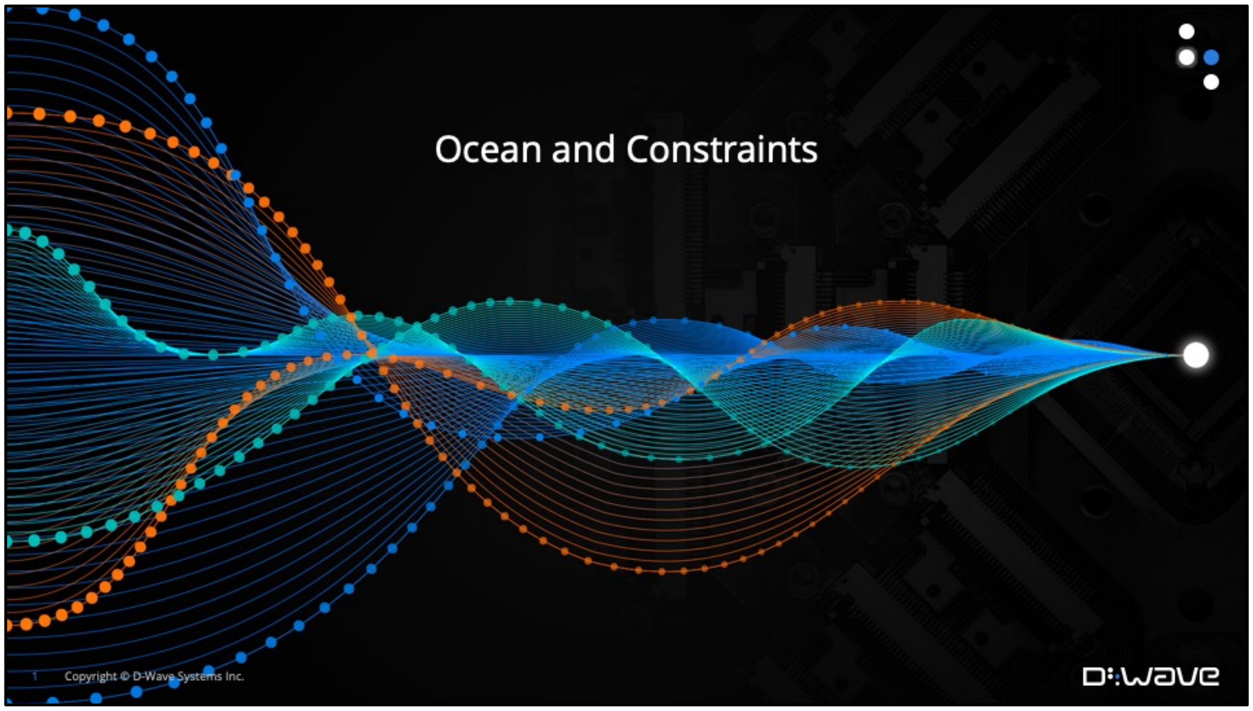
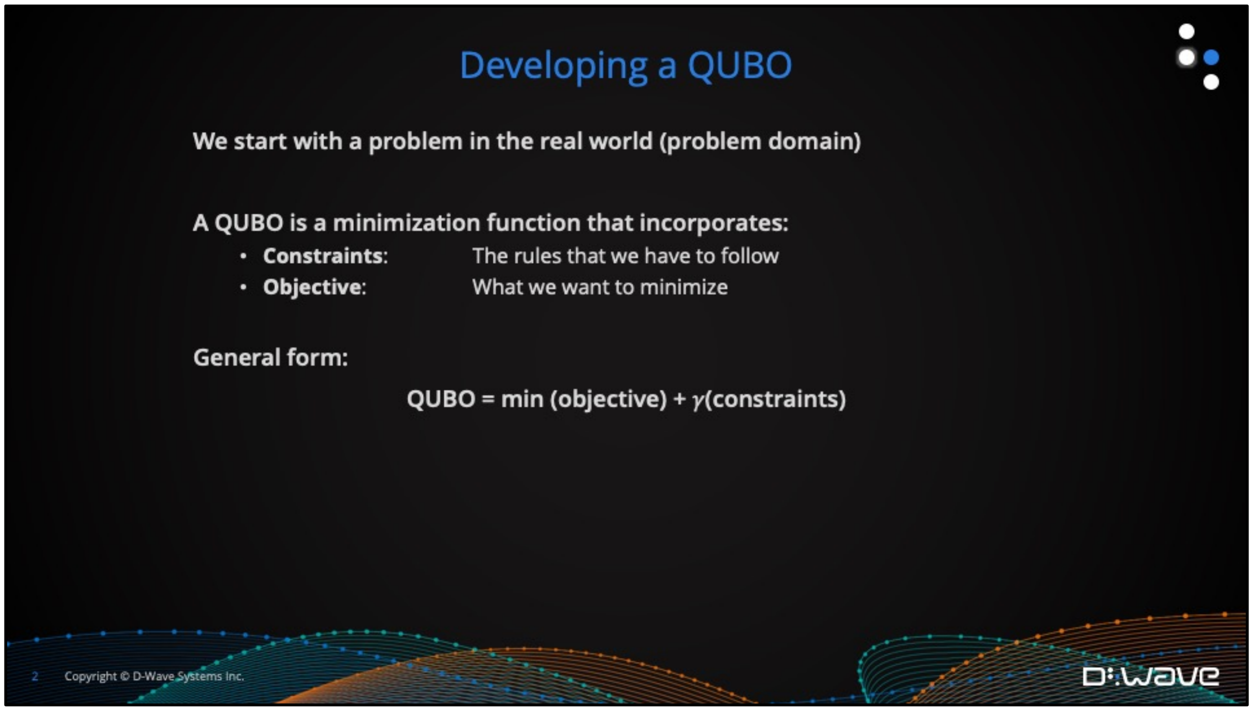


Ocean and Constraints

In this module, we'll build an Ocean program for a problem that has both an objective and a constraint. This will require us to look at the Lagrange parameter.



