A document containing background information for those interested   
in programming and understanding quantum computers

**Contents**

 [1.1 - Conventional computing](http://www.dwavesys.com/en/dev-tutorial-intro.html#section1p1)

 [1.2 - A new kind of computing](http://www.dwavesys.com/en/dev-tutorial-intro.html#section1p2)

 [1.3 - The light switch game](http://www.dwavesys.com/en/dev-tutorial-intro.html#section1p3)

 [1.4 - How does quantum mechanics help?](http://www.dwavesys.com/en/dev-tutorial-intro.html#section1p4)

 [2.1 - It's a math expression - who cares?](http://www.dwavesys.com/en/dev-tutorial-intro.html#section2p1)

 [2.2 - The energy program](http://www.dwavesys.com/en/dev-tutorial-intro.html#section2p2)

 [2.3 - Quantum computers can LEARN](http://www.dwavesys.com/en/dev-tutorial-intro.html#section2p3)

 [2.4 - A computer that programs itself](http://www.dwavesys.com/en/dev-tutorial-intro.html#section2p4)

 [2.5 - Uncertainty is a feature](http://www.dwavesys.com/en/dev-tutorial-intro.html#section2p5)

**Aim, audience and required background**

The aim of this document is to introduce people to the concepts and terminology used in Quantum Computing. It provides an overview of what a Quantum Computer is, and why you would want to program one.

The material here is written using very high level concepts and is designed to be accessible to both technical and non-technical audiences. Some background in physics, mathematics and programming is useful to help understand the concepts presented in this document, although this is not a solid requirement.

**What you will learn**

By following through the material in this primer, you will learn:

 How quantum physics gives us a new way to compute

 The similarities and differences between quantum computing and classical computing

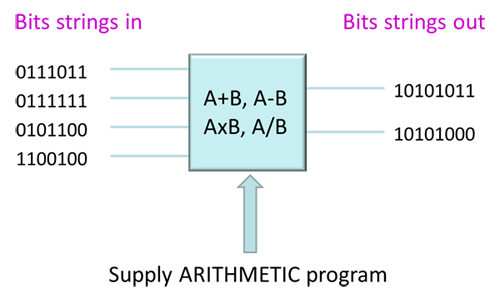
 How the fundamental units of quantum computing (qubits) are manipulated to solve hard problems

 Why Quantum Computing is well suited to AI and machine learning applications, and how quantum computers may be used as 'AI co-processors'

SECTION 1

1.1 - Conventional computing

To understand quantum computing, it is useful to first think about conventional computing. We take modern digital computers and their ability to perform a multitude of different applications for granted. Our desktop PCs, laptops and smart phones can run spreadsheets, stream live video, allow us to chat with people on the other side of the world, and immerse us in realistic 3D environments. But at their core, all digital computers have something in common. They all perform simple arithmetic operations. Their power comes from the immense speed at which they are able to do this. Computers perform billions of operations per second. These operations are performed so quickly that they allow us to run very complex high level applications. Conventional digital computing can be summed up by the diagram shown in figure 1.



*Figure 1. Dataflow in a conventional computer*

Although there are many tasks that conventional computers are very good at, there are still some areas where calculations seem to be exceedingly difficult. Examples of these areas are: Image recognition, natural language (getting a computer to understand what we mean if we speak to it using our own language rather than a programming language), and tasks where a computer must learn from experience to become better at a particular task. Even though there has been much effort and research poured into this field over the past few decades, our progress in this area has been slow and the prototypes that we do have working usually require very large supercomputers to run them, consuming a vast quantities of space and power.

We can ask the question: Is there a different way of designing computing systems, from the ground up? If we could start again from scratch and do something completely different, to be better at these tasks that conventional computers find hard, how would we go about building a new type of computer?

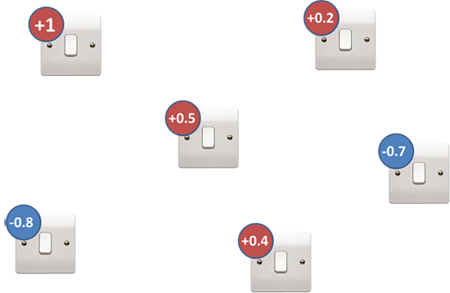
1.2 - A new kind of computing

Quantum computing is radically different from the conventional approach of transforming bits strings from one set of 0's and 1's to another. With quantum computing, everything changes. The physics that we use to understand bits of information and the devices that manipulate them are totally different. The way in which we build such devices is different, requiring new materials, new design rules and new processor architectures. Finally, the way we program these systems is entirely different. This document will explore the first of these issues, how replacing the conventional bit (0 or 1) with a new type of information - the qubit - can change the way we think about computing.

1.3 - The light switch game

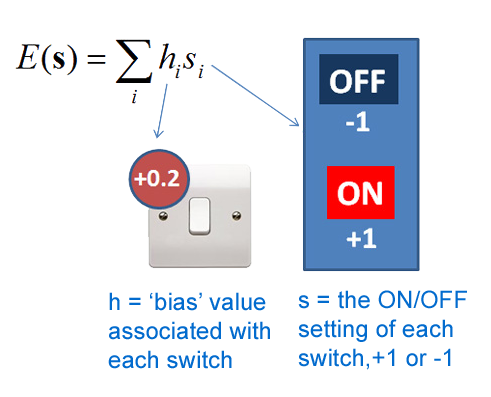
To begin learning about quantum computing, it is important to understand why we can't use conventional digital computers to solve certain problems. Let's consider a mathematical problem, which we'll call the light switch game, that illustrates this point.

The light switch game involves trying to find the best settings for a bunch of switches. Here's a graphical example introducing this problem:



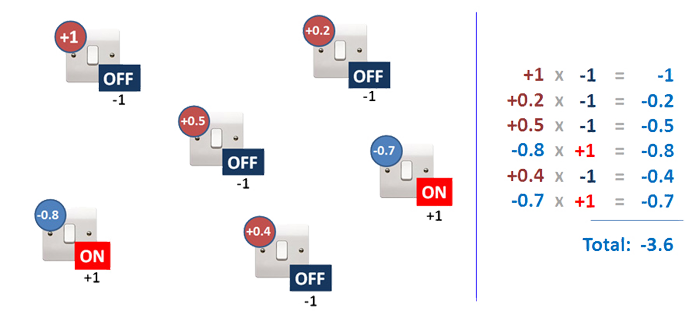
*Figure 2. The light switch game*

Let's imagine that each light switches has a number associated with it, which is chosen for you (you don't get to change this). We call this a 'bias value'. You get to choose whether to turn each light swtich ON or OFF. In our game, ON = +1 and OFF = -1. We then add up all the switches' bias values multiplied by their ON/OFF values. This gives us a number. The objective of the game is to set the switches to get the lowest number. Mathematically, we call the bias values of each switch *hi*and the switch settings are called *si*.



*Figure 3. Playing the light switch game - add up the bias values of each switch multiplied by their settings (which you have to choose)*

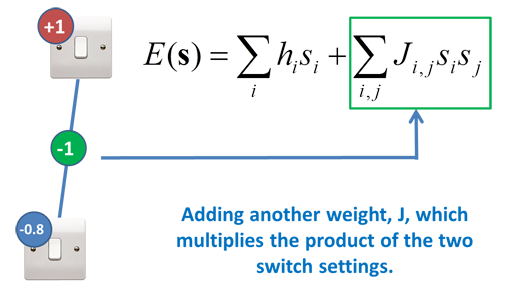
So depending upon which switches we set to +1 and which we set to -1, we will get a different score overall. You can try this game. Hopefully you'll find it easy because there's a simple rule to winning:



*Figure 4. Working out the answer for a particular "guess" at the switch settings*

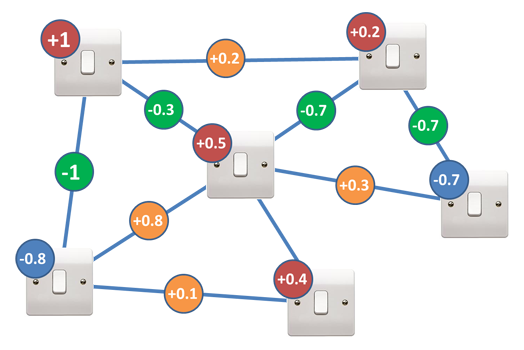
We find that if we set all the switches with positive biases to OFF and all the switches with negative biases to ON and add up the result then we get the lowest overall value. Easy, right? I can give you as many switches as I want with many different bias values and you just look at each one in turn and flip it either ON or OFF accordingly.

