.complex128)     # Channel matrix
    u = np.asarray(u, dtype=np.complex128).reshape(-1)   # User symbols
    Nr = H.shape[0]                            # Number of users
    P = zf_precoder(H)                         # ZF precoder
    F = build_basis_map(P)                     # Realify basis
    u_r = complex_to_real(u)                   # Realify symbols
    C, decode = build_integer_encoding(2*Nr, t=t)   # Encoding matrix
    G = gram_of_map(F)                         # Gram matrix
    Q, const = form_qubo_core(G, u_r, tau, C)  # QUBO form
    return {"Q": Q, "offset": const, "C": C, "decode": decode}
```

vpp.py explained line by line

```python
# Verification function (brute force)
def brute_force_check(H, u, tau, Q, offset, C, decode, t=1):
    Nr = H.shape[0]
    F = build_basis_map(zf_precoder(H))        # Build basis
    G = gram_of_map(F)
    y = complex_to_real(u)
    vals = list(range(-(2**t), 2**t))          # Candidate integers
    cand = np.array(list(product(*([vals] * (2*Nr)))), dtype=int)
    # Direct method: try every l
    bestE, bestl = np.inf, None
    for l in cand:
        v = y.reshape(-1) + tau * l
        E = float(v @ G @ v)
        if E < bestE:
            bestE, bestl = E, l.copy()
    # Brute-force QUBO: try every binary vector q
    M = Q.shape[0]
    if M > 26:                                 # Hard cutoff (too large)
        return {"ok": False, "reason": "too many variables",
                "bestE_direct": bestE, "best_l_direct": bestl}
    bestQ, bestq = np.inf, None
    for z in range(1 << M):                    # Check all 2^M binary vectors
        q = np.array([(z >> i) & 1 for i in range(M)], float)
        E = float(q @ Q @ q + offset)
        if E < bestQ:
            bestQ, bestq = E, q
    l_from_qubo = decode(bestq)                # Decode binary solution
    return {"ok": True, "E_direct": bestE, "E_qubo": bestQ,
          "l_from_qubo": l_from_qubo, "l_direct": bestl}
```

vpp.py explained line by line

```
# Random example generator
def sample_and_build(seed=0):
    rng = np.random.default_rng(seed)          # Random generator
    Nr = Nt = 2                                # Fixed small dimension
    # Channel matrix H: complex Gaussian i.i.d. entries, unit variance
    H = (rng.normal(size=(Nr, Nt)) + 1j * rng.normal(size=(Nr, Nt))) / np.sqrt(2)
    # QPSK constellation {±1 ± j}
    const = np.array([1+1j, 1-1j, -1+1j, -1-1j])
    u = rng.choice(const, size=Nr)             # Random user symbols
    tau = 4.0                                  # Arbitrary spacing parameter
    data = qubo_from_vpp(H, u, tau, t=1)
    return H, u, tau, data
```

vpp_qubo_example.py

Function sample_and_build sets $N_r = N_t = 2$. It draws a random complex channel H with independent standard normal real and imaginary parts then divides by $\sqrt{2}$ so each complex coefficient has unit variance. It draws user symbols u from QPSK. It sets $\tau = 4.0$. It calls qubo_from_vpp to build Q offset C and decode. In the main script we call sample_and_build and then call brute_force_check. The script prints H, u, τ, Q, offset, both best energies, both perturbation vectors, and a boolean that shows whether the energies match.

vpp_qubo_example.py explained

```
import numpy as np
from vpp_qubo.vpp import sample_and_build, brute_force_check

# -------------------------
# Example 1
# -------------------------

# Generate random VPP instance with fixed seed for reproducibility
H, u, tau, data = sample_and_build(seed=3)

# sample_and_build returns:
#   H    : random 2x2 channel matrix (complex Gaussian entries)
#   u    : random user symbols (QPSK constellation)
#   tau  : spacing parameter (set = 4.0 inside sample_and_build)
#   data : dictionary containing QUBO info:
#          Q (matrix), offset (scalar), C (encoding matrix), decode (function)

