Introduction to Machine Learning and Quantum Computing

CHEM 584: Lecture Notes

Prof. Victor S. Batista

SCL 18: Tuesdays and Thursdays 9:00 am – 10:15 am

Yale University - Department of Chemistry

Contents

1 Syllabus

Machine learning and quantum computing have emerged as leading technologies of the twenty-first century and are expected to be increasingly applied to address a wide variety of chemical and materials science challenges. The goal of this course is to introduce fundamental concepts of machine learning and quantum computing to chemists and materials science students through an overview of algorithms, computational methods, and applications. It is intended to empower students to engage with this emerging field and foster the growing field of artificial intelligence for accelerated scientific discoveries in the molecular and physical sciences.

Textbooks. Recommended textbooks for this class are:

R1:"Pattern Recognition and Machine Learning" by Christopher M. Bishop (Springer, 2006). (pdf) (matlab)

R2: "Deep Learning" by Ian Goodfellow, Yoshua Bengio and Aaron Courville. (pdf) (github).

R3: "Deep Learning for Coders with Fastai and PyTorch: AI Applications Without a PhD" by Jeremy Howard and Sylvain Gugger. (github)

R4: "Dive into Deep Learning" by Jeremy Howard and Sylvain Gugger. (pdf)

R5: "Quantum Computation and Quantum Information" by Michael A Nielsen and Isaac L. Chuang (Cambridge).

R6: "An Introduction to Quantum Computing" by Phillip Kaye, Raymond Laflamme and Michele Mosca (Oxford University Press). (pdf)

R7: "Learn Quantum Computation Using Qiskit" and notebook.

Our lecture notes will be updated according to the pace of the course and suggestions from the students. References to the textbooks listed above are indicated in the notes as follows: [R1(190)] indicates “from Reference 1, Page 190”.

Pytorch Tutorials and Documentation: Pytorch tutorials and Pytorch documentation will be essential for actual implementations, complemented with tutorials for understanding and implementing sequence-to-sequence (seq2seq) models .

TensorFlow Tutorials and Documentation: The TensorFlow tutorials are written as Jupyter notebooks and run directly in Google Colab.

Scikit-learn Tutorials and Documentation: Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

Qutip Tutorials and Documentation:

Qutip tutorials will be useful for simulations of light in cavities for quantum information.

Contact Information
Office hours will be held by zoom (Monday, 5:00pm). Zoom ID: 943 8610 8716, Passcode: victor

Grading: There will be no final exam for this class. The final grading evaluation is the same for both undergraduate and graduate students: homework (40%), two mid-terms (40%) on 2/22 and 3/22, and a final project (20 %). Homework will be assigned during lectures and also through Yale canvas. Computer assignments given on Thursdays will be due the following Thursday 9:00 am, and will be assisted for questions during office hours on Monday.

2 Feedforward Neural Networks

Figure (3) shows a simple example of a model neural network (NN), a so-called two-layer *feedforward* NN based on 3 lists of numbers $\mathbf{x} = [x_0, x_1, \cdots x_D]$, $\mathbf{z} = [z_0, z_1, \cdots, z_M]$, and $\mathbf{y} = [y_1, \cdots y_k]$ and two layers of adjustable parameters $\omega_{jk}^{(1)}$ and $\omega_{jk}^{(2)}$. The $\mathbf{x}$ values are inputs that define the values $\mathbf{z}$ in the so-called *hidden layer* of 'neurons', as follows:

$$z_j = \phi_1 \left(\sum_{k=0}^{D} \omega_{jk}^{(1)} x_k \right). \tag{1}$$

Here, $\phi_1(r)$ is the so-called *activation function* that could be linear (*e.g.*, $\phi_1(r) = r$) or non-linear (*e.g.*, $\phi_1(r) = r$, for $r \geq 0$ and $\phi_1(r) = 0$ for $r < 0$), as discussed further in Sec. 2.3. The outputs $\mathbf{y}$ are computed analogously,

$$y_j = \phi_2 \left(\sum_{k=0}^{M} \omega_{jk}^{(2)} z_k \right). \tag{2}$$

The resulting NN can then be parametrized by adjusting the values of the weights $\omega_{jk}^{(1)}$ and $\omega_{jk}^{(2)}$, with $x_0 = z_0 = 1$, so that for any given input $[x_1, \cdots, x_D]$ we can predict the corresponding output $[y_1, \cdots, y_k]$. The weight $\omega_{j0}^{(l)}$ is often called the *bias* of layer l.

Examples for molecular systems could have inputs $\mathbf{x}$ defined in terms of the atomic coordinates of a molecule while the outputs $\mathbf{y}$ could be the potential energy, $y_1 = V(\mathbf{x})$, and forces $y_k = F_k(\mathbf{x})$ acting on each atom (as in typical force-fields employed in molecular dynamics, or Monte Carlo simulations). Another NN could have inputs x_k defined in terms of Hammett parameters for the substitutional groups of a molecule, and the outputs x_k could be the energies of the frontier orbitals $y_1 = E_{HOMO}$, $y_2 = E_{LUMO}$ of that molecule. Alternatively, the NN could have input numbers x_k defining the name of a molecule, or the primary sequence of a protein, according to some dictionary or catalog while the outputs y_k could be the properties of the molecule (*e.g.,* solubility, molecular weight, toxicity, NMR chemical shifts).

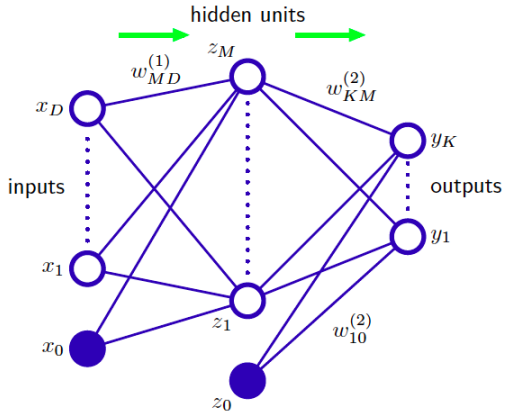


Figure 1: Network diagram for a two-layer feedforward neural network. The inputs x_j, hidden units z_j, and output variables y_j are represented by nodes (artificial neurons), and the links between the nodes represent the weight parameters ω_{jk}. The neurons $x_0 = z_0 = 1$ are kept constant and linked to other neurons with weights $\omega_{j,0}^2$ and $\omega_{j,0}^1$ called bias parameters. Arrows denote the direction of information flow through the network during forward propagation [R1(228)].

A simple way to parametrize the NN shown in Fig. (3) is to adjust the values of $\omega_{jk}^{(1)}$ and $\omega_{jk}^{(2)}$ to minimize the error (often called the *loss*) between the predicted values y_j and the correct values of outputs for a so-called *training set* of inputs for which the outputs $y_j^{(a)}$ are known,

$$\epsilon = \sum_{j=1}^{K} (y_j - y_j^{(a)})^2, \tag{3}$$

where y_j are the calculated values of the outputs, and $y_j^{(a)}$ are the correct values (or labels) corresponding to a given input training set. Note that ϵ is a function of the parameters weights $\omega_{jk}^{(1)}$ and $\omega_{jk}^{(2)}$ since, according to Eqs. (1) and (2),

$$y_j = \phi_2\left(\sum_{k'=0}^{M} \omega_{jk'}^{(2)} \phi_1\left(\sum_{i=0}^{D} \omega_{k'i}^{(1)} x_i\right)\right). \tag{4}$$

So, we can minimize ϵ by adjusting the parameters $\omega_{jk}^{(l)}$, as described below in Sec. 2.1.

2.1 Gradient Descent

The parameters of NNs are typically adjusted by using the gradient descent method, illustrated in Fig. (2). To visualize the minimization process, we consider that the weights $\omega_{jk}^{(l)}$ are functions of time t along the optimization process that advances with small integration time steps τ. Therefore, the loss ϵ decreases along the optimization time by evolving with values $\epsilon(0), \epsilon(\tau), \epsilon(2\tau), \cdots, \epsilon(N\tau)$, with $N\tau$ the total optimization time.

Given an initial set of randomly assigned values, the adjustable parameters $\omega_{jk}^{(l)}$ are updated along the direction of minus the gradient of the loss with respect to the parameters,

$$\frac{\partial \omega_{jk}^{(l)}}{\partial t} = -\frac{\partial \epsilon}{\partial \omega_{jk}^{(l)}}, \tag{5}$$

where s parametrizes the evolution of parameters, as follows:

$$\omega_{jk}^{(l)}(t+\tau) = \omega_{jk}^{(l)}(t) - \frac{\partial \epsilon}{\partial \omega_{jk}^{(l)}}\tau, \tag{6}$$

with the *learning rate* parameter τ defined as a small positive number.

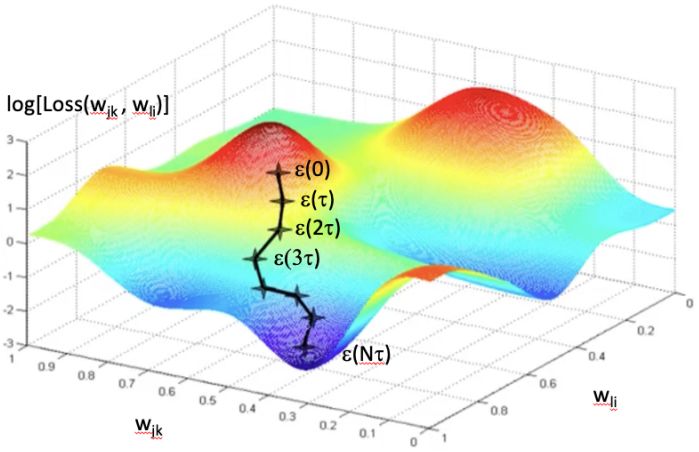


Figure 2: Minimization of the loss by gradient descent.

Such a choice of gradients ensures that ϵ decreases monotonically since

$$\begin{aligned} \epsilon(t+\tau) &= \epsilon(t) + \sum_{jkl} \frac{\partial \epsilon}{\partial \omega_{jk}^{(l)}} \frac{\partial \omega_{jk}^{(l)}}{\partial t}\tau, \\ &= \epsilon(t) - \sum_{jkl} \left|\frac{\partial \omega_{jk}^{(l)}}{\partial t}\right|^2 \tau. \end{aligned} \tag{7}$$

2.1.1 Stochastic Gradient Descent

The minimization of the error between the predicted and correct values of the outputs, for a training set of K inputs requires the evolution of the weights $\omega_{ik}^{(l)}$, according to Eqs. (6) and (3), as follows:

$$\begin{aligned} \omega_{ik}^{(l)}(t+\tau) &= \omega_{ik}^{(l)}(t) - \frac{\partial \epsilon}{\partial \omega_{ik}^{(l)}}\tau, \\ &= \omega_{ik}^{(l)}(t) - \sum_{j=1}^{K} \frac{\partial (y_j - y_j^{(a)})^2}{\partial \omega_{ik}^{(l)}}\tau, \end{aligned} \tag{8}$$

which is the so-called *standard* (or 'batch') gradient descent, corresponding to the batch of K outputs. However, when K is very large, evaluating the sums of gradients becomes very expensive.

The *stochastic gradient descent* (SGD) reduces the computational cost at every iteration by randomly sampling a subset of j output values at every step (effectively applying dropout on the outputs). Therefore, the key difference compared to standard gradient descent is that only a portion of the output data is used to approximate the gradient of the loss, and that portion is picked randomly at each step.

2.1.2 Exercise: Gradient Descent

To illustrate the gradient descent method as applied to optimize the parameters of a model, optimize the linear coefficients of a third-order polynomial $y_{pred}(x) = \sum_{j=0}^{3} c_j x^j$, to approximate the output $y(x) = \sin(x)$ for an input of 2000 equally spaced values of x in the range $x = [-\pi, \pi]$.

Go through the Pytorch tutorials learning with examples and see how to formulate the linear regression problem in terms of the parametrization of a 2-layer feedforward linear NN.

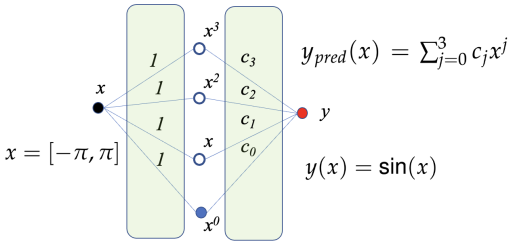


Figure 3: Diagram of a two-layer feedforward neural network for linear regression of $sin(x)$ as a third-order polynomial.

Solution: Download the Jupyter notebook for the solution from Exgradientdescent.ipynb where you can have a first exposure to the concepts that we will discuss in subsequent sections and the basic aspects of Pytorch, including:

- The concept of a PyTorch tensor
- How to define a neural network with learnable parameters (or weights).
- Forward propagation: How to process the input through the network
- How to compute the loss (how far the output is from being correct)
- Backward propagation: How to compute the gradients of the loss w.r.t. the parameters of the network.
- How to update the weights of the network, typically using a simple update rule: weight = weight - learning_rate.

2.2 Colab, Python, Tensorflow, Keras and Pytorch

Several software libraries and computational packages are currently available for high level implementations of machine learning methods, including pytorch, torch , tensorflow, and keras. A comparison of these three popular deep learning frameworks can be found here. Pytorch is the newer framework (based on Torch and open-sourced on GitHub in 2017 by Facebook's AI research group). Pytorch's popularity is rapidly growing among AI researchers due to simplicity, flexibility, efficient memory usage, speed, and dynamic computational graphs.

For this class, I recommend working in the Google Colab environment, so you can run your codes in Google's computers. In Colab, everything you need is already installed, or you can upload by mounting your Google drive as shown in the Navigating_tutorial.ipynb.zip Jupyter notebook for which you need to have the following csv file in the same folder. A brief (and fun) tutorial on how to work with RDKit at Google Colab is available at the RDKitEH.ipynb.zip notebook.

These notebooks also introduce the SMILES notation for representation of molecules in terms of strings of characters, which can be converted into lists of numbers to input molecules into neural networks.

The SMILEs notation follows the following rules (examples shown in Fig. 4):

1. Atoms are represented by their atomic symbols.
2. Hydrogen atoms are omitted (are implicit).
3. Neighboring atoms are represented next to each other.
4. Double bonds are represented by '=', triple bonds by '#'.
5. Branches are represented by parentheses.
6. Rings are represented by allocating digits to the two connecting ring atoms.
7. Aromatic rings are indicated by lowercase letters.

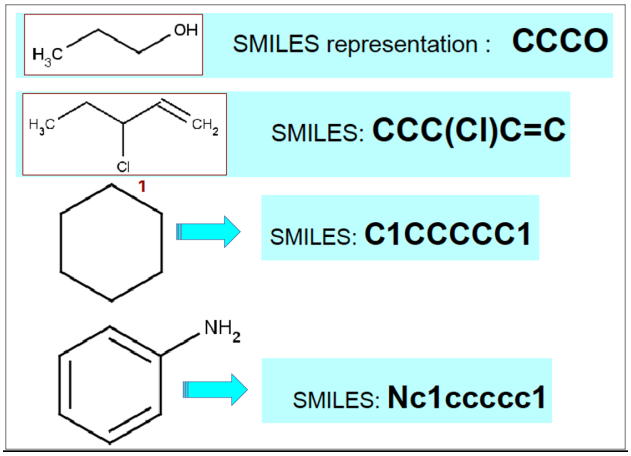


Figure 4: Examples of SMILES representations of molecules.

Molecules can also be represented by features defined according to the type of atoms and their corresponding neighborhoods in the molecular structure, using the extended-connectivity fingerprints (ECFPs), also known as *circular fingerprints* available at DeepChem. The main properties of ECFPs are that (i) they are defined by considering circular atom neighborhoods (Fig. 16, middle panel, Sec. 5); (ii) they are rapidly calculated; (iii) they represent substructures; (iv) they can account for a huge number of different molecular features (including stereochemical information); and (v) they represent both the presence and absence of functionality.

I also recommend brushing up your python knowledge with the excellent tutorial provided in Sec. 22, adapted by Kevin Zakka for the Spring 2020 edition of cs231n, and available at Python_tt.ipynb.

2.3 Activation Functions

Figure 5 shows some of the most commonly used activation functions which are essential building blocks of neural networks.

2.3.1 Linear Activation

The so-called 'Adaline' activation, or linear regression function, produces only linear models (*i.e.*, model NN without hidden layers), since the output is always a simple linear combination of the input data,

$$\begin{aligned} y_j &= \sum_{i=0}^{M} \omega_{ji}^{(2)} \sum_{i=0}^{D} \omega_{ik}^{(1)} x_k \\ &= \sum_{k=0}^{D} \left(\sum_{i=0}^{M} \omega_{ji}^{(2)} \omega_{ik}^{(1)} \right) x_k \end{aligned} \tag{9}$$

so even models with multiple hidden layers with linear activation functions are equivalent to models without hidden layers. Therefore, to go beyond simple linear regression models, it is imperative to include non-linear activation functions (*i.e.*, functions $\phi(z)$ with non-constant derivatives).

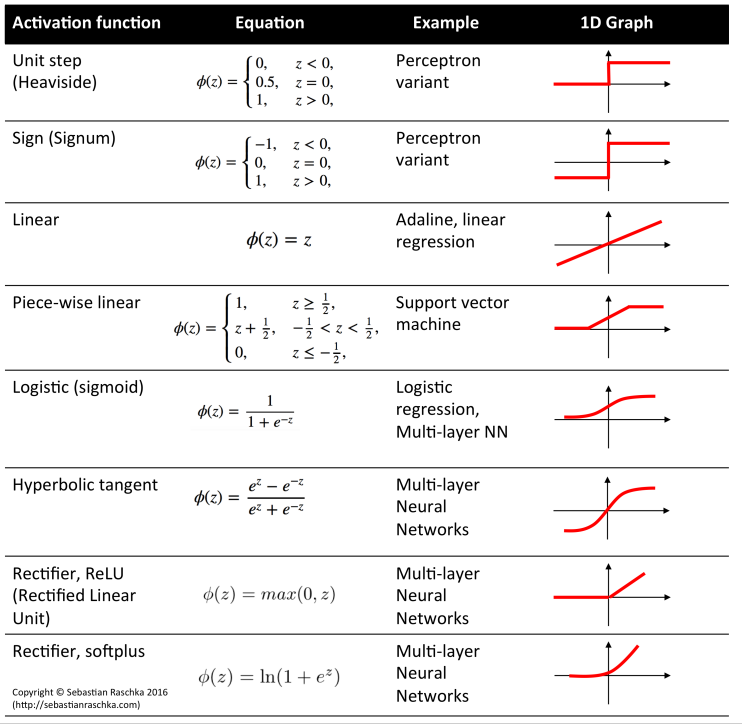


Figure 5: Activation Functions for Artificial Neural Networks [Copyright ©2014-2020 Sebastian Raschka].

2.3.2 Non-linear Activation

Two representative examples of non-linear activation functions, listed in Fig. 5 are the Logistic sigmoid and the ReLU functions. The main difference between these two activation functions is that the derivative of the sigmoid $\phi(z) = 1/(1 + e^{-z})$ is smaller than 1, in fact $\phi'(z) \leq 1/4$ since $\phi'(z) = e^{-z}/(1 + e^{-z})^2$, while the derivative of the ReLU is always equal to 1 when $z > 0$. As we discuss later in Sec. 2.3.3, activation functions with $\phi'(z) < 1$ can be problematic since they lead to the so-called *vanishing gradient* problem. As shown in Sec. 2.3.3, the gradients of the loss with respect to the weights are proportional to the product of the derivatives of the activation functions, and multiplying derivatives of activation functions that are < 1 leads to gradients of the loss that approximately equal to zero. So, the parameters of the NN cannot be adjusted and the NN cannot be trained. A solution to that problem has been the implementation of the ReLU, $\phi(z) = \max(0, z)$, and Leaky ReLU, $\phi(z) = \max(0.01\, z, z)$, activation functions that overcome the vanishing gradient problem since their derivatives are always equal to 1 when $z > 0$. So, the ReLU function has enabled training of NNs with many layers and thus the emergence of the field of deep-learning briefly described for image recognition in the following Youtube. The softplus function is essentially a smooth ReLU activation (*i.e.*, with continuous gradients).

An important observation is that non-linear activation functions, such as ReLU, introduce correlations between different inputs since all of the inputs of a neuron combined determine whether its argument is $z > 0$. So, the activation correlates the various inputs.

2.3.2.1 Classification and Non-linear Regression Problems Supervised learning problems involved training data sets that include both the inputs and the corresponding labels. Pattern recognition of a class or type (*e.g.*, type of amino acid, class of organic molecule, etc.) in which the aim is to assign each input one of a finite number of discrete categories (e.g., one of the 20 types of natural amino acids, or one of the various types of organic molecules, etc.) are called *classification problems*. In contrast when the desired output consists of continuous variables (e.g., the solubility value, the NMR chemical shift value, etc., based on the input molecular structure), then the task is called *regression*. Other examples of regression could be the prediction of the yield of a chemical reaction based on inputs corresponding to the concentrations of reactants, the temperature, and the pressure [R1(3)]. Non-linear activation functions are essential for both non-linear regression and classification problems.

Logistic Sigmoid: Nonlinear activation functions are typically used for output layers of classification problems. For example, the logistic sigmoid function $\phi(z_k) = 1/(1+e^{-z_k})$ is typically used for binary classification (*e.g.*, is the molecule toxic or not, is it flammable or not, etc.), predicting one class when $\phi > 0.5$ and the other when $\phi \leq 0.5$.

Softmax: The softmax activation function $\phi(z_k) = \exp(z_k)/\sum_j \exp(z_j)$ is a multiclass generalization of the logistic sigmoid that outputs a normalized probability distribution over the predicted output classes (*e.g.*, 97% probability that the molecule is and aldehyde, 2% probability that is a ketone, 1% that in an alcohol). The softmax function is typically used in conjunction with loss functions such as the *KL divergence* or *cross-entropy*, described below. For classification problems, the loss defined by the cross-entropy or the KL divergence allow for faster training than the sum-of-square differences as well as improved generalization [R1(235)].

KL Divergence and Cross-Entropy: The Kullback-Leibler (KL) divergence is defined, as follows:

$$KL(p\|\phi) = -\sum_k p(z_k)\log_2(\phi(z_k)/p(z_k)) = H(p,\phi) - H(p), \tag{10}$$

where $H(p) = -\sum_k p(z_k)\log_2(p(z_k))$ is the entropy corresponding to the target probabilities $p(z_k)$, and

$$H(p,\phi) = -\sum_k p(z_k)\log_2(\phi(z_k)). \tag{11}$$

is the cross entropy. Both functions allow for comparisons of two discrete probability distributions and are commonly used for training models to produce outputs corresponding to a target distribution. Note that minimizing the KL divergence corresponds exactly to minimizing the cross-entropy since the entropy $H(p)$ of the target distribution does not depend on the adjustable weights.

2.3.2.1.1 Optional Exercise: Use the bound $log(x) \leq x - 1$ to show that the KL divergence is always ≥ 0 (Gibbs Inequality). Therefore, the cross-entropy is always larger or equal than $H(p)$, and equal to $H(p)$ when the two distributions are the same.

