Slide 1: Title (about 1:00)

Good morning. Thank you very much for the introduction. I would also like to thank you for inviting me to give this talk. My name is Kenta Kasai, from Institute of Science Tokyo. Today I will speak under the title "Bringing Classical LDPC Design Principles to Quantum LDPC Codes." Classical error correction is now very mature, both as theory and as technology. For classical LDPC codes, belief propagation can get very close to the limits predicted by modern coding theory. Classical error correction is also everywhere: storage, communication, and digital infrastructure all rely on it. Quantum error correction is in a very different stage. It is essential for large-scale quantum information processing, but the coding theory is still much younger. In particular, many quantum LDPC codes have not yet shown the sharp waterfall behavior that we know from classical LDPC codes. In this talk, I ask how much of the classical LDPC design idea we can carry over to quantum LDPC codes under the CSS constraints. **[Next slide.]**

Slide 2: Waterfall vs. Error Floor (about 0:55)

I first explain how we usually read finite-length decoding curves for classical LDPC codes. In this talk, the quantum simulations are in the code-capacity model. This means that data qubits are noisy, but syndrome measurements and decoder input are assumed to be perfect; circuit-level faults are not included. There are two regimes: the waterfall and the error floor. Here FER means the frame error rate, that is, the fraction of frames where decoding fails. The waterfall is the steep drop of FER as the error rate p passes through the BP threshold. In physics language, this is the finite-length trace of a phase transition. In this region, when decoding fails, the remaining error is typically macroscopic: its size is proportional to the block length n. Degree distribution mainly

moves the threshold, while the block length mainly controls how sharp the finite-length transition looks. The simple formula on the slide summarizes this finite-length scaling. Here Z is a standard Gaussian random variable. $Q(x)$ is the probability that this Gaussian variable is larger than x. So the formula says that decoding fails when the finite-length fluctuation crosses the threshold boundary. For a fixed target FER, the input of the Q-function is fixed, so $\sqrt{n}(p_{\mathrm{th}} - p)$ is almost fixed. Therefore, if the block length becomes four times larger, the gap to threshold becomes one half. This is the main point I need here: longer codes make the waterfall sharper.
The error floor is different. It is caused by rare finite-length structures that remain visible at low error rates. Here the dominant residual errors are small patterns whose size does not grow with n. Large girth reduces harmful short-cycle effects, and minimum distance and trapping-set-type structures control the low-error tail. So a good sparse-graph code must be good in both senses. It should have a sharp waterfall from degree distribution and randomness, and a low error floor from large girth, minimum distance, and trapping-set control. Many earlier quantum LDPC codes did not show a clear waterfall; their curves were often dominated by error-floor-like behavior. This is the classical LDPC design language that I will use in the rest of the talk. **[Next slide.]**

Slide 3: Why the Waterfall Has a Q-Shape (about 1:00)

This slide explains why the waterfall of an LDPC ensemble is often written with a Q-function. The distribution in the picture is centered at the actual channel error rate p, not at the threshold. At finite length, the BP trajectory fluctuates around the density-evolution trajectory for that value of p. Near the BP threshold, the dominant fluctuation is asymptotically Gaussian, and its variance is described by covariance-evolution-type arguments. Decoding fails when the right tail

of this finite-length fluctuation crosses the threshold boundary. Therefore the waterfall part of FER is approximated by

$$\mathrm{FER}_{\mathrm{WF}}(p, n) \approx Q\left(\frac{\sqrt{n}(p_{\mathrm{th}} - p)}{\alpha}\right).$$

Here p_{th} is the BP threshold predicted by density evolution, and α is a scaling parameter depending on the LDPC ensemble and the channel. The important consequence is the $1/\sqrt{n}$ transition width. For a fixed target FER, the argument of the Q-function is almost fixed. Therefore, if the block length becomes four times larger, the gap to threshold becomes about one half. **[Next slide.]**

Slide 4: Density-Evolution Trajectory (about 1:00)

Here I use a simplified one-dimensional picture to explain the idea of density evolution. Please think of x_t as the fraction of noise that remains after the t-th BP iteration. Then density evolution is represented by the abstract recursion shown on the slide, $x_{t+1} = f_p(x_t)$, with $x_0 = p$. The graph compares this update curve with the diagonal line $y = x$. The broken vertical and horizontal line shows the density-evolution trajectory without listing the numerical values. Here, saying that the window is open means that there is no nonzero fixed point. Equivalently, the update curve stays below the diagonal for every positive x. Then the recursion moves downward and the remaining noise fraction goes to zero. This is the left figure, where p is below the BP threshold. If p is too large, the curve crosses the diagonal. Then a nonzero fixed point appears, and the recursion is trapped there. This is the right figure, where p is above the BP threshold. The BP threshold is the largest p for which no nonzero fixed point exists. At the threshold, a nonzero fixed point appears by tangency between the update curve and the diagonal. This is the cleanest way to see how density evolution gives a BP threshold. **[Next slide.]**