OK, let's make it harder. So now imagine that many of the pairs of switches have an additional rule, one which involves considering PAIRS of switches in addition to just individual switches... we add a new bias value (called J) which we multiply by BOTH the switch settings that connect to it, and we add the resulting value we get from each pair of switches to our overall number too. Still, all we have to do is decide whether each switch should be ON or OFF subject to this new rule.



*Figure 5. Making the game harder by adding additional terms that depend on the settings of pairs of switches.*

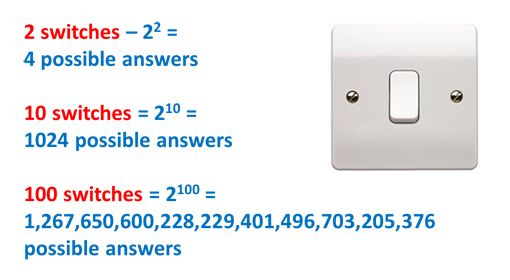
But now it is much, much harder to decide whether a switch should be ON or OFF, because its neighbours affect it. Even with the simple example shown with 2 switches in the figure above, you can't just follow the rule of setting them to be the opposite sign to their bias value anymore (try it!). With a complex web of switches having many neighbours, it quickly becomes very frustrating to try and find the right combination to give you the lowest value overall.



*Figure 6. The light switch game with connecting terms added, generating an 'interacting' web of light switches*

1.4 - How does quantum mechanics help?

With a couple of switches you can just try every combination of ON's and OFF's, there are only four possibilities: [ON ON], [ON OFF], [OFF ON] or [OFF OFF]. But as you add more and more switches, the number of possible ways that the switches can be set grows exponentially:



*Figure 7. The exponential problem with the light switch game.*

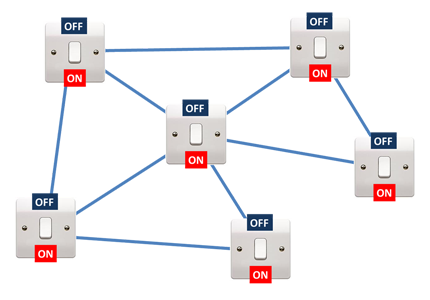
You can start to see why the game isn't much fun anymore. In fact it is even difficult for our most powerful supercomputers. Being able to store all those possible configurations in memory, and moving them around inside conventional processors to calculate if our guess is right takes a very, very long time. With only 500 switches, there isn't enough time in the Universe to check all the configurations.

Quantum mechanics can give us a helping hand with this problem. The fundamental power of a quantum computer comes from the idea that you can put bits of information into a superposition of states. You can think of this as being a situation where the qubit has not yet decided which state it wants to be in. Some people like to refer to the qubit in superposition as 'being in both states at the same time'. You can alternatively think of the qubit's state as being undecided as to whether it is +1 or -1. Which means that using a quantum computer, our light switches can be ON and OFF at the same time:



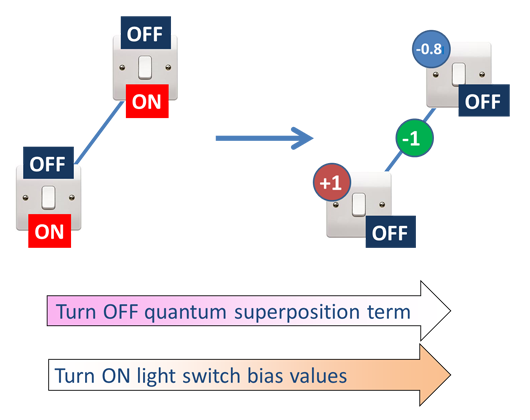
*Figure 8: A quantum mechanical bit of information (qubit) can reside in what is known as a superposition state, where it has not yet decided whether to be in the +1 or the -1 state (alternatively, you can think of it as being 'in both states').*

Now lets consider the same bunch of switches as before, but now held in a quantum computer's memory (notice that the bias values haven't been added yet):



*Figure 9. A network of connected quantum bits in superposition. The answer is in there somewhere!*

Because all the light switches are on and off at the same time, we know that the correct answer (correct ON/OFF settings for each switch) is represented in there somewhere - it is just currently hidden from us. But that is OK, because quantum mechanics is going to find it for us. The D-Wave quantum computer allows you to do is take a 'quantum representation' like this, and extract the configuration of ONs and OFFs with the lowest value. Here's is how it works:



*Figure 10. The computer begins with the bits in superposition, ends with them behaving as regular classical bits, and finds the answer along the way.*

You start with the system in its quantum superposition as described above, and you slowly adjust the quantum computer to turn off the quantum superposition effect. At the same time, you slowly turn up all those bias values (the h and J's from earlier). As this operation is performed, the switches slowly drop out of their superposition state and choose a classical state, either ON or OFF. At the end, each switch MUST have chosen to be either ON or OFF. The quantum mechanics working inside the computer helps the light switches settle into the right states to give the lowest overall value when you add them all up at the end. Even though with *N* switches there are 2*N* possible configurations it could have ended up in, it finds the lowest one, winning the light switch game. So we see that the quantum computer allows us to minimize expressions such as the one considered here:

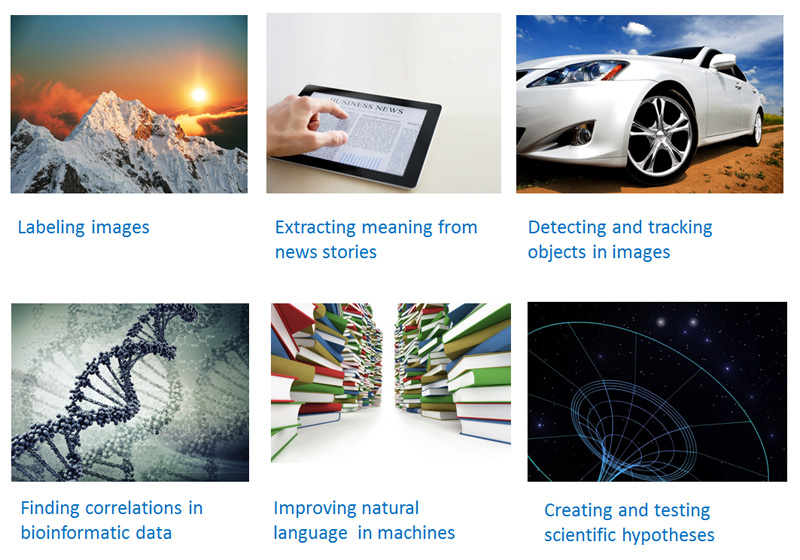
*E*(*s*)=∑*ihisi*+*Jijsisj*

which can be difficult (if not impossible) for classical computers.

SECTION 2

2.1 - It's a math expression - who cares?

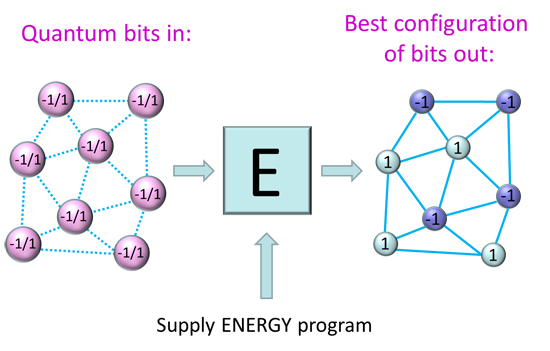
We didn't build a machine to play a strange masochistic light switch game. The concept of finding a good configuration of binary variables (switches) in this way lies at the heart of many problems that are encountered in everyday applications. A few are shown in figure below. Even the concept of scientific discovery itself is an optimization problem (you are trying to find the best 'configuration' of terms contributing to a scientific equation which match real world observations).



*Figure 11. Examples of applications which under the hood all involve finding good 'switch settings' and can be tackled more efficiently with quantum computers.*

2.2 - The energy program

To understand how these problems might be cast as finding settings of switches, let's consider how the quantum computer is programmed. Recall Figure 1, where bit strings in were transformed into other bits strings via the application of a logic program. Instead of that, we now have a resource where bits can be undecided, so the computation is performed in a fundamentally different way, as shown in Figure 12. In this case, a group of qubits are initialized into their superposition states, and this time an ENERGY PROGRAM (instead of a logic program) is applied to the group. The qubits go from being undecided at the beginning of the computation, to all having chosen either -1 or +1 states at the end of the computation. What is an Energy Program? It is just those h and J numbers - the bias settings - that we introduced earlier. In the light switch game, we said that the h and J's were given to you. Well, now we see where they come from - they are the definition of the problem you are trying to solve.