2.3.3 Vanishing Gradient Problem

The gradients of the loss ϵ with respect to the adjustable parameters $\omega_{jk}^{(l)}$, introduced in Sec. 2.1, are computed according to the chain rule:

$$\frac{\partial \epsilon}{\partial \omega_{jk}^{(l)}} = \sum_{i=1}^{K} \frac{\partial \epsilon}{\partial y_i} \frac{\partial y_i}{\partial \omega_{jk}^{(l)}}. \tag{12}$$

For example, according to Eq. (4),

$$\begin{aligned} \frac{\partial y_i}{\partial \omega_{jk}^{(1)}} &= \frac{\partial}{\partial \omega_{jk}^{(1)}} \phi_2 \left(\sum_{k'=0}^{M} \omega_{ik'}^{(2)} \phi_1 \left(\sum_{l=0}^{D} \omega_{k'l}^{(1)} x_l \right) \right), \\ &= \frac{\partial \phi_2(s)}{\partial s} \frac{\partial}{\partial \omega_{jk}^{(1)}} \sum_{k'=0}^{M} \omega_{ik'}^{(2)} \phi_1 \left(\sum_{l=0}^{D} \omega_{k'l}^{(1)} x_l \right), \\ &= \frac{\partial \phi_2(s)}{\partial s} \frac{\partial \phi_1(s)}{\partial s} \omega_{ij}^{(2)} x_k. \end{aligned} \tag{13}$$

Therefore, the gradient of the loss with respect to $\omega_{jk}^{(1)}$ is obtained, as follows:

$$\frac{\partial \epsilon}{\partial \omega_{jk}^{(1)}} = \frac{\partial \phi_2(s)}{\partial s} \frac{\partial \phi_1(s)}{\partial s} \sum_{i=1}^{K} \frac{\partial \epsilon}{\partial y_i} \omega_{ij}^{(2)} x_k, \tag{14}$$

which is proportional to the product of two gradients of activation functions. Analogously, for a NN with 3 layers (*i.e.*, 2 hidden layers) we would have gradients defined, as follows:

$$\begin{aligned} \frac{\partial \epsilon}{\partial \omega_{jk}^{(1)}} &= \sum_{i=1}^{K} \frac{\partial \epsilon}{\partial y_i} \frac{\partial y_i}{\partial \omega_{jk}^{(1)}}, \\ &= \frac{\partial h_3(s)}{\partial s} \frac{\partial \phi_2(s)}{\partial s} \frac{\partial \phi_1(s)}{\partial s} \sum_{i=1}^{K} \frac{\partial \epsilon}{\partial y_i} \sum_{k'=0}^{M} \omega_{ik'}^{(3)} \omega_{k'j}^{(2)} x_k, \end{aligned} \tag{15}$$

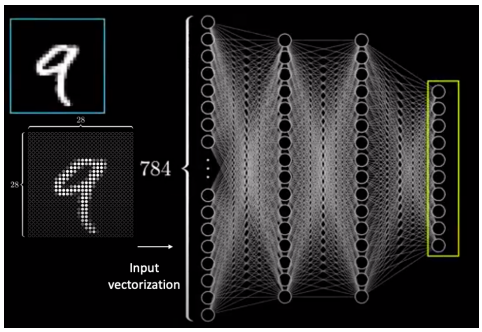


Figure 6: Dense neural network with multiple hidden layers, with input neurons corresponding to the pixels intensities of hand written numbers reshaped by vectorization into a 1-dimensional array.

which are clearly proportional to the product of 3 gradients of activation functions, making the gradient of the loss very small when the gradients of the activation functions are < 1. Therefore, deep neural networks with multiple hidden layers (Fig. 6) typically rely on ReLU or Leaky ReLU activation functions that enable efficient training since they do not suffer from the vanishing gradient problem. Fig. 6 also shows that a 2-dimensional array of data, corresponding to the 28×28 intensities of pixels of hand written numbers can be vectorized –*i.e.*, reshaped as a 1-dimensional array of 784 neurons.

2.3.4 Validation, Cross Validation and Bootstrapping

Validation of the model is essential to ensure that the NN works well on data that has not been used during the training process. Therefore, it is important to keep aside a portion of the data that is not used for training, and use it for testing and validation to ensure that the loss of the training and testing sets are comparable.

A simple approach is to split it into 70:30, as shown in Fig. 7 (left), which is a fine procedure when there is enough data so long as the split ensures that the training and testing samples have the same distribution (*e.g.*, randomized).

When we have limited data, the simple splitting procedure described above might introduce bias in the parametrization of the NN since the training set might miss some key sample points. So, neither the training set distribution nor the testing set might be representative of the original data set. Therefore, with limited data, it might be necessary to implement *cross-validation*.

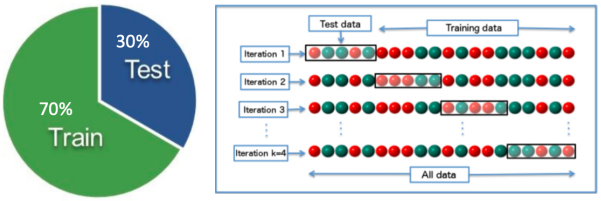


Figure 7: Left: Splitting of data into training and testing sets. Right: Representation of the sampling procedure implemented in k-fold cross-validation method.

The k-fold cross validation method, represented in Fig. 7 (right panel), is a popular resampling procedure that generates a less biased model even when having limited data since it ensures that every data point from the original dataset is included in the training and testing sets. Rather than splitting the data into 70:30, the data is split randomly into k folds (k is typically between 5-10). At each iteration, all but one of the folds are used for training while the fold that was not used for training is used for validation. The process is repeated until every fold has been included in the testing set. Finally, all k validation results are averaged (a process often called *bagging*).

Bootstrap sampling is another method commonly used when there is limited amount of data to generate models with less bias and less *overfitting*, a problem discussed in Sec. 2.6.2. Bootstrap sampling also enables quantification of uncertainty of the predictions. Similarly to cross-validation, bootstrap sampling generates multiple data sets as follows [R1 (23)]. Suppose our original big data set consists of N data samples $X = x_1, ..., x_N$ (Fig. 8). We create a data set X_A by drawing n samples at random from X. Then, we create X_B by drawing n samples at random from X, with replacement, so that some points in X_A may be replicated in X_B, whereas other points in X_A may be absent from X_B. The process is repeated N times to generate N data sets, each of size n and each obtained by sampling from the original data set X. Predictions from all data sets are averaged (a process called bootstrap *aggregation*, or *bagging*) and the prediction uncertainty is estimated with the standard deviation.

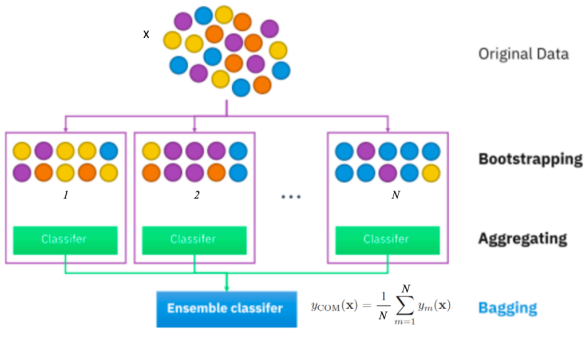


Figure 8: Bootstrap sampling of N batches of data by drawing n points at random from X with replacement, so some points in X may be replicated in X_B, whereas other points in X may be absent from X_B.

2.3.4.1 Exercise: Bootstrapping To see how bootstrapping can reduce the variance of an estimation by simply averaging multiple estimations, we consider the problem of estimating the average value of rolling a fair dice (*i.e.*, average of the uniform distribution for the six possible outcomes 1-6). Compute the average by rolling the dice $N = 1000$ times and show that the distribution of outcomes is uniform with variance σ^2. Repeat the process a large number of times and show that the average of the averages exhibits a normal distribution with a variance $\epsilon^2 = \sigma^2/N$.

- Show that the distribution of outcomes is uniform, since the dice is fair. Obtain the mean of outcomes, which is close to 3.5 and variance $\sigma^2 = 3$.
- Compute the distribution of averages obtained by repeating that calculation 2,000 times and show that the distribution of means is a Gaussian with variance $\epsilon^2 = \sigma^2/N = 0.003$.

This is a demonstration of the Central Limit Theorem at work. The theorem states that the distribution of a sufficiently large number of means obtained with N samples drawn with replacement from any arbitrary distribution with variance σ^2, is a Gaussian with variance $\epsilon^2 = \sigma^2/N$.

Solution: Download the Jupyter notebook with the solution from dice.ipynb.

2.4 Tutorial Assignment on Hammett Neural Networks with Keras/TensorFlow

[***This tutorial assignment has been designed and developed by Jessica Freeze***]

The tutorial on how to build and execute a Hammett neural network with Keras/TensorFlow for prediction of frontier orbital energies of tungsten-benzylidyne catalysts using Hammett parameters as input descriptors can be downloaded as a notebook:

Assignment_1_NeuralNetworks.ipynb pdf

2.5 Tutorial Regressive Models for Chemical Predictions with Scikit-Learn

Here's a Scikitlearn.zip with the notebooks An introduction to Machine Learning with Scikit Learn.pdf and Robust And Calibrated Estimators With Scikit Learn In Machine Learning.pdf. Unzip the file in your colab folder and update the name of your google folder (in each notebook). Furthermore, the file Introducing_Scikit_Learn.ipynb.zip contains the notebook Introducing Scikit Learn.pdf .

[***The following tutorial assignment has been designed and developed by Jessica Freeze***]

The tutorial on how to implement regressive models with Scikit-Learn for chemical properties predictions can be downloaded as a notebook:

Assignment_2_RegressiveLinearModelsForChemistryPrediction.ipynb, pdf.

2.6 Prediction of Molecular Toxicity by Linear Classification with DeepChem

A turn-key tutorial on how to make predictions of molecular toxicity with respect to 12 different assays, using a classification neural networks with DeepChem, can be downloaded as a notebook:

04_Molecular_Fingerprints.ipynb, 04_Molecular_Fingerprints.pdf

The 2-layer NN is trained with Tox21, a database that contains information about the toxicity of molecules with respect to 12 different assays. The input molecular features are defined according to the type of atoms and their corresponding neighborhoods in the molecular structure, using the extended-connectivity fingerprints (ECFPs), also known as *circular fingerprints* available in DeepChem. The main properties of ECFPs are that (i) they are defined by considering circular atom neighborhoods (Fig. 16, middle panel, Sec. 5); (ii) they are rapidly calculated; (iii) they

represent substructures; (iv) they can account for a huge number of different molecular features (including stereochemical information); and (v) they represent both the presence and absence of functionality.

2.6.1 Training

Figure 9 illustrates the typical iterative procedure implemented for training of neural networks. The training data is typically divided into N batches. The training process is initiated with the first batch of data that is input for *forward propagation*, a process that initializes the weights (often randomly chosen weights) and computes the values of neurons in the hidden and output layers. The resulting output values are then compared to the actual labels of the input data to compute the loss, introduced by Eq. (3), and the gradient of the loss with respect to the weights, according to Eq. (12). The gradients are then used for *back propagation*, the process that updates the weights by the gradient descent optimizer. Having updated the weights, the second batch is input and the forward and backward propagation steps are applied. The process is repeated for each batch until the first epoch is completed. Then, the loss is computed and if it is not sufficiently low the weights are further optimized by processing more epochs.

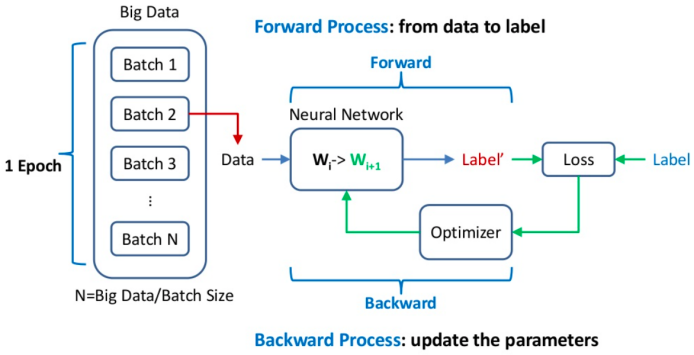


Figure 9: Representation of the training iterative procedure implemented for parametrization of neural networks by forward and backward propagation.

2.6.2 Overfitting Problem

The design of neural networks is usually based on previously developed successful neural networks or the result of work based on trial and error to tune the *hyperparameters* that define the number of layers, the number of neurons per layer, and the types of layers. The neural network is designed to achieve minimum loss for both the training and validation/testing data. The loss of the training set can always be minimized by increasing the complexity of the network, increasing the number of layers and number of neurons per layer. However, increasing too much the number of adjustable parameters can lead to the problem of *overfitting* where increasing the complexity reduces the loss of the training set but increases the loss of the validation data set, as shown in Fig. 10. By trial an error, one can increase the complexity and reach the ideal complexity range where the error for the validation/testing data is minimum, making sure not to fall into overfitting range where the error for the testing data set increases.

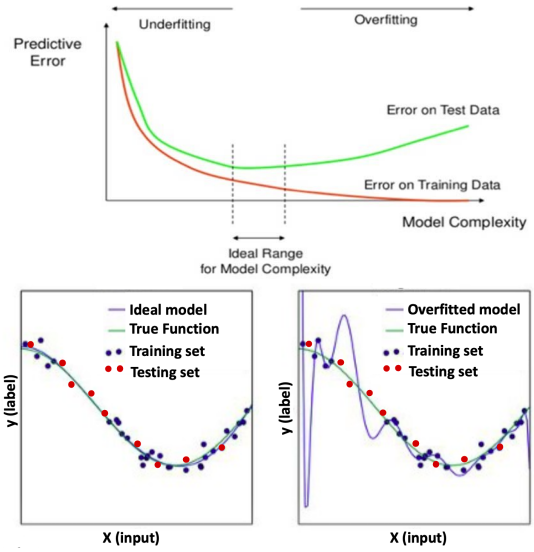


Figure 10: Top: Increasing the complexity of the model decreases the loss for both the training and testing data sets until reaching the bias trade-off point. Increasing the complexity further leads to overfitting, where the loss for the training set is reduced but the loss for the testing set increases. Bottom: Increasing the number of adjustable parameters introduces oscillations to reduce the loss of the noisy training data, increasing the loss of the testing set.

2.6.3 Regularization

Dropout is a simple regularization method, commonly used for reducing the number of adjustable parameters in artificial neural networks.

Randomly chosen weights are simply omitted or 'dropped out' by zeroing them during the training process, as shown in Fig. 11. The technique is also called *random pruning* and belongs to the family of dilution methods (*i.e.*, methods based on adding damping noise to the parameters of the model).

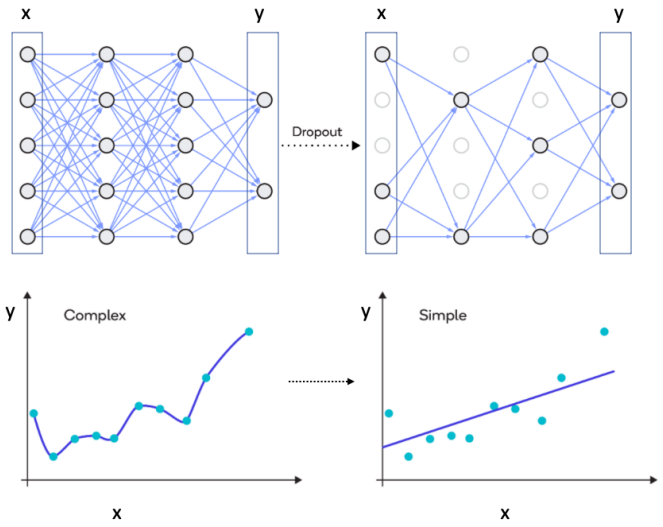


Figure 11: Top: Dilution by 'dropout', a technique that reduces the number of adjustable parameters by simply omitting randomly chosen weights (*i.e.*, links), dropping them out of the training process. Bottom: Representation of complex solutions, obtained by overfitting with dense neural networks (left), as compared to simple solutions obtained by regularization.

Other regularization techniques are typically included in most optimization methods used for parametrization of neural networks, including L_2 regularization (called *Ridge Regression*) where the loss includes an additional term to penalize according to the sum of the squares of the weights. For example, for the 2-layer NN introduced in Sec. 2, the loss for Ridge regression would be: $\epsilon = \sum_{j=1}^{K}(y_j - y_j^{(a)})^2 + \lambda\left(\sum_{i,k}(\omega_{ik}^{(1)})^2 + \sum_{i,k}(\omega_{ik}^{(2)})^2\right)$. L_1 regularization (also called *Lasso Regression*), by adding a term to the loss proportional to the sum of absolute values of the weights: $\epsilon = \sum_{j=1}^{K}(y_j - y_j^{(a)})^2 + \lambda\left(\sum_{i,k}\omega_{ik}^{(1)} + \sum_{i,k}\omega_{ik}^{(2)}\right)$.

The main difference between Ridge and Lasso regressions is that Lasso leads to *sparse* representations (with more coefficients equal to zero), since the regularization term introduces an aggressive gradient of the loss with respect to small coefficients, as defined by λ. In contrast, the corresponding gradient in Ridge regression is $2\lambda\omega_{jk}$ –*i.e.*, proportional to the small value of the coefficients ω_{jk}.

Note that the regularization terms added to the loss, transform the problem into a more constrained problem with fewer possible solutions, eliminating for example solutions with large oscillations, as shown in Fig. 10, allowing us to find a simple solution). In general, we look for simple solutions since data collected from systems is usually very simple and the experiments correspond to weak perturbations leading to an observable response that is very simple either linear in the perturbational field or of very low order.

3 Clustering and Regression Algorithms

3.1 Random Forest

Random forest is a recursive algorithm for classification of data into subgroups, using a 'forest' of decision binary trees with branches defined by selected features of the data (Fig. 12). Therefore, the algorithm involves an unsupervised process that profiles data samples into subgroups, after having created decision trees for classification based on selected features.

A simple example is a data set of molecules that we might want to classify into subgroups of certain toxicity, solubility, etc., by analyzing their features. At the same time, the classification process is used to build decision trees that enable classification. Molecules with unknown properties are run through the decision trees to find out in which leaf they end up branching out. Their properties are inferred from the properties of the molecules that usually branch out into that leaf.

Each tree is created by using a subset of the training data set (sampled with replacement), according to an iterative procedure defined by the following steps:

1. Randomly choose n features of the data to be analyzed as potentially splitting features, according to steps 2 and 3.

2. For each feature, find the splitting point that minimizes the entropy (or, linearized entropy, called *Gini impurity* as discussed in Sec. 3.1.1). [1]

3. Split the data into 2 subgroups, using the feature with best performance.

4. Stop, if stopping criterion is met (*e.g.*, the entropy decrease is smaller than a given threshold, or some other stopping criteria is met such as the maximum depth of the tree has been reached, or the number of samples in the node is smaller than a minimum value, etc., to ensure that the tree does not classify noise). Otherwise, goto 1 and repeat the process for the generated subgroups.

Clearly, the algorithm subdivides the data into subgroups, each of which with more uniform population (*i.e.*, more pure in a certain type of feature) than the complete subset of data used for construction of the tree. The purity of the subgroups can then be exploited to classify a sample of unknown type by running it through the decision tree and assigning to its type, the type of data corresponding to the subgroup where the sample branched out. Running the unknown sample through all of the trees of the forest and averaging the results from all trees gives the ensemble average sample classification either as a majority vote, or otherwise a weighted vote based on the level of confidence for the prediction from each tree.

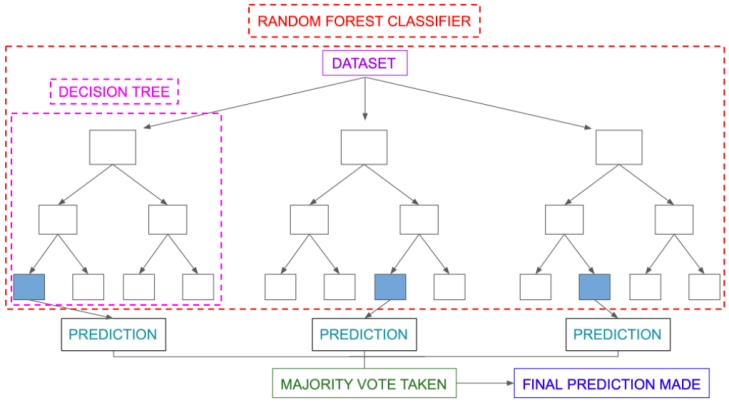


Figure 12: Schematic representation of the forest of binary decision trees generated by the random forest algorithm, and the outcome prediction for a testing sample (blue).

[1] Alternatively, one can simply choose a random splitting point (a variation of the method called extremely randomized trees).

As described above, the random forest algorithm is a random subspace method for ensemble learning where correlations between prediction estimators (trees) are suppressed by training the trees on random samples of data and random selection of features instead of using entire feature set. The bootstrap aggregation, or bagging, combines the predictions produced by several learners into an ensemble that performs better than the original learners.

3.1.1 Entropy and Gini

The *entropy* of a tree node k with N_k elements of N_t possible types is defined, as follows:

$$S_k = -\sum_{j=1}^{N_t} p_{jk} \log_2(p_{jk}), \tag{16}$$

where $p_{jk} = n(j,k)/N_k$ is the likelihood that an element of node k is of type j, as defined by the number $n(j,k)$ of elements of type j in node k over the total number N_k of elements in that node.

The so-called *Gini* impurity measure G_k of node k is defined as the linearized entropy, obtained by approximating $\log_2(p_{jk})$ by its first order expansion around $p_{jk} = 1$ (*i.e.*, using the expansion $\log_2(p_{jk}) \approx \log_2(1) + (p_{jk} - 1) + \cdots = p_{jk} - 1$): $G_k = -\sum_{j=1}^{N_c} p_{jk}(p_{jk} - 1) = 1 - \sum_{j=1}^{N_c} p_{jk}^2$. Note that $G_k = S_k = 0$ when all of the elements of node k are of a single type i (*i.e.*, the composition is pure in type i). In addition, both S_k and G_k are maximum when all the possible types have equally probable, (*i.e.*, with probability $p(j|k) = 1/N_t$), and the composition is maximally impure. Therefore, both S_k and G_k are equally valuable as metrics of impurity. In practice, however, G_k is more popular since it is less expensive to compute.

3.2 K-means Algorithm

The k-means clustering algorithm classifies the samples of a data set into K subgroups of elements that happen to be closer to the centroid of their cluster than to the centroids of any of the other $K-1$ clusters, as defined by a given measure of distance.

The k-means clustering procedure can be described, as follows. Starting with the complete data set, K centroids are defined at random in the space of features,

1. Assign samples to their corresponding nearest centroid.
2. Recompute the centroid of each cluster according to their own samples.
3. Stop, if stopping criterion is met (*e.g.*, the composition of each cluster has not changed). Otherwise, goto 1.

The number of clusters that defines the most unbiased classification, as determined by the composition of the clusters, is typically the number of clusters that maximizes the entropy $S_K = -\sum_{j=1}^{K}\sum_{i=1}^{N_t} p_{ij}\log_2(p_{ij})$, where $p_{ij} = n(i,j)/N_j$ is the likelihood that an element of cluster j is of type i, as defined by the number $n(i,j)$ of elements of type i in cluster j over the total number of elements N_j in that cluster. The maximum entropy principle gives the most unbiased distribution since an inference made on the basis of incomplete information should be drawn from the probability distribution that maximizes the entropy, subject to the constraints on the distribution.

3.3 K-Nearest Neighbors Algorithm

The K-nearest neighbors (KNN) algorithm is a classification method based on the plurality vote of the K nearest neighbors, in the space of features (K-NN classification). It can also be used as a regression method based on the average of the properties of the K nearest neighbors (K-NN regression). The average, or the vote, can also be weighted inversely proportionally to the distances of the sample to each of its K nearest neighbors. The underlying assumption of the method is that samples that are close together in the space of features have similar properties.

