ToQ.jl: A high-level programming language for D-Wave machines based on Julia

Daniel O'Malley
Computational Earth Science
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: omalled@lanl.gov

Velimir V. Vesselinov
Computational Earth Science
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: vvv@lanl.gov

***Abstract*—Quantum computers are becoming more widely available, so it is important to develop tools that enable people to easily program these computers to solve complex problems. To address this issue, we present the design and two applications of ToQ.jl, a high-level programming language for D-Wave quantum annealing machines. ToQ.jl leverages the metaprogramming facilities in Julia (a high-level, high-performance programming language for technical computing) and uses D-Wave's ToQ programming language as an intermediate representation. This makes it possible for a programmer to leverage all the capabilities of Julia, and the D-Wave machine is used as a co-processor. We demonstrate ToQ.jl via two applications: (1) a pedagogical example based on a map-coloring problem and (2) a linear least squares problem. We also discuss our experience using ToQ.jl with a D-Wave 2X, particularly with respect to a linear least squares problem which is of broad interest to the scientific computing community.**

I. Introduction

Julia [1], [2] is a fast dynamic language for technical computing. ToQ (pronounced "to cue") is a programming language developed by D-Wave [3] that transforms human-readable code into a "quantum machine instruction" that can be executed by a D-Wave quantum annealing machine (QAM). ToQ can be seen as a high-level programming language among tools available for programming QAMs, but it lacks many high-level constructs (e.g., loops, arrays, functions, etc.). One path forward for ToQ would be to incrementally add more high-level functionality, but this would be time consuming and result in yet another programming language. As an alternative to this path, we have developed an extension of the Julia language, ToQ.jl, that uses ToQ as an intermediate representation. With ToQ.jl, we immediately have a high-level programming language with a large set of available libraries that treats the QAM as a co-processor. We call ToQ.jl a programming language, but, more specifically, it is a Julia module that includes macros [4] which effectively extend the syntax of Julia to replicate the key features of ToQ.

In addition to ToQ, D-Wave has also developed application program interfaces (APIs) that are available for the C and Python programming languages. Like ToQ.jl, these APIs provide a mechanism to utilize a D-Wave QAM from a high-level programming language. However the APIs for these languages require the programmer to interact with the QAM at a lower level than ToQ. Using C or Python with these APIs to control the QAM gives the programmer a great deal of flexibility, but the flexibility comes with a degree of complexity that is not present when using ToQ. In contrast, ToQ.jl provides much (but, at present, not all) of the flexibility that comes with using the C or Python APIs, but without the additional complexity. For example, when using ToQ or ToQ.jl, the programmer is able to define variables that correspond to logical bits. In contrast, when using the C or Python APIs, the programmer has to keep track of indices that correspond to bits rather than using a C or Python variable to represent the bit. The examples we study demonstrate that relatively complex problems can be expressed in short ToQ.jl code (see the appendices).

II. ToQ.jl

The basic input to a QAM is a quadratic unconstrained binary optimization (QUBO) problem. The objective function associated with a QUBO can be formulated as

$$f(\mathbf{q}) = \sum_i v_i q_i + \sum_{i<j} w_{ij} q_i q_j \tag{1}$$

The QAM returns binary vectors, $\mathbf{q}$, that are preferentially sampled so that $f(\mathbf{q})$ is small. The samples have a distribution which is approximately Boltzmann [5]. The goal of ToQ.jl is to make it easy to represent problems in the form of (1). While this form may be restrictive, it has been shown that many NP-hard problems can be formulated in an equivalent Ising formulation [6].

The design of ToQ.jl is heavily influenced by JuMP (a modeling language for mathematical programming that extends Julia) [7] and ToQ itself. Writing a program with ToQ.jl generally follows these steps:

1) Create a ToQ.jl model object
2) Define the variables and parameters of the model (we will discuss the difference between these)
3) Add terms to the QUBO objective function (1)
4) Execute the model on the QAM
5) Load the samples

Creating a ToQ.jl model object is achieved by calling `ToQ.Model(...)` where the parameters to this function describe the QAM and give names for where files associated with the model will be stored seamlessly in the background.