*Figure 12. The basic operation of a quantum computer is to supply an energy program (a series of h and J numbers) and let the computer find the switch settings (+1 and -1).*

Crafting an energy program as a series of h and J values - to encode a real world problem that you care about - is exceedingly hard and time-consuming. It would be the equivalent of programming you desktop PC by sending in machine code to the microprocessors inside! Luckily, there is a better way to program quantum computers by using a quantum compiler. This process is explained in more detail in the [QC Software Tutorial](http://www.dwavesys.com/en/dev-tutorial-software.html) and hands-on demonstrations of this technique form the majority of the [applications programming](http://www.dwavesys.com/en/dev-tutorials.html) tutorials.

2.3 - Quantum computers can LEARN

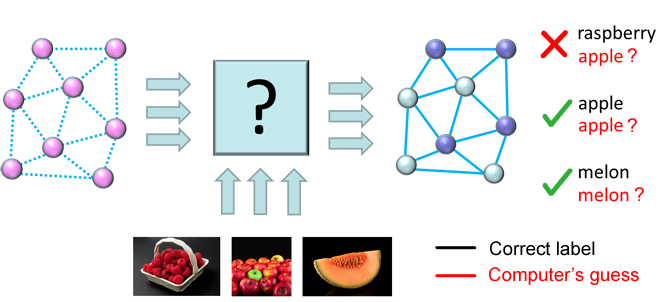
The discipline of teaching computers to reason about the world and learn from experience is known as machine learning. It is a sub-branch of the field of artificial intelligence. Most of the code we write is fairly static - that is, given new data it will perform the same computation over and over again and make the same errors. Using machine learning we can design programs which modify their own code and therefore learn new ways to handle pieces of data that they have never seen before.

The type of applications that run very well on D-Wave's hardware are applications where learning and decision making under uncertain conditions are required. For example, imagine if a computer was asked to classify an object based on several images of similar objects you had shown it in the past. This task is very difficult for conventional computing architectures, which are designed to follow very strict logical reasoning. If the system is shown a new image, it is hard to get it to make a general statement about the image, such as 'it looks similar to an apple'. D-Wave's processors are designed to support applications that require high level reasoning and decision making.

How can we use a quantum computer to implement learning, for example, if we want the system to recognize objects? Writing an energy program for this task would be very difficult, even using a quantum compiler, as we do not know in detail how to capture the essence of objects that the system must recognize. Luckily there is a way around this problem, as there is a mode in which the quantum computer can tweak its own energy program in response to new pieces of incoming data. This allows the machine to make a good guess at what an object might be, even if it has never seen a particular instance of it before. The following section gives an overview of this process, but the best way to understand the concepts in detail is by working through specific examples of coding such applications. Examples can be found on the [Tutorials](http://www.dwavesys.com/en/dev-tutorials.html) page.

2.4 - A computer that programs itself

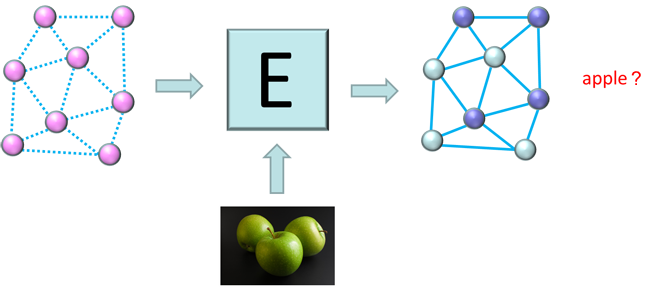
In order to get the system to tweak its own energy program, you start by showing the system lots and lots of instances of the concept that you want it to learn about. An example in shown in Figure 13. Here the idea is to try to get the computer to learn the difference between images of different types of fruit. In order to do this, we present images (or rather, a numeric representation of those images) to the system illustrating many different examples of apples, raspberries and melons. We also give the system the 'right' answer each time by telling it what switch settings (labels) it should be ending up selecting in each case. The system must find an energy program (shown as a question mark as we do not know it at the beginning) so that when an image is shown to the system, it gets the labels correct each time. If it gets many examples wrong, the algorithm knows that it must change its energy program.



*Figure 13. Teaching the quantum chip by allowing it to write its own energy program.   
The system tweaks the energy program until it labels all the examples that you show it correctly. This is also known as the 'training' or 'learning' phase.*

At first the system chooses an energy program (remember that it is just a bunch of h and J values) at random. It will get many of the labellings wrong, but that doesn't matter, as we can keep showing it the examples and each time allow it to tweak the energy program so that it gets more and more labels (switch settings) correct. Once it can't do any better on the data that it has been given, we then keep the final energy program and use that as our 'learned' program to classify a new, unseen example (figure 14).

In machine learning terminology this is known as a supervised learning algorithm because we are showing the computer examples of images and telling it what the correct labels should be in order to help it learn. There are other types of learning algorithms supported by the system, even ones that can be used if labeled data is not available. Subsequent tutorials in this series will go into these concepts in much more depth.



*Figure 14. After the system has found a good energy program during the   
training phase, it can now label unseen examples to solve a real world problem. This is known as the 'testing' phase.*

2.5 - Uncertainty is a feature

Another interesting point to note about the quantum computer is that it is probabilistic, meaning that it returns multiple answers. Some of these might be the answer that you are looking for, and some might not. At first this sounds like a bad thing, as a computer that returns a different answer when you ask it the same question sounds like a bug! However, in the quantum computer, this returning of multiple answers can give us important information about the confidence level of the computer. Using the fruit example above, if we showed the computer an image and asked it to label the same image 100 times, and it gave the answer 'apple' 100 times, then the computer is pretty confident that the image is an apple. However, if it returns the answer apple 50 times and raspberry 50 times, what this means is that the computer is uncertain about the image you are showing it. And if you had shown it an image with apples AND raspberries in, it would be perfectly correct! This uncertainty can be very powerful when you are designing systems which are able to make complex decisions and learn about the world.

How D-Wave processors are built, and how they use the physics of spin systems to implement quantum computation

**Contents**

 [1.1 - The building blocks of QC](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section1p1)

 [1.2 - A fabric of programmable elements](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section1p2)

 [1.3 - Support circuitry: Addressing, programming and reading the qubits](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section1p3)

 [1.4 - Manufacturing quantum processors](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section1p4)

 [2.1 - The processor packaging](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section2p1)

 [2.2 - Computer cooling](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section2p2)

 [2.3 - Computer shielding and wiring](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section2p3)

 [2.4 - Computer form factor](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section2p4)

 [2.5 - Cloud based access](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section2p5)

 [3.1 - The future of the hardware](http://www.dwavesys.com/en/dev-tutorial-hardware.html#section3p1)

**Aim, audience and required background**

The aim of this document is to describe how a quantum computer is physically built, how quantum bits and their associated circuitry are created, addressed, and controlled, and what is happening inside the computer when programmers send information to a D-Wave quantum machine.

The material here is written using very high level concepts and is designed to be accessible to both technical and non-technical audiences. Some background in physics, mathematics and programming is useful to help understand the concepts presented in this document, although this is not a solid requirement. In order to understand the material here you may also wish to read the [Quantum Computing Primer](http://www.dwavesys.com/en/dev-tutorial-intro.html) document first.

**What you will learn**

By following through the material in this primer, you will learn:

 How we can craft hardware to take advantage of quantum physics

 How quantum computer processors are radically different to conventional computing

 Why new materials are required to implement quantum computing

 How quantum bits (qubits) are fabricated, addressed, and controlled to manipulate quantum information

 What the various components of the D-Wave OneTM quantum computer look like, and how they are combined to form a computing system

SECTION 1: Inside the processor

1.1 - The building blocks of QC

In the previous tutorial, our representation of qubits as bits of information has been in a symbolic way, 0, 1 and a superposition state of both 0 and 1. But how are the qubits physically made? What do they look like?

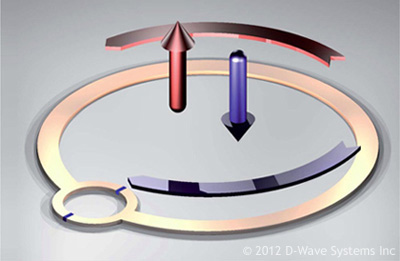
**Classical CMOS transistors**

The way that we encode and access information in modern digital computers is by adjusting and monitoring voltages that are present on tiny transistor switches inside integrated circuits. Each transistor is addressed by a bus which is able to set it to a state of either 0 (a low voltage) or 1 (a high voltage). So we use the idea of an electrical voltage to 'encode' bits of information in a physical device.