3.4 Unsupervised Classification Assignment: K-means and Random Forest

[***This tutorial assignment has been designed and developed by Jessica Freeze***] The assignment for unsupervised classification can be downloaded as a notebook UnsupervisedClassification.ipynb or UnsupervisedClassification.pdf

4 Convolutional Neural Networks (CNN): Alphafold

Regularization can also be accomplished by convolution, as shown in Fig. 13 for the convolution of a 7×7 neural layer, using a 3×3 convolutional kernel of weights $\omega_{k,l}$. Starting with the kernel placed at the top-left corner of the input layer, the input neurons overlapping with the kernel are multiplied with the corresponding kernel weights and the products are summed to generate the value of the neuron $z'_{i,j}$ at the convoluted layer, aligned with the center of the kernel:

$$z'_{i,j} = \phi\left(\sum_{k=1}^{3}\sum_{l=1}^{3} z_{i+k,j+l}\omega_{k,l}\right), \qquad (17)$$

where ϕ is an activation function responsible for the so-called 'detector stage'. The kernel is then displaced to overlap with other neurons in the input layers and the same weights of the convolution kernel $\omega_{k,l}$ are used to generate the value of another neuron in the convoluted layer, according to Eq. (17).

The process is then repeated until the whole input layer is convoluted, as seen by clicking in the animation of the lower panel of Fig. 13. Convolution is often followed by a *pooling* layer that 'summarizes' sections of the convoluted layer into a single output value for example by computing the average of the values of the convoluted layer, or the maximum value (maxpooling). Pooling is often applied since it makes the output invariant with respect to global changes in the same image such as translation or rotation of the input in applications to image processing.

When compared to feedforward NN layers, CNN offer significant advantages for enhanced performance since they typically include fewer links between neurons (*i.e.*, sparcity), and fewer parameters (*i.e.*, shared weights). Even for the simple example shown in Fig. 13, we note that a feedforward transformation from 7×7 to 5×5 layer would require $7 \times 7 \times 5 \times 5$ weights, while the CNN requires only 3×3 since the same parameters are shared by all of the transformations that generate the convoluted layer. In addition, the CNN introduces sparsity of connectivity, with much fewer links between input and output neurons. Note, for example, that the neuron at the top-left corner of the input layer is linked only to the neuron at the

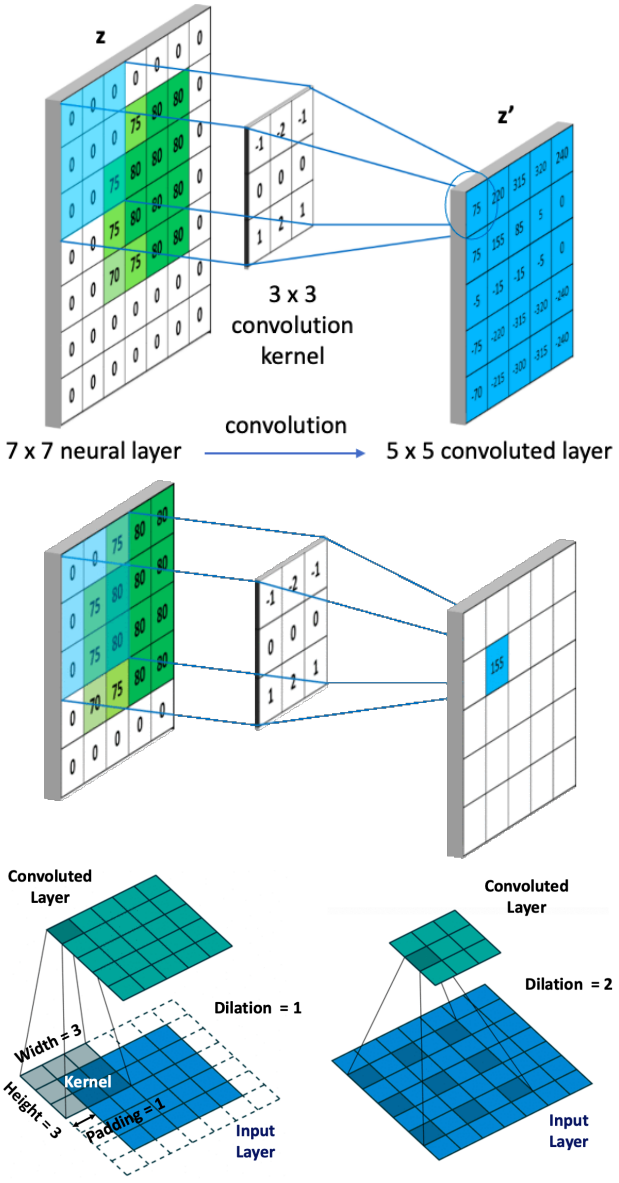


Figure 13: Top: Convolution of a 7×7 neural layer, using a 3×3 kernel with stride=1, padding=0, and dilation=1 to generate a 5×5 convoluted layer. Middle: Animation of convolution. Bottom: Comparison of convolution layers obtained with 3×3 kernels, with dilation = 1 (left) and dilation = 2 (right), respectively.

top-left corner of the output layer in that CNN, while it would be linked to all neurons of the output layer in a corresponding dense feedforward network.

Convolution layers are defined by the hyperparameters of the kernel (parameters defined by the user), shown in Fig. 13 (bottom panel), including the kernel dimensions as defined by the *height* (k_h) and *width* (k_w) for a 2-dimensional kernel, the *stride* (s) (i.e., the step size for striding across the input layer), the *padding* (p) (or, border parameter defining whether the kernel stops when it reaches the border or beyond), and the *dilation rate* (d) defining the spacing between the elements of the kernel. The resulting dimensions (width W_{out} and height H_{out}) of the resulting convoluted layer can be computed in terms of the kernel parameters and the width W_{in} and height H_{in} of the input layer, as follows:

$$\begin{aligned} W_{out} &= \left[\frac{W_{in} + 2 \times p - d \times (k_w - 1) - 1}{s} + 1 \right] \\ H_{out} &= \left[\frac{H_{in} + 2 \times p - d \times (k_h - 1) - 1}{s} + 1 \right] \end{aligned} \tag{18}$$

When the input has C_{in} channels (*e.g.*, pixel intensities for blue, red and green), the kernel can have multiple channels and transform the input into an output layer with a single channel. Further, C_{out} kernels can be applied to generate an output layer with C_{out} channels, as shown in Fig. 14.

Convolutional layers are particularly suitable for extracting features, as in applications to image processing. For example, the edges of an image can be generated simply by convolution of the image with a 3×3 kernel with positive and negative values,

$$k = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}. \tag{19}$$

In addition, convolution has valuable properties for image processing since it commutes with translation (*i.e.*, the shift of an input simply leads to a shifted output) and thus generates similar outputs for similar inputs. In addition, convolutions are local due to the finite scope of the kernel and, therefore, preserve the local structure of the input. Various different convolution kernels are typically applied so that various different features can be extracted.

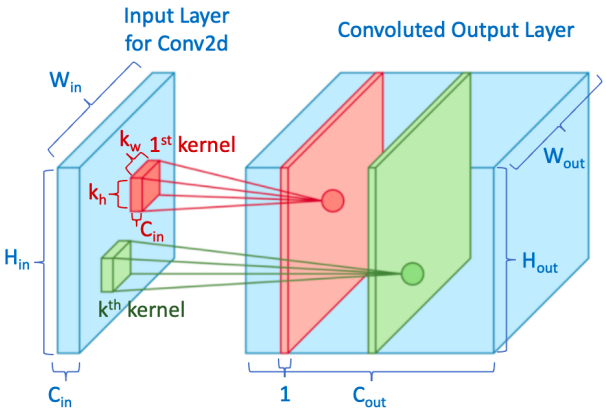


Figure 14: Convolution of a $W_{in} \times H_{in}$ input layer with C_{in} channels using multiple $k_w \times k_h$ kernels to generate the $W_{out} \times H_{out}$ output layer with C_{out} channels.

The development of the CNN AlexNet, shown in Fig. 15 (top panel), and its celebrated victory in the *2012 ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) has revolutionized the field of deep learning. The network achieved a top-5 performance with 15.3% error for recognition and classification of images of 1000 categories, including about 1,200 image per category. AlexNet, with 5 convolutional layers and 3 fully connected layers, has shown that the depth of the model is essential for high performance. Furthermore, AlexNet laid the foundation for the traditional CNN scheme based on a *convolutional layer* followed by an *activation function* followed by a *max pooling* operation, although the pooling operation is sometimes omitted to preserve the spatial resolution of the image. Training of AlexNet, with 61 million parameters, and 600 million

connections, as summarized here was possible due to the utilization of graphics processing units (GPUs), and the feasibility of parallelizing the convolutions based on two pathways representing the split between two GPUs (Fig. 15 (top panel)).

Since AlexNet, the state-of-the-art in CNN architectures has gone deeper and deeper (*i.e.*, AlexNet has only 5 convolutional layers, while the VGG network has 19 layers, and GoogleNet (also codenamed Inception_v1) has 22 layers) (Fig. 15, middle panel). Training even deeper neural networks has been enabled by the discovery of residual layers that skip over convolutional layers to avoid the vanishing gradient problem, resulting in one of the most ground-breaking developments in the last few years. ResNet allowed training of thousands of layers.

Convolutional neural networks have made a significant contribution toward solving the folding protein challenge, with the development of AlphaFold1, a neural network (Fig. 15, bottom panel) for prediction of 3-dimensional protein structures. AlphaFold1 created high-accuracy structures for 24 out of 43 free modeling domains, in the blind assessment of the state of the field competition *Critical Assessment of Protein Structure Prediction5* (CASP13), whereas the next best achieved accuracy for only 14 out of 43 domains. This link provides a notebook to play with AlphaFold.

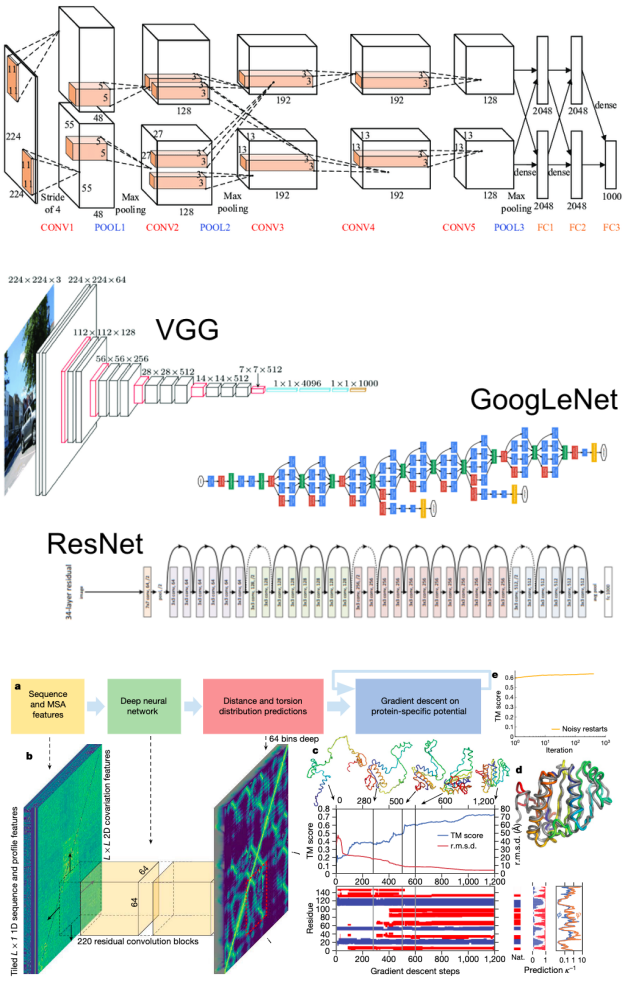


Figure 15: Top: CNN AlexNet for image recognition, including 5 convolutional layers, and 3 fully connected layers. The two pathways represent the split between two GPUs. Middle: Representation of VGG and Google ResNet networks. Bottom: Diagram of AlphaFold1, including a CNN that generates the 2-dimensional contact map of distances d_{jk}^{cnn} between amino acid residues j and k, and a function that computes the position of alpha carbons and thus the inter-residue distances, as a function of torsion angles.

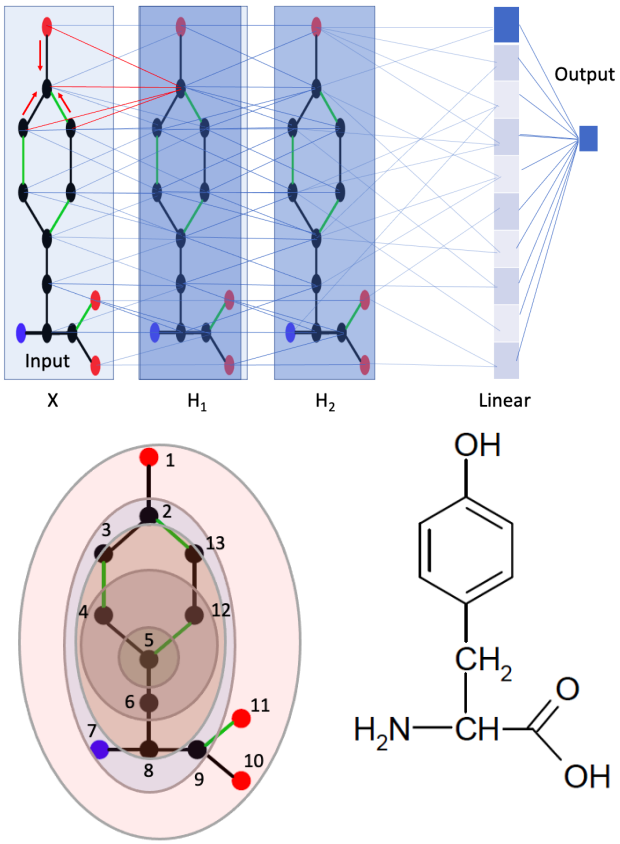
The AlphaFold1 neural network takes a 2-dimensional input based on the protein primary sequence of amino acid residues and features of the amino acids, and implements a CNN that generates the 2-dimensional contact map of distances d_{jk}^{cnn} between amino acid residues j and k. The predicted d_{jk}^{cnn} values are then used to calculate the protein backbone torsional angles ϕ and ψ of all residues by gradient descent optimization of a function $\mathbf{x} = G(\phi, \psi)$ that computes the coordinates $\mathbf{x}_j$ of all alpha carbons j and thus the inter-residue distances $d_{jk} = \|\mathbf{x}_j - \mathbf{x}_k\|$. SGD optimization minimizes the loss between d_{jk} and d_{jk}^{cnn} w.r.t. ϕ and ψ. More recently, the CNN of AlphaFold1 has been replaced by an attention transformer, exploiting advances in natural language processing developments, discussed in Sec. 7.1, leading to the development of AlphaFold2 with performance higher than 87 % at the CASP14 assessement.

5 Graph Convolutional Networks (GCN)

Graph convolutional networks (GCN) are ideally suited for describing molecules, since molecules can be represented by graphs (*i.e.*, nodes connected by edges). The nodes are defined by $N \times F^{(j)}$ matrices $H^{(j)}$, corresponding to N atoms with $F^{(j)}$ features (Fig. 16). The edges (bonds) are defined by the $N \times N$ adjacency matrix, A (with $A_{jk} = 1$, if atoms j and k are linked, and $A_{jk} = 0$, otherwise). The adjacency martix defines the degree matrix, as follows: $D_{i,l} = \delta_{i,l} \sum_k A_{i,k}$, a diagonal matrix defining the number of edges of each node.

The features $H^{(j)}$ of nodes in hidden layer j are computed with so-called *propagation rules*, described later in this section, pooling transformations (represented by red arrows in Fig. 16, top) that compute the features of each node by convolution with those of its neighbors as defined in the previous layer $(j-1)$. The convolutional kernels are defined by $F^{(j-1)} \times F^{(j)}$ matrices of weights $\omega^{(j-1)}$, corresponding to the numbers $F^{(j-1)}$ and $F^{(j)}$ of features in layers $j-1$ and j, respectively (*vide infra*). The weights are trained by gradient descent to ensure that the final output of the GCN matches the labels of a training set of molecules (all of them defined by their corresponding different graphs although with common atomic features).

Inputs: The features are initialized as defined according to the atom types and their corresponding neighborhoods in the molecular structure as encoded, for example, by the extended-connectivity fingerprints (ECFPs), also known as *circular fingerprints* available at DeepChem. The main properties of ECFPs are that (i) they are defined by considering circular atom neighborhoods (Fig. 16, middle panel); (ii) they are rapidly calculated; (iii) they represent substructures; (iv) they can account for a huge number of different molecular features (including stereochemical information); and (v) they represent both the presence and absence of functionality. Also, a differentiable generalization of circular fingerprints has been



	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0	0	0	1
3	0	1	0	1	0	0	0	0	0	0	0	0	0
4	0	0	1	0	1	0	0	0	0	0	0	0	0
5	0	0	0	1	0	1	0	0	0	0	0	1	0
6	0	0	0	0	1	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	1	1	0	1	0	0	0	0
9	0	0	0	0	0	0	0	1	0	1	1	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0	0
11	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	1	0	0	0	0	0	0	0	1
13	0	1	0	0	0	0	0	0	0	0	0	1	0

Figure 16: Top: GCN with two hidden layers for predicting a molecular property (*e.g.*, solubility). Middle: Representation of the molecule with a graph where nodes correspond to atoms, colored according to their circular fingerprints, and edges link the nodes according to the molecular connectivity. Bottom: Adjacency matrix defining the edges of the graph as determined by the molecular connectivity.

developed and is available at DeepChem, among other molecule featurizers.

5.1 Propagation Rules

Each hidden layer of features $H^{(j)}$, with $H^{(0)} = X$, is obtained from the previous layer, according to a propagation rule $H^{(j)} = f(A, H^{(j-1)}\omega^{(j-1)})$. In general, $f(A, H^{(j-1)}\omega^{(j-1)}) = \phi(\mathcal{C}(A, H^{(j-1)}\omega^{(j-1)}))$, where $\mathcal{C}$ is a convolution that aggregates the features of each node with those of its linked neighbors, using weights that are optimized by gradient descent. The activation ϕ is typically a ReLU function. Often, the adjacency matrix is incremented with the identity ($\tilde{A} = A + I$) to aggregate the features of the central node with those of its neighbors during the computation of hidden layers.

Examples of popular propagation rules are:

(i) the *sum rule* $(AH\omega)$ computes the features of the i-th node in the j-th hidden layer as the entries of the i-th row of the convolution of A and $H^{(j-1)}$, as follows:

$$\mathcal{C}(A, H^{(j-1)}\omega^{(j-1)})_i = \sum_{k=1}^{N} A_{i,k} H_k^{(j-1)} \omega^{(j-1)}. \tag{20}$$

(ii) the *mean rule* $(D^{-1}AH\omega)$ averages the values of the neighbors and scales the average with the node degree, as follows:

$$\begin{aligned} \mathcal{C}(A, H^{(j-1)}\omega^{(j-1)})_i &= \sum_{l=1}^{N} D_{i,l}^{-1} \sum_{k=1}^{N} A_{i,k} H_k^{(j-1)} \omega^{(j-1)}, \\ &= \sum_{k=1}^{N} D_{i,i}^{-1} A_{i,k} H_k^{(j-1)} \omega^{(j-1)}, \end{aligned} \tag{21}$$

where $D_{i,l} = \delta_{i,l} \sum_k A_{i,k}$ are the elements of the degree matrix. The resulting normalization allows for balanced training of all weights regardless of the node degree (*i.e.*, atom covalency), keeping the aggregated feature on the same order or magnitude as the input features to avoid the problem of exploding gradients. (iii) the *spectral rule* $(D^{-1/2}AD^{-1/2}H\omega)$ normalizes the features of a node, not only taking into consideration the degree of the node, but also the degree of its neighbors, as follows:

$$\begin{aligned} \mathcal{C}(A, H^{(j-1)}\omega^{(j-1)})_i &= \sum_{l=1}^{N} D_{i,l}^{-1/2} \sum_{k=1}^{N} A_{i,k} \sum_{l'=1}^{N} D_{k,l'}^{-1/2} H_k^{(j-1)} \omega^{(j-1)}, \\ &= \sum_{k=1}^{N} D_{i,i}^{-1/2} A_{i,k} D_{k,k}^{-1/2} H_k^{(j-1)} \omega^{(j-1)}. \end{aligned} \tag{22}$$

Therefore, the spectral rule also keeps the features roughly on the same scale as the input features. The main difference when compared to the mean rule is that it weighs more strongly neighbors with low-degree and lower if they have a high-degree. So, it s particularly useful when low-degree neighbors provide more useful information than neighbors with high-degree.

5.2 Prediction of NMR Chemical Shifts by Graph Convolutional Networks

[***This tutorial has been designed and developed by Mr. Cantarella (Haote) Li***, based on the recent publication by Eric Jonas and Stefan Kuhn, *J Cheminform* (2019) **11**:50 *Rapid prediction of NMR spectral properties with quantified uncertainty*] A PyTorch tutorial showing how to implement a Graph Convolutional neural network for predictions of NMR chemical shifts of molecules using molecular graphs with atomic features as inputs, can be downloaded as a notebook: GCN_NMR_Cantarella.zip.

5.3 Prediction of Solubilities by Graph Convolutional Networks with DeepChem

A turn-key tutorial on how to make predictions of molecular solubilities using Graph Convolutional neural networks with DeepChem, using molecular graphs as inputs, can be downloaded as a notebook:

- 01_The_Basic_Tools_of_the_Deep_Life_Sciences.ipynb,
- 01_The_Basic_Tools_of_the_Deep_Life_Sciences.pdf

5.4 Introduction to classification by Graph Convolutional Networks with DeepChem

A turn-key tutorial on how to classify molecular structures using Graph Convolutional neural networks with DeepChem, using molecular graphs as inputs, can be downloaded as a notebook:

- 06_Introduction_to_Graph_Convolutions.ipynb ,
- 06_Introduction_to_Graph_Convolutions.pdf

6 Recurrent Neural Networks (RNN)

Up to this point, we have discussed feedforward networks, convolutional neural networks and graph convolutional networks for supervised learning. Those networks take all of the input values of the input layer at once and predict a single output corresponding to the input values. In contrast, *recurrent neural networks* (RNN) take one input value at a time, and recurrently produce outputs in context of all the previously processed input values. So, RNN's are ideally suited for predicting the next element of a sequence as for the problem of completing a sentence (*i.e.*, fill in the blank). For example, if the input is 'TNT is' the output would be 'explosive', or if the input is 'H_2S is called' the output would be 'hydrogen sulfide'). Isn't that cool? Obviously, a RNN model trained with enough sentences from chemistry books and publications would be quite useful. So, can we build it? What is the structure of an RNN?

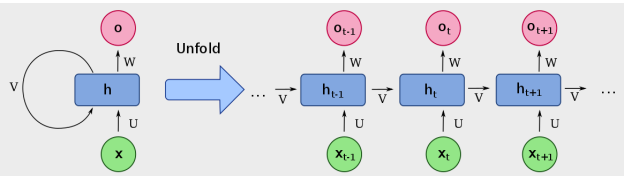


Figure 17: Folded (left) and unfolded (right) representations of a recurrent neural network layer where the inputs x_t are processed in a sequence, one at a time, to generate an output value O_t as well as a hidden state h_t that is combined with the next input x_{t+1} to produce the next element of the output, and a hidden state h_{t+1} that is passed and combined with the next input to continue the process until the end of the input sequence.