If we think of the QAM as being analogous to a GPU, then a model is analogous to a GPU kernel.

One can define scalar, array, matrix, *etc* variables with ToQ.jl's `@defvar` macro. For example, `@defvar model mybit`, `@defvar model mybits[1:m]`, and `@defvar model mybits[1:m, 1:n]` would create a scalar, m-element array, and an m-by-n matrix variable, respectively, as part of `model` where `m` and `n` are Julia integer variables. Parameters are defined similarly with the `@defparam` macro, but only scalar parameters are permitted. It can be easy to conflate "variables" and "parameters" (a nomenclature which we borrowed from ToQ), but they are distinct. A variable corresponds to a q_i in (1), but a parameter does not. Note that set of functions that can be written as (1) form a vector space. In particular, if f and g are in this form, then so is $\alpha f + \beta g$. Parameters are akin to the α and β. The parameters make it possible to vary the weights of different parts of (1) without having to redo a (potentially expensive) operation called embedding. The utility of parameters will be illustrated in the applications.

Initially the QUBO ($f(\mathbf{q})$ from (1)) associated with the model is equal to zero. One can add terms to $f(\mathbf{q})$ with the `@addterm` macro. For example `@addterm model exp(2)*mybit` where `mybit` is a ToQ.jl variable would add a linear term to $f(\mathbf{q})$ where the v_i corresponding to `mybit` would be incremented by e^2. Note that the ability to use the exponential function in these terms comes from Julia's underlying implementation of the exponential function. Parameters can also be included in the terms, but the value of the parameters must be set before executing on the QAM. For example, it is possible to do `@addterm model param*mybit` where `param` is a ToQ.jl parameter. Quadratic terms can also be added, `@addterm model param*mybits[j]*mybits[i]` where `mybits` is a variable array and `i` and `j` are Julia integer variables. The terms must be products, contain one or two variables, and zero or one parameters. They can contain an arbitrary number of other valid Julia expressions that evaluate to a scalar.

By calling `ToQ.solve!(model; kwargs...)`, one causes the QAM to produce samples for the QUBO associated with `model` using a set of keyword arguments `kwargs`. ToQ.jl accomplishes this by translating the model into a ToQ program and using the 'dw' command line interface [3] to compile, embed (if desired), and execute the QUBO on the QAM. There are several keyword arguments that can be passed to this function. There is a keyword argument for each of the parameters – it is at this time that the parameter values are set. Other keyword arguments include `numreads` (how many samples the machine should produce) and `doembed`. The `doembed` argument controls whether or not the model should be "embedded." ToQ.jl allows one to pose a logical QUBO in the form of (1), but the QAM imposes additional restrictions (sparsity on w_{ij}). The process of embedding transforms the logical QUBO into one that matches the sparsity structure imposed by the machine. This introduces an additional keyword argument `param_chain` that is effectively a parameter that must be set. Embedding adds new terms to the QUBO, and `param_chain` controls the magnitude of these terms. One can reset the parameters' values and call `ToQ.solve!` again without having to do the embedding again by passing `doembed=false` as a keyword argument.

After calling `ToQ.solve!`, `ToQ.getnumsolutions` can be used to obtain the number of unique samples returned by the QAM. Then the i^{th} solution can be loaded with `@loadsolution model energy occs isvalid i` where now `energy` contains the energy of the sample (reported by the QAM), `occs` contains the number of occurrences of that sample, and `isvalid` is true if the sample is valid (i.e., the embedding process did not break the logical structure for this sample). The values of the ToQ.jl variables associated with this solution can then be accessed via `varname.value` where `varname` is the name of the variable.