**The SQUID - a quantum transistor**

Quantum computers have similarities to and differences from this CMOS transistor idea. Figure 1 shows a schematic illustration of what is known as a superconducting qubit (also called a SQUID), which is the basic building block of a quantum computer (a quantum 'transistor', if you like). The name SQUID comes from the phrase Superconducting QUantum Interference Device. The term 'Interference' refers to the electrons - which behave as waves inside a quantum waves, interference patterns which give rise to the quantum effects. The reason that quantum effects such as electron waves are supported in such a structure - allowing it to behave as a qubit - is due to the properties of the material from which it is made. The large loop in the diagram is made from a metal called niobium (in contrast to conventional transistors which are mostly made from silicon). When this metal is cooled down, it becomes what is known as a superconductor, and it starts to exhibit quantum mechanical effects.

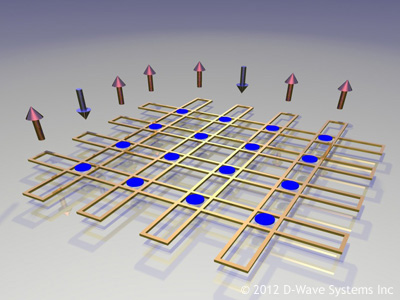
A regular transistor allows you to encode 2 different states (using voltages). The superconducting qubit structure instead encodes 2 states as tiny magnetic fields, which either point up or down. We call these states +1 and -1, and they correspond to the two states that the qubit can 'choose' between. Using the quantum mechanics that is accessible with these structures, we can control this object so that we can put the qubit into a superposition of these two states as described earlier. So by adjusting a control knob on the quantum computer, you can put all the qubits into a superposition state where it hasn't yet decided which of those +1, -1 states to be.



*Figure 1. Schematic of a superconducting qubit, the basic building block of the D-Wave OneTM   
Quantum Computer. The arrows indicate the magnetic spin states which encode the bits of information as +1 and -1   
values. Unlike regular bits of information, these states can be put into quantum mechanical superposition.*

1.2 - A fabric of programmable elements

In order to go from a single qubit to a multi-qubit processor, the qubits must be connected together such that they can exchange information. This is achieved through the use of elements known as couplers. The couplers are also made from superconducting loops. By putting many such elements (qubits and couplers) together, we can start to build up a fabric of quantum devices that are programmable. Figure 2 shows a schematic of 8 connected qubits. The loop shown in the previous diagram has now been stretched out to form one of the long gold rectangles. At the points where the rectangles cross, the couplers have been shown schematically as blue dots.



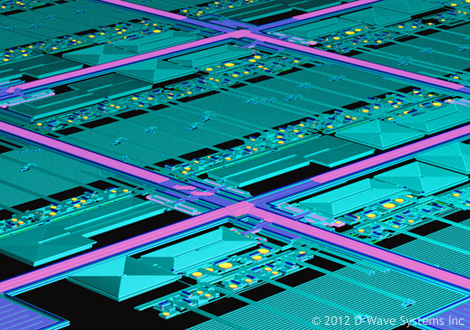
*Figure 2. Left: A schematic illustration of 8 qubit loops (gold). The blue dots are the locations of the 16 coupling elements that allow the qubits to exchange information. Mathematically, these elements couple together variables in a problem that you wish the computer to solve.*

1.3 - Support circuitry: Addressing, programming and reading the qubits

There are several additional components necessary for processor operation. A large part of the circuitry that surrounds the qubits and couplers is a framework of switches (also formed from Josephson junctions) forming circuitry which both addresses each qubit (routes pulses of magnetic information to the correct places on chip) and stores that information in a magnetic memory element local to each device. The majority of the 25,000 Josephson junctions in a Rainier processor are used to make up this circuitry. Additionally, there are readout devices attached to each qubit. During the computation these devices are inactive and do not affect the qubits' behaviour. After the computation has finished, and the qubits have settled into their final (classical) 0 or 1 states, the readouts are used to query the value held by each qubit and return the answer as a bit string of 0's and 1's to the end user. Here is a video showing how some of the processor elements are combined to produce the computational fabric at the core of the D-Wave OneTM Rainier processor:

*Inside the chip: Video showing 3D Animation of how the components in a Rainier processor fit together*

The image in figure 3 shows the layout of the actual circuit, as drawn in a CAD program by a D-Wave circuit layout designer, and is ready to be sent off to the processor fabrication foundry. Here the full complexity of the processor is revealed. In this image, the qubits are now shown as long pink strips, which have been stretched out even more than in the previous figure. The green and yellow elements that sit in the spaces between qubits are components which make up the programmable circuitry mentioned above. The yellow dots are Josephson junctions embedded within this circuitry.

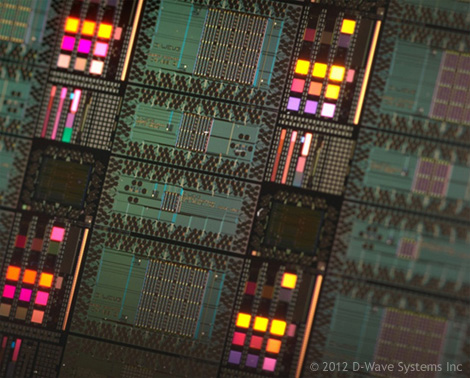
[](http://www.dwavesys.com/en/images/tut-hardware-pmmcloseup-big.jpg)

*Figure 3. (Click to open a larger version in a new tab) False-colour view of part of a CAD layout of the   
Rainier chip architecture. This image is from a real processor design layout file, which is sent to the manufacturer and   
from which the processors are fabricated layer by layer. The long qubit loops are now shown as the pink areas, the control   
circuitry lines which carry currents to the programmable are indicated by the green features and the Josephson junctions are shown in yellow.*

Note that this architecture is very different from conventional computing. The processor has no large areas of memory (cache), rather each qubit has a tiny piece of memory of its own. In fact, the chip is architected more like a biological brain than the common 'Von Neumann' architecture of a conventional silicon processor. One can think of the qubits as being like neurons, and the couplers as being like synapses that control the flow of information between those neurons. We'll see in later programming tutorials how this 'brain-like' architecture helps us to solve problems in machine learning and artificial intelligence using quantum computers.

1.4 - Manufacturing quantum processors

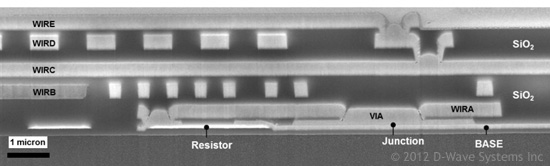
Figure 4 shows an image of the final chips after fabrication in a superconducting electronics foundry. The chips are 'stamped' onto a silicon wafer using techniques modified from the processes used to make semiconductor integrated circuits. There are several processors visible on this wafer image. The largest, near the bottom center, has 128 qubits connected together with 352 connection elements between them. The qubit/coupler circuits on each individual processor are the cross-hatched looking patches visible in this image. This is known as a Rainier processor and it is the type of processor found inside the D-Wave One.



*Figure 4. Photograph of a wafer of Rainier processors, including the 128-qubit processor used in the D-Wave OneTM QC system.*

It is important to try and use as much knowledge from the semiconductor industry as possible when fabricating large-scale integrated circuits, as through decades of testing, good parameters have been found to make integrated circuit yields good enough for the processors to be used commercially.

The techniques learned from the semiconductor industry have resulted in the construction of a Large-Scale Integration (LSI) fabrication capability owned by D-Wave in Santa Clara, USA (2006 - present). This fabrication capability is unique. Shown in Figure 5 is a cross section of one of the processors fabricated at D-Wave's superconducting electronics foundry. The fabrication process that has been developed is able to yield LSI (50,000+ Josephson junctions for Vesuvius) superconducting circuits. It is the only superconducting fabrication facility capable of yielding superconducting processors of this complexity. Fabrication yield is critical to improving processor performance and requires on-going significant investment, and in order to maintain historical qubit doubling rates, investments are being made to improve the capability into VLSI (1,000,000+ Josephson junction per processor) territory over the next five years.