One way of looking at the structure of a RNN is as a linear chain of feedforward neural networks (from left to right in Fig. 17). From bottom to top, each feedforward network has an input layer (green), hidden layer (blue) and output (red) from bottom to top. The key distinct structural aspect of the RNN is that the hidden layers of the feedforward networks are connected with directional links from left to right. In each feedforward layer, the input generates a hidden state $h_t = \phi(Ux_t + Vh_{t-1} + b)$ by using an input x_t and the hidden state h_{t-1} from the previous feedforward layer. The activation function is typically $\phi = \tanh$ for which $\phi' = (1-\phi^2)$. So, for each input at a time, a feedforward generates an output $o_t = c + Wh_t$, and passes the hidden state to the next layer. So, in contrast to feedforward and convolutional neural networks, RNNs generate a Markov chain of hidden states h_t determined by the current input x_t and the hidden state h_{t-1} which in turn is determined by all previous inputs $x_1, x_2, \cdots x_{t-1}$. So, each output is determined from all previous inputs. Regularization is achieved by sharing the same parameters for all times.

Training a RNN can be tricky since the gradient of the loss L associated with a large number N of iterative composition steps with $h_t = \phi(Ux_t + Vh_{t-1} + b)$, can quickly vanish, or diverge. To see the origin of this difficulty, let us compute the gradient of L with respect to the hidden states, for the simple example where $o_t = Wh_t$, with $L = L(o_{t_N}, o_{t_{N-1}}, \cdots, o_{t_0})$,

$$\begin{aligned}\frac{\partial L}{\partial h_{t_{N-1}}} &= \frac{\partial L}{\partial o_{t_{N-1}}} W + \frac{\partial L}{\partial o_{t_N}} \frac{\partial o_{t_N}}{\partial h_{t_N}} \frac{\partial h_{t_N}}{\partial h_{t_{N-1}}} = \frac{\partial L}{\partial o_{t_{N-1}}} W + \frac{\partial L}{\partial o_{t_N}} WV\phi', \\ &= \frac{\partial L}{\partial o_{t_{N-1}}} W + \frac{\partial L}{\partial o_{t_N}} WV(1 - (\phi(Ux_t + Vh_{t-1} + b))^2).\end{aligned} \tag{23}$$

So, the gradient with respect to $h_{t_{N-1}}$ depends on V. Analogously, we can show that the gradient of L with respect to $h_{t_{N-2}}$ depends on V^2, and with respect to h_{t_0} depends on V^N. So, for sufficiently large N, the gradients either vanish when $V < 1$, or diverge when $V > 1$, and very quickly! In fact, exponentially quickly with the number of steps N. Multiple strategies have been explored to address this numerical challenge.

Echo State Network: One simple approach that also works as a regularizer is the so-called ***echo state network*** where constraints are imposed on the parameters W, V and U so that the gradients will not vanish or diverge. For example, training only W, with $V = U = 1$ and $\phi = ReLU$.
Clipping Gradients: Another approach is the so-called ***clipping gradient*** method where a maximum value for the gradient is imposed and the weights are evolved with a value of the gradient that is the minimum between the actual gradient and the maximum allowed value. That trick often helps to prevent the divergent gradient problem. Those are problematic since they tend to evolved the weights with big jumps into undesired regions of the parameter space where the loss is very high and the gradients are even higher thus preventing any reasonable convergence. However, the *clipping gradient* method is not a panacea, so we need other approaches.
Long Delays: Another approach is called ***long delays***, where the hidden states are not connected to the nearest neighbors but rather to the second, or to the n-th neighbor. That design delays the process of vanishing or diverging gradients since the gradients would no longer depend on V^N but rather on $V^{N/n}$.
Leaky Units: Another approach is based on the so-called ***leaky units*** where the hidden units summarizing the past are defined, as follows: $h_{t,j} = \left(1 - \frac{1}{\tau_j}\right) h_{t-1,j} + \frac{1}{\tau_j}\phi(Ux_t + Vh_{t-1} + b)$, where the index j labels various different components of the vector h_t. Note that the model has two limits, including the limit of $\tau_j = 1$ where the model is equivalent to the standard RNN with $h_{t,j} = \phi(Ux_t + Vh_{t-1} + b)$, and the limit when $\tau_j \to \infty$ where $h_{t,j} = h_{t-1,j}$ so the model is analogous to the *echo state*, with $V = 1$. Implementing different values of τ_j for different directions j, allows to implement the influence of the past differently along different components j corresponding to different features of the model.
Gated RNN: Starting with the ***Long Short-Term-Memory*** (LSTM) algorithm, the ***Gated Recurrent Units*** (GRU) method proposed in 2014, and the many different variations that have been developed during the past few years address the central issue of either forgetting the past or allowing information from previous steps to pass through a gate and influence the current hidden state.

GRUs are more recent and more flexible that LSTM. Rather than updating the hidden units $h_{t,j} = \phi(Ux_t + Vh_{t-1} + b)$ as in the standard RNN, GRUs are based on so-called 'gate units'. The original version by Cho et al introduced the so-called **update gate** layer $z_t = \sigma(U^{(z)}x_t + V^{(z)}h_{t-1})$ usually defined with a sigmoid activation function, and a **reset gate** that resets the memory to forget the past. It is similarly defined although with different weights $r_t = \sigma(U^{(r)}x_t + V^{(r)}h_{t-1})$ that can be open or closed, for letting information flow, or not. Here, $U^{(r)}$, $V^{(r)}$, $U^{(z)}$, and $V^{(z)}$ are weight matrices that are learned. The gates update the hidden layer, as follows:

$$\begin{aligned} h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t, \\ \tilde{h}_t &= \tanh(Wx_t + U(r_t \circ h_{t-1})), \end{aligned} \tag{24}$$

where the symbol $\circ$ in the first term corresponds to the element-wise Hadamard product of elements of vector z_t times corresponding elements of vector h_{t-1} in the same position. The temporary memory $\tilde{h}_t = \tanh(Wx_t + U(r_t \circ h_{t-1}))$ is defined just by x_t when the reset gate r_t is closed (equal to 0) and therefore zeroes the vector Uh_{t-1} that brings information from the past. Many other variations of GRUs have been proposed.

The equations for LSTM are more complicated with various types of gates which provide furher flexibility at combining current information with information from the past. In fact, GRU was developed to simplify LSTM for problems where that extra flexibility was not necessary. Nevertheless, in practice, LSTM and GRU are implemented as modules that take inputs h_t and x_t and generate output h_{t+1}, as implemented for example in PyTorch with torch.nn.RNN and torch.nn.GRU.

7 Autoencoders

Autoencoders are made by connecting two neural networks with a bottleneck layer called *latent space*, as shown in Fig. 18. Autoencoders are typically trained for reconstruction of the input by minimizing the loss defined by the difference $|\mathbf{x} - \hat{\mathbf{x}}|$ between the input $\mathbf{x}$ and the output $\hat{\mathbf{x}}$, for example, in applications to data compression. The underlying dimensionality reduction when transforming $\mathbf{x} \in R^d$ to $\mathbf{z} \in R^n$ with the encoder neural network is analogous to the transformation $\mathbf{z} = \mathbf{U}^T\mathbf{x}$ of principal component analysis (PCA) where $\mathbf{U}$ is a $d \times n$ matrix. The reconstruction with the decoder neural network is analogous to the transformation $\hat{\mathbf{x}} = \mathbf{U}\mathbf{z}$, with $\mathbf{U}\mathbf{U}^T = I$. Autoencoders can operate like PCA when they are built with linear activation. However, they can also be more general (*i.e.*, generalizations of PCA) by including multiple layers with nonlinear activation functions. Autoencoders can also be overcomplete, when the latent space is larger than the input space (*i.e.*, $n > d$). A possible application of overcomplete autoencoders is noise reduction since the trivial 'solution' obtained by copying the input into the latent space, and from latent vector to the output would not reconstruct the clean image simply because the input is noisy.

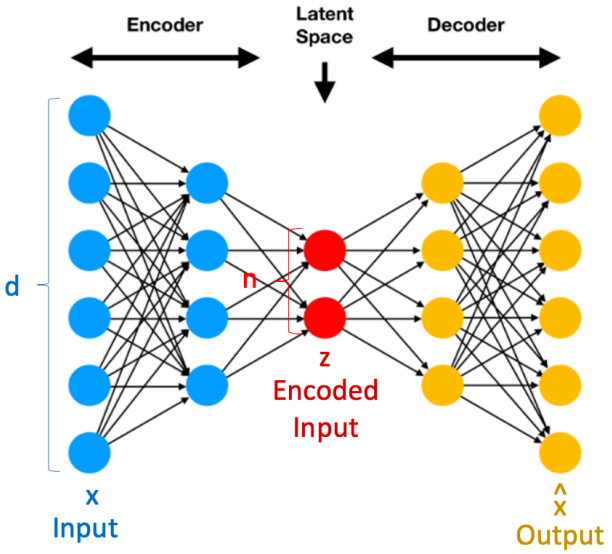


Figure 18: Schematic representation of an autoencoder, constructed by connecting two neural networks with a layer called the latent space.

7.1 RNN, CNN and Multi-Head Attention Autoencoders

Autoencoders have been applied to the problem of sequence to sequence reconstruction/translation, as analyzed by Sutskever et al also described with LSTM RNNs for both the encoder and decoder networks (Fig. 19, top) by the excellent online tutorial. The corresponding transformer based on multi-head attention (Fig. 19, bottom) has been introduced by the article Attention is All You Need also analyzed by the online tutorial. The attention transformer of text and images has generated the amazing GPT transformers, demonstrating design 'creativity', already applied by companies. Beyond captioning and avocado sofas, AI creativity can also generate innovative hypotheses beyond the imagination of humans.

In those implementations, the input is a sentence that is tokenized (*i.e.*, broken into the constituent words and symbols which are *embedded* (*i.e.*, converted to numbers) with the `nn.Embedding` PyTorch module as 256 entry vectors) while the output is the corresponding sentence in a different language. The resulting transformer is trained by teacher forcing, with an additional input to the decoder corresponding to the translated sentence (Fig. 19).

We note that the context vector generated in latent space by the LSTM encoder summarizes not only the content of the sentence, as defined by the words and symbols, but also the relative order of the sequence of words which is essential for the meaning of the sentence.

Analogous transformers can also be achieved by using non-RNN encoders and decoders, including convolutional models and multi-head attention that are much faster because all of the input values of the sequence are provided at the same time. Furthermore, contrary to the RNNs that require a slow

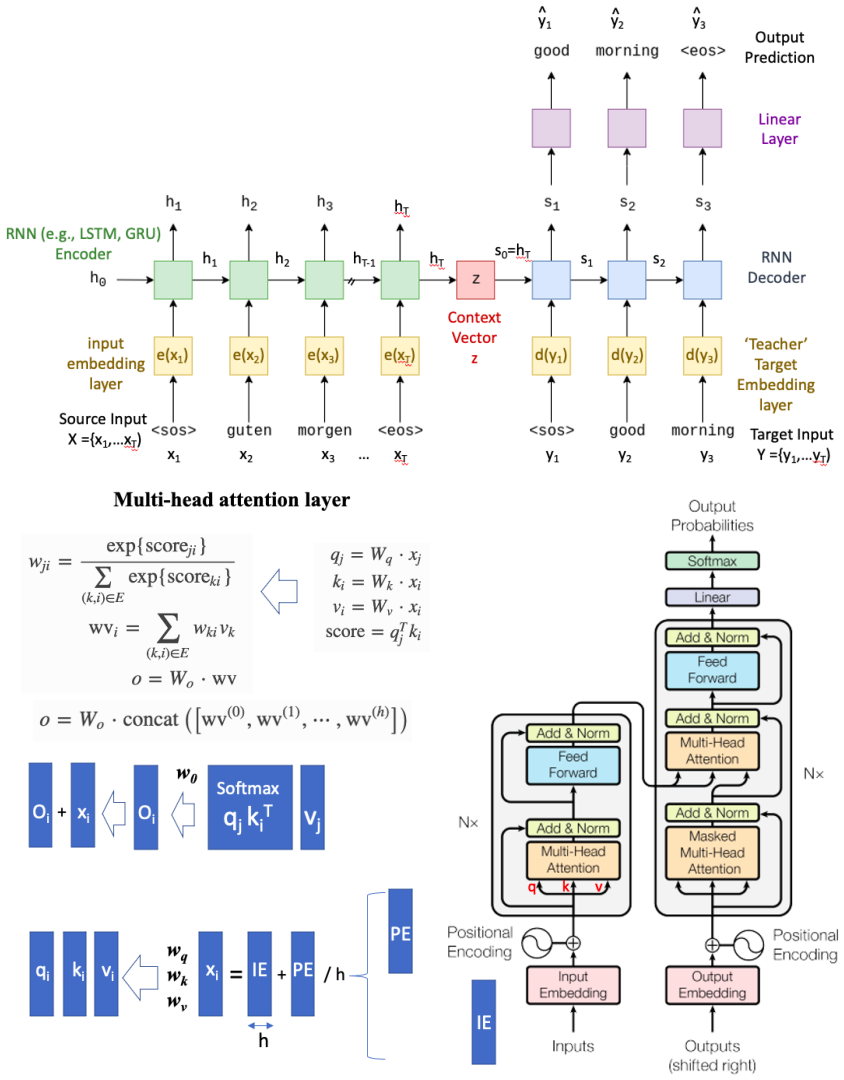


Figure 19: Schematic representations of the sequence to sequence autoencoders based on RNNs with teacher forcing (top) and multi-head attention (bottom).

sequential training process, the convolutional models and multi-head attention schemes have the advantage that can be parallelized, so they can be trained much faster than RNNs.

7.1.1 Attention Mechanism

Figure 19 (lower panel) shows the implementation of the autoencoder for sequence to sequence prediction based on the so-called *multi-head attention* mechanism (Figure 20). Instead of using an RNN for getting dependency between input values, the encoder implements self-attention to captures information of each input token in context of the sequence, as described below.

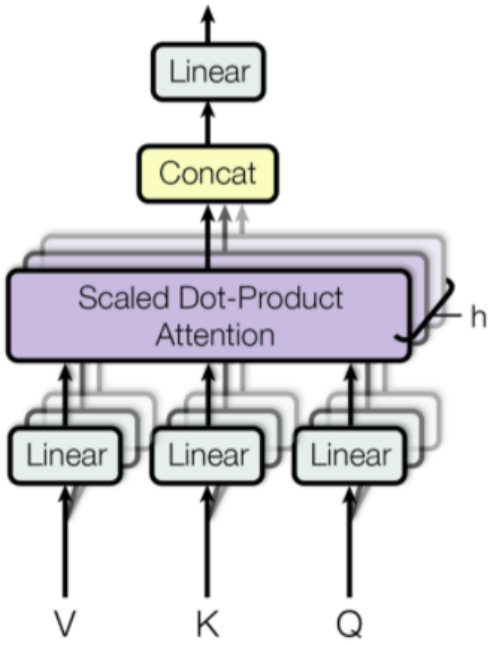


Figure 20: Schematic representation of the multi-head attention module with h parallel attention processes.

Input to Attention: Figure 19 (lower panel) shows that each element of the input sequence (*i.e.*, each input token) is embedded into a vector and combined with a vector that provides the position of that token i in the sequence (*i.e.*, a positional encoding vector), generating the input vector x_j. Key, query and value vectors ($k_i = W_k x_i$, $q_i = W_q x_i$ and $v_i = W_v x_i$, respectively) are then generated from x_i using learnable weights W_k, W_q and W_v.

Dot-Product Attention: The key and query vectors are used to compute softmax attention values

$$w_{ij} = exp(score_{ij}))/\sum_{kl} exp(score_{kl}),$$

from the score of similarity $score_{ij} = q_i^T k_j$ is higher when i and j are correlated elements of the input (related elements of the input). These attention values w_{ij} define the level of alignment between input tokens, and thus the level of attention that the i-th input should pay to the value v_j of the j-th token, as follows $wv_i = \sum_j w_{ij} v_j$. These weighted value vectors are then passed by a linear layer with learnable weights W_o to produce the attention output context vector, as follows: $o = W_o \cdot wv$.

Parallel Multi-Head: The so-called 'multi-head' implementation applies the attention mechanism in parallel to various components of the input and then concatenates and linearly transforms the output into the expected dimension for the context vector, as shown in Figure 20. Such a parallel implementation allows for attending to different parts of the sequence differently.

Example: Image Captioning The jupyter notebook provides a code for image captioning, as described in the github, a fundamental task in vision-language understanding, where the model predicts a textual informative caption to a given input image, as described in this article.

7.2 Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN)

Variational autoencoders (VAE) are generalized versions of autoencoders where the decoder receives an input sample from a uniform or Gaussian distribution and generates an output that resembles the type of labels used in the training. One form of VAE is based on a context vector defining the parameters of a desired probability distribution that generates the input data for the decoder. In that case, the model could be trained with a loss defined by the reconstruction error plus the KL divergence between the desired sampling distribution and the distribution defined by the context vector. The Gaussian distribution allows for analytic computation of the gradients of the KL divergence.

A decoder of a generative model can also be parametrized to take samples from a uniform, or Gaussian distribution and generate outputs with a distribution similar to the distribution that characterizes the labels of the training set. The similarity between the two distributions can be measured by the so-called *maximum mean discrepancy* (MMD) distance described below in Sec. 7.2.1.

Another form of generative models are the generative adversarial networks (GAN) that also feed a decoder with data from a uniform or Gaussian distribution and generate outputs that resemble the distribution of labels, as applied to molecular design. GAN are trained by using an adversarial classifier that is trained to become better and better at distinguishing between the generated data and the labels while the generator is trained to become better at generating data that the discriminator cannot discriminate from the original data. A tutorial on how to train a GAN model with MNIST is available here, and a version with conditional generation with additional inputs to the generator and discriminator to condition the output. Also, Python-GAN provides a wonderful discussion and Pytorch implementation.

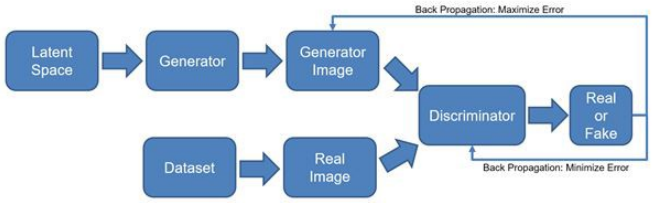


Figure 21: Schematic representation of the GAN model based on a generator from latent space and a discriminator.

Both VAE and GAN models provide generative methods based on sampling functions capable of feeding the decoder with inputs that generate new samples resembling the type of data used in the training set. For example, when the model is trained with pictures of human faces, the decoder generates outputs that resemble human faces. When the training set is based on small drug like molecules, such as those from the Zinc library, the sampling process generates drug like molecules, as shown in the tutorials mentioned below. Isn't that amazing?

VAE Implementation with Pytorch on Colab: [Tutorial designed and developed by Mr. Haote Li] A turn-key tutorial on how to implement the VAE transformer, based on multi-head attention and MMD to generate small drug like molecules, such as those from the Zinc library can be downloaded as the following notebook: VAE_Tutorial.zip.

CVAE Implementation with Pytorch on Colab: [Tutorial designed and developed by Mr. Haote Li] A turn-key tutorial on how to implement the *conditional* VAE transformer, based on multi-head attention and MMD to generate small drug like molecules with specific properties, such molecules similar to those in the Zinc library with a specific solubility can be downloaded as the following notebook: TCVAEtutorial.zip.

7.2.1 Maximum Mean Discrepancy Method

The goal of this section is to introduce the so-called *maximum mean discrepancy* (MMD) method for comparisons of distribution functions, using the *radial basis function* (RBF) kernel.

We consider n samples x_j from distribution function $p(x_j)$ and m samples y_k from distribution function $q(y_k)$. Each of the samples is transformed according to a transformation ϕ and the means of the transformed data are computed, as follows: $\mu_p = \sum_{j=1}^{n} \phi(x_j)$ and $\mu_q = \sum_{k=1}^{m} \phi(y_k)$ to obtain the MMD distance between the two distributions $d_{pq} = |p - q|^2$ as estimated by the distance between the two means, $d_{pq} = |\mu_p - \mu_q|^2$:

$$
\begin{aligned}
d_{pq} &= \left(\frac{1}{n} \sum_{j'=1}^{n} \phi(x_{j'})^T - \frac{1}{m} \sum_{k'=1}^{m} \phi(y_{k'})^T \right) \left(\frac{1}{n} \sum_{j=1}^{n} \phi(x_j) - \frac{1}{m} \sum_{k=1}^{m} \phi(y_k) \right), \\
&= \sum_{j,j'=1}^{n} \phi(x_{j'})^T \phi(x_j) + \sum_{k,k'=1}^{m} \phi(y_{k'})^T \phi(y_k) - \sum_{j=1}^{n} \sum_{k'=1}^{m} \phi(y_{k'})^T \phi(x_j) - \sum_{k=1}^{m} \sum_{j'=1}^{n} \phi(x_{j'})^T \phi(y_k), \qquad (25) \\
&= \frac{1}{n^2} \sum_{j,j'=1}^{n} K(x_{j'}, x_j) + \frac{1}{m^2} \sum_{k,k'=1}^{m} K(y_{k'}, y_k) - \frac{1}{nm} \sum_{j=1}^{n} \sum_{k'=1}^{m} K(y_{k'}, x_j) - \frac{1}{nm} \sum_{j'=1}^{n} \sum_{k=1}^{m} K(x_{j'}, y_k)
\end{aligned}
$$

where the *radial basis function* (RBF) kernel $K(x_j, y_{k'}) = exp(-|x_j - y_{k'}|^2)/(2\sigma^2)$, with the hyper-parameter σ defining the variance, ensures matching of all of the moments of the two distributions.

7.3 Time Series Prediction: Dynamical Mode Decomposition

The goal of this section is to introduce the so-called *dynamical mode decomposition* (DMD) method to predict the evolution of a dynamical system, after learning the principal modes of propagation from observations of a time series of data $v_1, v_2, \cdots, v_n$, collected at n times $t_j = (j-1)\tau$, equally spaced by time intevals τ. Typically, the instantaneous data v_j is an array of N numbers with $N \gg n$. When v_j are the concentrations of components of a mixture at time t_j during a chemical reaction, the task is to predict the evolution of concentrations at subsequent times after analyzing the evolution during the first n time steps.

The DMD method finds the eigenvalues λ_k and eigenvectors Γ_k of the eigenvalue problem

$$A\Gamma_k = \lambda_k \Gamma_k, \tag{26}$$

where A is the $N \times N$ transformation matrix defined by the following equation,

$$Y = AX, \tag{27}$$

where $X = v_1, v_2, \cdots, v_{n-1}$, and $Y = v_2, v_3, \cdots, v_n$ are the matrices of data collected at n equally spaced times $t_j = j\tau$, with $j = 1-n$. We note that A can be diagonalized according to the similarity transformation $\Gamma^\dagger A \Gamma = \lambda$, where λ is the diagonal matrix of eigenvalues, and Γ the matrix of eigenvectors of A.