Slide 5: Classical LDPC Benchmark: BP Threshold (about 2:00)

Now I turn the previous lesson into a concrete design question. Among the LDPC design indicators, this slide focuses on degree distributions. Here, degree distribution means the column-weight and row-weight distribution of the parity-check matrix. Equivalently, it is the degree distribution of variable nodes and check nodes in the Tanner graph. The figure is a density-evolution benchmark for degree optimization. Density evolution is a way to analyze BP decoding for a large random LDPC ensemble. Instead of simulating one finite graph, it tracks how the distribution of BP messages changes from iteration to iteration when the graph is locally tree-like. Let me first explain how to read this plot. The horizontal axis is the physical error rate p, and the vertical axis is the code rate R. Each plotted point represents a degree distribution and its density-evolution BP threshold at that rate. Farther to the right means that BP decoding is expected to tolerate a higher physical error rate at the same rate. The question behind the figure is asked before constructing a finite-length CSS code. If the X- and Z-decoding graphs behaved like random LDPC ensembles with a chosen degree distribution, what BP threshold should we expect? So this plot is a coding-theory benchmark for choosing degree distributions that are worth trying to build under CSS constraints. The colored curves show regular ensembles with fixed column weight J. Among these regular degree distributions, $J = 3$ gives the best BP threshold. The arXiv-labeled degree points are not only benchmarks. For these points, large CSS codes have already been constructed. Their decoding experiments show BP performance close to the matching DE threshold. This includes the $(3, 12)$ case of the present talk. The black curve is the hashing bound. The black diamonds show optimized irregular degree distributions found by the restricted DE search. They should be read as design targets, not as CSS codes that are already built. The message is that degree distribution is a knob that can move the BP threshold toward the hashing bound. But

the BP threshold is only one side of the classical LDPC design story. The next slide shows the matching distance-side benchmark. **[Next slide.]**

Slide 6: Classical LDPC Benchmark: Minimum Distance (about 1:10)

This slide uses the same style of benchmark, but now for minimum distance. The horizontal axis is the normalized minimum distance, namely the typical minimum distance divided by the block length. The vertical axis is again the code rate R. The important point is that this does not yet guarantee the distance of the CSS codes we will construct. As in the BP-threshold slide, we first ignore the CSS orthogonality constraint and ask a classical LDPC question. If we randomly construct a (J, L)-regular LDPC code without CSS commutativity, what normalized minimum distance should a typical large code have? This plot answers that benchmark question for regular ensembles. The lesson is simple. As J increases from 3 to 4 to 5, the normalized minimum distance becomes larger. From the BP-threshold viewpoint, $J = 3$ was the best regular choice. But $J = 4$ is still not bad, and it gives a better distance-side margin. There is also one important tradeoff. When we increase the column weight J, each variable node has more edges. So the parity-check matrix becomes denser, and short cycles become easier to create. In other words, larger J helps the distance side, but it can make it harder to keep the girth large. So the design problem is not one-dimensional. We have to balance BP threshold, minimum distance, sparsity, and the CSS commutativity constraint. **[Next slide.]**

Slide 7: Design Freedom vs. CSS Commutativity (about 1:00)

This slide states the main design difficulty. For a classical LDPC code, we choose one sparse parity-check matrix H. Then we try to tune degree distribution, randomness, large girth, and minimum distance. For a quantum CSS code, we must choose two sparse matrices, H_X and H_Z. They cannot be chosen independently. The CSS condition says that they are orthogonal, as shown on the slide. Here and below, the prime means transpose. Equivalently, all X-checks and Z-checks must commute. So the problem is simple to state but hard to solve: can we keep the classical LDPC design controls under this commutativity constraint? That is the position of this work. **[Next slide.]**

Slide 8: Minimum Distance (about 1:20)