III. Application to Map Coloring

Having described the basic usage of ToQ.jl, we now illustrate the usage with an application to coloring the map of Canada with three colors so that no two neighboring provinces/territories (hereafter, just provinces for brevity) have the same color. A C implementation of this map-coloring problem is part of D-Wave's educational materials (e.g., [8]). A condensed version of the source code for solving this problem is given in appendix A. The first handful of lines set up an array of the provinces of Canada and a dictionary describing their neighbor relationships, then a ToQ.jl model object is created. Next, the variables that will be used are defined via `@defvar m rgb[1:length(provs), 1:3]`, which is a matrix with one row for each province and 3 columns. The idea behind this is that each row will contain exactly one 1. The column in which the 1 is contained determines which color is assigned to the province corresponding to that row. For example, the first row is the row for British Columbia. If `rgb[1, 1]==1`, `rgb[1, 2]==1`, or `rgb[1, 3]==1`, British Columbia would be assigned the color red, green, or blue, respectively.

The QUBO must be designed in such a way that each row is encouraged to contain exactly one nonzero bit and so that if two provinces neighbor one another, they do not have a one in the same column. The QUBO can be represented in the form

$$f(\mathbf{q}) = \alpha \sum_{p \in \mathcal{P}} g_p(\mathbf{q}) + \beta \sum_{\{p_1,p_2\} \in \mathcal{N}} h_{\{p_1,p_2\}}(\mathbf{q}) \qquad (2)$$

where $\mathbf{q}$ corresponds to `rgb`, $\mathcal{P}$ is the set of provinces, g_p is minimized when the row of `rgb` corresponding to province p has exactly one nonzero element, $\mathcal{N}$ is a set containing all sets of neighboring provinces, and $h_{\{p_1,p_2\}}$ is minimized when the the rows in `rgb` corresponding to provinces p_1 and p_2 do not contain a 1 in the same column. At a high level, the first term of (2) encourages the QAM to produce samples such that each province is assigned exactly one color and the second

term encourages the QAM to produce samples such that no two neighboring provinces have the same color. The constants α and β will correspond to ToQ.jl parameters that control how strongly the QAM should be encouraged to achieve each of these goals. α corresponds to the ToQ.jl parameter, `c_p`, defined via `@defparam m c_p` and β corresponds to `n_p`.

The first term is constructed with 3 nested loops:

```
for i = 1:length(provs)
 for j = 1:3
  @addterm m -1*c_p*rgb[i, j]
  for k = 1:j - 1
   @addterm m 2*c_p*rgb[i, j]*rgb[i, k]
   ...
```

The second term is constructed with 3 nested loops where the innermost loop is executed only if the two provinces are neighbors:

```
for j = 1:length(provs)
 for k = 1:j - 1
  if provs[k] in nbors[provs[j]]
   for i = 1:3
    @addterm m n_p*rgb[j, i]*rgb[k, i]
    ...
```

Since ToQ.jl extends Julia, the high level constructs we used to describe the provinces and their neighbor relationships (an array of strings and a dictionary) can be readily used to construct the QUBO.

The model is executed on the QAM by calling `ToQ.solve!(m; ...)`, where in the listing in appendix A, `c_p`, `n_p`, and `param_chain` are set to 1, 5, and 2 (respectively), the number of samples (`numreads`) is set to 100, and the embedding will be performed (`doembed=true`). The values of `c_p`, `n_p`, and `param_chain` must be tuned to obtain good results. For example, if `c_p` is too large in comparison to `n_p` and `param_chain`, the samples will tend to assign a unique color to the provinces, but will be prone to have neighbors assigned the same color and/or inconsistencies with the logical QUBO. Similar undesired behavior results if `n_p` or `param_chain` is too large in comparison to the other parameters. Note that on subsequent solves, the parameters could be set to different values without needing to embed the problem again. That is, in subsequent solves, one can set `doembed=false` and this will speed the process up. Finally, solutions can be loaded with `@loadsolution`.