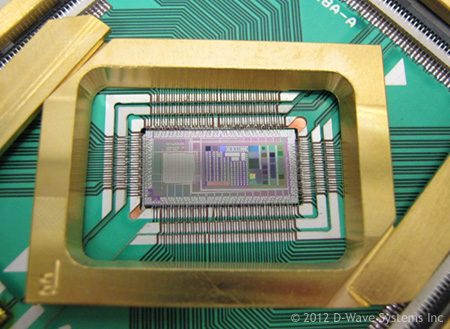


*Figure 5. A microscope cross-section of a D-Wave processor, fabricated using a 6-metal wiring layer process. The layer which is used to form the Josephson junctions is shown near the bottom of this sandwich structure.*

SECTION 2: Outside the processor

2.1 - The processor packaging

To build the quantum computer, one of these chips is selected from the wafer, and placed in the center of the processor packaging system, as shown in Figure 6. This image shows the chip area open, just after it has been wire bonded to connect it to the signal lines. It is possible to see the signal lines on the surrounding printed circuit board. There are far fewer incoming lines than there are programmable elements on the processor, which is made possible by additional circuitry - in the form of demultiplexers and signal routing and addressing - all implemented in superconducting logic circuitry on the chip.



*Figure 6. A photograph of the chip after being bonded to the circuit board which allows signals lines to be connected.*

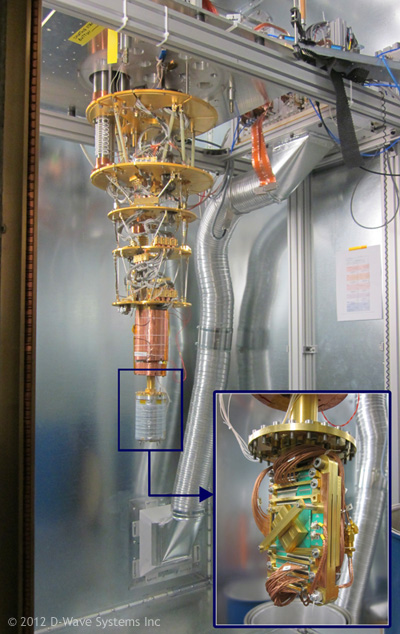
2.2 - Computer cooling

Reduction of the temperature of the computing environment below approximately 80mK is required for the processor to function, and generally performance increases as temperature is lowered - the lower the temperature, the better. 20mK is targeted as the lowest temperature that can be easily reached as an operating point. The processor and parts of the input/output (I/O) system, comprising roughly 10kg of material, must be cooled to these temperatures. Most of the physical volume of the current system is due to the large size of the refrigeration system. The refrigeration system used to cool the processors is known as a dilution refrigerator.

The inset in Figure 7 shows the chip packaging attached to the cooling apparatus. Note that the area around the chip has now been closed up to protect it from being damaged. When the computer is being operated, this part is sealed inside a vacuum chamber Because quantum processors require low temperatures for the quantum effects to be sustained, the entire piece shown in the inset of figure 7 is cooled to around 20mK, which is approximately 100 times colder than interstellar space.

To reach the near-absolute zero temperatures at which the system operates, the refrigerators use liquid Helium as a coolant. The type of refrigerator inside the D-Wave OneTM system is known as a "dry" dilution refrigerator. This means that all the liquid helium resides inside a closed cycle system, where it is recycled and recondensed using a pulse-tube technology. This makes them are suited to remote deployment, as there is no requirement for liquid helium replenishment on-site.

The specialized equipment to allow cooling to these temperatures is available commercially and runs reliably. The refrigeration technology is also mature enough that the system has a turnkey operation. The computer can be cooled down to operating temperature within several hours, and once this temperature is reached remain cold for months or years.

[](http://www.dwavesys.com/en/images/tut-hardware-system-assembly-big.jpg)

*Figure 7. (Click to view large version in new window) Schematic of the inside of the quantum computing system   
(inside its dedicated room) before the shielding and vacuum containers are attached. The chip packaging is housed   
inside the silver cylinder at the bottom of the assembly, as shown opened up in the insert.*

2.3 - Computer shielding and wiring

The I/O subsystem is responsible for passing information from the user to the processor and back. The signals are low frequency (<30MHz) analog currents, carried on metal lines, transitioning to superconducting lines at low temperatures. Key components of the I/O subsystem include the processor mount and wirebonding to it; low frequency bandpass filters for removing noise from the lines; room temperature electronics for converting signals coming from a front end server to analog currents; and the front end server which receives programming instructions from a user.

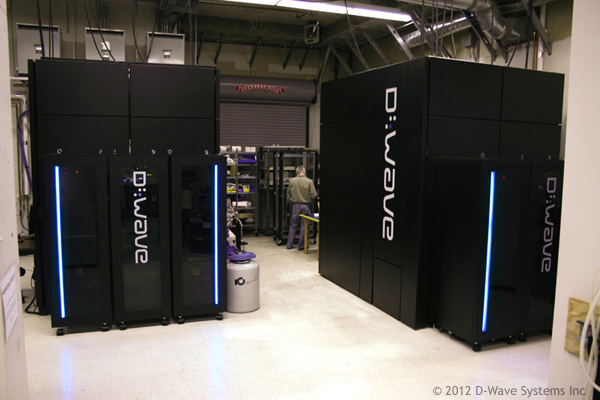
Nearly all aspects of the I/O subsystem are designed, manufactured, and tested by D-Wave in their Burnaby facility. Many of the specifications of the I/O system place unusual demands on the materials and processes involved. For example, much of the I/O subsystem must function at 20mK and be robust against multiple warming / cooling cycles between room temperature and base. Much of the subsystem must be made using superconducting metals, such as tin, which are typically non-standard for manufacturers. Additionally none of the materials close to the processor can be magnetic. To enforce this requirement, the company individually tests the magnetic character of every single component of each I/O subsystem at base temperature, and includes only those components that pass.

The current I/O subsystems provide 192 heavily filtered lines from room temperature to the processor, and are designed for optimal operation of a single 512-qubit Vesuvius processor. The D-Wave processor design is adversely affected by stray magnetic fields, and extreme care must be taken to exclude these. The current magnetic shielding system achieves fields less than 1 nanoTesla (nT) in three dimensions across the entire volume of the processor. This is achieved by a system comprising five concentric cylindrical shields, some of them high permeability metals and some of them superconducting. Integrated, on the processors, are magnetic sensors that measure the ambient field. Countering magnetic fields are applied that zero the field at these sensors. The temperature of the assembly is then slowly reduced, and the superconducting shields go superconducting, and 'lock' the zeroed field in place.

2.4 - Computer form factor

In addition to the magnetic shielding, the system sits inside a shielded room which screens out RF electromagnetic noise. The only path for signals between the inside and outside of the shielded room is a digital optical channel carrying programming information in, and results of computations out.

This shielded room doubles as the housing for the D-Wave OneTM system. Figure 8 shows a photograph of two D-Wave OneTM systems in their final, assembled form. The main black cube measures approximately 10'x10'x10', and there is an auxiliary cabinet where the server and control systems are housed. This footprint of the entire system is fairly large, being comparable in size to modern supercomputers which are housed in datacenter facilities. For this reason, the current mode of use of these systems is as a cloud computing resource.

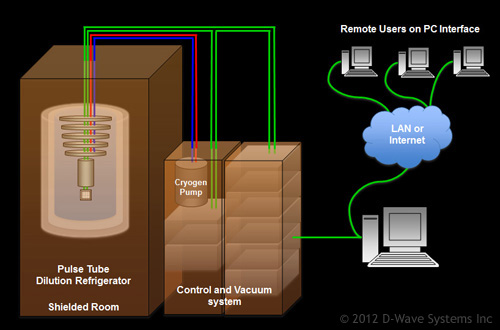


*Figure 8. Left: Photograph of two D-Wave OneTM Systems being tested in the lab.   
The large black cube unit houses the refrigerated and shielded quantum computing system, whilst the adjoining cabinet   
contains the control racks for the unit and the server for remote connection to the system.*

2.5 - Cloud based access

The way that the quantum processors are programmed is using a cloud-based model. This means that the systems can be programmed remotely from any location with an internet connection. Figure 9 shows an overview of how a user interacts with the system. Each D-Wave OneTM system has its own server. The system server handles the job queuing and scheduling, so that multiple users can access the system simultaneously. As more systems are brought online, this model will transition to the quantum processing ability being a co-processing resource that is accessible through (and used in conjuction with) classical cloud computing methods.

For more information on programming the system and the software side of quantum computing, please also read the [QC software document](http://www.dwavesys.com/en/dev-tutorial-software.html), which follows on from this hardware overview.