Developing a QUBO

We start with a problem in the real world (problem domain)

A QUBO is a minimization function that incorporates:

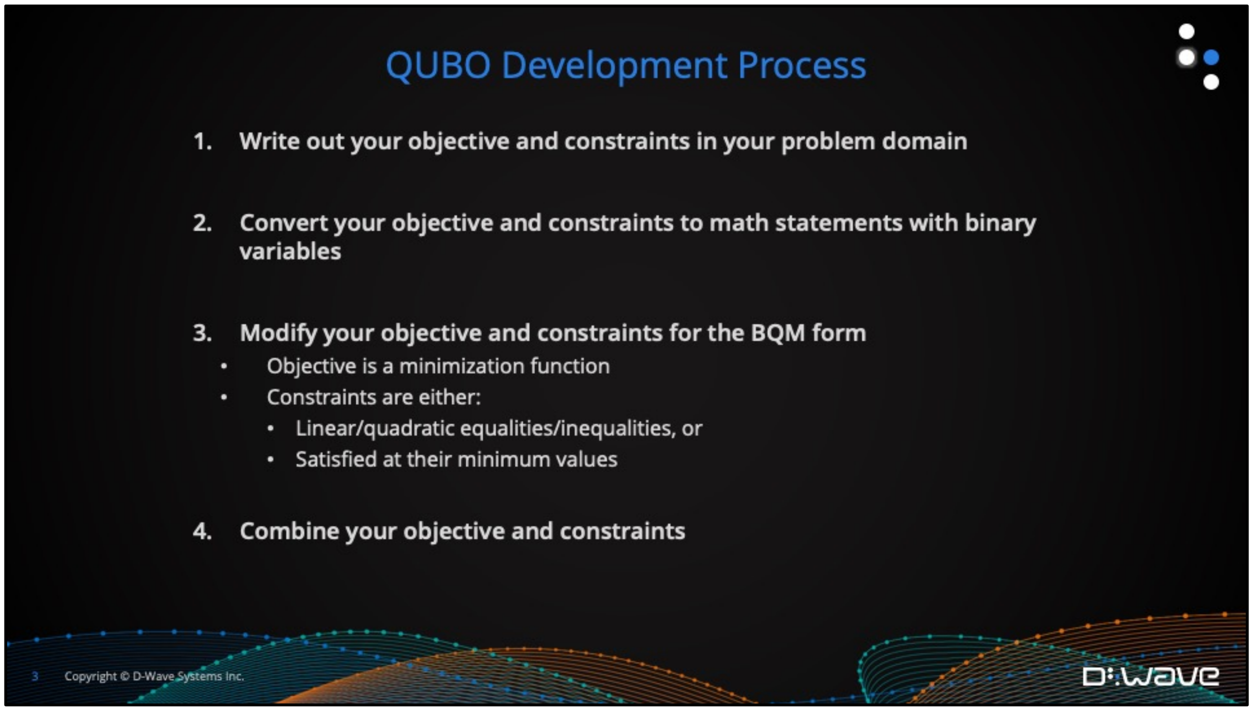
• **Constraints**:	The rules that we have to follow
• **Objective**:	What we want to minimize

General form:

QUBO = min (objective) + γ(constraints)

As a reminder, let’s review the steps to develop a QUBO.

A QUBO starts with a problem in our problem domain. It consists of some combination of objectives and constraints. We can combine these using addition and Lagrange parameters.

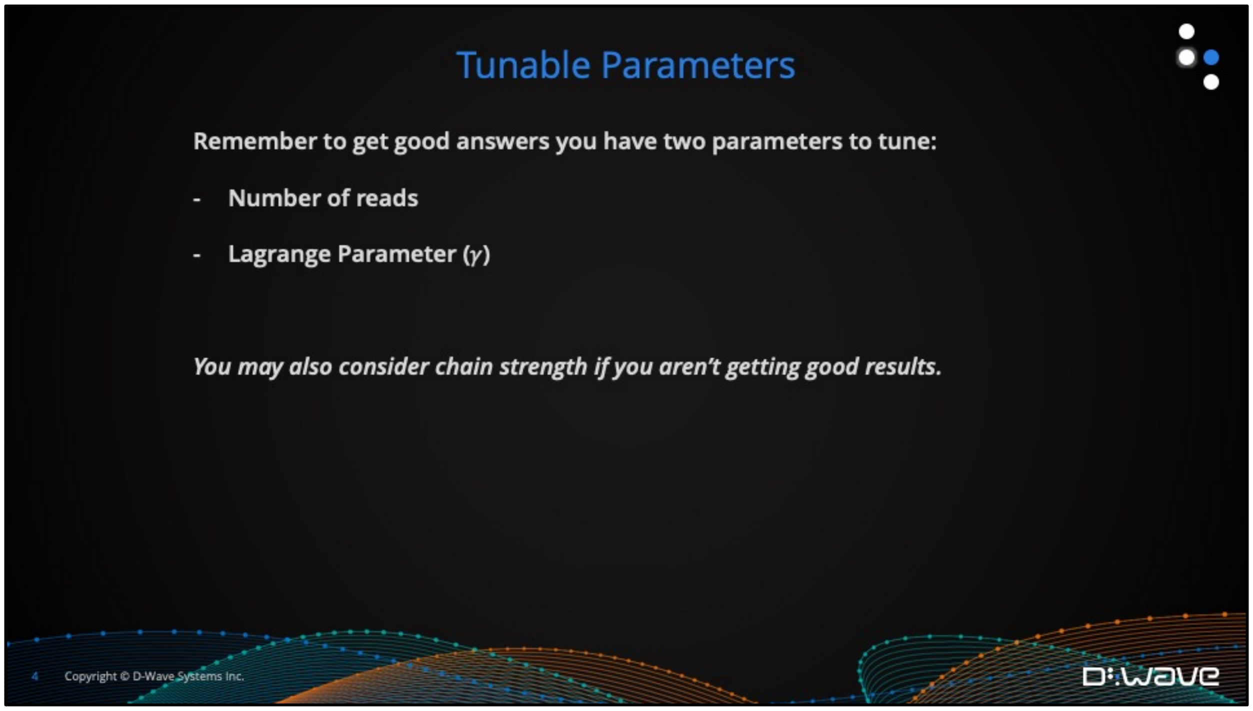


QUBO Development Process

1. **Write out your objective and constraints in your problem domain**
2. **Convert your objective and constraints to math statements with binary variables**
3. **Modify your objective and constraints for the BQM form**
   - Objective is a minimization function
   - Constraints are either:
     - Linear/quadratic equalities/inequalities, or
     - Satisfied at their minimum values
4. **Combine your objective and constraints**

3

To formulate our QUBO, we first need to clearly define what are our objectives and constraints. Then we can figure out how to define our binary variables. Once we've written out our objectives and constraints as math expressions using these binary variables, we may need to make some adjustments so that they can be combined with a Lagrange parameter to build the QUBO.