# Check correctness using brute force:
#   1. Direct method: search all possible integer perturbations l
#   2. QUBO method  : search all possible binary vectors q
#   3. Compare resulting minimum energies and decoded perturbations
res = brute_force_check(H, u, tau, data["Q"], data["offset"],
                        data["C"], data["decode"])
```

vpp_qubo_example.py explained

```
# Print results for Example 1
print("\n")
print("Example 1: ")
print("Channel matrix H:\n", H)
print("User symbols u:", u)
print("Tau:", tau)
print("QUBO matrix Q:\n", data["Q"])
print("Offset:", data["offset"])
print("Direct best energy:", res["E_direct"])
print("QUBO best energy:", res["E_qubo"])
print("Decoded perturbation vector l* from QUBO:", res["l_from_qubo"])
print("Decoded perturbation vector l* direct:", res["l_direct"])

# Numerical tolerance: check if results match
# Here, 1e-8 was arbitrarily chosen as tolerance threshold
print("Match:", abs(res["E_direct"] - res["E_qubo"]) < 1e-8)
```

vpp_qubo_example.py explained

```
# -------------------------
# Example 2
# -------------------------

# Another random instance with different seed
H2, u2, tau2, data2 = sample_and_build(seed=4)

# Perform brute force check again
res2 = brute_force_check(H2, u2, tau2,
                         data2["Q"], data2["offset"],
                         data2["C"], data2["decode"])

# Print results for Example 2
print("\n")
print("Example 2: ")
print("Channel matrix H:\n", H2)
print("User symbols u:", u2)
print("Tau:", tau2)
print("QUBO matrix Q:\n", data2["Q"])
print("Offset:", data2["offset"])
print("Direct best energy:", res2["E_direct"])
print("QUBO best energy:", res2["E_qubo"])
print("Decoded perturbation vector l* from QUBO:", res2["l_from_qubo"])
print("Decoded perturbation vector l* direct:", res2["l_direct"])
print("Match:", abs(res2["E_direct"] - res2["E_qubo"]) < 1e-8)
```

How to read output

The printed channel H is the complex channel of size two by two. The vector u has two QPSK symbols. The scalar τ is the spacing. The QUBO matrix Q is real symmetric with size equal to the number of binary variables M. The offset is $y^T G y$. The direct best energy is the minimum of the original energy over integer v. The QUBO best energy is the minimum of $q^T Q q +$ offset over binary q. The decoded perturbation from QUBO and the best perturbation from the direct search should match on the toy examples which confirms the construction.

raw program output)

```
Example 1:
Channel matrix H:
 [[ 1.44314775-0.32007138j -1.80712807-0.15245022j]
 [ 0.29564053-1.42834589j -0.40147374-0.16400096j]]
User symbols u: [-1.+1.j -1.+1.j]
Tau: 4.0
QUBO matrix Q:
 [[ 1.87529499e+01 -1.05755497e+01 ... -7.90783412e+00]
  ...
 [-7.90783412e+00  3.95391706e+00 ...  1.49070452e+01]]
Offset: 1.2904706720279664
Direct best energy: 1.2904706720279664
QUBO best energy: 1.2904706720279664
Decoded perturbation vector l* from QUBO: [0 0 0 0]
Decoded perturbation vector l* direct: [0 0 0 0]
Match: True