IV. Application to Linear Least Squares Problems

We begin by considering the binary linear least squares problem

$$\hat{\mathbf{q}} = \arg\min_{\mathbf{q}} ||A\mathbf{q} - \mathbf{b}||_2^2 \tag{3}$$

where A is a real-valued matrix, $\mathbf{b}$ is a real-valued vector and $\mathbf{q}$ is a binary vector. The objective function for this optimization problem is readily represented in the form of (1) with

$$v_j = \sum_i A_{ij}(A_{ij} - 2b_j) \tag{4}$$

$$w_{jk} = 2\sum_i A_{ij}A_{ik} \tag{5}$$

where A_{ij} is the component of A in the i^{th} row and j^{th} column and b_i is the i^{th} component of $\mathbf{b}$. The ToQ.jl code for solving this problem is listed in appendix B. Note that the code in appendix B is implemented as a Julia function which can be called just like any other Julia function.

Suppose that we now wish to solve a (non-binary) linear least squares problem

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} ||A\mathbf{x} - \mathbf{b}||_2^2 \tag{6}$$

where $\mathbf{x}$ is not a binary vector, but a real-valued vector with N components. To solve this problem using the binary least squares routine, we discretize $\mathbf{x}$ with an n-bit fixed point approximation

$$x_i^d = \sum_{j=1}^{n} 2^{j_0 - j} q_{(i-1)n+j} \tag{7}$$

where j_0 sets the fixed point, $x^d = (x_1^d, x_2^d, \ldots, x_N^d)$ is the discretized version of $\mathbf{x}$ and $\mathbf{q} = (q_1, q_2, \ldots, q_{Nn})$ is a binary vector. Then we form a new matrix, A^d with Nn columns that is constructed in such a way that $A\mathbf{x}^d = A^d\mathbf{q}$. This is achieved by setting the $[(i-1)n+j]^{\text{th}}$ column of A^d equal to 2^{j_0-j} times the i^{th} column of A where i ranges from 1 to N and j ranges from 1 to n. Using A^d, we can approximately reformulate the linear least squares problem in terms of $\mathbf{q}$

$$\hat{\mathbf{q}} = \arg\min_{\mathbf{q}} ||A^d\mathbf{q} - \mathbf{b}||_2^2 \tag{8}$$

Now, the problem can be optimized by the QAM using the QUBO formulation given in equations 4 and 5 with A_{ij} replaced by A_{ij}^d.

A. Laplace's Equation

Our development of the linear least squares problems arose out of an attempt to use the D-Wave 2X to solve a one-dimensional discretization of Laplace's equation,

$$L\mathbf{x} \equiv \begin{bmatrix} -2 & 1 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \ldots & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \ldots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 1 & -2 \end{bmatrix} \mathbf{x} = \mathbf{b} \tag{9}$$

This equation has many physical interpretations. For example, in subsurface hydrology, this is a discretized version of the steady-state groundwater equation where $\mathbf{x}$ would be a discretization of the pressure and $\mathbf{b}$ would be a discretization of the groundwater recharge.

Using the fixed-point least squares approach described previously, we obtained 10,000 samples from the D-Wave 2X

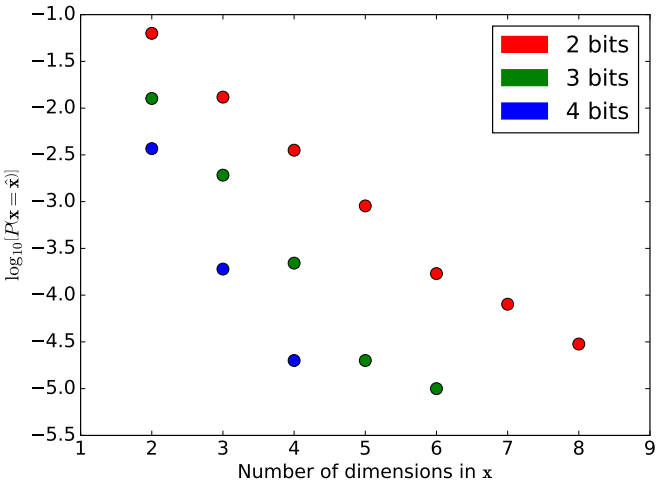

Fig. 1. The probability of obtaining the optimal fixed-point solution to a discrete, one-dimensional Laplace equation as a function of the number of (real-valued) unknowns is plotted for several different fixed-point representations. No optimal solutions were obtained when the dimension of $\mathbf{x}$ was 9 or 10.