Tunable Parameters

Remember to get good answers you have two parameters to tune:

- Number of reads
- Lagrange Parameter (γ)

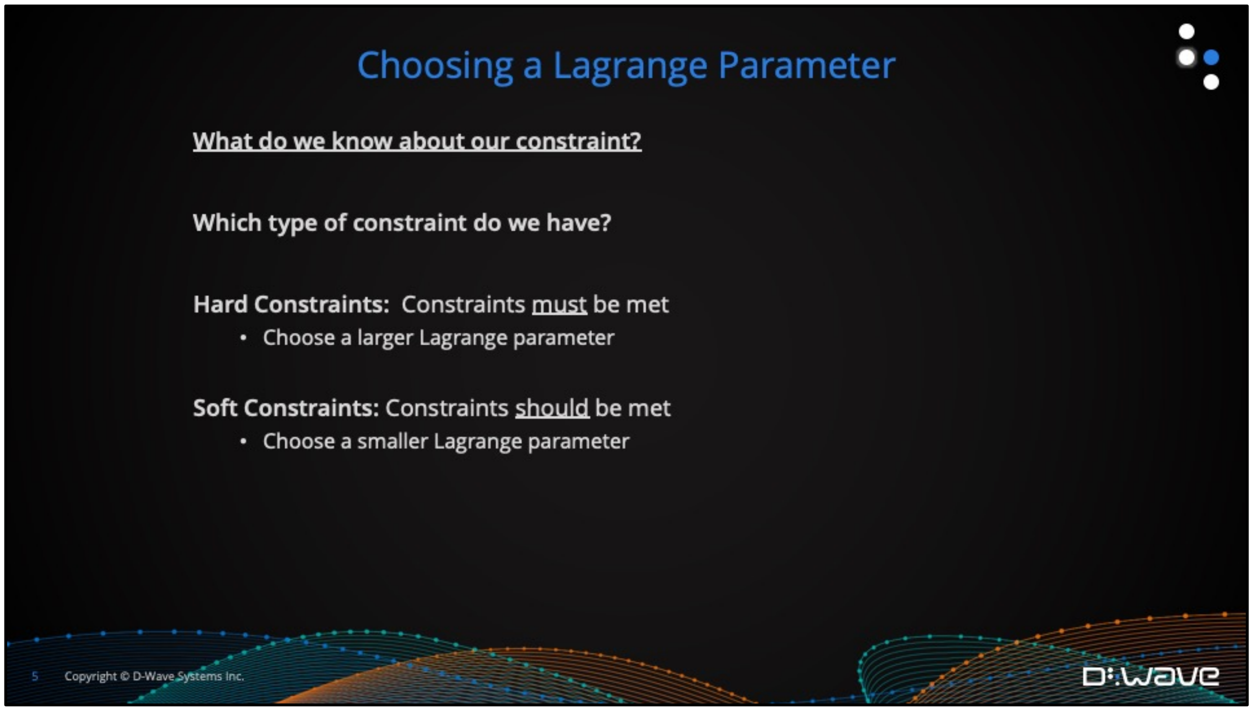
You may also consider chain strength if you aren't getting good results.

There are a few different parameters that we should try tuning to improve results when we build our QUBO and run it on the QPU.

First is the number of reads. Remember that the QPU is probabilistic, so we should set this number to 10, 100, or even 1000 depending on how complex our problem is.

Next is the Lagrange parameter. This parameter helps us to determine how strongly to weight each constraint against the objective and against each other. If you are seeing solutions returned where a constraint is not satisfied, consider increasing the Lagrange parameter that corresponds to that constraint.

Lastly, we can choose to manually tune the chain strength value for our problem if we are not seeing good results with Ocean's auto tuning tool.



Choosing a Lagrange Parameter

What do we know about our constraint?

Which type of constraint do we have?