Example 2:
Channel matrix H:
 [[-0.46088594-1.16064316j -0.12354378-0.00367926j]
 [ 1.17643052-0.44085544j  0.46608784+0.10509836j]]
User symbols u: [-1.+1.j -1.-1.j]
Tau: 4.0
QUBO matrix Q:
 [[ 2.93803688e+02 -1.40341844e+02 ...  1.20976166e+02]
```

Output Meaning

Example 1 and 2 results:

- **Channel matrix H:** A 2×2 complex matrix where each entry represents the gain of the channel between one transmit antenna and one user. Generated randomly using i.i.d. complex Gaussian samples (Rayleigh fading model).
- **User symbols u:** A length-2 vector, one QPSK symbol per user. For Example 1, $u = [-1 + j, -1 + j]$.
- **Tau (τ):** Spacing parameter used for perturbations. Here fixed to $\tau = 4.0$.
- **QUBO matrix Q:** The quadratic binary optimization matrix. It encodes the cost function in binary form. Large numbers come from inner products of basis vectors; near-zero values are numerical rounding errors.
- **Offset:** Constant term in the cost function. Both Q and offset are needed to evaluate the QUBO energy.
- **Direct best energy:** Minimum energy found by brute-force search over integer perturbations l.

Output meaning

Example 1 and 2 results:

- **QUBO best energy:** Minimum energy found by brute-force search over binary vectors q.
- **Decoded perturbation vector l^*:** The integer perturbation that minimizes the energy. Both methods produced $[0, 0, 0, 0]$.
- **Match = True:** Confirms that the QUBO formulation exactly matches the original mathematical formulation for these examples.

Assumptions in toy implementation

- Restricted examples to 2×2 channels to keep exhaustive checks feasible
- Generated H with independent standard normal real and imaginary parts, scaled by $1/\sqrt{2}$ so each complex coefficient has unit variance
- Used QPSK for u
- Set $\tau = 4.0$ as a simple spacing for the example
- Used two's complement with $t + 1$ bits per integer
- Limited brute force QUBO search to at most 26 binary variables to avoid explosive runtime
- Validated derivation by comparing direct and QUBO searches on small cases

Why the QUBO is quadratic

The norm squared of a linear function is a quadratic function. We have $\|F(y+\tau v)\|_2^2 = (y+\tau v)^T G(y+\tau v)$ where $G = F^T F$. This is quadratic in v. After encoding $v = Cq$ it becomes quadratic in q. Therefore a QUBO form is natural for this problem.

Why fold the linear term into the diagonal

For binary variables we have $q_i^2 = q_i$. A linear term $b_i q_i$ can be written as $b_i q_i^2$ and added to the diagonal entry Q_{ii}. This keeps the objective in pure quadratic form $q^T Q q$. The constant offset remains unchanged.

Receivers and the perturbation

Users do not need to know the explicit vector v if the receiver applies a modulo mapping that wraps symbols back into the original constellation region. The transmitter chooses v to reduce power. The receivers then map the received symbol into the basic region. This is the standard idea in vector perturbation schemes.

Why QPSK and why tau equals four

QPSK is the simplest complex constellation with nonzero real and imaginary parts and gives a clean toy example. The spacing parameter τ must be positive and of suitable scale relative to the constellation. I chose $\tau = 4.0$ for clarity in the example. Other values are possible and would change numeric entries in Q but not the derivation.

Channel sampling in the toy example

In rich scattering environments a complex baseband channel coefficient is often modeled as complex Gaussian with zero mean and independent real and imaginary parts. I draw H with independent standard normal real and imaginary entries and divide by $\sqrt{2}$ so that each complex coefficient has unit variance. This produces a Rayleigh fading model at the magnitude level. This choice is only for the toy script. The derivation does not depend on it.

Effect of $t = 1$ in Binary Encoding