using 10 different embeddings for 100,000 samples in total. We explored fixed-point representations with 2, 3, and 4 bits; and attempted to find a least squares solution to (9) with the number of unknowns (dimension of $\mathbf{x}$) ranging from 2 to 10. The fixed-point version of the discrete Laplacian, L^d, and $\mathbf{b}$ were then passed to the function listed in appendix B to obtain the samples.

Fig. 2 shows an estimate of the probability of obtaining a least squares solution for a subset of these problems. The subset in the figure contains those where at least one of the 100,000 samples was a least squares solution. In the other cases, we estimate that the probability of obtaining a least squares solution is less than or approximately one in 100,000. For least squares problems, this probability is a natural performance metric. An alternative would be to compute the expected time to obtain the least squares solution (which would be inversely proportional to the probability and proportional to the annealing time). Doing so would bring the annealing time into the picture (potentially an important component of performance), but we have left the annealing time fixed at the lowest possible value of 20 microseconds in all the cases examined here. Another important performance metric is the asymptotic time required to assemble the QUBO. In the case of an $N \times N$ dense matrix, this time is $O(N^3)$. This is the same as solving a dense least squares problem with a classical computer, so it is unlikely that solving this problem with a QAM will ever provide an enormous performance increase. Sparse systems or binary least squares problems [9], [10] may have more potential for performance improvements with a QAM.

V. User Experience

From our perspective, there are two main complications that arise when programming for the QAM:

1) The values of v_i and w_{ij} are bounded. They correspond to physical machine settings that have limits.
2) If we think of w_{ij} as a matrix, the matrix is sparse and the structure of the sparseness is fixed by the QAM. A given w_{ij} can be nonzero only if there is a physical coupling between the bits q_i and q_j.

The last complication can be resolved by "chaining" bits together, but this resolution introduces a complication of its own.

In the current hardware (the D-Wave 2X), no three bits are mutually coupled. That is if w_{ij} is allowed to be nonzero and w_{jk} is allowed to be nonzero, then w_{ik} must be zero. The idea of chaining is to "chain" two physical bits together to represent one logical bit. For example if the logical bits q_1, q_2, and q_3 needed to be mutually coupled to represent the desired QUBO, the logical q_1 could be represented with two physical bits that have the following properties: they are coupled to one another, one of them is coupled to the physical bit for q_2, and the other is coupled to the physical bit for q_3. Terms would have to be added to the objective function in (1) so that the QAM is encouraged to return samples where the two physical bits that represent the logical bit q_1 are equal to one another. This is the embedding process that is automatically carried out for ToQ.jl models. The automatic process embeds a logical QUBO into a QUBO that fits within the sparsity structure imposed by the hardware. Of course, it is not always possible to do so, and the process may fail.

The complication introduced by chaining is that one must determine how to scale the terms added to the original objective function by the chaining process. If $f(\mathbf{q})$ represents our original objective function and $g(\mathbf{q})$ consists of the terms introduced in the chaining process, the new objective function is