Hard Constraints: Constraints must be met

- Choose a larger Lagrange parameter

Soft Constraints: Constraints should be met

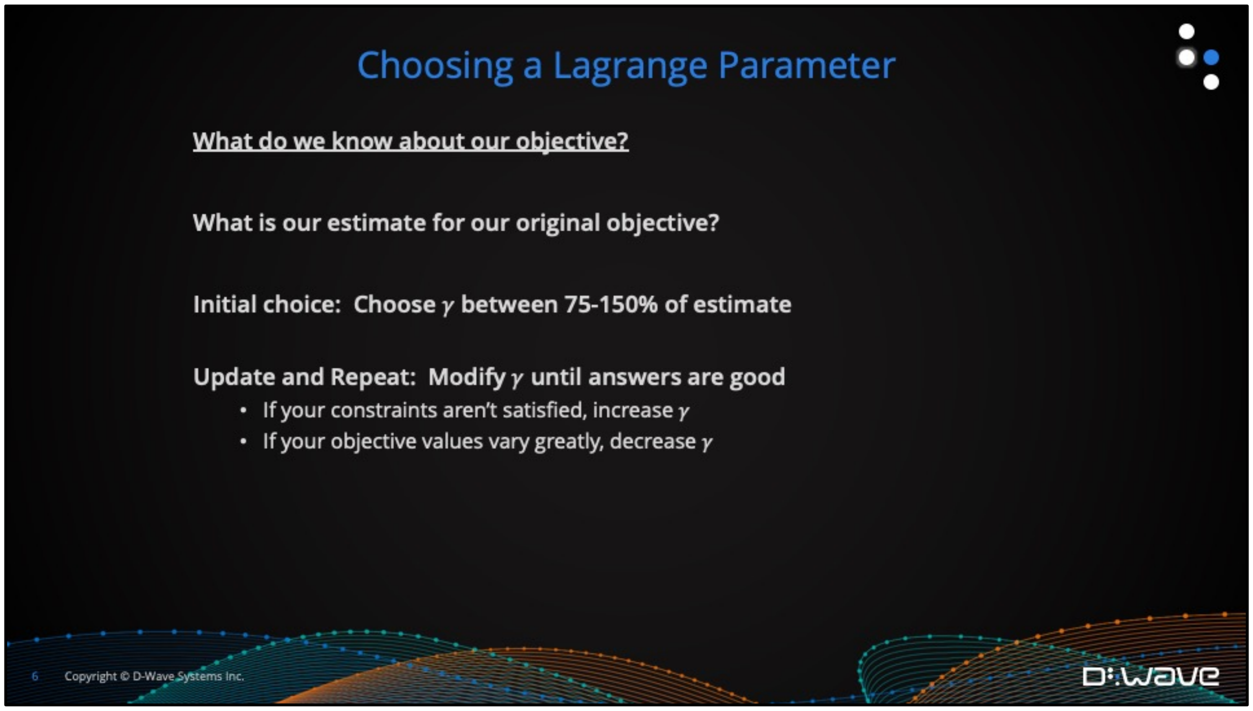
- Choose a smaller Lagrange parameter

Let's look at how we can choose a good value for a Lagrange parameter. We want to choose a value that is large enough that our constraint is satisfied, but small enough that we aren't overpowering the other constraints and objectives in the problem.

To do this, we need to consider how important it is that our constraint is satisfied in the original problem.

If a constraint is a hard constraint, it means that it absolutely must be satisfied. This is like a salesman being in only one city at one time in the traveling salesman problem – a person can't be in two places at the same time! For a hard constraint we'll need to choose a larger value for the Lagrange parameter.

If a constraint is a soft constraint, it means that we might have some flexibility on whether the constraint is absolutely satisfied or almost satisfied. This is like placing antennas with little to no interference – a little interference is ok, but ideally we would like none. For a soft constraint, we can choose a smaller value for the Lagrange parameter.



Choosing a Lagrange Parameter

What do we know about our objective?

What is our estimate for our original objective?

Initial choice: Choose γ between 75-150% of estimate

Update and Repeat: Modify γ until answers are good

- If your constraints aren't satisfied, increase γ
- If your objective values vary greatly, decrease γ

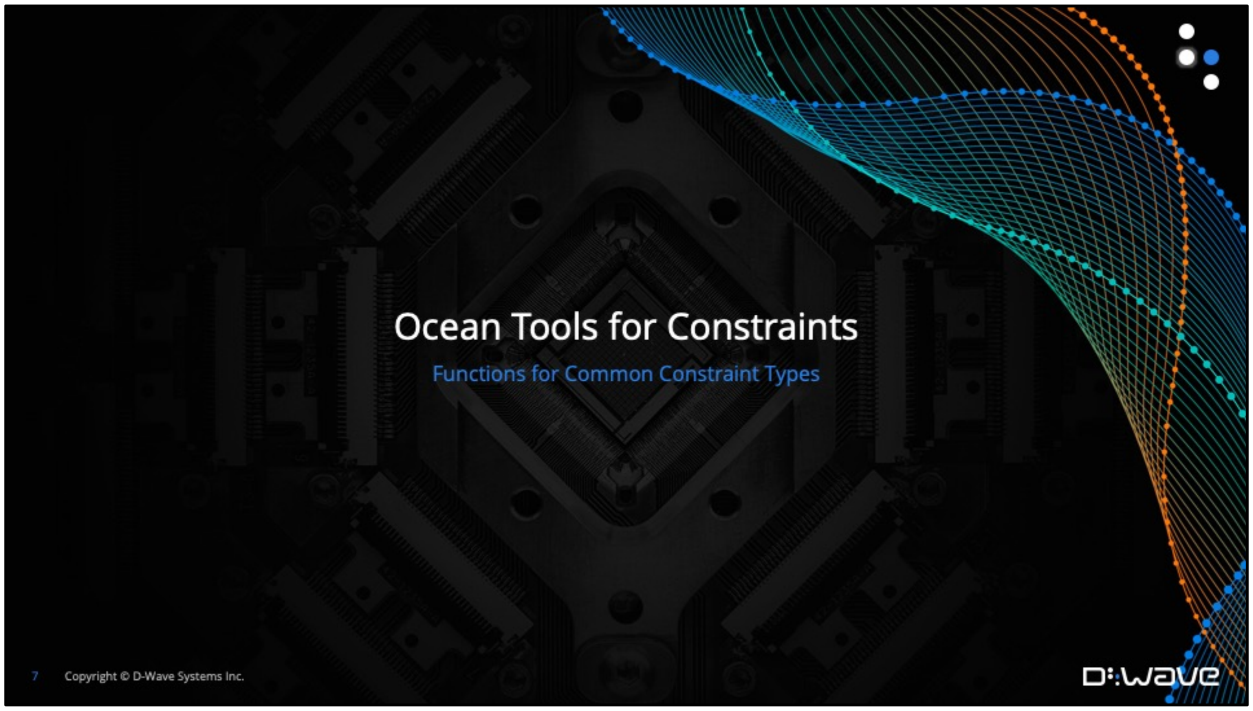
6 Copyright © D-Wave Systems Inc.

D:WAVE

So how can we choose a good starting value for this tunable parameter?

A good rule of thumb is to think about what value the objective portion of your problem will have in an optimal solution. Start with that as a value, try it out, and then increase or decrease the value depending on the solutions that are returned. Essentially, we want to have the values being produced from the constraint portion of the QUBO to be on the same order as the values produced from the objective portion. If my objective is going to produce values like 10 or 20, but my constraint has a penalty of 1 when it's not satisfied, I would choose a Lagrange parameter of 10 or 20 to make those penalties around the same number as the original objective.

Eventually you should settle with the smallest value you've found that consistently returns solutions with the constraint satisfied.