Linear Model: The dynamics describing the evolution of the data is simulated by considering that X is a continuous time-dependent variable,

$$X(\tau) = AX(0), \tag{28}$$

with $A = e^{\mathcal{A}\tau}$ defined as the propagator so

$$X(t) = e^{\mathcal{A}t} X(0), \tag{29}$$

and $\dot{X}(t) = \mathcal{A}X(t)$. Note that $\lambda = \Gamma^\dagger e^{\mathcal{A}\tau}\Gamma = e^{\Gamma^\dagger \mathcal{A}\Gamma\tau}$, showing that Γ also diagonalizes $\mathcal{A}$ since the exponential of a matrix is diagonal only if the matrix is diagonal. Thus, A and $\mathcal{A}$ have the same eigenvalues λ_k and eigenvectors Γ_k.

Any arbitrary initial state of the data of interest $X(0) = \Psi_0$ can then be expanded in the basis of eigenvectors of A, as follows:

$$\Psi_0 = \sum_{k=1}^{\tilde{n}} c_k \Gamma_k, \tag{30}$$

where the coefficients $c_k = \sum_j S^{-1}_{kj} \langle \Gamma_j | \Psi_0 \rangle$ are computed in terms of the inverse of the overlap matrix $S_{kj} = \langle \Gamma_k | \Gamma_j \rangle$, since the eigenvectors Γ_j are not orthogonal when the matrix A is not symmetric, or Hermitian. The dynamics of the system can then be propagated, according to Eq. (29), as follows:

$$\begin{aligned} \Psi_t = e^{\mathcal{A}t}\Psi_0 &= \sum_{k=1}^{\tilde{n}} c_k e^{\mathcal{A}t}\Gamma_k, \\ &= \sum_{k=1}^{\tilde{n}} c_k e^{\lambda_k t}\Gamma_k. \end{aligned} \tag{31}$$

Eigenvalues and Eigenfunctions: The eigenvalues and eigenvectors of A are efficiently computed, according to the DMD method, by using the singular value decomposition svd of X, as follows.

We introduce the substitution $X = U\Sigma V^T$ into Eq. (27), where U is the matrix of eigenvectors of XX^T with diagonal matrix of eigenvalues Σ^2 since $XX^T = (U\Sigma V^T)(V\Sigma U^T) = U\Sigma^2 U^T$, thus $XX^T U = U\Sigma^2$. Analogously, we can show that V is the matrix of eigenvectors of $X^T X$ with the same diagonal matrix of eigenvalues Σ^2. The elements of Σ^2 thus define the entries of Σ (the so-called *singular values*) which can be all positive (since Σ^2 defines Σ out of an arbitrary phase). Singular values smaller than a given threshold ϵ can be neglected to keep only $\tilde{n} \leq n$ non-zero singular values larger than a desired threshold value, ϵ. Dropping out singular values smaller than ϵ is a form of regularization that filters out noise in the data and define a reduced dimensionality model that can be solved very efficiently.

Substituting the svd $X = U\Sigma V^T$ into Eq. (27), we obtain: $Y = AU\Sigma V^T$, where U is an $N \times \tilde{n}$ matrix. Having, U, Σ and V, we can compute the small $\tilde{n} \times \tilde{n}$ matrix $\tilde{A} = U^T AU$, as follows:

$$\tilde{A} = U^T Y V \Sigma^{-1}. \tag{32}$$

Considering that according to Eq. (26), $A\Gamma_k = \Gamma_k \lambda_k$, we obtain $U\tilde{A}U^T\Gamma_k = \Gamma_k\lambda_k$, so

$$\tilde{A}\tilde{\Gamma}_k = \tilde{\Gamma}_k \lambda_k, \tag{33}$$

with

$$\tilde{\Gamma}_k = U^T \Gamma_k. \tag{34}$$

Solving Eq. (33), we obtain the $\tilde{n}$ eigenvalues λ_k and eigenstates $\tilde{\Gamma}_k$, corresponding to the first $\tilde{n}$ eigenvalues and eigenstates of the large $N \times N$ matrix A since $\tilde{A}$ and A are related by the similarity transformation $\tilde{A} = U^T AU$. Furthermore, we can obtain the first $\tilde{n}$ eigenvectors Γ_k of A, from the eigenvectors of $\tilde{A}$, according to Eq. (34), as follows: $\Gamma_k = U\tilde{\Gamma}_k$.

Alternatively, when $\lambda_k \neq 0$, we can compute Γ_k, as follows:

$$\Gamma_k = \lambda_k^{-1} Y V \Sigma^{-1} \tilde{\Gamma}_k, \tag{35}$$

since $\Gamma_k = U\tilde{\Gamma}_k$, and $A\Gamma_k = \lambda_k\Gamma_k$, so, $\lambda_k^{-1}AU\tilde{\Gamma}_k = \Gamma_k$. Furthermore, $Y = AX$, so $\Gamma_k = \lambda_k^{-1}YX^{-1}U\tilde{\Gamma}_k$, and considering that $X = U\Sigma V^T$ and $X^{-1} = V\Sigma^{-1}U^T$, we obtain $\Gamma_k = \lambda_k^{-1}YV\Sigma^{-1}\tilde{\Gamma}_k$ which proves Eq. (35).

Alternatively, the eigenstates of A can also be written, as follows:

$$\begin{aligned} \tilde{\Gamma}_k &= \lambda_k \Gamma_k, \\ &= Y V \Sigma^{-1} \tilde{\Gamma}_k, \end{aligned} \tag{36}$$

since any multiple of the eigenstate Γ_k is also an eigenstate.

Exercise: Implement the DMD method for analyzing and predicting the evolution of a 2-dimensional Gaussian $\psi(x,y) = e^{-(x-x_e(t))^2-(y-y_e(t))^2}$ with an oscillatory dynamics defined by $x_e(t) = 0.5\cos(w_x t)$ and $y_e(t) = 0.1\cos(w_y t)$, with $w_x = \pi/t$ and $w_y = 0.5 + \pi/t$, where $t = 30$ is the total propagation time.

Solution: The following python script shows the implementation of the DMD method for analyzing the evolution of a 2-dimensional Gaussian. The script also includes the tt implementation which requires installation of scikit tt as described at https://github.com/PGelss/scikit_tt.

```
import numpy as np
import os
import sys
import scipy.linalg as lin
```

```
from scikit_tt.tensor_train import TT
import scikit_tt.data_driven.tdmd as tdmd
import matplotlib.pyplot as plt
import scikit_tt.utils as utl
import time as _time
def gau(r,r0):
    x=r-r0
    return(np.exp(-x**2))
def dmd_exact(x_data, y_data):
    # decompose x
    u, s, v = lin.svd(x_data, full_matrices=False, overwrite_a=True,
                      check_finite=False, lapack_driver='gesvd')
    # construct reduced matrix
    reduced_matrix = u.T @ y_data @ v.T @ np.diag(np.reciprocal(s))
    # find eigenvalues
    eigenvalues, eigenvectors = lin.eig(reduced_matrix, overwrite_a=True,
                                        check_finite=False)
    # sort eigenvalues
    ind = np.argsort(eigenvalues)[::-1]
    dmd_eigenvalues = eigenvalues[ind]
    # compute modes
    dmd_modes = y_data @ v.T @ np.diag(np.reciprocal(s)) \
    @ eigenvectors[:, ind] @ np.diag(np.reciprocal(dmd_eigenvalues))
    # overlap matrix
    nm = np.size(dmd_eigenvalues)
    S = np.zeros((nm,nm),dtype=complex)
    for j in range(nm):
        for k in range(nm):
            S[j][k]=np.vdot(dmd_modes[:,j],dmd_modes[:,k])
    # invert S
    Sinv = lin.inv(S)
    return S, Sinv, dmd_eigenvalues, dmd_modes


nt=30
npt=50
nmo=4
wx=np.pi/npt
wy=np.pi/npt + .5
dz=4/npt
z=(np.arange(npt)-npt/2)*dz
# generate time dependent data
data = np.zeros((npt,npt,nt))
for k in range(nt):
    rx=0.7*np.cos(k*wx)
    ry=0.3*np.cos(k*wy)
    for i in range(npt):
        for j in range(npt):
            data[i][j][k] = gau(z[i],rx) * gau(z[j],ry)
# visualize time-dependent data
nt2=nt
stri=np.int(nt/nt2)
f = plt.figure(figsize=plt.figaspect(1.))
for j in range(nt2):
```

```
    i=j*stri
    ax = f.add_subplot(1, 1, 1, aspect=0.5)
    ax.imshow(np.real(data[:, :,i]), cmap='jet')
    plt.axis('off')
    plt.pause(.25)
    plt.clf()
# construct tensors y and x corresponding to y = A x
number_of_snapshots = data.shape[-1] - 1
x = data[:, :, 0:number_of_snapshots].reshape(\
    data.shape[0] * data.shape[1], number_of_snapshots)
y = data[:, :, 1:number_of_snapshots + 1].reshape(\
    data.shape[0] * data.shape[1], number_of_snapshots)

# apply exact DMD
S, Sinv, eigenvalues_dmd, modes_dmd = dmd_exact(x, y)
# Check S*Sinv
#print("ov=",S.dot(Sinv)

# reshape result for comparison
modes_dmd = modes_dmd.reshape([data.shape[0], data.shape[1], \
 number_of_snapshots])

# plot 4 modes
f = plt.figure(figsize=plt.figaspect(1.75))
for j in range(nmo):
    ax = f.add_subplot(2, 2, j + 1, aspect=0.5)
    ax.imshow(np.real(modes_dmd[:, :,j]), cmap='jet')
    plt.axis('off')
    ev = eigenvalues_dmd[j]
    plt.title(r'$\lambda ~ = ~ $' + str("%.2f" % np.real(ev)) + '+' + \
                  str("%.2f" % np.imag(ev)) + r'$i$')

# tt DMD implementation
# construct x and y tensors and convert to TT format
x = TT(data[:, :, 0:number_of_snapshots, None, None, None])
y = TT(data[:, :, 1:number_of_snapshots + 1, None, None, None])
# define lists
eps=0
eigenvalues_tdmd = [None]
modes_tdmd = [None]
# apply exact TDMD TT
eigenvalues_tdmd, modes_tdmd = tdmd.tdmd_exact(x, y, threshold=eps)
# convert to full format for comparison and plotting
modes_tdmd = modes_tdmd.full()[:, :, :, 0, 0, 0]
# plot 4 modes
ff = plt.figure(figsize=plt.figaspect(1.75))
for j in range(nmo):
    ax = ff.add_subplot(2, 2, j + 1, aspect=0.5)
    ax.imshow(np.real(modes_tdmd[:,:,j]), cmap='jet')
    plt.axis('off')
    ev = eigenvalues_tdmd[j]
    plt.title(r'$\lambda ~ = ~ $' + \
                  str("%.2f" % np.real(ev)) + '+' + \
                  str("%.2f" % np.imag(ev)) + r'$i$')
```

```
# Expansion coefficients of initial state in terms DMD modes
rx=0.7
ry=0.3
nm=np.size(eigenvalues_dmd)
nm2 = 20
#print("np.size(eigenvalues_dmd)=",nm)
ck=np.zeros(nm,dtype=complex)
for k in range(nm2):
    for i in range(npt):
        for j in range(npt):
            for jj in range(nm):
                ck[k] = ck[k] + Sinv[k][jj]*np.conjugate(modes_dmd[i,j,jj]) \
                    * gau(z[i],rx) * gau(z[j],ry)
# time-dependent reconstructed data
data = np.zeros((npt,npt,nt),dtype=complex)
for k in range(nt):
    norma=0.0
    for i in range(npt):
        for j in range(npt):
            for kk in range(nm2):
                aa = np.angle(eigenvalues_dmd[kk])
                ra = np.absolute(eigenvalues_dmd[kk])
                data[i][j][k] = data[i][j][k] + \
                    modes_dmd[i,j,kk] * ck[kk] * ra**k * np.exp(1j*k*aa)
            norma = norma + data[i][j][k] * dz**2
    print("norma2=",norma)
# visualize time-dependent reconstructed data
f = plt.figure(figsize=plt.figaspect(1.))
for j in range(nt):
    ax = f.add_subplot(1, 1, 1, aspect=0.5)
    ax.imshow(np.real(data[:, :,j]), cmap='jet')
    plt.axis('off')
    plt.pause(.25)
    plt.clf()
plt.show()
```

7.4 Hybrid Quantum-Classical Neural Network

This section shows how to include a quantum layer into a neural network and create a so-called hybrid quantum-classical neural network (QCNN), as described in the qiskit tutorial.

Figure 22 shows the implementation of a quantum hidden layer by using a quantum circuit that evolves a given input quantum state using unitary transformations parametrized by the output of a previous classical layer in the neural network. The result of measurements of the evolved quantum state provides expectation values that are used as inputs for a subsequent classical layer in the neural network. In Figure 22, σ is a nonlinear function and h_i is the value of neuron i at each hidden layer. $R(h_i)$ represents any rotation of the quantum state about an angle equal to h_i, while y is the final prediction generated from the network.

A tutorial notebook with the implementation of the hybrid QCNN with PyTorch, as applied to the classification of images of two types of digits (0 or 1) from the MNIST dataset is available at vic_hqcnn.ipynb.zip and vic_hqcnn.pdf. For simplicity, the quantum circuit evolves a single qubit and involves a single trainable parameter θ corresponding to the unitary R_y-rotation by the angle θ:

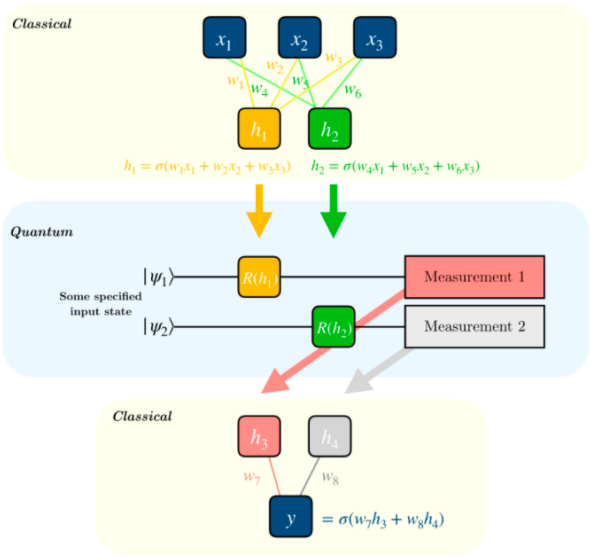


Figure 22: Schematic representation of a hybrid quantum-classical neural network where the quantum layer evolves a quantum state by using unitary transformations parametrized by the output of the previous (classical) layer, and the expectation value computed with the evolved quantum state provides the activation values for the next (classical) layer in the network.

$$R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \tag{37}$$

Since the quantum circuit involves 1 parameter, it is necessary to ensure the network condenses neurons down to size 1, which is accomplished by creating a typical CNN with two fully-connected layers. The value of the last neuron of the fully-connected layer is fed as the parameter θ into the quantum circuit. The circuit measurement then serves as the final prediction for 0 or 1, as provided by a σ_z measurement.

The tutorial also includes examples of quantum circuits for implementation of quantum algorithms that we will discuss in future lectures, that could be run on a classical quantum simulator, or on the IBM quantum computer after setting up an account.

What about backpropagation? how do we calculate gradients when the quantum circuit is involved. We can view the quantum circuit as a black box and the gradient of this black box with respect to its parameters can be calculated by finite differences, incrementing the inputs θ of the quantum circuit by $\pm s$. The gradient is simply the difference between our quantum circuit evaluated at $\theta + s$ and $\theta - s$. Thus, we can systematically differentiate our quantum circuit as part of a larger backpropogation routine. This closed form rule for calculating the gradient of quantum circuit parameters is known as the parameter shift rule.

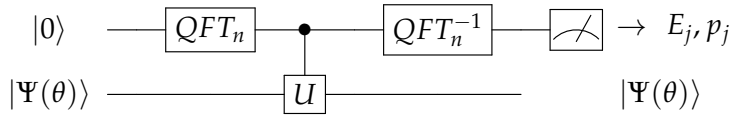
7.5 Variational Optimization with Hybrid Neural Networks

The goal of this subsection is to explain how to use the hybrid quantum-classical neural network, shown in Fig. 22, to find the ground state of a molecule described by a Hamiltonian $\hat{H}$.

The ground state is typically obtained according to the variational theorem by optimization of the parameters θ that define a trial wavefunction $|\Psi(\theta)\rangle = \hat{U}(\theta)|0\rangle$, so that the expectation value of the energy $E_\theta = \langle\Psi(\theta)|\hat{H}|\Psi(\theta)\rangle$ is as small as possible. Therefore, we can initialize the input layer as a single neuron with input $x_1 = 1$, a linear hidden layer with as many neurons as necessary parameters θ for the unitary transformations that would be applied in the quantum circuit. Each of the qubits are rotated with parameters defined by the activation of the hidden layer. The resulting quantum state is then used to compute the loss, defined as the expectation value of the energy.

7.5.1 Quantum Computation

The expectation value of the energy can be computed by *quantum phase estimation*, as described later in Sec. 16.2, using the following quantum circuit:



(38)

where $U = e^{-iH\tau/\hbar}$ and $E_\theta = \langle\Psi(\theta)|\hat{H}|\Psi(\theta)\rangle = \sum_j\ p_j\ E_j$.

7.5.2 Hybrid Quantum-Classical Computation

The expectation value of the energy can also be computed with a classical computer by computing a single point with $\Psi(\theta)$ using a standard software for electronic structure calculations (*e.g.*, Gaussian, PySCF, etc.) leading to a hybrid quantum-classical method.

7.5.3 Variational Quantum Eigensolver

A particular case of the hybrid quantum-classical methodology is the so-called variational quantum eigensolver (VQE) method, where the quantum circuit is a variational form to facilitate the implementation by using as few parameters as possible.

Implementation on the IBM Quantum: A turn-key tutorial on how to implement the VQE algorithm with Qiskit-Nature on Colab or the IBM Quantum is available as a qiskit-nature tutorial, and can be downloaded as a notebook: QiskitNature_vic.ipynb.zip, pdf. In addition, the following tutorials provide examples of QiskitNature as applied to calculations of HF electronic structure HF_vic.ipynb.zip, pdf as well as vibrational structure, pdf and 03_ground_state_solvers.zip, pdf. I recommend going through those notebooks after familiarization with the qiskit syntax explained in the beautiful online tutorials.

8 Qubits and Gates

This section introduces the representation of qubits as points on the surface of a unit sphere (the so-called Bloch sphere). The states of qubits can be represented in the basis of eigenstates $|0\rangle$ and $|1\rangle$ of the σ_z operator,

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{39}$$

with eigenvectors

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{40}$$

with eigenvalues 1 and -1, respectively, since $\sigma_z|0\rangle = |0\rangle$ and $\sigma_z|1\rangle = -|1\rangle$. Therefore, an arbitrary qubit $|\psi\rangle$ can be represented in that basis as follows:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{41}$$

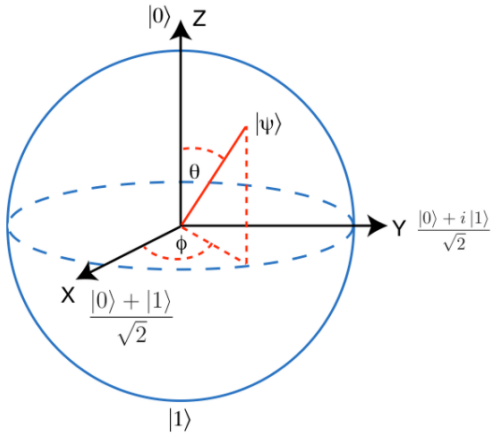


Figure 23: Bloch representation of an arbitrary qubit $|\psi\rangle$.

with α and β two complex numbers whose squares give the probability of observing the qubit in state $|0\rangle$ and $|1\rangle$, respectively. Therefore, $\alpha^2 + \beta^2 = 1$.

When both α and β are real numbers, we can represent them with a single real number $0 \leq \theta \leq \pi$, as follows: $\alpha = \cos\frac{\theta}{2}$ and $\beta = \sin\frac{\theta}{2}$. An arbitrary state can be represented with two real numbers: $0 \leq \theta \leq \pi$, and $0 \leq \phi < 2\pi$, as follows (Fig. 23):

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle = \begin{pmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2}\,e^{i\phi} \end{pmatrix}. \tag{42}$$

Therefore, all qubit states can be represented by a point on the sphere of unit radius, while the operators are represented by the axes. For example, the state $|0\rangle$ is represented by the point on the z axis with $z = 1$, while the state $|1\rangle$ is also on the z axis with $z = -1$.

8.1 Single qubit gates

Single qubit gates are unitary transformations represented by 2×2 matrices that rotate the qubit on the Bloch sphere, without changing the norm. An example is the NOT gate $X = \sigma_x$ that converts $|0\rangle$ into $|1\rangle$, and viceversa. For example,

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{43}$$

Another example is the Hadamard gate $H = (\sigma_x + \sigma_z)/\sqrt{2}$,

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{44}$$

that transforms $|0\rangle$ into $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and $|1\rangle$ into $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Therefore, the Hadamard gate transforms $|s_1\rangle$ with $s \in \{0,1\}$, as follows:

$$\begin{aligned} H|s_1\rangle &= 2^{-1/2}\left(|0\rangle + (-1)^{s_1}|1\rangle\right), \\ &= 2^{-1/2}\sum_{x\in\{0,1\}}(-1)^{s_1\cdot x}|x\rangle. \end{aligned} \tag{45}$$

8.2 Rotations

The Pauli matrices σ_x, σ_y and σ_z define the rotation operators,

$$R_r(\theta) = e^{-i\frac{\theta}{2}\sigma_r} = \cos\left(\frac{\theta}{2}\right)\ I - i\,\sin\left(\frac{\theta}{2}\right)\ \sigma_r \tag{46}$$

that rotate any qubit state by an angle θ about the axis $r = \{x,y,z\}$. Equation (46) can be obtained by Taylor expansion, since the square of a Pauli matrix is equal to the identity (*i.e.*, $\sigma_r^2 = I$). Therefore, $e^{-i\frac{\theta}{2}\sigma_r} = \cos(\frac{\theta}{2}\sigma_r) - i\,\sin(\frac{\theta}{2}\sigma_r)$, with $\cos(\frac{\theta}{2}\sigma_r) = 1 - \frac{1}{2!}\left(\frac{\theta}{2}\sigma_r\right)^2 + \frac{1}{4!}\left(\frac{\theta}{2}\sigma_r\right)^4 + \cdots = 1 - \frac{1}{2!}\left(\frac{\theta}{2}\right)^2 + \frac{1}{4!}\left(\frac{\theta}{2}\right)^4 + \cdots = \cos(\frac{\theta}{2})$. Analogously, $\sin(\frac{\theta}{2}\sigma_r) = \frac{\theta}{2}\sigma_r - \frac{1}{3!}\left(\frac{\theta}{2}\ \sigma_r\right)^3 + \frac{1}{5!}\left(\frac{\theta}{2}\ \sigma_r\right)^5 + \cdots = \frac{\theta}{2} - \frac{1}{3!}\left(\frac{\theta}{2}\right)^3\ \sigma_r + \frac{1}{5!}\left(\frac{\theta}{2}\right)^5\ \sigma_r + \cdots = \sin(\frac{\theta}{2})\ \sigma_r$.