$$\hat{f}(\mathbf{q}) = f(\mathbf{q}) + \alpha g(\mathbf{q}) \tag{10}$$

where $\alpha > 0$ is the scaling factor (corresponding to the `param_chain` keyword argument of `ToQ.solve!`). If α is too small, the samples returned by the QAM will be inconsistent with the logical QUBO given by $f(\mathbf{q})$. That is, the physical bits that are supposed to represent a single logical bit will not have the same value, so there is not a unique mapping from the physical bits to the logical bits. If such a situation occurs, we say that the chain is broken. If the chain is broken, the logical bits are not properly coupled. On the other hand, if α is too large, chains will be unlikely to be broken, but the samples are less likely to make $f(\mathbf{q})$ small. That is, many samples may be required from the QAM before one is obtained that minimizes the original QUBO. One way to conceptualize the QUBOs is as a vector space, but the QUBOs that can be executed by a QAM all fit inside a box; so this conceptualization can be misleading. Once α reaches a certain threshold, the QUBO represented by $\alpha g(\mathbf{q})$ will be on the boundary of the box. Further increasing α will effectively

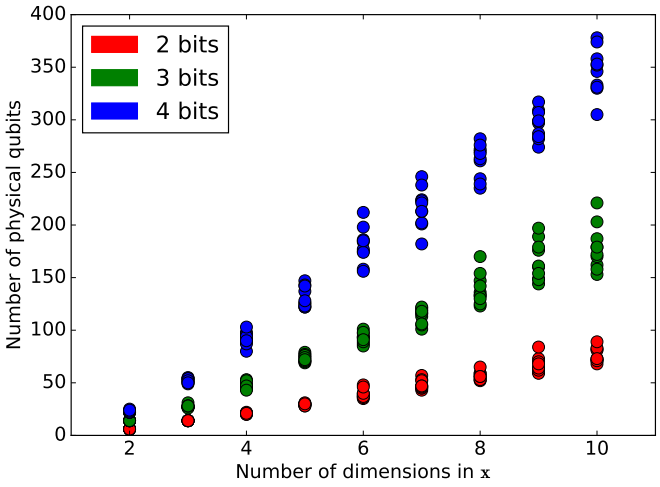

Fig. 2. The number of physical bits used to represent represent the fixed-point solution as a function of the number of (real-valued) unknowns is plotted for several different fixed-point representations.

diminish the weight of the $f(\mathbf{q})$ term in (10), since the QUBO will be automatically rescaled to fit in the box.

The process of chaining/embedding also significantly increases the number of bits required to represent the QUBO. Fig. 2 shows the number of bits required to solve the embedded version of Laplace's equation. There is some variability in the number of physical bits in the embedded problem because the software that performs the embedding uses a stochastic algorithm. The number of physical bits that must be utilized to represent the logical QUBO grows much faster than the number of logical bits needed to represent the logical QUBO. For example, the 4-bit, 10-dimensional Laplace's equation typically requires approximately 350 physical bits whereas the logical problem requires only 40 bits.

Note that w_{jk} in (5) is nonzero if and only if there is a row of A with nonzero components in the j^{th} and k^{th} columns. This means that if A has a sparsity structure that is consistent with the sparsity of the couplings between the bits in the QAM, the chaining trick described above is not necessary. We expect that problems which fit naturally into the QAMs topology (i.e., the sparsity structure of w_{ij}) would provide much better performance in terms of the probability of obtaining the least squares solution. This would provide the additional benefit of being able to solve larger problems, as the number of bits in the logical problem would be the same as the number of bits used in the QAM.

In these early stages of using the QAM, the goal is often seen as finding a real-world problem that is a member of the (relatively small) set of problems for which the QAM provides good performance. An alternative goal would be to find a QAM for which a given problem can be solved efficiently. This search could be carried out with QAM simulators. If a flexible simulator were built that allowed users to set the size, topology, temperature (in the Boltzmann distribution), and size of the box that the QUBO coefficients must fit in, we could then answer questions like "What is the simplest QAM that can realize a desired performance metric for a given problem?". Answering such a question would provide guidance about the development of future QAMs in hardware.