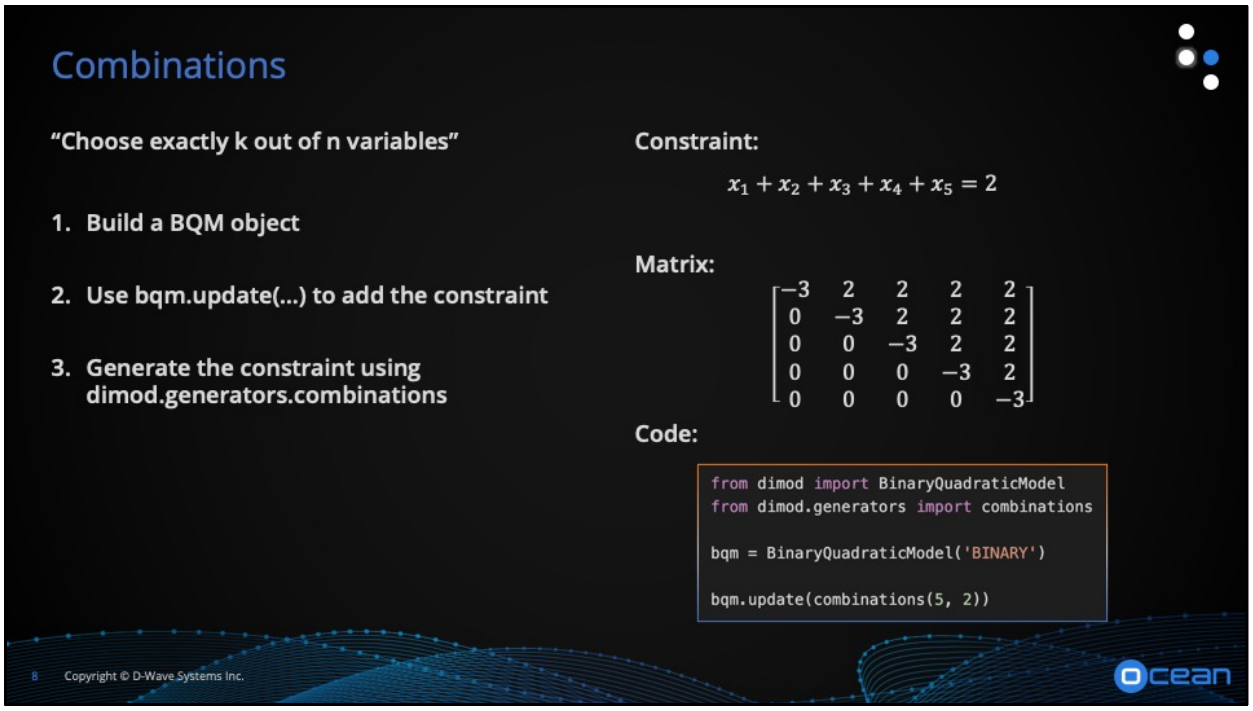


Ocean Tools for Constraints

Functions for Common Constraint Types

7 Copyright © D-Wave Systems Inc.

Ocean has several special functions that can make it easier to build some common constraints into your QUBO with Python. To get started, you'll need to make sure that you have at least version 0.10.0 of dimod installed in your virtual environment. You can check this by typing "pip freeze" on the command-line.



Combinations

"Choose exactly k out of n variables"

1. Build a BQM object
2. Use bqm.update(...) to add the constraint
3. Generate the constraint using dimod.generators.combinations

Constraint:

$$x_1 + x_2 + x_3 + x_4 + x_5 = 2$$

Matrix:

$$\begin{bmatrix} -3 & 2 & 2 & 2 & 2 \\ 0 & -3 & 2 & 2 & 2 \\ 0 & 0 & -3 & 2 & 2 \\ 0 & 0 & 0 & -3 & 2 \\ 0 & 0 & 0 & 0 & -3 \end{bmatrix}$$

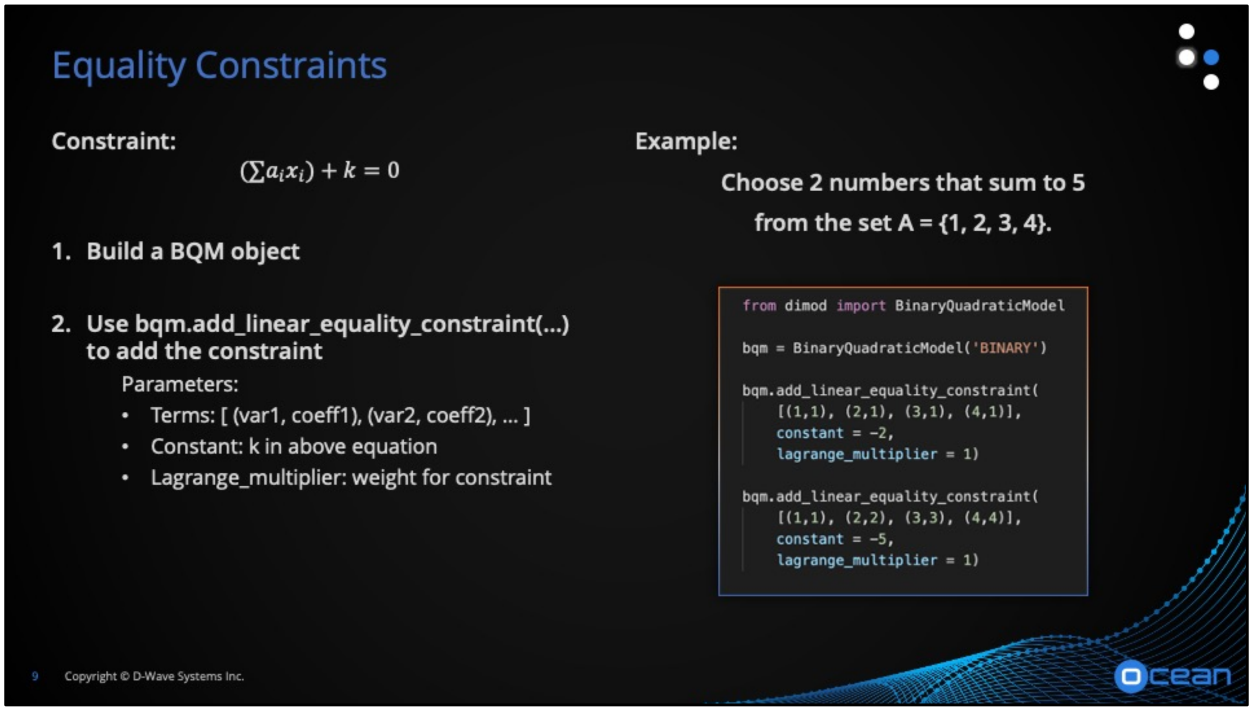
Code:

```
from dimod import BinaryQuadraticModel
from dimod.generators import combinations

bqm = BinaryQuadraticModel('BINARY')

bqm.update(combinations(5, 2))
```

8 Copyright © D-Wave Systems Inc.

One type of constraint that we have seen a lot so far is combinations. The combinations function in dimod allows you to add a "choose exactly k out of n" constraint to a BQM object. Here you can see the syntax for using this function in an Ocean program. Our constraint is to choose exactly 2 out of 5 variables, or the sum of the binary variables needs to equal 2. We can convert this to a matrix form as shown here, and then add in the linear and quadratic coefficients to our BQM. Alternatively, we can use the combinations function in the dimod package to do this for us.