As an example, we consider the rotation of $|0\rangle$ by an angle θ about the y axis, implemented as follows:

$$e^{-i\frac{\theta}{2}\sigma_y}|0\rangle = \cos\frac{\theta}{2}\ |0\rangle - i\,\sin\frac{\theta}{2}\ \sigma_y\ |0\rangle \tag{47}$$

or

$$e^{-i\frac{\theta}{2}\sigma_y}|0\rangle = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{48}$$

In particular, when $\theta = \pi/2$, the state $|0\rangle$ that points along the z axis (as shown in Fig. 23) is rotated about the y axis by $\pi/2$ so it becomes $|+\rangle$ pointing along the x axis (*i.e.*, the eigenstate of σ_x with eigenvalue equal to 1), as follows:

$$e^{-i\frac{\pi}{4}\sigma_y}|0\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle \tag{49}$$

Note that the rotation of $|0\rangle$ by $\theta = -\pi/2$ generates the state $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, corresponding the eigenstate of σ_x with eigenvalue equal to -1. Analogously, rotation of the state $|1\rangle$ by $\theta = \pi/2$ generates the state $-|-\rangle = e^{\pm i\pi}|-\rangle$, corresponding to minus the eigenstate of σ_x with eigenvalue equal to -1 (also pointing in the same direction as $|-\rangle$, although it is multiplied by a global phase $e^{\pm i\pi}$):

$$e^{-i\frac{\pi}{4}\sigma_y}|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}\left(-\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \frac{1}{\sqrt{2}}(-|0\rangle + |1\rangle) = -|-\rangle \tag{50}$$

We note that rotation operators can be computed with a variety of gates. For example, we can implement a rotation about the y axis, as follows: $R_y(-\pi/2) = H\sigma_x$, since

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \tag{51}$$

In the previous section, we showed that a state in the basis of σ_x, (i.e., $|+\rangle$ and $|-\rangle$) can be rotated to the basis of σ_z by applying the Hadamard gate. Analogously, a state in the basis of σ_y can be rotated to the basis of σ_z by applying the following gate:

$$R_x(\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \tag{52}$$

Note that a set of eigenvectors of σ_y can be defined as $|y^+\rangle = (|0\rangle + i|1\rangle)/\sqrt{2}$ and $|y^-\rangle = (i|0\rangle + |1\rangle)/\sqrt{2}$ since $\sigma_y|y^+\rangle = |y^+\rangle$ and $\sigma_y|y^-\rangle = -|y^-\rangle$:

$$\sigma_y|y^+\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 \\ i \end{pmatrix} = \begin{pmatrix} 1 \\ i \end{pmatrix} = |y^+\rangle, \tag{53}$$

and

$$\sigma_y|y^-\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} i \\ 1 \end{pmatrix} = \begin{pmatrix} -i \\ -1 \end{pmatrix} = -|y^-\rangle, \tag{54}$$

Those eigenstates of σ_y are transformed into eigenstates of σ_z, as follows:

$$R_x(\pi/2)|y^+\rangle = \frac{1}{2} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \begin{pmatrix} 1 \\ i \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle, \tag{55}$$

and

$$R_x(\pi/2)|y^-\rangle = \frac{1}{2} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \begin{pmatrix} i \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle. \tag{56}$$

Analogously, we can rotate a state in the basis of $|0\rangle$ and $|1\rangle$ to the basis of $|y^+\rangle$ and $|y^-\rangle$ by applying the rotation matrix,

$$R_x(-\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix}, \tag{57}$$

which is the inverse of $R_x(\pi/2)$. Therefore, σ_y can be applied according to the following similarity transformation, $\sigma_y = R_x(-\pi/2)\sigma_z R_x(\pi/2)$ since

$$R_x(-\pi/2)\sigma_z R_x(\pi/2) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}. \tag{58}$$

8.3 Multiple qubits

The state of a register with mutliple qubits is defined by the tensor product of the states of the qubits. A simple example is the state of a register with only two qubits in states $|0\rangle$ and $|1\rangle$, respectively. In that case, the state of the register is the $|0\rangle \otimes |1\rangle$ which can also be written as $|0\rangle|1\rangle$, or simply as $|01\rangle$. In vector representation, it is written as follows:

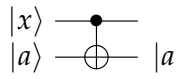
$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{59}$$

8.3.1 The CNOT gate

The controlled NOT gate, or CNOT gate, is defined by the following matrix:[2]

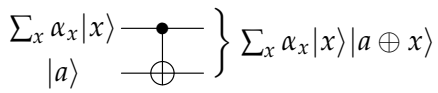
$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{60}$$

Note that CNOT is a 2-qubit gate that flips the second qubit (*i.e.*, the target qubit) only when the first one (*i.e.*, the control qubit) is $|1\rangle$ by applying a sum mod(2), as follows: [3]



(61)

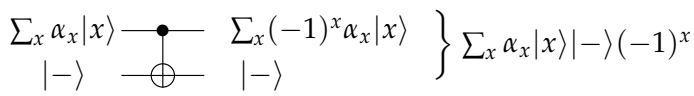
It is essential in the construction of quantum circuits since it can entangled qubits, as follows:



(62)

Note that for the resulting superposition state, measurement of the first qubit determines the outcome of a measurement of the second qubit, and viceversa.

8.3.1.1 Phase kickback: An interesting case is when the ancilla qubit $|a\rangle$ is initialized as $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$. Remember that the NOT gate transforms $|0\rangle$ into $|1\rangle$, and $|1\rangle$ into $|0\rangle$, so the NOT gate transforms $|-\rangle$ into $-|-\rangle$. As a result, the CNOT gate flips the sign of the second qubit only when the first one is $|1\rangle$, so the outcome is, as follows:



(63)

Note that the second qubit remains unchanged since the phase is kicked back to the first qubit, the so-called phase 'kickback' trick implemented in many quantum algorithms.

8.3.1.2 Conditional gates: correlation functions Conditional gates are generalizations of the CNOT gate, where a gate is applied to a second set of qubits only when the control qubit is $|1\rangle$ (represented by a black circle in Fig. 24), or only when the control qubit is $|0\rangle$ (represented by a white circle in Fig. 24). Here, we show that conditional gates A and B can be applied to measure the correlation function,

$$C_{AB} = \langle \Psi_0 | A^\dagger B | \Psi_0 \rangle, \tag{64}$$

by measuring the expectation value $\langle 2\sigma^+ \rangle$ for an ancilla qubit according to the circuit shown in Fig. 24 (panel a).

[2] **Implementation on the IBM Quantum:** A turn-key tutorial on how to define quantum circuits with Qiskit on Colab or the IBM Quantum is available as a Qiskit tutorial, and can be downloaded as a notebook CNOT_vic.ipynb, and CNOT_vic.pdf.

[3] **The CNOT gate cannot be constructed by single qubit gates:** This statement is important because it shows that single qubit gates are not universal, since at least the CNOT cannot be computed by single qubit gates. **One line Proof:** Assume single qubit operators are universal. Then there must exist some $A \otimes B = CNOT = (P_0 \otimes I) + (P_1 \otimes X)$, with $P_0 = |0\rangle\langle 0|$ and $P_1 = |1\rangle\langle 1|$. Therefore, $A_{00}B = I, A_{11}B = X$, and $A_{01}B = A_{10}B = 0$. Since $A_{00}B = I$ then A_{00} must be 1, and $B = I$. But, $A_{11}B = X$, so $A_{11} = 1$ and $B = X$, in contradiction to the previous statement. So, there is no set of single qubit gates such that $A \otimes B = CNOT$.

This can be shown by considering that $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and the initial state is $||\Psi_i\rangle = |+\rangle|\Psi_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle|\Psi_0\rangle + |1\rangle|\Psi_0\rangle)$. Thus, the final state is $||\Psi_f\rangle = \frac{1}{\sqrt{2}}(|0\rangle A|\Psi_0\rangle + |1\rangle B|\Psi_0\rangle)$. Therefore, the measurement of the ancilla qubit with respect to $2\sigma^+ = \sigma_x + i\sigma_y$, can be computed as follows:

$$\langle 2\sigma^+\rangle = \frac{2}{\sqrt{2}}(\langle\Psi_f||\sigma^+|0\rangle A|\Psi_0\rangle + \langle\Psi_f||\sigma^+|1\rangle B|\Psi_0\rangle), \tag{65}$$

with $\sigma^+|0\rangle = 0$, and $\sigma^+|1\rangle = |0\rangle$. So,

$$\begin{aligned}\langle 2\sigma^+\rangle &= \frac{2}{\sqrt{2}}\langle\Psi_f||0\rangle B|\Psi_0\rangle, \\ &= \langle\Psi_0|A^\dagger B|\Psi_0\rangle.\end{aligned} \tag{66}$$

Analogously, we can define $B(t_1) = U_{t_1}^\dagger B U_{t_1}$ and $A(t_2) = U_{t_2}^\dagger A U_{t_2}$ (where $U_t = e^{-\frac{i}{\hbar}\hat{H}t}$) to obtain:

$$\begin{aligned}C_{AB}(t_1,t_2) &= \langle\Psi_0|A(t_2)B(t_1)|\Psi_0\rangle, \\ &= \langle\Psi_0|U_{t_2}^\dagger A U_{t_2} U_{t_1}^\dagger B U_{t_1}|\Psi_0\rangle, \\ &= \langle\Psi_0|U_{12}^\dagger A U_{12} B|\Psi_0\rangle,\end{aligned} \tag{67}$$

with $U_{12} = U_{t_2-t_1}$. So, $C_{AB}(t_1,t_2)$ can be determined according to the circuit shown in Fig. 24 (panel b) by measuring the expectation value $\langle 2\sigma^+\rangle$ for an ancilla qubit. Note that both circuits shown in panel (b) are equivalent since U_{12} can be applied in two consecutive steps, by first conditioning the first qubit to be $|1\rangle$ and then $|0\rangle$ (top, b), or unconditionally (bottom, b). Here, we note the final state is $||\Psi_f\rangle = \frac{1}{\sqrt{2}}(|0\rangle A^\dagger U_{12}|\Psi_0\rangle + |1\rangle U_{12} B|\Psi_0\rangle)$. So, the measurement of the ancilla qubit with respect to $2\sigma^+ = \sigma_x + i\sigma_y$ is computed analogously, as follows:

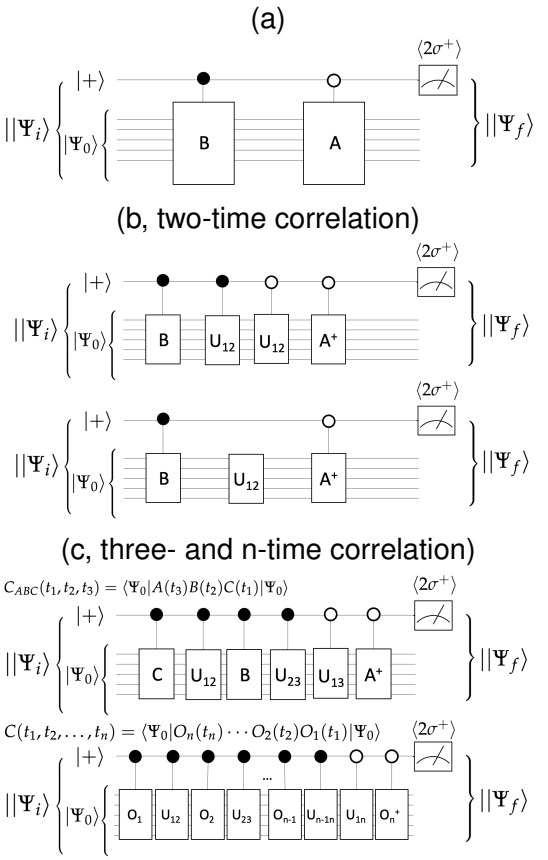


Figure 24: Measurement of the correlation function $C_{AB} = \langle\Psi_0|A^\dagger B|\Psi_0\rangle$ (panel a), the two-time correlation function $C_{AB}(t_1,t_2) = \langle\Psi_0|A(t_2)B(t_1)|\Psi_0\rangle$ (panel b), and the n-time correlation function $C(t_1,t_2,\ldots,t_n) = \langle\Psi_0|O_n(t_n)\cdots O_2(t_2)O_1(t_1)|\Psi_0\rangle$ (panel c) by measuring $\langle 2\sigma^+\rangle$ of the ancilla.

$$\langle 2\sigma^+\rangle = \frac{2}{\sqrt{2}}(\langle\Psi_f||\sigma^+|0\rangle A^\dagger U_{12}|\Psi_0\rangle + \langle\Psi_f||\sigma^+|1\rangle U_{12} B|\Psi_0\rangle), \tag{68}$$

with $\sigma^+|0\rangle = 0$, and $\sigma^+|1\rangle = |0\rangle$. So,

$$\begin{aligned}\langle 2\sigma^+\rangle &= \frac{2}{\sqrt{2}}\langle\Psi_f||0\rangle U_{12} B|\Psi_0\rangle, \\ &= \langle\Psi_0|U_{12}^\dagger A U_{12} B|\Psi_0\rangle.\end{aligned} \tag{69}$$

Next, we define $C(t_1) = U_{t_1}^\dagger C U_{t_1}$ $B(t_2) = U_{t_2}^\dagger B U_{t_2}$ and $A(t_3) = U_{t_3}^\dagger A U_{t_3}$ to obtain:

$$\begin{aligned}C_{ABC}(t_1,t_2,t_3) &= \langle\Psi_0|A(t_3)B(t_2)C(t_1)|\Psi_0\rangle, \\ &= \langle\Psi_0|U_{t_3}^\dagger A U_{t_3} U_{t_2}^\dagger B U_{t_2} U_{t_1}^\dagger C U_{t_1}|\Psi_0\rangle, \\ &= \langle\Psi_0|U_{13}^\dagger A U_{23} B U_{12} C|\Psi_0\rangle,\end{aligned} \tag{70}$$

with $U_{12} = U_{t_2-t_1}$, $U_{23} = U_{t_3-t_2}$ and $U_{13} = U_{t_3-t_1}$. Therefore, the final state is $||\Psi_f\rangle = \frac{1}{\sqrt{2}}(|0\rangle A^\dagger U_{13}|\Psi_0\rangle + |1\rangle U_{23}BU_{12}C|\Psi_0\rangle)$. So, the measurement of the ancilla qubit with respect to $2\sigma^+ = \sigma_x + i\sigma_y$, gives (Fig. 24, c):

$$\begin{aligned}\langle 2\sigma^+\rangle &= \frac{2}{\sqrt{2}}(\langle\Psi_f||\sigma^+|0\rangle A^\dagger U_{13}|\Psi_0\rangle + \langle\Psi_f||\sigma^+|1\rangle U_{23}BU_{12}C|\Psi_0\rangle),\\ &= \langle\Psi_0|U_{13}^\dagger A U_{23}BU_{12}C|\Psi_0\rangle.\end{aligned} \tag{71}$$

The n-time correlation function, $C(t_1, t_2, \ldots, t_n) = \langle\Psi_0|O_n(t_n)\cdots O_2(t_2)O_1(t_1)|\Psi_0\rangle$, is obtained as follows (Fig. 24, c):

$$\begin{aligned}\langle 2\sigma^+\rangle &= \frac{2}{\sqrt{2}}(\langle\Psi_f||\sigma^+|0\rangle O_n^\dagger U_{1n}|\Psi_0\rangle + \langle\Psi_f||\sigma^+|1\rangle U_{n-1n}O_{n-1}\cdots U_{23}O_2U_{12}O_1|\Psi_0\rangle),\\ &= \langle\Psi_0|U_{1n}^\dagger O_n U_{n-1n}O_{n-1}\cdots U_{23}O_2U_{12}O_1|\Psi_0\rangle.\end{aligned} \tag{72}$$

8.3.2 Hadamard Gate

The Hadamard gate for multiple qubits transforms each of the qubits according to the Hadamard gate. A simple example is the Hadamard transformation for a register with only $n = 2$ qubits for which the Hadamard gate is represented, as follows: $H^{\otimes 2} = H \otimes H$. When applied to a register initialized in the $|00\rangle$ state, we obtain:

$$\begin{aligned}H^{\otimes 2}|00\rangle &= H|0\rangle \otimes H|0\rangle = |+\rangle \otimes |+\rangle,\\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle),\\ &= \frac{1}{\sqrt{2}}\sum_{x_1\in\{0,1\}}|x_1\rangle \otimes \frac{1}{\sqrt{2}}\sum_{x_2\in\{0,1\}}|x_2\rangle = \frac{1}{2^{n/2}}\sum_{x_1,x_2\in\{0,1\}}|x_1x_2\rangle,\\ &= \frac{1}{2^{n/2}}\sum_{x\in\{0,1\}^2}|x\rangle.\end{aligned} \tag{73}$$

Analogously, for $n = 2$ with $|s\rangle = |s_1s_2\rangle$, we obtain that according to Eq. (45):

$$\begin{aligned}H^{\otimes 2}|s\rangle &= H^{\otimes 2}|s_1s_2\rangle = H|s_1\rangle \otimes H|s_2\rangle,\\ &= \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{s_1}|1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{s_2}|1\rangle),\\ &= \frac{1}{\sqrt{2}}\sum_{x_1\in\{0,1\}}(-1)^{s_1\cdot x_1}|x_1\rangle \otimes \frac{1}{\sqrt{2}}\sum_{x_2\in\{0,1\}}(-1)^{s_2\cdot x_2}|x_2\rangle,\\ &= \frac{1}{2^{n/2}}\sum_{x_1,x_2\in\{0,1\}}(-1)^{s_1\cdot x_1 + s_2\cdot x_2}|x_1x_2\rangle,\\ &= \frac{1}{2^{n/2}}\sum_{x\in\{0,1\}^2}(-1)^{s\cdot x}|x\rangle.\end{aligned} \tag{74}$$

For n qubits, the Hadamard gate transforms the state $|00\cdots0\rangle$, or simply as $|0\rangle$, as follows:

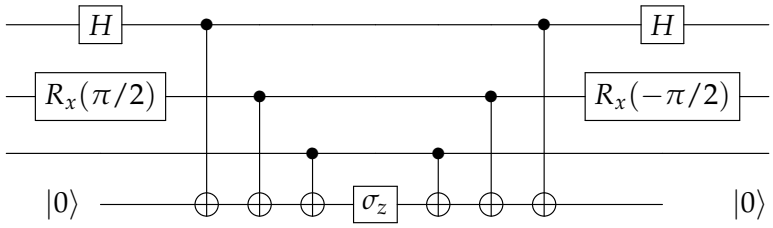
$$H^{\otimes n}|0\rangle = \frac{1}{2^{n/2}}\sum_{x\in\{0,1\}^n}|x\rangle \tag{75}$$

Also, it is important to note that the Hadamard gate is its own inverse, so $H^{\otimes n}H^{\otimes n} = 1$ and therefore the Hadamard transform of a uniform superposition generates the state $|0\rangle$, as follows:

$$H^{\otimes n}|0\rangle = H^{\otimes n}\frac{1}{2^{n/2}}\sum_{x\in\{0,1\}^n}|x\rangle = |0\rangle \tag{76}$$

8.3.3 Tensor Product of Pauli Matrices

As an example of the application of Hadamard and rotation matrices, we show below a circuit for applying the tensor product of 3 Pauli matrices $\sigma_x \otimes \sigma_y \otimes \sigma_z$ to an arbitrary state of 3 qubits:



(77)

Note that we have transformed σ_x and σ_y to σ_z operations, according to the following substitutions: $\sigma_x = H\sigma_z H$, and $\sigma_y = R_x(-\pi/2)\sigma_z R_x(\pi/2)$. Also, the same circuit can be used to apply any function $f(\sigma_x \otimes \sigma_y \otimes \sigma_z)$ products of Pauli matrices simply by applying $f(\sigma_z)$ to the ancilla qubit instead of applying σ_z.

9 Grover's Algorithm

The goal of this section is to introduce Grover's algorithm , an algorithm that can find a specific state out of N possibilities in $\sqrt{N}$ steps. Considering that a classical search algorithm would have to check one by one each of the possible states, it would take on average $N/2$ steps and $N-1$ steps in the worst case. So, the Grover's algorithm is quadratically faster than a classical search. It is particularly useful when there is a large number of possible states (so the target state is hard to find 'the needle in the haystack') although it is easy to check whether a given state is the target state of interest, or not. For that case, one can define a function $f(x) = 1$ when x is the desired state ('the needle') and $f(x) = 0$, otherwise. With $f(x)$, we can define the so-called 'oracle' operator $\hat{O} = e^{i\pi f(\hat{x})}$ that changes the sign of the component along the target state, as follows: $\hat{O}|x\rangle = -|x\rangle$ when $|x\rangle$ is the desired state. Otherwise, it leaves the state unchanged: $\hat{O}|x\rangle = |x\rangle$.

Algorithm: Starting with a uniform superposition $|s\rangle = N^{-1/2}\sum_{x=1}^{N}|x\rangle$ of all possible states $|x\rangle$, the algorithm repeatedly applies the oracle followed by the so-called diffusion operator,

$$|\hat{D}\rangle = 2|s\rangle\langle s| - I, \tag{78}$$

that changes the sign of the component perpendicular to $|s\rangle$.

Simple Example: A simple example is a system with $N = 4$ possible states, prepared in the uniform superposition

$$|s\rangle = \frac{1}{\sqrt{N}}(|00\rangle + |10\rangle + |01\rangle + |11\rangle), \tag{79}$$

with an oracle designed to change the sign of the $|11\rangle$ component (*e.g.*, $\hat{O} = I - 2|11\rangle\langle 11|$), as follows:

$$\hat{O}|s\rangle = \frac{1}{\sqrt{N}}(|00\rangle + |10\rangle + |01\rangle - |11\rangle), \tag{80}$$

Remarkably, the state $\hat{D}\hat{O}|s\rangle$ is more aligned with the target state $|11\rangle$ than the initial state $|s\rangle$, as shown geometrically in Fig. (25). The procedure is repeated $\sqrt{N}$ times to maximize the alignment with the desired state.

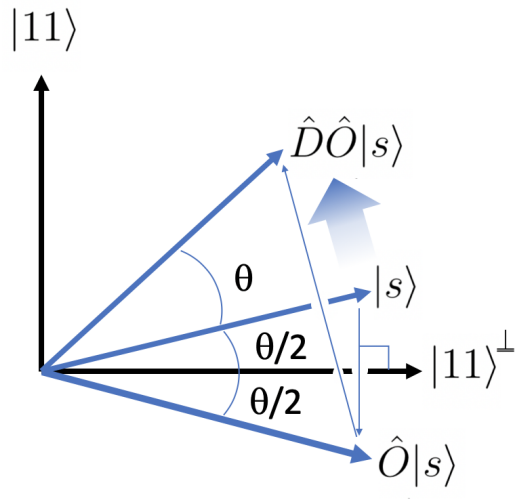


Figure 25: Geometric representation of the first iteration of Grover's algorithm. The initial superposition state $|s\rangle$ is rotated by an angle θ towards the target vector $|11\rangle$ by first applying the oracle $\hat{O}$ that changes the sign of the component along the direction of the target state $|11\rangle$, and then the diffusion operator $\hat{D}$ that changes the sign of the component perpendicular to the uniform superposition $|s\rangle$.