VI. Conclusion & Future Directions

We have presented the design and a couple uses of ToQ.jl, a high-level programming language for QAMs. ToQ.jl makes it relatively easy to write short codes that can solve diverse problems on a QAM. We demonstrated this with applications to map-coloring and linear least squares problems. In its current form, ToQ.jl replicates what we see as the key features of ToQ (variables, parameters, and the ability to add terms to the QUBO) while leveraging the capabilities of Julia so that complex QUBOs can be constructed easily. There are a number of ways that ToQ.jl could be improved to give programmers more flexibility without adding too much additional complexity.

Our perception is that getting the best performance out of the QAM requires a problem which fits naturally in the QAMs topology. For such problems, the process of chaining/embedding would have a negative impact on performance since the stochastic embedding algorithm is unlikely to find a perfect match between the physical bits and the logical bits. Because of this issue, it is imperative to give the programmer more control over how ToQ.jl's variables are mapped onto the physical bits. ToQ.jl's `@defvar` macro can be extended to permit this, e.g., `@defvar model logbits[1:n] physbits` could be used to give control over the physical bits that correspond to the logical bits. In this case, `physbits` would be an integer array with `n` elements containing the indices of the physical bits that correspond to the logical bits in `logbits`. Doing this would require more work on the back end. Currently, we are using ToQ and the 'dw' command line interface as the back end, but the Python or C API would be a more natural back end in this situation. Reworking the back end to use one of these APIs would be relatively straightforward, but it remains to be done. The ability to associate ToQ.jl variables with physical bits is the main feature that limits the flexibility of ToQ.jl in comparison to using the Python or C APIs directly. Adding this feature would largely bridge the gap so that ToQ.jl would have the flexibility of the Python and C API while being more expressive.

Some other features that would be beneficial include constructs that allow the programmer to express concepts at a higher level. These would enhance the expressiveness of ToQ.jl. For example, the only way to construct a QUBO presently is with the `@addterm` macro where each term must be added one at a time. This could be extended so that something like `@addterms model 2*q1+3*q1*q2` so that multiple terms could be added at once. It would also be useful to automatically expand quadratic expressions – `@addterms model (q[i-1]-2*q[i]+q[i+1])^2` could be used where ToQ.jl would automatically expand the quadratic expression and add the terms to the model.

For example, including such an expression in a loop (and dealing with the first and last rows separately) would make it very natural to represent the discrete Laplace equation as a QUBO where the unknown is a binary vector. Higher order expressions (cubic, quartic, *etc*.) could also be automatically expanded and represented as a QUBO. However, we are somewhat hesitant to implement the higher order expressions because it requires the automatic creation of extra variables and increases parameter chaining, both of which are likely to have a detrimental impact on performance.

There are many possible directions for future improvements to ToQ.jl. We are currently in the process of releasing ToQ.jl as open-source software; we welcome and encourage feedback, feature requests, and contributions.

Appendix A
Condensed map coloring code

```
using ToQ
provs = ["BC", "YK", "NW", "AB", ...
nbors = Dict()
nbors["BC"] = ["YK", "NW", "AB"]
nbors["YK"] = ["BC", "NW"]
...set up more neighbors...
nbors["PE"] = []
nbors["NL"] = ["QB"]
m = ToQ.Model("canada_model",...
@defvar m rgb[1:length(provs), 1:3]
@defparam m c_p #color penalty
@defparam m n_p #neighbor penalty
#add color penalties
for i = 1:length(provs)
 for j = 1:3
  @addterm m -1*c_p*rgb[i, j]
  for k = 1:j - 1
   @addterm m 2*c_p*rgb[i, j]*rgb[i, k]
  end
 end
end
#add neighbor penalties
for j = 1:length(provs)
 for k = 1:j - 1
  if provs[k] in nbors[provs[j]]
   for i = 1:3
    @addterm m n_p*rgb[j, i]*rgb[k, i]
   end
  end
 end
end
#solve the system on the QAM
ToQ.solve!(m; c_p=1, n_p=5, param_chain=2,
  numreads=100, doembed=true)
for i = 1:ToQ.getnumsolutions(m)
 @loadsolution m energy num_occ valid i
 #do something with the solution
end
```