Here I recall the definition of minimum distance. For a classical code, it is the smallest nonzero vector in the kernel of the check matrix H, measured by Hamming weight. This is the standard distance related to large correctable errors and to the error-floor tail. For a CSS code, the important objects are nontrivial logical operators. The Z-distance looks for a vector that passes all X-checks but is not generated by the Z-checks. The X-distance is the same definition with X and Z exchanged. The quantum distance is the smaller of these two. So the practical message is simple: low-weight logical operators are dangerous, and a larger distance pushes the error floor down. **[Next slide.]**

Slide 9: Problem: Latent Rows Become Logicals (about 1:30)

Let me first define what I mean by latent rows. In this construction, we start from larger parent check matrices. We keep some rows as the

active stabilizer checks. The rows that remain in the parent matrix but are not used as active checks are called latent rows. The problem on this slide is that these unused parent rows can become logical operators. The Hagiwara–Imai construction makes it easy to impose a regular degree distribution. It and bicycle codes fall into this row-removal type CSS construction. Other popular quantum LDPC constructions can make the minimum distance large, but it has been hard to design them with a target degree distribution. First, we build fully orthogonal parent matrices $\hat{H}_X$ and $\hat{H}_Z$. Here, fully orthogonal means that every X-side parent row is orthogonal to every Z-side parent row. These parent matrices have many rows. So if we use them as they are, the rate is low, or even zero. To increase the rate, we keep only the top rows as the active matrices H_X and H_Z. The remaining rows are denoted by $\tilde{H}_X$ and $\tilde{H}_Z$. These are the latent rows shown on the slide. The problem is that the whole parent matrices were made orthogonal. So orthogonality also remains between the active rows and the latent rows. In other words, every active row on one side is automatically orthogonal to every latent row on the other side. This means that the latent rows pass all active checks. Unless we design the construction to avoid this, such a latent row is usually not generated by the stabilizers. Then it can easily become a low-weight logical operator in the normalizer. Its weight is usually on the order of the row weight. So the minimum distance is bounded above by the row weight. This is the structural limit that we want to overcome. **[Next slide.]**

Slide 10: Design Principle (about 1:20)

The design principle of this work is to avoid building a fully orthogonal parent matrix at the start. What a CSS code needs is orthogonality of the active matrices that we actually use. So we still require the active X-checks and active Z-checks to commute. On the other hand, the latent rows do not have to be orthogonal to the active rows. As we saw on the previous slide, if we also impose active–latent orthogonality, the

latent rows can become logical operators. Therefore, we impose orthogonality only on the active part, and we intentionally keep the latent part non-orthogonal. More concretely, we design the construction so that the active rows and the latent rows still interact with each other. Then the latent rows do not automatically pass all active checks. This makes it much harder for the latent rows themselves to become small logical operators. Here, however, we especially want to measure logical operators that come from linear combinations of latent rows. For that purpose, we define the latent distance. It is the smallest weight of a nontrivial logical operator that can be made from latent rows. Since these are actual logical operators, the true minimum distance is at most the smaller of the two latent distances. This directly measures the risk that latent rows create logical operators. In standard row-removal methods, this latent distance was bounded above by the row weight. In this work, we try to keep the minimum distance large by making this latent distance large. **[Next slide.]**

Slide 11: Generalized Hagiwara–Imai Construction (Example J=3, L=12) (about 2:10)

Now I move from the design principle to the construction. The construction generalizes the block-circulant Hagiwara–Imai construction. First, J and L denote the column and row weights. We prepare $L/2$ binary permutation matrices F_i and G_j of size P. In the experiments shown later, we use $P = 768$. We build $\hat{H}_X, \hat{H}_Z$ by arranging F_i and G_j so that both the left and right halves are block-circulant. Here, block-circulant means that each block row is made from the previous one by a cyclic shift, both for the F_i blocks and for the G_j blocks. Because of this symmetry, the product depends only on the cyclic difference between block-row indices. We use the top J block rows of $\hat{H}_X, \hat{H}_Z$ as the active matrices, and the remaining block rows as the latent matrices.

Here a prime $(')$ means transpose. For permutation blocks, it is also the inverse. This structure reduces the parent product to a small family of blocks, called Ψ_r. Each Ψ_r is a sum of $L/2$ swapped-order pairs, or commutator terms. So commutativity between F_i and G_j controls when Ψ_r becomes zero. The key idea is to control commutativity between F_i and G_j by looking at $\hat{H}_X(\hat{H}_Z)'$ as four blocks. The upper-left block must be zero, because it is the active–active product and gives the CSS condition. In this construction, because of the block-circulant symmetry, the lower-right latent–latent block has the same form as the upper-left block. Therefore, once the upper-left block is made zero, the lower-right block automatically becomes zero as well. The upper-right and lower-left blocks should not be zero, because they are the active–latent products. These two blocks are also the same block in this construction. If they were also zero, the latent rows could again create low-weight logical operators. On the next slide, I show how to make the upper-left block zero while keeping the active–latent block nonzero. **[Next slide.]**