Number of queries: For the simple example of 4 states, introduced above, Grover's algorithm transforms the initial superposition into the target state $|11\rangle$ in one step, thus outperforming a classical search that would need on average 2 steps –*i.e.*, $\log_2(\#states)$. We note that

$$\langle 11|s\rangle = \frac{1}{\sqrt{N}}, \tag{81}$$

so

$$\hat{O}|s\rangle = |s\rangle - \frac{2}{\sqrt{N}}|11\rangle, \tag{82}$$

and

$$\hat{D}\left(|s\rangle - \frac{2}{\sqrt{N}}|11\rangle\right) = (2|s\rangle\langle s| - I)\left(|s\rangle - \frac{2}{\sqrt{N}}|11\rangle\right). \tag{83}$$

Therefore,

$$\begin{aligned}\hat{D}\hat{O}|s\rangle &= (2|s\rangle\langle s|)\left(|s\rangle - \tfrac{2}{\sqrt{N}}|11\rangle\right) - \left(|s\rangle - \tfrac{2}{\sqrt{N}}|11\rangle\right) &(84)\\ &= \left(2|s\rangle - \tfrac{4}{N}|s\rangle\right) - |s\rangle + \tfrac{2}{\sqrt{N}}|11\rangle &(85)\\ &= \tfrac{N-4}{N}|s\rangle + \tfrac{2}{\sqrt{N}}|11\rangle &(86)\end{aligned}$$

so for a system with 2 qubits, $N = 4$, we obtain $(\hat{D}\hat{O})^n|s\rangle = |11\rangle$ when $n = 1$.

More generally we note that, according to Fig. 25, the projection of $(\hat{D}\hat{O})^n|s\rangle$ along the direction of $|11\rangle$ is $\sin((2n+1)\frac{\theta}{2})$, with $\frac{\theta}{2} = \frac{1}{\sqrt{N}}$. To maximize the projection, we need to have $(2n+1)\frac{\theta}{2} = \frac{\pi}{2}$. So, $(2n+1)\frac{1}{\sqrt{N}} = \frac{\pi}{2}$ and when n is large, $2n\frac{1}{\sqrt{N}} \approx \frac{\pi}{2}$, giving $n \approx \frac{\pi\sqrt{N}}{4}$. So, remarkably, $n \approx \sqrt{N}$ when N is large –*i.e.*, quadratically faster than the classical search where n is of order N.

Another way of showing that $n \approx \sqrt{N}$ for large N is by considering how much the amplitude of the target state x^* is amplified by one step of the algorithm, when its amplitude is $\alpha_{x^*} = 2^{-1/2}$ and therefore the average amplitude is $\bar{\alpha} \approx (2^{-1/2} + N2^{-1/2}N^{-1/2})/N \approx (2N)^{-1/2}$ since the average amplitude of all other states $x^* \neq x$ is $2^{-1/2}(N-1)^{-1/2} \approx (2N)^{-1/2}$. Upon applying the oracle, we obtain $\alpha_{x^*} = -2^{-1/2}$, as shown in Fig. (26), bottom panel and inverting about the mean $\bar{\alpha}$ by applying $\hat{D}$, we increase its amplitude to $2^{-1/2} + 2\bar{\alpha}$ (*i.e.*, we increment by $2\bar{\alpha} = 2(2N)^{-1/2} = (N/2)^{-1/2}$). Therefore, the number of steps necessary to reach an amplitude of $2^{-1/2}$ by increments of the order of $(N/2)^{-1/2}$ is $n = 2^{-1/2}/(N/2)^{-1/2} = \sqrt{N}$. So, the algorithm finds the solution with 50% probability in $\mathcal{O}(\sqrt{N})$ steps.

Inversion about the mean: As shown below, the effect of the diffusion operator on an arbitrary state $|\psi\rangle = \sum_x \alpha_x|x\rangle$, is to change the sign of the component perpendicular to the initial superposition state which is equivalent to inverting the amplitudes relative to their mean value (*i.e.*, amplitude inversion about the mean), as follows:

$$\hat{D}\sum_{x=1}^{N}\alpha_x|x\rangle = \sum_{x=1}^{N}(2\bar{\alpha} - \alpha_x)|x\rangle, \tag{87}$$

with $\bar{\alpha} = N^{-1}\sum_{j=1}^{N}\alpha_j$ the amplitude mean value. According to Eq. (87), $\hat{D}$ inverts the amplitudes about the mean since $(\alpha_x - \bar{\alpha}) = -((2\bar{\alpha} - \alpha_x) - \bar{\alpha})$ (Fig. 26). To obtain Eq. (87), we consider the uniform superposition $|s\rangle = N^{-1/2}\sum_{x=1}^{N}|x\rangle$, so $\hat{D} = 2|s\rangle\langle s| - I = 2N^{-1}\sum_{x=1}^{N}\sum_{y=1}^{N}|y\rangle\langle x| - I$, and

$$\begin{aligned}\hat{D}|\psi\rangle &= 2N^{-1}\sum_{x=1}^{N}\sum_{y=1}^{N}|x\rangle\sum_{x'=1}^{N}\alpha_{x'}\langle y|x'\rangle - \sum_{x'=1}^{N}\alpha_{x'}|x'\rangle,\\ &= \sum_{x=1}^{N}|x\rangle 2N^{-1}\sum_{y=1}^{N}\alpha_y - \sum_{x=1}^{N}\alpha_x|x\rangle, &(88)\\ &= \sum_{x=1}^{N}(2\bar{\alpha} - \alpha_x)\,|x\rangle.\end{aligned}$$

Quantum Circuit: The oracle $\hat{O} = (-1)^{f(j)}$ produces a phase inversion on the amplitude of the target state while leaving all the other amplitudes unchanged. It can be implemented by using the so-called *phase kickback* algorithm, according to the following circuit:

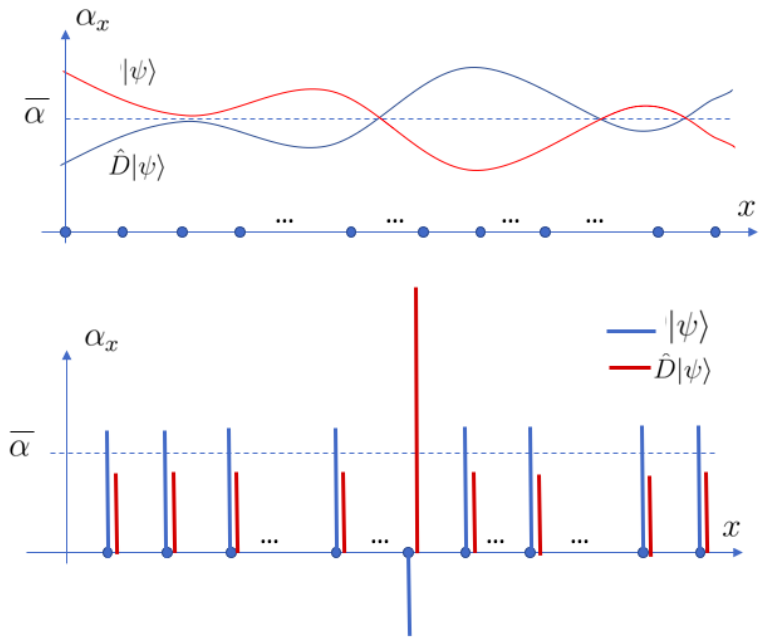


Figure 26: Two examples showing how $\hat{D}$ inverts the amplitudes α_x about the mean $\bar{\alpha}$.

Figure 27: Quantum circuit exploiting the phase kickback algorithm for implementing the unitary U_f corresponding to the *conditional phase inversion* oracle $\hat{O} = (-1)^{f(x)}$ using $|b\rangle = |-\rangle$ as the second set of qubits. The state $|x\rangle|b\rangle$ is transformed by the unitary, as follows: $U_f|x\rangle|b\rangle = |x\rangle|f(x)\oplus b\rangle$. When $|b\rangle = |-\rangle$ (right panel), we obtain: $U_f|x\rangle|-\rangle = |x\rangle(-1)^{f(x)}|-\rangle$ since b is unchanged when $f(x) = 0$ (*i.e.*, $f(x)\oplus b = b$) while $f(x) = 1$ transforms the state $|-\rangle = 2^{-1/2}(|0\rangle - |1\rangle)$ into $2^{-1/2}(|f(x)\oplus 0\rangle - |f(x)\oplus 1\rangle) = 2^{-1/2}(|1\rangle - |0\rangle) = (-1)|-\rangle$.

Uniform superposition: The uniform superposition can be prepared by starting with all qubits in the state $|0\rangle$ and applying the Hadamard operator to each qubit, as follows: $\hat{H}^{\otimes 2}|00\rangle = (\hat{H}_1 \otimes \hat{H}_1)|00\rangle$, since the Hadamard operator $\hat{H}_1 = \frac{1}{\sqrt{2}}(\hat{\sigma}_z + \hat{\sigma}_x)$ transforms $|0\rangle$ into the symmetric linear combination $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The Hadamard transformation is also its own inverse, so applying it to the uniform superposition generates the state with all qubits in state $|0\rangle$. Furthermore, applying it to an arbitrary state, the Hadamard transform rotates it leaving the component of the uniform superposition pointing along the direction $|0\rangle$. The resulting rotation can be exploited to apply the diffusion operator to an arbitrary state by first applying the Hadamard transform (or inverse quantum Fourier transform) to have the component of the uniform superposition pointing along the direction of $|0\rangle$, inverting the phases of all components orthogonal to $|0\rangle$ and then applying the Hadamard transform (or quantum Fourier transform) to leave the state with its original orientation, according to the following circuit:

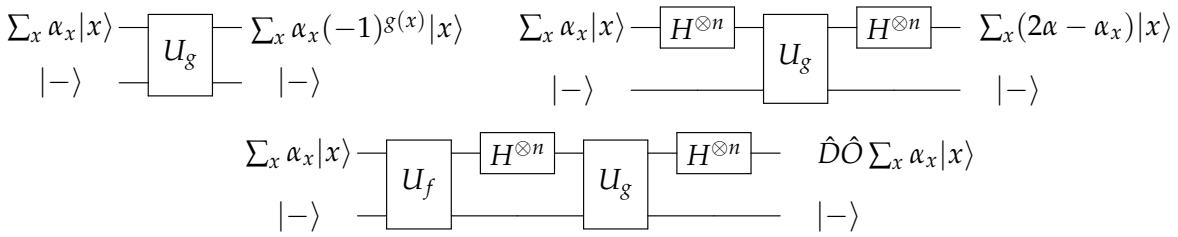
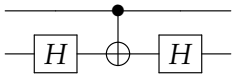


Figure 28: Top Left: Quantum circuit for inversion of the phases of all components orthogonal to $|0\rangle$ by implementing the unitary $U_g = (-1)^{g(x)}$, where $g(x) = 0$ if $|x\rangle = |00\cdots0\rangle$ and $g(x) = 1$, otherwise. Top Right: Quantum circuit for inverting the amplitudes about their mean, according to the diffusion operator $\hat{D} = 2|s\rangle\langle s| - I$, by first applying a Hadamard gate that rotates the state to leave the component of the uniform superposition pointing along the direction $|0\rangle$, applying U_g to invert the phases of all other components, and then applying a Hadamard that rotates the state back to the original orientation. Bottom: Quantum circuit for implementation of one step of the Grover's algorithm.

Oracle operator: To explain how to construct $\hat{O}$ in terms of unitary operators for the example described above, we note that

$$-|1\rangle = \hat{H}_1\hat{\sigma}_x\hat{H}_1|1\rangle. \tag{89}$$

since the Hadamard operator $\hat{H}_1 = \frac{1}{\sqrt{2}}(\hat{\sigma}_z + \hat{\sigma}_x)$ transforms $|1\rangle$ into the antisymmetric linear combination $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, the NOT operator $\hat{x}$ changes the sign of the state $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ (by changing $|0\rangle$ into $|1\rangle$ and $|1\rangle$ into $|0\rangle$), and applying the Hadamard operator to $|-\rangle$ returns the original state $|1\rangle$. By using CNOT instead of NOT, as shown in the circuit, below:

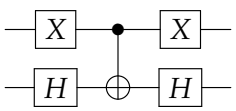


(90)

we change the sign of $|1\rangle$ in the second qubit only when it is preceded by the control qubit $|1\rangle$, as necessary when applying $\hat{O}$ to the uniform superposition, according to Eq. (80).

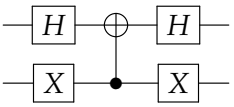
Diffusion operator: The diffusion operator changes the sign of all of the terms orthogonal to the uniform superposition. Therefore, it is possible to implement it by first orienting the state along a convenient direction where it is easier to change the sign of the orthogonal state and then rotate it back to its original orientation. For example, rotating the state $\hat{O}|s\rangle$ so that its component along the direction of the uniform superposition points along one of the computational states (*e.g.*, along the $|00\rangle$ direction) so one can change the sign of all of the terms that are orthogonal to that direction (*e.g.*, $|00\rangle$) and then rotate the resulting state back so that the component along the superposition state points back along its original direction. As mentioned above, to rotate a state so that its component along the uniform superposition points to the direction $|00\rangle$ we need to apply the Hadamard gate $H^{\otimes 2}$.

The circuit introduced by Eq. (90) changes the sign of the component along the $|11\rangle$ direction. The analogous circuit but with a NOT gate previously applied to the first qubit (and subsequently applied as well to avoid modifying that qubit) would change the sign of the term along $|01\rangle$, as follows:



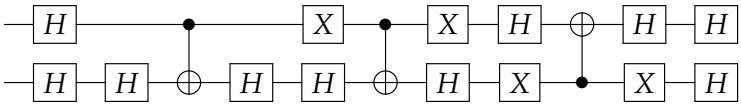
(91)

and the same circuit but with an exchanged roles for the control and target qubits would change the sign of the term associated with the direction $|10\rangle$, as follows:



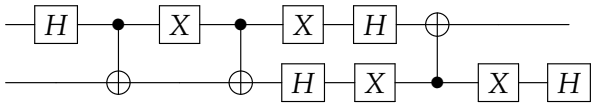
(92)

so, the complete diffusion operator $\hat{D}$ can be implemented, as follows:



(93)

which can be simplified, as follows:



(94)

since $\hat{H}\hat{H} = I$.

Implementation on the IBM Quantum: A turn- key tutorial on how to implement the Grover's algorithm with Qiskit on Colab or the IBM Quantum is available as a Qiskit tutorial, and can be downloaded as a notebook: grover_vic.ipynb.zip, and grover_vic.pdf. For a tutorial on visualization, please, check the following link, including a discussion on the Qsphere representation.

Matlab function: The gsa.m Matlab function simulates the Grover algorithm.

Grover Optimization: Grover's quantum computational search procedure can provide the basis for implementing adaptive global optimization algorithms. An example of such methods is the *Grover adaptive search* (GAS) algorithm where the global minimum of a cost function V is iteratively searched for with an adaptive oracle, as follows. Given an initial state $|j_0\rangle$ and its corresponding expectation value $V_0 = \langle j_0|V|j_0\rangle$, the oracle $\hat{O} = e^{i\pi f(j)}$ is defined with $f(j) = 1$ for states $|j\rangle$ with expectation value $\langle j|V|j\rangle < V_0$. Applying the Grover algorithm to a uniform superposition, we find a state $|j_1\rangle$ whose expectation value $V_1 < V_0$ after $r = \pi\sqrt{N}/4$ rotations. The oracle is then adapted with $f(j) = 1$ for states $|j\rangle$ with $\langle j|V|j\rangle < V_1$, and the process is iterated m times until convergence to find the global minimum state $|m\rangle$ with $V_m < V_{m-1} < \cdots < V_0$.

As an example of Grover minimization (Fig. 29), we consider the problem of finding the configuration of a conjugated polyene chain with Cartesian atomic coordinates $x_1, \cdots x_n$, assuming that bond-lengths and bending angles are known but the 1-4 dihedrals are yet to be determined since their π or $-\pi$ (*cis* or *trans*) configurations must fulfill a constraining set S of interatomic distances $d_{ij} = \|x_i - x_j\|$ determined by NMR. If all interatomic distances are determined, then the problem is trivial and can be solved in n steps. However, the problem is NP-hard when only some of the distances are known.

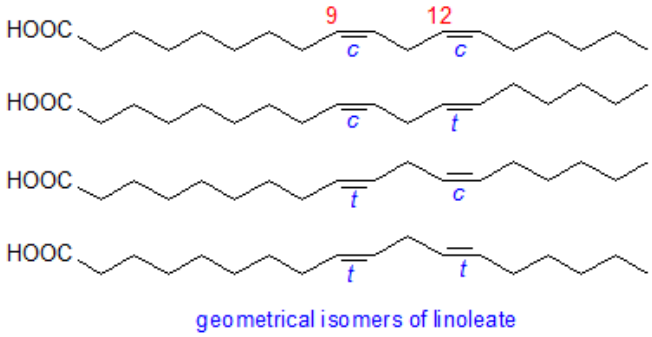


Figure 29: Isomers of linoleate.

Therefore, we need to find the coordinates $x_1, \cdots x_n$ that minimize the following cost function:

$$g(x_1, x_2, \cdots, x_n) = \sum_{(i,j)\epsilon S} (d_{ij} - \|x_i - x_j\|)^2, \tag{95}$$

and thus make $g = 0$. We note that only $n-3$ dihedrals have to be specified to define all interatomic distances, since all bond-lengths and bending angles are given and the positions of the first 3 atoms are defined by the bond-lengths and bending angles. Since each dihedral can be either *cis* or *trans*, we have a total of 2^{n-3} possible configurations, with only some of them satisfying Eq. (95).

To implement the Grover optimization algorithm, we prepare a register with $n-3$ qubits in a uniform superposition, where the state $|0\rangle$ of the j-th qubit corresponds to the *cis* state of the j-th dihedral and $|1\rangle$ corresponds to the *trans* state of that dihedral. The oracle $\hat{O} = e^{i\pi f(j)}$ is defined so that $f(j) = 1$ if state $|j\rangle$ satisfies Eq. (95) and $f(j) = 0$, otherwise.

9.1 Supplement I: Average Deviation Caused by the Oracle

The goal of this subsection is to show that

$$D_j = \sum_{x=0}^{N-1} \|\psi_j^x - \psi_j\|^2 \leq 4j^2, \tag{96}$$

where $|\psi_j\rangle = \sum_y \alpha_{y,j}|y\rangle|\phi_y\rangle$, with $|\psi_j^x\rangle = UO_x|\psi_{j-1}\rangle$ and $|\psi_j\rangle = U|\psi_{j-1}\rangle$. According to Eq. (96), the averaged squared deviation D_j caused by j calls to the oracle, relative to the state evolving with an empty-oracle, increases no faster than $\mathcal{O}(j^2)$.

Solution: Defining $U_x = UO_x$ and $\Delta U = U - U_x$, we obtain:

$$\begin{aligned} |\psi_j\rangle &= (\Delta U + U_x)|\psi_{j-1}\rangle, \\ &= \Delta U|\psi_{j-1}\rangle + U_x|\psi_{j-1}\rangle, \\ &= \Delta U|\psi_{j-1}\rangle + |\psi_j^x\rangle, \end{aligned} \tag{97}$$

so, $|\psi_j\rangle - |\psi_j^x\rangle = \Delta U|\psi_{j-1}\rangle$. Now, substituting $|\psi_{j-1}\rangle = U|\psi_{j-2}\rangle = (\Delta U + U_x)|\psi_{j-2}\rangle$ into Eq. (97), we obtain:

$$\begin{aligned} |\psi_j\rangle &= \Delta U|\psi_{j-1}\rangle + U_x|\psi_{j-1}\rangle. \\ &= \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2|\psi_{j-2}\rangle, \\ &= \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + |\psi_j^x\rangle, \end{aligned} \tag{98}$$

so, $|\psi_j\rangle - |\psi_j^x\rangle = \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle$.

Now, substituting $|\psi_{j-2}\rangle = U|\psi_{j-3}\rangle = (\Delta U + U_x)|\psi_j\rangle$ into Eq. (98), we obtain:

$$\begin{aligned} |\psi_j\rangle &= \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2|\psi_{j-2}\rangle, \\ &= \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2(\Delta U + U_x)|\psi_{j-3}\rangle, \\ &= \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2\Delta U|\psi_{j-3}\rangle + |\psi_j^x\rangle, \end{aligned} \tag{99}$$

Therefore, $|\psi_j\rangle - |\psi_j^x\rangle = \Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2\Delta U|\psi_{j-3}\rangle$.

Analogously, we can repeat the procedure j times to obtain: $|\psi_j\rangle - |\psi_j^x\rangle = U_x^0\Delta U|\psi_{j-1}\rangle + U_x\Delta U|\psi_{j-2}\rangle + U_x^2\Delta U|\psi_{j-3}\rangle + \cdots + U_x^{(j-1)}\Delta U|\psi_0\rangle = \sum_{k=0}^{j-1} U_x^k\Delta U|\psi_{j-1-k}\rangle$, so

$$\begin{aligned} \||\psi_j\rangle - |\psi_j^x\rangle\| &= \|\sum_{k=0}^{j-1} U_x^k\Delta U|\psi_{j-1-k}\rangle\|, \\ &= \|\sum_{k=0}^{j-1} U_x^{j-1-k}\Delta U|\psi_k\rangle\|. \end{aligned} \tag{100}$$

Now, we show that

$$\| |\psi_j\rangle - |\psi_j^x\rangle \|^2 = \| \sum_{k=0}^{j-1} U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2 \leq j \sum_{k=0}^{j-1} \| U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2, \tag{101}$$

as follows. We consider that $\sum_{i,k=0}^{j-1}(a_k - a_i)^2 = \sum_{i,k=0}^{j-1}(a_k^2 + a_i^2 - 2a_k a_i)$, so

$$\begin{aligned} \sum_{i,k=0}^{j-1}(a_k - a_i)^2 &= \sum_{i,k=0}^{j-1} a_k^2 + \sum_{i,k=0}^{j-1} a_i^2 - 2\sum_{i,k=0}^{j-1} a_k a_i, \\ &= j\sum_{k=0}^{j-1} a_k^2 + j\sum_{i=0}^{j-1} a_i^2 - 2\sum_{i,k=0}^{j-1} a_k a_i, \\ &= 2j\sum_{k=0}^{j-1} a_k^2 - 2\| \sum_{k=0}^{j-1} a_k \|^2, \end{aligned} \tag{102}$$

and solving for $\| \sum_{k=0}^{j-1} a_k \|^2$, we obtain:

$$\| \sum_{k=0}^{j-1} a_k \|^2 = j\sum_{k=0}^{j-1} a_k^2 - \frac{1}{2}\sum_{i=0}^{j-1}\sum_{k=0}^{j-1}(a_k - a_i)^2. \tag{103}$$

Considering that $(a_k - a_i)^2$ are positive numbers, we obtain the following bound:

$$\| \sum_{k=0}^{j-1} a_k \|^2 \leq j\sum_{k=0}^{j-1} a_k^2. \tag{104}$$

Therefore, according to Eqs. (104) and (100) with $a_k = \| U_x^{j-1-k} \Delta U |\psi_k\rangle \|$, we obtain:

$$\begin{aligned} \| \sum_{k=0}^{j-1} U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2 &\leq j\sum_{k=0}^{j-1} \| U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2, \\ &\leq j\sum_{k=0}^{j-1} \| \Delta U |\psi_k\rangle \|^2, \\ &\leq j\sum_{k=0}^{j-1} \| |\psi_{k+1}\rangle - |\psi_{k+1}^x\rangle \|^2, \end{aligned} \tag{105}$$

where the second line of Eq. (105) was obtained by considering that U_x is unitary so $U_x^\dagger U = 1$ and thus $(U_x^{j-1-k})^\dagger U^{j-1-k} = 1$.[4] So, $\| U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2 = \langle \psi_k | (\Delta U)^\dagger (U_x^{j-1-k})^\dagger U_x^{j-1-k} \Delta U |\psi_k\rangle = \| U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2 = \langle \psi_k | (\Delta U)^\dagger \Delta U |\psi_k\rangle = \| \Delta U |\psi_k\rangle \|^2$. Now, according to Eq. (**??**), we obtain:

$$\| \sum_{k=0}^{j-1} U_x^{j-1-k} \Delta U |\psi_k\rangle \|^2 = \| |\psi_j\rangle - |\psi_j^x\rangle \|^2 \leq 4j \sum_{k=0}^{j-1} \| \alpha_{x,k} \|^2. \tag{106}$$

Therefore, summing over all possible strings x, we obtain:

$$\begin{aligned} \sum_x \| |\psi_j\rangle - |\psi_j^x\rangle \|^2 &\leq 4j \sum_{k=0}^{j-1} \sum_x \| \alpha_{x,k} \|^2, \\ &\leq 4j^2, \end{aligned} \tag{107}$$

since $\sum_x \| \alpha_{x,j} \|^2 = 1$.