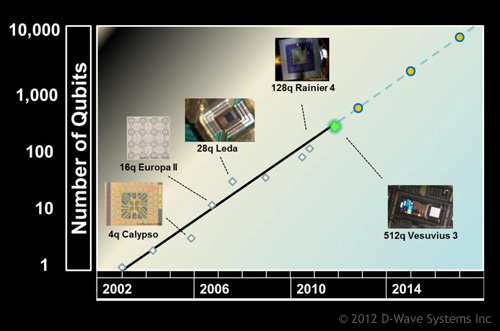


*Figure 9. Schematic of the system infrastructure and connection to LAN/internet*

SECTION 3

3.1 - The future of the hardware

For the past 8 years, the number of qubits on D-Wave's processors has been steadily doubling each year (see figure 10). This trend is expected to continue. To create processors with numbers of qubits up to around 10,000, the current fabrication process can simply be scaled to add more qubits in the same way that they are arranged currently. To go beyond ten-thousand into hundreds of thousands or millions of qubits will require major processor redesign, but there are certainly ways in which this can be achieved and it is not seen as a fundamental obstacle to improving the hardware.



*Figure 10. 'Rose's law' for quantum computing mimics the conventional 'Moore's law' paradigm   
seen in semiconductor processor development*

In addition to doubling the qubit count every 12 months, further improvements are planned, which include increasing the precision of the programmable elements, reducing the footprint of the entire system, reducing the energy consumption of the system, and the inclusion of parallel cores (i.e. multiple processors) within a single refrigeration unit.

  Quantum Computer Software

Understanding the software stack that has been built around the D-Wave OneTMquantum computing hardware

**Contents**

 [1.1 - Introduction](http://www.dwavesys.com/en/dev-tutorial-software.html#section1p1)

 [1.2 - Software stack components](http://www.dwavesys.com/en/dev-tutorial-software.html#section1p2)

 [2.1 - Coding at the BlackBox Compiler level](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p1)

 [2.2 - Key concept: The generating function](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p2)

 [2.3 - Crafting a generating function matched to your application](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p3)

 [2.4 - Adding the D-Wave OneTM System](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p4)

 [2.5 - Coding at the frameworks level](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p5)

 [2.6 - Summary and key take-aways](http://www.dwavesys.com/en/dev-tutorial-software.html#section2p6)

 [APPENDIX - BlackBox details](http://www.dwavesys.com/en/dev-tutorial-software.html#Appendix)

**Aim, audience and required background**

This material was developed to help those interested in programming the D-Wave OneTM System at a high level. This document introduces the basic ideas behind the software stack through which developers can program quantum computers. In order to understand the material here you may wish to read the [Quantum Computing Primer](http://www.dwavesys.com/en/dev-tutorial-intro.html) document first.

**What you will learn**

By following through this background reading material, you will learn:

 How te software stack built around quantum hardware is constructed

 Examples of programming at different levels of the software stack

 How to use the D-Wave OneTM System as a co-processor to a conventional computer in a scalable way

 How quantum computer software is changing and improving over time

SECTION 1

1.1 - Introducing the software stack

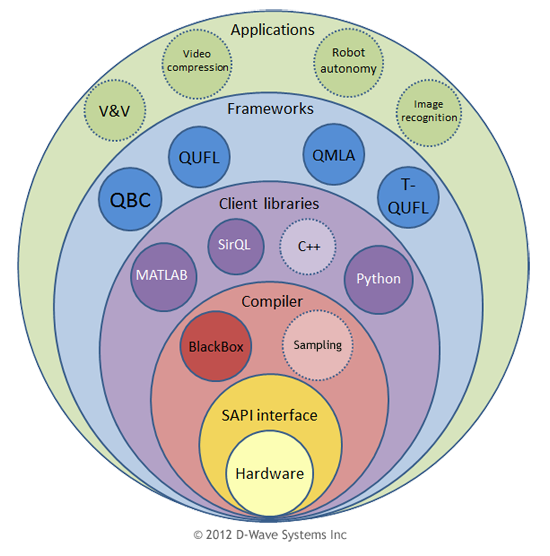
The D-Wave One is the world's first commercially available quantum computer system. The system is designed to tackle discrete optimization problems - a hard problem type which bottlenecks the development of many applications. If you can imagine a use of computers that is currently beyond our capabilities, it is likely that the hardness of discrete optimization problems is the main culprit.

While quantum computers are usually thought of as primarily hardware, this is mostly because of the immaturity of the field. Prior to the D-Wave One, there were no operational quantum computers, and therefore there was no reason to think too hard about software. The D-Wave One changed this, and opened up a whole new set of challenges and opportunities for software systems designed for quantum computers.

Any useful computing system must provide robust and general software access to the hardware's capabilities. The processors at the heart of the D-Wave One system are unlike any ever built, and therefore the software stack for connecting a programmer to the hardware had to be developed from scratch.

This document will describe our approach to developing the software architecture for the D-Wave One. The focus will be on the way that a user of the system can interact with the various levels of abstraction we've built.

The software stack which connects the hardware to a user is shown in Figure 1 on the following page. Each layer within the stack will be explained in turn.



*Figure 1. The software architecture for the D-Wave One. The dotted circles are elements which are planned for inclusion in future releases of the D-Wave developer packs.*

1.2 - Software Stack components

**Layer 1: Hardware**

The D-Wave One system contains a hardware component which is a large physical machine. The computer hardware (microprocessor and surrounding infrastructure) is housed in a 10'x10' shielded room. The current product system is sold complete with cooling and shielding equipment, and is fully calibrated and factory tested. The system has a turnkey operation and is ready to solve problems as soon as it is installed at a user's site. For more information on this layer, please see the dedicated [QC hardware tutorial](http://www.dwavesys.com/en/dev-tutorial-hardware.html).

**Layer 2: SAPI interface (machine code layer)**

This layer represents the low-level settings of qubit parameters; the 'machine code' of the quantum computer. The code is sent to the machine using a web interface; a local webserver parses the incoming requests and queues and sorts code automatically, so multiple users can be handled at the same time.

At this software level, the key difference between the D-Wave quantum computing system and a conventional computer is that the cloud-based interfacing is implemented within this layer; in a conventional machine it is done at a much higher layer. A conventional computing analogy to the way that D-Wave's stack is architected is that of a microprocessor being fed machine code over the internet. The fact that D-Wave's hardware incorporates a cloud-based methodology (regardless of the fact that it is done at a low level) means that the system is already architected to support being added to a high performance computing center or server farm.

Learning to program the machine at this level is very difficult, but it is upon this layer that algorithmic and software development is based. The end user would never normally program at this level unless a.) They wished to write a new kind of compiler which used the underlying quantum mechanics in a new way, in order to develop a language/programming framework that is not currently supported by the D-Wave software stack, or b.) The end user wishes to explore the system as a research tool; control at the machine code level can be used to explore fundamental physics or computer science experiments.

**Layer 3: Compiler layer**

This layer allows full access to the optimization power of the hardware without the user needing to know anything about the underlying hardware architecture or the physics of the machine. A programmer using this level of abstraction only needs to provide a function to be optimized, where the function returns a real number given a bit string. This function can literally be anything. We call this the compiler layer because abstracts away the intricate details of the computational hardware from the programmer, (which allows thinking in more general terms about problem solving), although it is not exactly analogous to a conventional classical compiler.

**Layer 4: Client library wrappers**

For those wishing to develop applications, this layer of software abstraction is usually the most natural starting point. This layer provides the ability to use standard high level programming languages to access the underlying parts of the software system. In many of the applications developed to date for the D-Wave One, conventional high performance computing systems are required to compute the values of the function requiring optimization. This layer supports a natural synergy between conventional supercomputers and the D-Wave One, where the workload for 'big data' problems can naturally be subdivided between a conventional supercomputing system and the D-Wave One.

**Layer 5: Frameworks**

The modes of use layer allows the creation of a much higher level code base consisting of code which uses optimization at its core but may have complicated algorithmic steps wrapped around this central hard problem. Modes of use are implementations of algorithms (such as those found in the area of machine learning) that can be applied to many different applications areas. The layer allows the construction of complex methods for attacking problems. Examples of modes of use in the D-Wave One software stack include supervised binary classification, supervised multiple label assignment, and unsupervised feature learning, which have been neatly bundled into libraries of commonly used functions to form programming frameworks that use quantum mechanics under the hood.

**Layer 6: Applications layer**

This is the level where an end-user (customer, client, etc.) would interact with quantum applications, which have been developed using the lower layers.

Examples of applications that have been coded using the D-Wave OneTM quantum computer include: Binary classification for object detection in images and polarity labeling of movie review text; correlating text sentiment extracted from news feeds with stock market prices; video compression; lattice protein folding; and assignment of category labels to images, blog posts and news stories.

SECTION 2

2.1 - Coding at the BlackBox Compiler level

As with conventional processors and computing systems, it is possible to develop on a D-Wave OneTM System by programming directly in the machine language of the processor. D-Wave's developer tools, up to and including devpack 1.3, provided this level as the only option available.