Docs:
- Combinations: https://docs.ocean.dwavesys.com/en/latest/docs_dimod/reference/generated/dimod.generators.combinations.html#dimod.generators.combinations



Equality Constraints

Constraint:

$$\left(\sum a_i x_i\right) + k = 0$$

1. **Build a BQM object**
2. **Use bqm.add_linear_equality_constraint(...) to add the constraint**

   Parameters:
   - Terms: [(var1, coeff1), (var2, coeff2), ...]
   - Constant: k in above equation
   - Lagrange_multiplier: weight for constraint

Example:

Choose 2 numbers that sum to 5 from the set A = {1, 2, 3, 4}.

```
from dimod import BinaryQuadraticModel

bqm = BinaryQuadraticModel('BINARY')

bqm.add_linear_equality_constraint(
    [(1,1), (2,1), (3,1), (4,1)],
    constant = -2,
    lagrange_multiplier = 1)

bqm.add_linear_equality_constraint(
    [(1,1), (2,2), (3,3), (4,4)],
    constant = -5,
    lagrange_multiplier = 1)
```

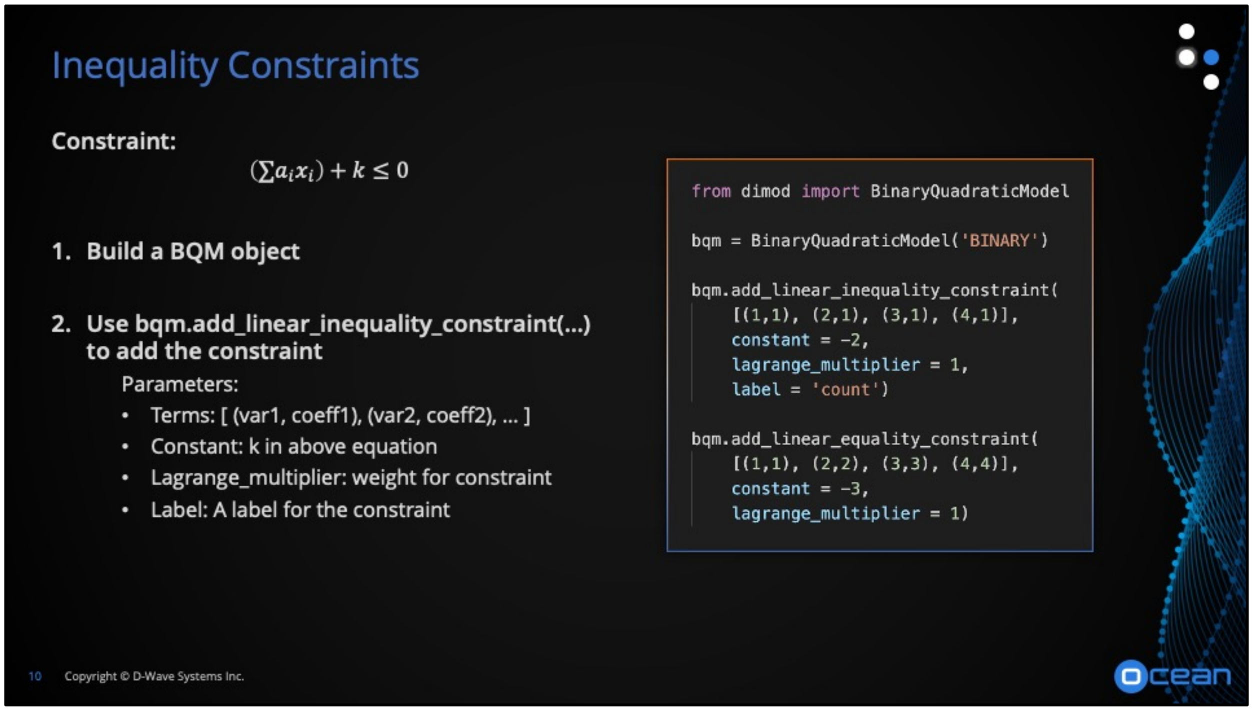
9 Copyright © D-Wave Systems Inc.

In many of our earlier modules we developed QUBOs that had equality constraints. Multiplying out the squared polynomials to get a matrix is a lot of work! Instead, we can use Ocean to add these equality constraints directly to a binary quadratic model using the "add_linear_equality_constraint". This function takes 3 parameters. The first parameter is a list of tuples that represent the variables involved in the constraint and the coefficient on each variable. The second parameter is the constant. Notice that we've moved it to the other side of the equation so our constraint is equal to 0! The last parameter is a Lagrange parameter.

For example, let's say we want to build a QUBO model that tells us the best way to choose 2 numbers that sum to 5 from the set of numbers 1, 2, 3, and 4. This problem has two constraints.

The first constraint is "choose exactly two numbers". We can add up our binary variables and set it equal to 2. On lines 5 through 8 in the code snippet we can see how we can use this function to build this equality constraint into our BQM. Remember that our constraint should equal 0, so our constant becomes -2 instead of +2. We can also add in a Lagrange parameter here. Note that we could also have used the combinations function for this constraint.

The second constraint is “the numbers sum to 5”. We can add up a weighted sum of our binary variables: 1x1+2x2+3x3+4x4 and set that equal to 5. Since this is a weighted sum, we can’t use the combinations function to help us here! Instead, the coefficients in our constraint are included in our list of terms.