Slide 12: Setting Δ and Γ (about 1:30)

Let us look more closely at the parent product on the left. Because of the block-circulant form, every block of this product is one of the Ψ_r's. The index r is determined only by a cyclic difference of block-row indices. For $J = 3, L = 12$, the active part is the upper-left 3×3 block. In this upper-left active block, the blocks $\Psi_0, \Psi_1, \Psi_2, \Psi_4, \Psi_5$ appear, but Ψ_3 does not appear. This is the important counting fact on this slide. We call the set of appearing indices Δ. In words, Δ is the set of cyclic differences that appear inside the active block.
Now recall that each Ψ_r is a sum of commutator terms of the form $F_iG_j + G_jF_i$ over binary matrices. Therefore, if the relevant pairs (F_i, G_j) commute, then the commutator term becomes zero over $\mathbb{F}_2$. On the right, I color the (i, j) pairs by the same Ψ_r colors. The set Γ is the set of pairs that contribute to the Ψ_r's with $r \in \Delta$. These are exactly the pairs where we impose commutativity. Then every Ψ_r

appearing in the active block becomes zero. So the active product, namely the upper-left block of $\hat{H}_X(\hat{H}_Z)^{'}$, is zero.
Pairs outside Γ are deliberately left free. This freedom lets us keep the upper-right and lower-left active–latent blocks nonzero. In the example, the free color is Ψ_3. It does not appear in the active block, but it does appear in active–latent positions. So we can keep $\Psi_3 \neq 0$ without violating the CSS condition. **[Next slide.]**

Slide 13: CPM, APM, and GPM (about 1:30)

Before introducing affine permutation matrices, I want to place them between two familiar extremes. The three pictures on the slide are actual 8×8 permutation matrices. Each blue cell shows the position of a one in the permutation matrix, and every row and every column has exactly one blue cell. A CPM, or circulant permutation matrix, is the most structured case. It is just a cyclic shift $x \mapsto x + b$ modulo P. This gives very clean algebra, and products of CPMs are again CPMs, but the randomness is very limited. More importantly for this construction, all CPM pairs commute. So CPMs cannot create the non-commuting active–latent pairs that we need here.
At the other extreme, GPM allows any permutation $x \mapsto \pi(x)$. This adds much more randomness to the graph, and from the LDPC viewpoint that sounds attractive. However, the algebra becomes hard to use. Commutativity is no longer a small congruence condition; it is a relation between large permutation matrices. Short-cycle conditions also become graph checks rather than simple arithmetic tests. So GPMs give randomness, but they do not give an easy algebraic handle for the CSS constraints.
APMs are the useful middle ground. They use affine maps $x \mapsto ax + b$, so they are more flexible than CPMs, but still closed under composition. Unlike CPMs, APMs can include non-commuting pairs, while still keeping

enough algebra to control the CSS constraints. This is why we can add randomness while keeping commutativity and cycle tests as modular arithmetic. **[Next slide.]**

Slide 14: Affine Permutation Matrices (about 1:20)

Here is the search idea behind the construction. At this point, the algebraic target is clear. For pairs (F_i, G_j) in Γ, we must impose commutativity, because those pairs contribute to the Ψ_r's in the active–active block. For pairs outside Γ, we want to keep some non-commutativity, because this keeps the active–latent products nonzero. At the same time, the Tanner graphs defined by the active matrices should avoid short cycles. So the construction has to satisfy three constraints at once: commutativity where it is required, non-commutativity where it is useful, and large girth for the active sparse graphs.

If F_i and G_j were arbitrary binary permutation matrices of size P, this search would be too large. There are $P!$ possible permutation matrices, and checking all commutation and cycle conditions by direct graph search would be too expensive. The key simplification is to restrict the permutation blocks to affine permutation matrices over $\mathbb{Z}_P$. That means each permutation is represented by an affine map, for example $f_i(x) = a_i x + b_i$ and $g_j(x) = c_j x + d_j$. This is still a binary permutation matrix, but it is described by only a few parameters.

The important point is that affine maps are closed under composition. Therefore, commutativity of F_i and G_j becomes a simple congruence condition on a_i, b_i, c_j, d_j, not a large matrix equation. The same representation also makes short-cycle tests cheap: products around a candidate cycle are again affine maps, so the existence of a fixed point can be checked by arithmetic modulo P. In this way, the conditions become arithmetic tests that are almost independent of P.