In devpack 1.4, we have released a new, higher level programming interface to the D-Wave OneTM System which is called the D-Wave BlackBox compiler - BlackBox for short. Under the hood, the D-Wave OneTM System is an extremely exotic and complex machine. Programming at the machine language level is extremely difficult, even for our own internal developers. Abstracting away from this underlying complexity is critical for making the system broadly accessible.

The BlackBox compiler framework simplifies the ease of use of the D-Wave OneTM System, and dramatically extends its possible range of uses. Programming using BlackBox is so easy, users can be up and running problems on it within minutes. This is the level at which nearly all programmers interact with the D-Wave OneTMsystem. Here is how it works.

2.2 - Key concept: the generating function

First, a developer codes a function which takes as input a string of bits - zeroes and ones - and outputs a real number. We'll call the function provided by the developer the generating function, or G(x). Many of the applications-level programming examples in the [Tutorials](http://www.dwavesys.com/en/dev-tutorials.html) section involves taking a real world problem and crafting it into a generating function. Mathematically we write the generating function like this:

*G*(*x*1,*x*2,...,*xN*)

where the *xk* variables (there are a total of N of these) are binary (0 or 1) variables. The value returned by the function *G*(*x*1,*x*2,...,*xN*) has to be a real number.

Let's take a look at a simple example:

*G*(*x*1,*x*2)=*x*1+2*x*2

In this case, we have N=2 variables, and the function *G*(*x*1,*x*2) evaluates to real numbers (well actually integers, but that's OK too!). Specifically, plugging in the 2N = 4 possible inputs, these are *G*(0,0)=0, *G*(1,0)=1, *G*(0,1)=2, and *G*(1,1)=3.

This particular example is really simple. However you as a developer can make G as gnarly and complicated as you want, as long as it takes as input binary variables and outputs real numbers. For example it could be something bizarre like:

*G*(*x*1,*x*2)=2(*x*1/(1+*x*2))*exp*[*sin*(*x*1)]*x*1*x*2

However it doesn't even need to arise from a closed form mathematical expression. For example, it could be related to the outcome of a series of settings of control switches on a complex system, such as a flight control system for an airplane.

This function can be coded up using any programming language (here we will use Python), and will be evaluated entirely using a conventional computing system. For some problems, the conventional computing system you'd want to use to do this is a high performance computing system, such as a cluster, a supercomputer or a cloud-based solution; however it could just as easily be your laptop - it's up to you, and depends on how hard it is to evaluate the value of the generating function given an input.

2.3 - Crafting a generating function matched to your application

**Overview**

A developer chooses *G*(*x*1,*x*2,...,*xN*) so that the lower the real number returned by the function, the 'better' the input bit string (*x*1,*x*2,...,*xN*) was. The input bit string represents a 'potential solution', and the number returned by the function given that input gives a measure of the goodness of the potential solution - the lower *G*(*x*1,*x*2,...,*xN*) is, the better the potential solution (*x*1,*x*2,...,*xN*) is.

Let's take a look at our first example, where *G*(*x*1,*x*2)=*x*1+2*x*2. In this case if we look at the returned values for all possible input combinations *G*(0,0)=0, *G*(1,0)=1, *G*(0,1)=2, and *G*(1,1)=3, we see that the choice of *G*(*x*1=0,*x*2=0)=0 gives the lowest (and therefore 'best') value of G. The generating function is sometimes called an optimization objective function, although as a developer you can use the output of BlackBox for more than just optimization (more on this shortly).

So where does G come from in the first place? It comes from your needs as an applications developer. If you are building an application where the above concept is useful (or even critical) to your work, the D Wave One provides a new tool to attack that problem. This is a little abstract, so we'll now provide a couple of examples to help illustrate the process of 'crafting a generating function'.

**Example 1: The Subset Sum Problem**

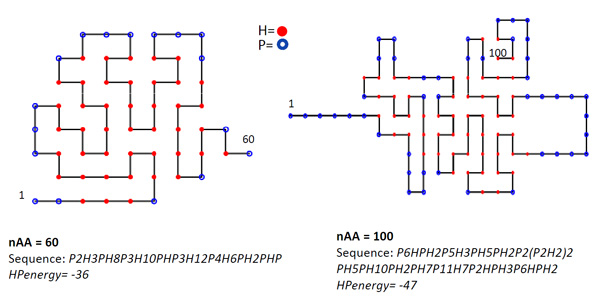
For example, let's say that you wanted to solve the Subset Sum Problem (SSP). In this problem, you are given a set of integers, and you want to determine whether there is a non-empty subset whose sum is zero. For example, the set might be −7,−3,−2,5,8 - this is the instance provided in the Wikipedia page on SSP. In this case the answer is 'yes', because the subset {−3,−2,5} sums to zero. As an applications developer, you could code up the SSP in the following way (note that there are many ways you could try to do this). Define five variables (*x*1,*x*2,*x*3,*x*4,*x*5) where if a variable is equal to 1 it means "please include me in the sum". In this case the generating function for the above example could be

*G*(*x*1,*x*2,*x*3,*x*4,*x*5)=(−7*x*1−3*x*2−2*x*3+5*x*4+8*x*5)2+∏(*k*=1)5(1−*xk*)

where the values of the coefficients are just the values in the original set. The first term fixes the minimum possible value of that part of G to be zero, and the second term penalizes the solution with all variables equal to zero (which otherwise would show up as an allowed solution!) but not any of the others. If we can find a configuration of variables for which *G*(*x*1,*x*2,*x*3,*x*4,*x*5)=0 we've got a solution to our SSP instance. This subset sum example is the focus of the hands-on[Getting Started](http://www.dwavesys.com/en/dev-tutorial-getting-started.html) tutorial, so you can follow along with coding up this problem to help further understand the process.

**Example 2: Lattice protein folding**

Another example is using the compiler to write a protein folding application. The user would again need to craft a function which somehow encoding a 'fold' into a bitstring. For example, in a simple 2D lattice protein folding model, for each segment of the protein one could encode a fold to the left as 01, a fold to the right as 10 and no fold to 11. A long bit string could therefore contain a complicated series of folds. (Figure 2)



*Figure 2. To build a lattice protein folding application, all that is required is a function that returns the energy of a suggested fold - the D-Wave One will then find the fold that gives the lowest energy. Here is an example of a third-party implementation of the two-dimensional HP lattice protein folding model built using the D-Wave software tools.*

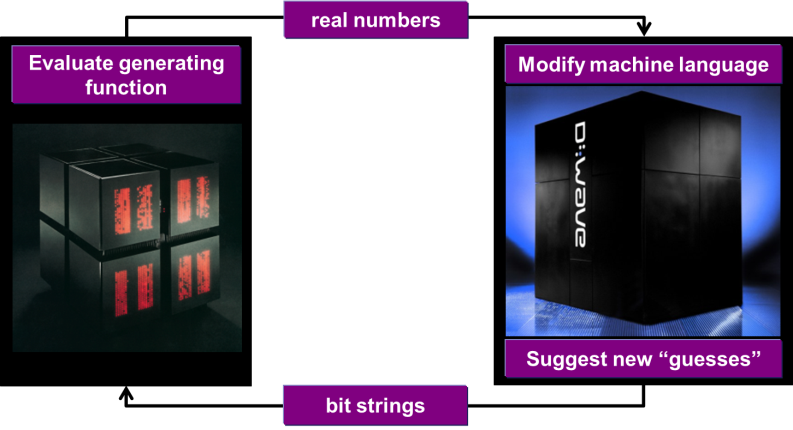
Given a particular fold, one must be able to compute the energy of this fold (how 'good' the fold is). The user would write a function that was able to take in a bit string corresponding to a fold, and return an energy value. Once this function has been written, it is passed as an argument into the BlackBox function in the SynDist layer. This layer will again repeatedly call the lower layers to send the relevant problem data to the hardware, and return the best set of folds that it finds, in terms of the 01,10,11 encoding scheme.

2.4 - Adding the D-Wave OneTM System

Up to this point everything has been done in a conventional framework, and all we've done is set up some machinery defining a function. Now we add the D-Wave OneTM System in the following way.

Imagine the entire computing system as consisting of two complimentary parts - think of these as the 'right brain' and 'left brain' of a hybrid computing system (see Figure 1 below). The 'left brain' is a conventional computing system whose rational, tedious job it is to compute the value of the generating function given 'guesses' at potential solutions. In general this involves a large amount of computation of the sort conventional computing systems excel at. The 'right brain' is the D-Wave One, which suggests 'creative' potential solutions, using the results obtained by the conventional computing system to quickly hone in on better and better solutions.