Inequality Constraints

Constraint:

$$(\sum a_i x_i) + k \le 0$$

1. **Build a BQM object**
2. **Use bqm.add_linear_inequality_constraint(...) to add the constraint**

   Parameters:
   - Terms: [(var1, coeff1), (var2, coeff2), ...]
   - Constant: k in above equation
   - Lagrange_multiplier: weight for constraint
   - Label: A label for the constraint

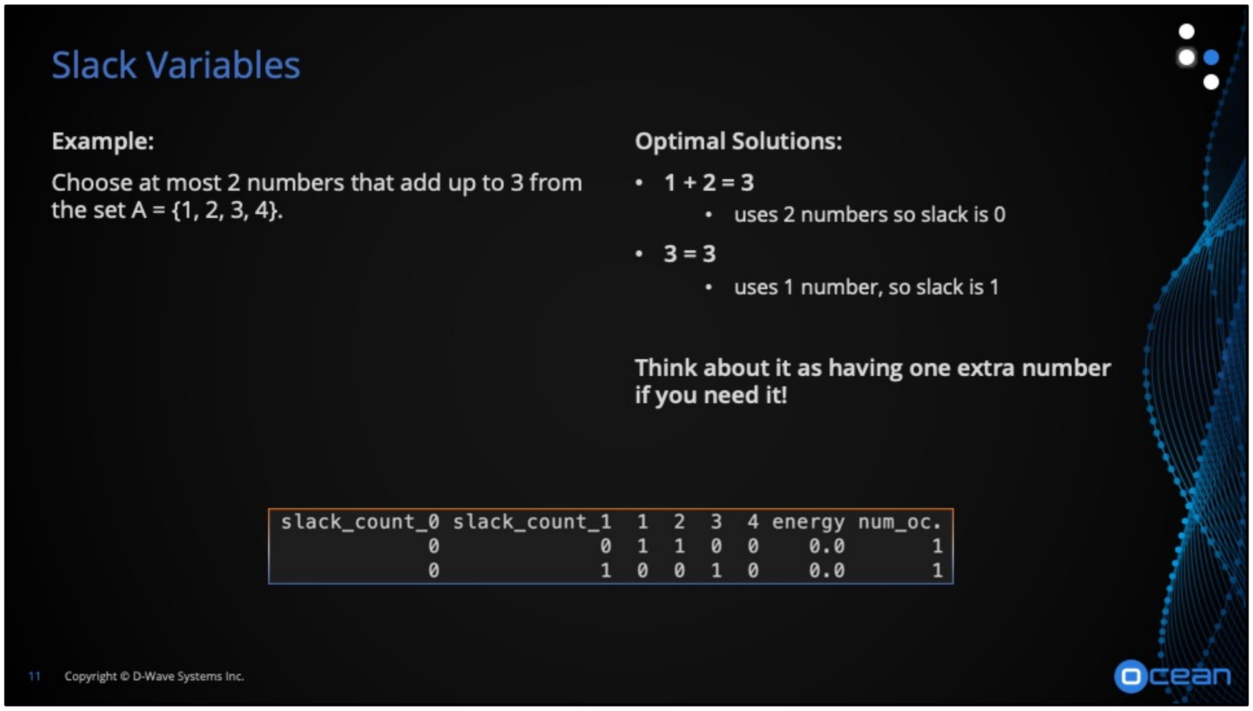
```
from dimod import BinaryQuadraticModel

bqm = BinaryQuadraticModel('BINARY')

bqm.add_linear_inequality_constraint(
    [(1,1), (2,1), (3,1), (4,1)],
    constant = -2,
    lagrange_multiplier = 1,
    label = 'count')

bqm.add_linear_equality_constraint(
    [(1,1), (2,2), (3,3), (4,4)],
    constant = -3,
    lagrange_multiplier = 1)
```

10 Copyright © D-Wave Systems Inc.

Similar to add_linear_equality_constraint there is a function available called add_linear_inequality_constraint. On the right we see how we can use both the inequality and equality functions to build a QUBO for the problem “choose at most two numbers from the set {1, 2, 3, 4} that sum to 3”.

The usage is almost exactly the same, except we must provide a label as an input when using this function. The label allows us to look at additional variables that need to be introduced, called slack variables.

When we add an inequality to a BQM, slack variables are additional binary variables that are used by the software to convert the inequality to an equality.



Slack Variables

Example:

Choose at most 2 numbers that add up to 3 from the set A = {1, 2, 3, 4}.

Optimal Solutions:

- **1 + 2 = 3**
  - uses 2 numbers so slack is 0
- **3 = 3**
  - uses 1 number, so slack is 1

Think about it as having one extra number if you need it!

```
slack_count_0 slack_count_1  1  2  3  4 energy num_oc.
            0             0  1  1  0  0    0.0       1
            0             1  0  0  1  0    0.0       1
```

11 Copyright © D-Wave Systems Inc.

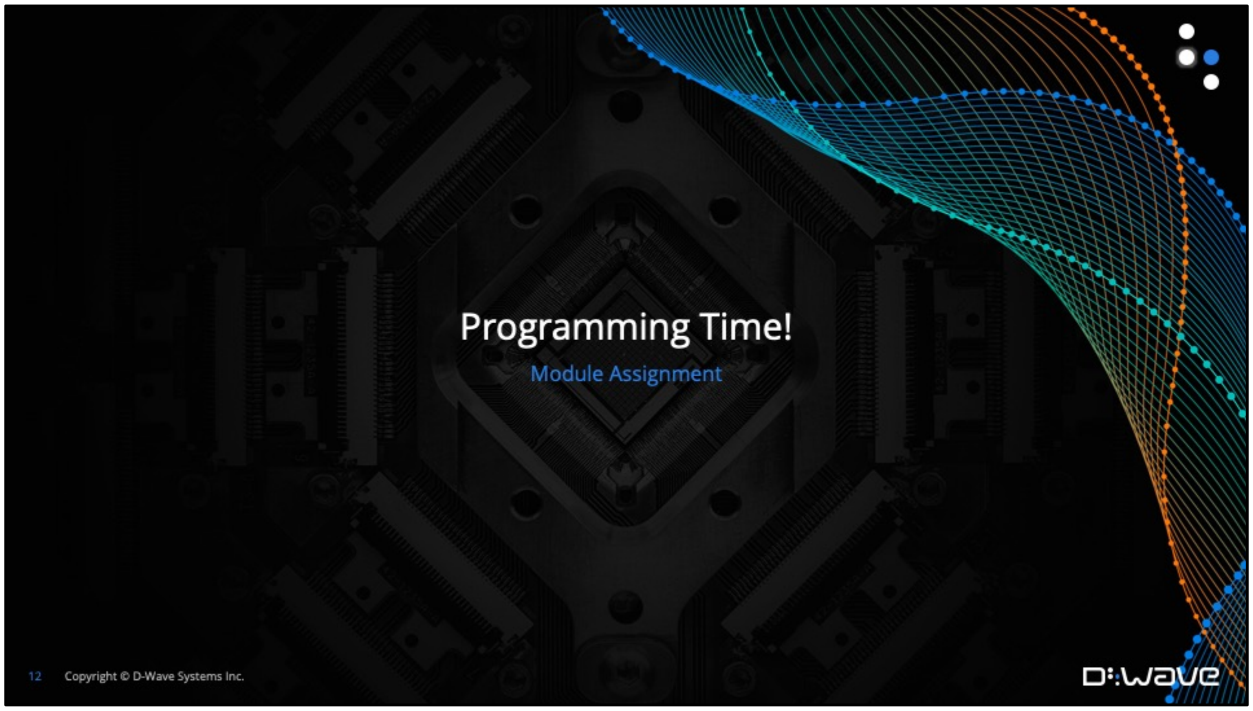
Here's an example of how slack variables work. Suppose that we need to choose at most 2 numbers that add up to 3 from our set A. There are two solutions: 1+2 and 3 by itself.