Appendix B
Condensed least squares code

```
function binlin(A, b; e_pval=1 / 8,...
 m = ToQ.Model("binlin_model", ...
 @defparam m e_p
 @defvar m x[1:size(A, 2)]
 #set up each equation
 v = zeros(size(A, 2))
 w = zeros(size(A, 2), size(A, 2))
 for i = 1:length(b)
  for j = 1:size(A, 2)
   v[j] += A[i, j] * (A[i, j] - 2 * b[i])
   for k = 1:j - 1
    w[j, k] += 2 * A[i, j] * A[i, k]
   end
  end
 end
 for i = 1:length(v)
  if v[i] != 0
   @addterm m e_p * v[i] * x[i]
  end
  for j = 1:length(v)
   if w[i, j] != 0
    @addterm m e_p * w[i, j] * x[i] * x[j]
   end
  end
 end
 #solve the system
 ToQ.solve!(m; e_p=e_pval, param_chain=1,
  numreads=numreads, doembed=true)
 #load the solutions
 sols = Array{Float64, 1}[]
 phys_OFs = Float64[]
 log_OFs = Float64[]
 num_occ = Float64[]
 for i = 1:ToQ.getnumsolutions(m)
  @loadsolution m energy num_occi valid i
  push!(sols, copy(x.value))
  push!(phys_OFs, energy)
  logical_obj_func = sumabs2(A*x.value-b)
  push!(log_OFs, logical_obj_func)
  push!(num_occ, num_occi)
 end
 return sols,phys_OFs,log_OFs,num_occ,m
end
```


Acknowledgment

The authors acknowledge Edward D. Dahl for his instruction and helpful discussions during the preparation of the manuscript. DO acknowledges the support of a Los Alamos National Laboratory Director's Postdoctoral Fellowship. VVV acknowledges the support of the DiaMonD project (An Integrated Multifaceted Approach to Mathematics at the Interfaces of Data, Models, and Decisions, U.S. Department of Energy Office of Science, Grant #11145687).

REFERENCES

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *arXiv preprint arXiv:1411.1607*, 2014.

[3] D.-W. S. Inc., "Software," 2016, accessed 20-April-2016. [Online]. Available: http://www.dwavesys.com/software

[4] JuliaLang, "Metaprogramming," 2016, accessed 20-April-2016. [Online]. Available: http://docs.julialang.org/en/release-0.4/manual/metaprogramming/

[5] Z. Bian, F. Chudak, W. G. Macready, and G. Rose, "The ising model: teaching an old problem new tricks," D-Wave Systems Inc., Burnaby, British Columbia, Canada, Tech. Rep., 2010, accessed 20-April-2016. [Online]. Available: http://www.dwavesys.com/sites/default/files/weightedmaxsat_v2.pdf

[6] A. Lucas, "Ising formulations of many np problems," *Frontiers in Physics*, vol. 12, 2014.

[7] M. Lubin and I. Dunning, "Computing in operations research using julia," *INFORMS Journal on Computing*, vol. 27, no. 2, pp. 238–248, 2015. [Online]. Available: http://dx.doi.org/10.1287/ijoc.2014.0623

[8] E. D. Dahl, "Programming with d-wave: Map coloring problem," D-Wave Systems Inc., Burnaby, British Columbia, Canada, Tech. Rep., 2013, accessed 20-April-2016. [Online]. Available: http://www.dwavesys.com/sites/default/files/Map%20Coloring%20WP2.pdf

[9] S. Chrétien and F. Corset, "Using the eigenvalue relaxation for binary least-squares estimation problems," *Signal Processing*, vol. 89, no. 11, pp. 2079–2091, 2009.

[10] E. Tsakonas, J. Jaldén, and B. Ottersten, "Robust binary least squares: Relaxations and algorithms," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 3780–3783.