This split allows hybrid systems to be built that can deal with enormous amounts of data and extremely complex generating functions. The information that passes between the conventional computing system and the D-Wave OneTM System is extremely small - it's just the bit strings coming from the system representing its guesses at good answers flowing from the D-Wave OneTM System to the conventional system, and the real numbers characterizing how good those guesses were, flowing from the conventional system to the D-Wave OneTM System. The amount of data that might be required to compute the value of the generating function could be (and often is) enormous - but we can use all of the standard tactics for dealing with this using conventional systems architecture.



*Figure 3. The BlackBox programming paradigm separates the evaluation of the generating function, and the process of generating potential solutions. The evaluation of the function happens in a conventional computing system, while the solution generation happens in the D-Wave One. Information flow between the two is extremely low bandwidth and is restricted to bit strings representing potential solutions flowing from the D-Wave OneTM System to the conventional system, and real numbers representing the values of the generating function evaluated for those guesses flowing from the conventional system to the D-Wave OneTM System.*

Using this hybrid system is simple. The developer provides the generating function, initiates the computation, and then the system starts 'thinking' about the generating function in the following, iterative way:

 1 - First, a series of random solutions are generated by the conventional computing system.

 2 - The quality of these guesses is evaluated by passing them into the generating function.

 3 - The real numbers characterizing the goodness of the solutions are sent to the D-Wave OneTM System.

 4 - The D-Wave OneTM System automatically adjusts itself based on this feedback, and then generates a series of guesses, informed by the results received.

 5 - These new guesses are sent back to the conventional system, where their goodness is evaluated.

 6 - Steps 3-5 are repeated until exit criteria are met.

When the D-Wave OneTM System generates a series of guesses at Step 4, it is doing something rather complicated. Here we won't go into any detail as to specifically what's going on in this step, but we'll outline the process. The following section is rather technical; you can skip it if you'd just like to start using BlackBox.

2.5 - Coding at the Frameworks level

As quantum computer programming is a relatively new art, a lot of the programming is currently done at the BlackBox (compiler) level. However, even though this is much easier than coding at the machine code level, this is still a long way from the high level applications based coding that is performed with classical digital computers today. To code applications, more layers of abstraction are needed.

For a programmer to use a D-Wave One at the frameworks level, they would import a library such as dwave\_ssfl, which contains a bunch of functions which are useful for feature learning (a commonly performed operation in machine learning applications). These functions would already have built into them concepts such as the objective functions described in the BlackBox section above, and the user would only have to supply the data over which they wanted to perform the learning task, learn how to use the SSFL library, and set a few parameters. Such libraries are being developed currently, and will be available for use soon.

2.6 - Summary and key take-aways

The D-Wave One software stack has been architected to allow developers to build code at several different levels of abstraction from hardware. From directly coding at the machine code level, all the way to high level frameworks, there is a lot of flexibility available.

The key take-away is that the software stack ensures that the end user need not have any understanding of the underlying hardware or the physics of the computer to be able to use D-Wave's technology to build powerful applications.

There is no requirement for a developer to understand or care about the details of the functioning of the processor at the machine code level - sophisticated industrial applications can, and in most cases should, be constructed using D-Wave tools that abstract away the underlying details.

For most applications, the best place to start prototyping new ideas and developing code is at the level of the compiler (through the client libraries). As this is a key concept, most of the applications programming examples featured on the [Tutorials](http://www.dwavesys.com/en/dev-tutorials.html) page are dedicated to showcaseing many examples of how to do this.

APPENDIX - BlackBox details

Probability distributions

In order to give some intuition for what the D-Wave OneTM System is doing in Step 4, we have to take a brief detour into statistical physics.

In physics, often a physical system is described by writing down a function whose inputs can be thought of as specific physical states of the system, and whose outputs are the energies of those states.

If the physical system of interest (call it the "central system") is in a state called thermal equilibrium, the probability of being in any particular state has a known functional form called the Boltzmann distribution. This distribution has the form

*P*(*x*1,*x*2,...,*xN*)=1*Zexp*[−*G*(*x*1,*x*2,...,*xN*)/*kT*]

which means, "if the system whose allowed energies are given by *G*(*x*1,*x*2,...,*xN*) is in thermal equilibrium at temperature T, the probability of the system being in state (*x*1,*x*2,...,*xN*) is *P*(*x*1,*x*2,...,*xN*)". Here k is the Boltzmann constant (don't worry too much about its specific value, it won't matter for us here) and Z is something called the partition function, which is

*Z*=Σ(*k*=1)*N*Σ(*xk*=0,1)*exp*[−*G*(*x*1,*x*2,...,*xN*)/*kT*]

Note that actually calculating *P*(*x*1,*x*2,...,*xN*) given *G*(*x*1,*x*2,...,*xN*) is not easy at all. But let's not worry about that right now. Let's just imagine that we had a physical system where the probabilities of measuring the bits (*x*1,*x*2,...,*xN*) were somehow magically given by the Boltzmann distribution. What would this look like, and what could we do with it?

The algorithms underlying the BlackBox procedure attempt to sculpt the probability distribution returned by the D-Wave OneTM System to be close to the Boltzmann distribution you'd get over the generating function. As the iteration proceeds, the probability distribution changes and, if everything goes well, starts converging to something close to the Boltzmann distribution described in the previous section.

What BlackBox returns is the best solutions found during this iterative procedure. These solutions can be used in two modes.

Often a developer is looking for the best solution found. In this optimization mode, the output of this procedure is the bit string BlackBox found that returned the lowest value of the generating function.

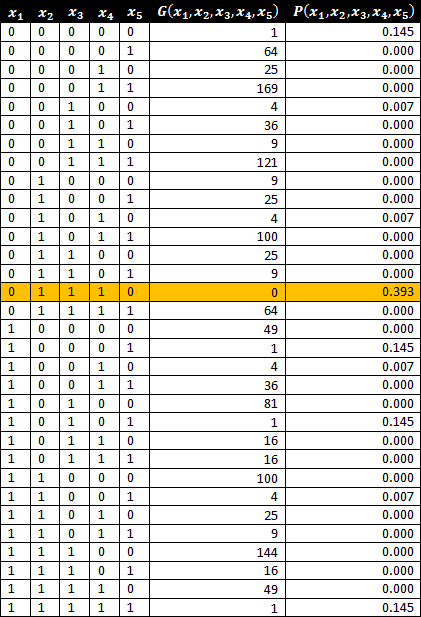
Sometimes a larger set of samples is desirable; we call this the sampling mode. For example, if the user wants to generate not only the best solution, but a set of 'good' alternate solutions, the system natively provides this capability - the user records the guesses generated at Step 4, the number of times these guesses were generated, and the quality of the guesses. BlackBox can be configured to return a set of the best answers found, and how many times each was seen.

**Probabilities and energies in our Subset Sum Problem example**

Let's take the specific example of the Subset Sum Problem instance we used earlier:

*G*(*x*1,*x*2,*x*3,*x*4,*x*5)=(−7*x*1−3*x*2−2*x*3+5*x*4+8*x*5)2+∏(*k*=1)5(1−*xk*)

Because we only have five bits here, we can explicitly write down all of the 32 terms we need to compute all of the terms in the Boltzmann distribution. It's a little tedious, but let's do it anyway!



*Figure 2. Enumerating the function.*

Notice that there is a single solution (highlighted in orange) that gives *G*(*x*1,*x*2,*x*3,*x*4,*x*5)=0: the solution with (*x*1=0,*x*2=1,*x*3=1,*x*4=1,*x*5=0).

Let's assume that the probability distribution over these results is a Boltzmann, and let's for the moment just arbitrarily set kT=1. We can then compute the actual probabilities for each state. These are the values in the rightmost column.

Now every time we measure the state of the system, we get one of the 32 states above with the probabilities in the righthand column (as long as the system is in thermal equilibrium at kT=1). If we wanted to find a solution to the Subset Sum Problem, given a physical system with this probability distribution, we would repeatedly sample from the probability distribution until we either found a solution or got bored of drawing samples. In this case, we see that there is a *PSS*=39.3 (SS stands for "single shot") chance of drawing the interesting state (*x*1=0,*x*2=1,*x*3=1,*x*4=1,*x*5=0) per sample.

So if we draw a total of K samples, the probability of seeing the state we want at least once is

*P*=1−(1−*PSS*)*K*

So if we drew say K=10 samples in this case, the probability of seeing the state we want at least once is

*P*=1−(1−0.393)10=99.3