This is what makes the construction practical. We can build the blocks one by one, check commutativity on Γ, keep the desired non-commutativity outside Γ, and reject choices that create short cycles. The construction software implementing this search has been released on GitHub. **[Next slide.]**

Slide 15: Example: Proposed Code Construction (about 1:00)

Let me now give the concrete example used in the experiments. With $P = 768$, the construction gives a girth-8, $(3, 12)$-regular code with parameters $[[9216, 4612, \leq 48]]$. Earlier row-removal-type constructions were limited by row-weight logicals, but this example avoids that immediate problem. We can explicitly build weight-48 logical operators, so the slide writes the distance as at most 48. For latent logicals, we can prove that the two latent distances are both 48; I omit the proof here. To prove the exact minimum distance, we would still need to rule out all other smaller logical operators. That part remains open, but in deep error-floor experiments we saw no logical failures, so we expect the true distance to be close to 48. **[Next slide.]**

Slide 16: Decoding Algorithm (BP + Post-Processing) (about 1:20)

Let me describe the decoder. First, we run joint BP on H_X and H_Z, using X/Z correlations. Most frames are corrected by BP alone, but sometimes BP stops with a small residual syndrome. When the number of unsatisfied checks is small, we use post-processing. The figure on the right is the ETS library figure from the paper. Here ETS stands for Elementary Trapping Set. It means a small set of variable nodes whose induced Tanner subgraph leaves a small number of odd checks. Here b

means the number of those odd checks. Equivalently, it is the number of unsatisfied checks left by the trapping set. The decoder precomputes the harmful $b = 2$ ETS patterns in the Tanner graph. It stores each entry by its variable set and its pair of odd checks. When BP stops with a residual syndrome, we compare the remaining odd-check pair with this library. If it matches one of the stored patterns, the variables in that pattern are added to the suspect set K. OSD and flip history are used in the same way, to enlarge this suspect set only where the residual suggests it. We then solve the restricted residual problem on K, and apply the update only when it gives a small-weight correction. Thus the ETS library is not a second full decoder. It is a targeted way to remove the main finite-length structures that make BP stop. We have released the decoder software on GitHub. **[Next slide.]**

Slide 17: Performance Highlight (about 1:20)

Let me highlight the main result. We constructed a girth-8, $(3, 12)$-regular quantum LDPC code with parameters $[[9216, 4612, \leq 48]]$. The channel here is the code-capacity depolarizing channel, and the horizontal axis is the physical error rate p.
The main curve is the FER of joint BP followed by the targeted post-processing from the previous slide. At $p = 4\%$, the FER reaches 10^{-8}. This is the region where I want to emphasize three points. First, the waterfall is very steep: once p is below the transition region, the frame error rate drops quickly. Second, we observed many successful corrections of weight-12 degenerate errors. This is important because, in a quantum code, a high-weight physical error can be equivalent to a much smaller error after multiplying by stabilizers. So the decoder is not simply acting like a classical bounded-distance decoder for the raw Hamming weight. It is already using the degeneracy of the CSS code, at least in these simulations. Third, we can already look at the beginning of

the error-floor region. At $p = 0.04$, we see early signs of an error floor, but the remaining failures are not large chaotic failures. They usually leave only a small number of unsatisfied checks. This suggests that the ETS library or the final post-processing step can still be improved.
The most important comparison is with the DE benchmark. Here it is the classical-style reference for a random non-orthogonal $(3, 12)$-regular sparse-graph code, under the cycle-free assumption. It tells us what BP would do without CSS orthogonality and short-cycle effects. The observed BP curve is close to that benchmark. So the message of this slide is that the CSS constraints have not destroyed the classical sparse-graph advantage: the code behaves much like the ideal random sparse-graph ensemble, while still satisfying quantum commutativity.
[Next slide.]

Slide 18: Conclusion (about 1:00)

Let me summarize the main message. The point is not only that we found one good code. The main point is that we built a quantum LDPC setting where the lessons of classical LDPC design naturally remain useful. In particular, degree distribution determines the BP threshold, so it has to be treated as a real design target. Randomness, large girth, and distance are also meaningful design targets. APMs give algebraic room to optimize these targets while keeping CSS commutativity. This lets the code behave close to the classical sparse-graph benchmark under BP decoding. This concludes the main talk. Thank you for your attention.