In the solution where we pick 1+2, we are using exactly two numbers. In this case our inequality to choose at most 2 is actually an equality – we chose exactly 2 numbers. This means that our slack variables should all have the value 0. We don't need to increase how many numbers were chosen to meet the value 2 that is our upper bound.

On the other hand, in the solution where we pick only the number 3, we do need the slack variables to play a part. Our upper bound of how many numbers we can choose is 2, and we chose only 1. We should expect to see one of our slack variables to have a value of 1 to help us meet the upper bound for our inequality.

In the sample set shown, we can see the slack variables for our constraint that we labeled on the last slide. There are two slack variables because our upper bound was 2. In the worst case we might not choose any numbers, so in order to get up to the upper bound of 2 we need two slack variables. For our solution of 1+2, both slack variables are 0, whereas for the solution 3 we see one of them has the value one.

The most efficient way to define and use slack variables is define them as exponents so that we can add a logarithmic number of slack variables rather than the number of slack variables growing at a linear rate with the upper bound.

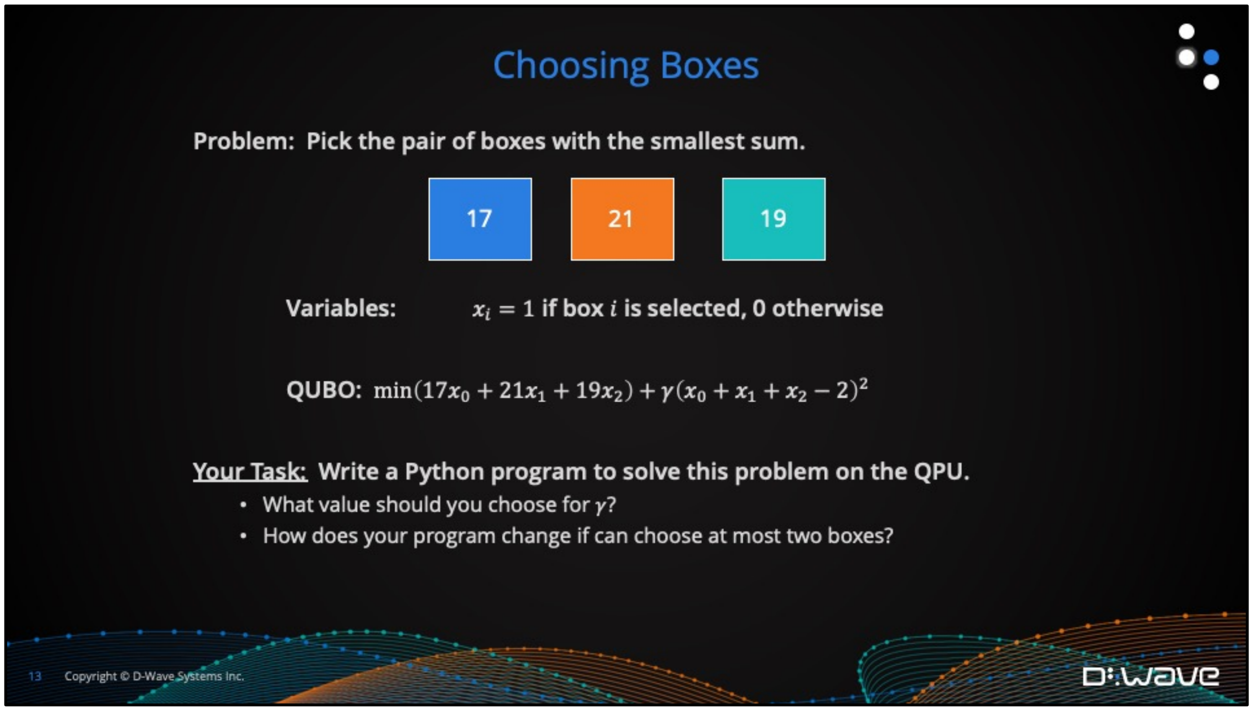


Programming Time!

Module Assignment

12 Copyright © D-Wave Systems Inc.

Now it’s time for you to practice tuning the Lagrange parameter.



Choosing Boxes

Problem: Pick the pair of boxes with the smallest sum.

17	21	19

Variables: $x_i = 1$ **if box** i **is selected, 0 otherwise**

QUBO: $\min(17x_0 + 21x_1 + 19x_2) + \gamma(x_0 + x_1 + x_2 - 2)^2$

Your Task: Write a Python program to solve this problem on the QPU.

- What value should you choose for γ?
- How does your program change if can choose at most two boxes?

13 Copyright © D-Wave Systems Inc.

We saw how to formulate the QUBO for this problem in Module 3. Now we'll work on an Ocean program and solve it using the QPU.

For this problem, you can leave out the chain strength parameter and let Ocean's chain strength tuning tool determine that parameter for you. You should focus on number of reads and the Lagrange parameter. What value might be good to start with for the Lagrange parameter? Will any of the special Ocean functions help you to write your program?