The parameter t controls the number of magnitude bits for each integer component of the perturbation vector. When $t = 0$, each real component uses one bit. When $t = 1$, each component uses one sign bit and one magnitude bit, doubling the number of binary variables:

$$M = (t + 1) \times 2N_r.$$

For a 2×2 complex channel, $2N_r = 4$ and $M = 8$. Thus the QUBO matrix becomes 8×8. This is expected and indicates an expanded search range of integer values.

Reading the Output in Plain English

Channel and Symbols:

$$H \in \mathbb{C}^{2\times 2}, \quad u = [-1+j,\ -1+j], \quad \tau = 4.$$

H represents the downlink channel, u are QPSK symbols, and τ defines the lattice spacing.
QUBO matrix Q: An 8×8 matrix defining the quadratic energy function

$$E(q) = q^T Q q + \text{offset}.$$

The entries encode binary interactions between all bits of the realified perturbation vector.

Direct and QUBO Verification Methods

Two independent checks confirm the QUBO formulation:

1. **Direct method:** Evaluate

$$E(v) = \|F(y + \tau v)\|_2^2$$

   for integer vectors v directly using the real-valued model.

2. **QUBO method:** Encode $v = Cq$, minimize

$$q^T Q q + \text{offset},$$

   then decode $q \mapsto v$.

Matching energies and decoded vectors verify that the QUBO correctly reproduces the classical problem.

Example Results for $t = 1$

For both random 2×2 channel examples:

$$\text{Direct best energy} = \text{QUBO best energy},$$

and

$$l^*_{\text{QUBO}} = l^*_{\text{direct}} = [0, 0, 0, 0].$$

The perturbation vector is zero, meaning no integer offset reduced power further. "Match: True" confirms perfect equivalence of both formulations.

Summary

- Setting $t = 1$ doubles the binary variable count, yielding an 8×8 QUBO matrix.
- The dimension increase is correct and expected due to realification and two-bit encoding.
- The two verification paths—direct computation and QUBO minimization—produce identical results.
- This confirms the consistency and correctness of the VPP-to-QUBO mapping for the realified system.

Overview: From Complex Model to QUBO

1. Complex domain formulation:

$$x = P(u + \tau v), \quad v \in \mathbb{Z}[j]^{N_r}.$$

Minimize transmit power $\|P(u + \tau v)\|_2^2$ by choosing integer perturbation v.

2. Realification: Convert complex quantities into real form:

$$\phi(u) = \begin{bmatrix} \Re(u) \\ \Im(u) \end{bmatrix}, \quad \Phi(P) = \begin{bmatrix} \Re(P) & -\Im(P) \\ \Im(P) & \Re(P) \end{bmatrix}.$$

After this step, $v \in \mathbb{Z}^{2N_r}$.

3. Binary encoding: Represent each integer as $v = Cq$ with two's complement encoding. Parameter t controls the number of bits per integer:

$$M = (t + 1) \cdot 2N_r, \quad q \in \{0, 1\}^M.$$

4. QUBO formulation: Substitute $v = Cq$ into the quadratic energy:

$$E(q) = q^T Q q + \text{offset}, \quad Q = \tau^2 C^T G C + 2\tau C^T G y.$$

Minimizing this QUBO yields the same optimal perturbation as direct minimization in v.

Understanding the Parameter t

The parameter t controls the **binary precision** used to represent each integer variable in the perturbation vector v.

1. Purpose: t determines how many bits we use to describe the possible integer values of v_i. Each v_i is encoded using **two's complement** representation.

2. Bits used: Each integer variable uses $(t+1)$ bits:

$$\text{Weights: } [-2^t, 2^0, 2^1, \dots, 2^{t-1}].$$

3. Representable range:

$$v_i \in [-2^t,\ 2^t - 1].$$

Example:

- $t = 0$: weights $[-1] \rightarrow$ integers $\{0, -1\}$, 1 bit per variable.
- $t = 1$: weights $[-2, 1] \rightarrow$ integers $\{-2, -1, 0, 1\}$, 2 bits per variable.
- $t = 2$: weights $[-4, 1, 2] \rightarrow$ integers $\{-4, -3, -2, -1, 0, 1, 2, 3\}$, 3 bits per variable.

4. Impact: Increasing t increases the **search range and resolution** of the perturbation vector, and therefore increases the **QUBO size**:

$$M = (t+1) \times 2N_r.$$