[4] **Unitary trick.** For any unitary operator U (*i.e.*, for which $U^\dagger U = 1$) we have $\| U|\psi\rangle \|^2 = \| |\psi\rangle \|^2$ since = $\| U\psi \|^2 = \langle \psi | U^\dagger U | \psi \rangle = \langle \psi | \psi \rangle = \| \psi \|^2$.

9.2 Supplement II: Optimal Number of Queries

Show that the number of queries j necessary to identify one out of a sufficiently large number N of possible states with probability of at least 50 % is bound by $j \geq c\sqrt{N}$, with c a small constant.

Solution:

We require $\|\langle x|\psi_j^x\rangle\|^2 \geq 1/2$ for any x to have at least 50 % of successfully identifying x out of N possibilities regardless of x. Replacing $|x\rangle$ by $e^{i\theta}|x\rangle$ does not change the probability of success, so we can assume $\langle x|\psi_j^x\rangle = \|\langle x|\psi_j^x\rangle\|$. To obtain the bound we compute the distance,

$$\begin{aligned} D_j &= \sum_x \|\psi_j^x - \psi_j\|^2, \\ &= \sum_x \|(\psi_j^x - x) - (\psi_j - x)\|^2, \\ &= \sum_x \|(\psi_j^x - x)\|^2 + \|\psi_j - x\|^2 - 2\|\psi_j - x\|\|\psi_j^x - x\|\cos(\theta), \\ &= \sum_x u_x^2 + \sum_x v_x^2 - 2\sum_x u_x v_x, \end{aligned} \tag{108}$$

where $u_x = |\psi_j^x\rangle - |x\rangle$ and $v_x = |\psi_j\rangle - |x\rangle$, so

$$\begin{aligned} E_j &= \sum_x \|(\psi_j^x - x)\|^2, \\ &= \sum_x u_x^2 = \mathbf{u}\cdot\mathbf{u}, \end{aligned} \tag{109}$$

and

$$\begin{aligned} F_j &= \sum_x \|(\psi_j - x)\|^2, \\ &= \sum_x v_x^2 = \mathbf{v}\cdot\mathbf{v}, \end{aligned} \tag{110}$$

and $\mathbf{u}\cdot\mathbf{v} = \sum_x u_x v_x$. Furthermore, $\mathbf{u}\cdot\mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos(\theta) \leq |\mathbf{u}||\mathbf{v}|$, since $\cos(\theta) \leq 1$, so

$$\sum_x u_x v_x \leq |\mathbf{u}||\mathbf{v}| = \sqrt{\sum_x u_x^2}\sqrt{\sum_x v_x^2}. \tag{111}$$

Substituting Eqs. (109), (110) and (111) into Eq. (108), we obtain:

$$D_j \geq E_j + F_j - 2\sqrt{E_j F_j} = (\sqrt{F_j} - \sqrt{E_j})^2. \tag{112}$$

In addition, we have the following bounds for E_j and F_j:

$$\begin{aligned} E_j &= \sum_x \||\psi_j^x\rangle - |x\rangle\|^2, \\ &= \sum_x \langle\psi_j^x|\psi_j^x\rangle - \langle\psi_j^x|x\rangle - \langle x|\psi_j^x\rangle + \langle x|x\rangle, \\ &\leq \sum_x (1 - \frac{2}{\sqrt{2}} + 1). \\ &\leq N(2 - \sqrt{2}), \end{aligned} \tag{113}$$

Furthermore, for any normalized state $|\psi_j\rangle$ and complete set of orthonormal states $|x\rangle$, we have

$$\begin{aligned} F_j &= \sum_x \| |\psi_j\rangle - |x\rangle \|^2, \\ &= 2N - 2\sum_x \| \langle\psi_j|x\rangle \| \cos(\theta_x), \\ &\geq 2N - 2\,\mathrm{Max}(\sum_x \| \langle\psi_j|x\rangle \|), \end{aligned} \tag{114}$$

To maximize $\sum_x \| \langle\psi_j|x\rangle \|$ with the constraint $\sum_x \| \langle\psi_j|x\rangle \|^2 = 1$, we maximize the function $f(c_0, \cdots c_{N-1}) = \sum_{x=0}^{N-1} c_x + \gamma(1 - \sum_{x=0}^{N-1} c_x^2)$, with respect to $c_y = \| \langle x|\psi_j\rangle \|$, as follows:

$$\frac{\partial f}{\partial c_y} = 1 - \gamma 2 c_y = 0, \tag{115}$$

giving $c_y = 1/(2\gamma)$ for all y, and since $\sum_{y=0}^{N-1} c_y^2 = 1 = N/(4\gamma^2)$, we obtain $\gamma = \sqrt{N}/2$ and $c_y = 1/\sqrt{N}$. Therefore, according to Eq. (114),

$$F_j \geq 2N - 2N/\sqrt{N} = 2N - 2\sqrt{N}. \tag{116}$$

Substituting Eq. (113) and (116) into Eq. (112), we obtain:

$$\begin{aligned} 4j^2 \geq D_j &\geq (\sqrt{F_j} - \sqrt{E_j})^2, \\ &\geq \left(\sqrt{2N - 2\sqrt{N}} - \sqrt{N(2-\sqrt{2})} \right)^2, \\ &\geq 4N - 2\sqrt{N} - N\sqrt{2} - 2\sqrt{N}\sqrt{(2N - 2\sqrt{N})(2-\sqrt{2})}, \\ &\geq N\left(4 - 2\sqrt{\frac{1}{N}} - \sqrt{2} - 2\sqrt{(2 - \frac{2}{\sqrt{N}})(2-\sqrt{2})} \right), \end{aligned} \tag{117}$$

So, for sufficiently large N, we obtain:

$$j \geq \sqrt{N}\sqrt{\frac{4 - \sqrt{2} - 2\sqrt{2(2-\sqrt{2})}}{4}}, \tag{118}$$

9.3 Supplement III: Average Success Probability of Grover's Algorithm

Show that the Grover's algorithm has an average probability p of finding the quantum state in one of the N possible states given by the following equation:

$$2N - 2\sqrt{pN} - 2\sqrt{N(N-1)(1-p)} = D_j = \sum_{x=0}^{N-1} \| \psi_j^x - \psi_j \|^2 \leq 4j^2, \tag{119}$$

where $|\psi_j^x\rangle = \sqrt{p}|x\rangle + \sqrt{\frac{1-p}{N-1}} \sum_{y \neq x} |y\rangle$, with $|\psi_j\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} |y\rangle$. Therefore, asymptotically and for $p = 1$ Eq. (119) gives the lower bound $j \geq \sqrt{\frac{N}{2}}$.

Solution: Computing the difference $|\psi_j^x\rangle - |\psi_j\rangle$, we obtain:

$$|\psi_j^x\rangle - |\psi_j\rangle = \left(\sqrt{p} - \frac{1}{\sqrt{N}}\right)|x\rangle + \left(\sqrt{\frac{1-p}{N-1}} - \frac{1}{\sqrt{N}}\right)\sum_{y\neq x}|y\rangle, \tag{120}$$

so

$$\begin{aligned}
\sum_{x=0}^{N-1} \||\psi_j^x\rangle - |\psi_j\rangle\|^2 &= \sum_{x=0}^{N-1}\left(\sqrt{p} - \frac{1}{\sqrt{N}}\right)^2 + \left(\sqrt{\frac{1-p}{N-1}} - \frac{1}{\sqrt{N}}\right)^2 (N-1), \\
&= N\left(\sqrt{p} - \frac{1}{\sqrt{N}}\right)^2 + N\left(\sqrt{\frac{1-p}{N-1}} - \frac{1}{\sqrt{N}}\right)^2 (N-1), \\
&= Np + \frac{N}{N} - 2\sqrt{pN} + N(1-p) + (N-1) - 2\sqrt{\frac{N(1-p)}{(N-1)}}(N-1), \\
&= -2\sqrt{pN} + 2N - 2\sqrt{N(1-p)(N-1)},
\end{aligned} \tag{121}$$

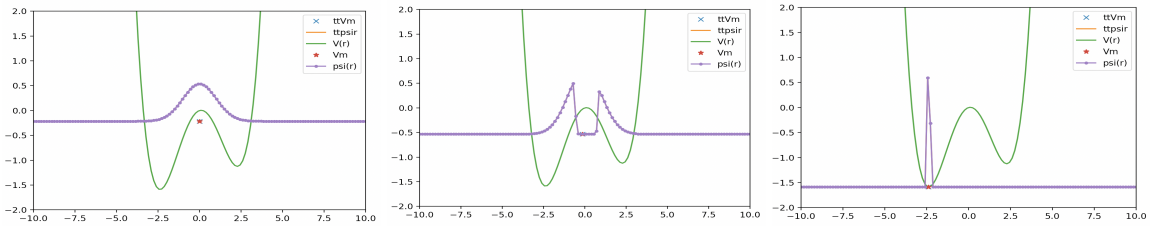
Therefore, for sufficiently large N, the Grover's algorithm finds one out of N states in $\sqrt{N}$ queries, which according to Eq. (118) is optimal. For example, to find one state out of N with $> 50\%$, we need:

$$j \geq \sqrt{\frac{N}{2} - \sqrt{2N}\frac{(1+\sqrt{N-1})}{4}}. \tag{122}$$

10 Iterative Power Algorithm: Classical Amplitude Amplification

The goal of this section is to introduce the iterative power algorithm (IPA) that can be thought of as classical computing analogue of a quantum computing algorithm, where an initial state is transformed so that a measurement can reveal the answer to the problem with high probability. We illustrate IPA as applied to global optimization and factorization.

A simple application involves finding the global minimum of the asymmetric double well potential $V(r) = -0.5 * r^2 + 1.0/(16.0 * 1.3544) * r^4 + 0.1 * r$, shown in the figure.



To find the global minimum, according to the IPA: (1) initialize a *pointer state*, such as the Gaussian $\psi(r) = \pi^{-1/4} e^{-r^2/2}$ illustrated in the figure (left panel); (2) use the pointer state to compute the expectation value of $V(r)$, as follows: $V_m = \langle\psi|V|\psi\rangle$; (3) update the pointer state by projecting out the amplitude components where $V(r) > V_m$, as follows: $\psi(r) \rightarrow \psi(r) \times [1 - sgn(V(r) - V_m)]$, and normalize the resulting pointer state (middle panel); (4) Goto (2). Iterate until reaching convergence. The right panel of the figure shows the resulting pointer obtained after 15 iterations. It can be used to reveal the position of the global minimum $r_m = \langle\psi|r|\psi\rangle$ and the value of the potential at the minimum $V_m = \langle\psi|V|\psi\rangle$. Alternatively, the oracle could be defined to transform the state, as follows: $\psi(r) \rightarrow \psi(r) \times \exp(-V(r))/\|\psi(r) \times \exp(-V(r))\|$.

We note that IPA is a particular version of amplitude amplification, as defined by an oracle that iteratively projects and renormalizes the pointer state. It can also be applied for factorization when using the remainder as the cost function in the subspace of prime numbers.

The tarball file that can be downloaded from here includes several versions of the implementation of the IPA introduced in this section, as applied to global optimization in a double well potential (ttdw.py), in a 4-well potential (tt4w.py), in a multiple well (ttmw.py), revealing a transition state proximal to the global minimum (ttts.py), geometry optimization of a cluster of atoms linked by harmonic oscillators (ttho.py and ftho.py), geometry optimization of a H_2 molecule using pyscf for the Hartree-Fock calculations (gh2.py), and factorization (gfa.py). Note that some of the programs require Ivan Oseledets' tensor train toolbox that can be installed from http://github.com/oseledets/ttpy or Alex Gorodetsky's functional train C3 package. Furthermore, the H_2 example requires PySCF.

10.1 Convergence

The goal of this section is to compare the convergence rate of the IPA and Grover's algorithms.

$$\begin{aligned} 1 = |v_k|^2 &= (N-1)\left(\frac{\lambda_2}{\lambda_1}\right)^{2k} v_{k,max}^2 + v_{k,max}^2, \\ &= v_{k,max}^2 \left(1 + (N-1)\left(\frac{\lambda_2}{\lambda_1}\right)^{2k}\right), \end{aligned} \tag{123}$$

$$v_{k,max} = \frac{1}{\sqrt{1+(N-1)\left(\frac{\lambda_2}{\lambda_1}\right)^{2k}}} \geq \frac{1}{\sqrt{2}}, \tag{124}$$

$$\sqrt{N} \geq k \geq \frac{\ln\left(\frac{1}{N-1}\right)}{2\ln\left(\frac{\lambda_2}{\lambda_1}\right)} \tag{125}$$

$$\left(\frac{1}{N-1}\right)^{\frac{1}{2\sqrt{N}}} \geq \frac{\lambda_2}{\lambda_1} \tag{126}$$

The logarithmic scaling is comparable to or better than in optimal quantum search algorithms (*e.g.*, the Grover quantum search method, where the number of queries necessary to amplify the amplitude of one out of N possible states scales as $O(\sqrt{N})$.

11 Bernstein-Vazirani Algorithm: Exponential Speedup from Superpositions

The goal of this section is to explain the Bernstein-Vazirani algorithm. It is one of the earliest quantum algorithms that was able to demonstrate exponential speedup relative to the fastest possible solution achievable with a classical computer. The algorithm introduces fundamental concepts of superposition states that are essential for understanding many other quantum algorithms.

The Bernstein-Vazirani algorithm solves the following problem. Given a function $f(x) = s \cdot x$ as a black box that transforms the n-bit string $|x\rangle$ into a single bit $f(x) = \{0,1\}$ using the secret string $|s\rangle = |s_1 s_2 \cdots s_n\rangle$, find the n bits of $|s\rangle$, namely $s_1, s_2, \cdots s_n$ by evaluating f as fewer times as possible.

We note that the classical solution requires n calls to the function because the string is composed of n bits and each call to the function returns a single bit (note that to reveal the bit s_j of the secret string one would call the function with $|x_1 x_2 \cdots x_n\rangle$ such that $x_k = \delta_{jk}$ for $j = 1, 2, \cdot n$). Remarkably, the quantum computing solution based on the Bernstein-Vazirani algorithm requires a single evaluation of the function, using the quantum circuit shown in Fig. 30. Can you believe it?

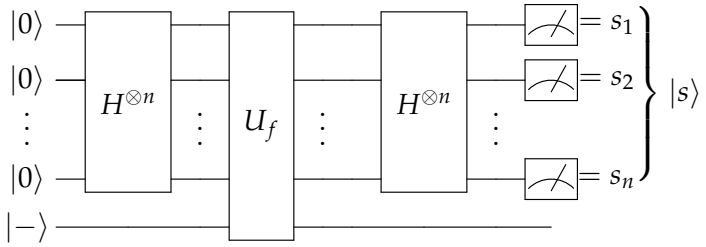


Figure 30: Quantum circuit for the Bernstein-Vazirani algorithm.

To understand how the algorithm works, we first note that the Hadamard gate applied on n qubits, $H^{\otimes n} = H \otimes H \otimes \cdots \otimes H$, transforms the n-bit string $|s\rangle = |s_1 s_2 \cdots s_n\rangle$, as follows:

$$H^{\otimes n}|s\rangle = \sum_{x \in \{0,1\}^n} \frac{(-1)^{s \cdot x}}{2^{n/2}} |x\rangle. \tag{127}$$

as shown below in Sec. 11.1. Also, the Hadamard transform is its own inverse (*i.e.*, $|s\rangle = H^{\otimes n}H^{\otimes n}|s\rangle$). Therefore, the Bernstein-Vazirani algorithm first prepares the state $H^{\otimes n}|s\rangle$ by making a single call to $f(x)$, and then reveals the string $|s\rangle$ by applying the Hadamard transform and measuring.

To prepare the state $H^{\otimes n}|s\rangle$, we initialize n working qubits as $|0\rangle$ and the ancilla qubit $|a\rangle$ as $|-\rangle$, and we apply the Hadamard transform to the working qubits to put them in the uniform superposition state, according to Eq. (127):

$$\begin{aligned} |-\rangle H^{\otimes n}|00\cdots0\rangle &= |-\rangle \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}|x\rangle, \\ &= \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}|-\rangle|x\rangle. \end{aligned} \tag{128}$$

Next, we evaluate the function $f(x)$ with the n working qubits still in the uniform superposition, and we transform the ancilla qubit by performing a control-NOT operation with $f(x)$, as follows: $|a \oplus f(x)\rangle$. The strings $|x\rangle$ for which $f(x) = 0$ leave the ancilla qubit unchanged, while the strings for which $f(x) = 1$ transform $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$ into $(-1)|-\rangle = \frac{1}{\sqrt{2}}|1\rangle - \frac{1}{\sqrt{2}}|0\rangle$. Therefore, the resulting state is

$$\begin{aligned} U_f|-\rangle H^{\otimes n}|00\cdots0\rangle &= \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}(-1)^{f(x)}|-\rangle|x\rangle, \\ &= |-\rangle \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}(-1)^{f(x)}|x\rangle, \\ &= |-\rangle \sum_{x\in\{0,1\}^n} \frac{1}{2^{n/2}}(-1)^{s\cdot x}|x\rangle \end{aligned} \tag{129}$$

with the working qubits in the desired state. Applying a Hadamard gate to the working qubits thus reveals the secret string s.

11.1 Hadamard Transform of Arbitrary Strings

To show that

$$H^{\otimes n}|s\rangle = \sum_{x\in\{0,1\}^n} \frac{(-1)^{s\cdot x}}{2^{n/2}}|x\rangle, \tag{130}$$

we consider first the simple case of 2-bit string $|s\rangle = |s_1\, s_2\rangle$, where $s_j = \{0, 1\}$. When $s_1 = s_2 = 0$, we have $s \cdot x = s_1 \cdot x_1 + s_2 \cdot x_2 = 0$, regardless of the values of x_1 and x_2. So, Eq. 130 gives the expected result $H^{\otimes 2}|00\rangle = |++\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle + |01\rangle)$. Next, when $s_1 = 1$ and $s_2 = 0$, we have $H^{\otimes 2}|10\rangle = |-+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle - |10\rangle - |11\rangle)$. Note that if both the input and output strings have a 1 in the same position, then the output is multiplied by (-1). That pattern of negative signs is properly captured by the term $(-1)^{s\cdot x}$. As an exercise, check that the pattern of negative signs is confirmed for the input string $|11\rangle$, and then work out 3-bit strings.

To show that the Hadamard gate is reversible, we compute

$$\begin{aligned} H^{\otimes n}H^{\otimes n}|s\rangle &= \sum_{y\in\{0,1\}^n} \sum_{x\in\{0,1\}^n} \frac{(-1)^{s\cdot x}}{2^{n/2}} \frac{(-1)^{y\cdot x}}{2^{n/2}}|y\rangle, \\ &= \sum_{x\in\{0,1\}^n} \sum_{y=s} \frac{(-1)^{s\cdot x}}{2^{n/2}} \frac{(-1)^{y\cdot x}}{2^{n/2}}|y\rangle + \sum_{x\in\{0,1\}^n} \sum_{y\neq s} \frac{(-1)^{s\cdot x}}{2^{n/2}} \frac{(-1)^{y\cdot x}}{2^{n/2}}|y\rangle, \end{aligned} \tag{131}$$

Therefore,

$$
\begin{aligned}
H^{\otimes n}H^{\otimes n}|s\rangle &= \sum_{x\in\{0,1\}^n}\frac{(-1)^{s\cdot x}}{2^{n/2}}\frac{(-1)^{s\cdot x}}{2^{n/2}}|s\rangle + \sum_{x\in\{0,1\}^n}\sum_{y\neq s}\frac{(-1)^{s\cdot x}}{2^{n/2}}\frac{(-1)^{y\cdot x}}{2^{n/2}}|y\rangle, \\
&= |s\rangle\sum_{x\in\{0,1\}^n}\frac{1}{2^n} + \sum_{x\in\{0,1\}^n}\sum_{y\neq s}\frac{(-1)^{s\cdot x}}{2^{n/2}}\frac{(-1)^{y\cdot x}}{2^{n/2}}|y\rangle, \\
&= |s\rangle + \sum_{y\neq s}\sum_{x\in\{0,1\}^n}\frac{(-1)^{(y+s)\cdot x}}{2^n}|y\rangle, \\
&= |s\rangle + \sum_{y\neq s}\sum_{x\perp(y+s)}\frac{(-1)^{(y+s)\cdot x}}{2^n}|y\rangle + \sum_{y\neq s}\sum_{x\parallel(y+s)}\frac{(-1)^{(y+s)\cdot x}}{2^n}|y\rangle, \\
&= |s\rangle + \sum_{y\neq s}\sum_{x\perp(y+s)}\frac{1}{2^n}|y\rangle + \sum_{y\neq s}\sum_{x\parallel(y+s)}\frac{(-1)}{2^n}|y\rangle,
\end{aligned}
\tag{132}
$$

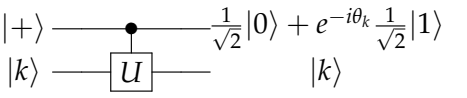
where the second and third term cancel each other since exactly half of the strings x are orthogonal to $s+y$.

Implementation on the IBM Quantum: A turn-key tutorial on how to implement the Vazirani algorithm with Qiskit on Colab or the IBM Quantum is available as a Qiskit tutorial, and can be downloaded as a notebook: vic_vazirani.ipynb, or vic_vazirani.pdf.

12 Phase Kickback

An important step introduced by Eq. (129), is the factorization of $|-\rangle$ that multiplies all strings $|x\rangle$, leaving behind its phase $(-1)^{f(x)}$ as acquired during the conditional-f (XOR) operation when going from the first to the second row of that equation. As a result, the ancilla $|-\rangle$ remained unchanged while its acquired phase was 'kicked back' to the strings of the superposition. Such a procedure that transfers the phase acquired by a qubit to another qubit(s) in a superposition state is called 'phase kickback' and is exploited by many quantum algorithms.

A simple illustrative example of phase kickback is the conditioned implementation of the unitary $U = e^{-\frac{i}{\hbar}\hat{H}t}$ that transforms the state $|\phi_k\rangle$, as follows: $U|k\rangle = e^{-i\theta_k}|k\rangle$, with $\theta_k = E_k t/\hbar$. When applied to a qubit prepared in state $|k\rangle$, conditioned to the state of another qubit prepared in a superposition state (e.g., $|+\rangle$), we transfer the phase θ_k to this other qubit, as follows:



(133)

since $\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) cU|k\rangle = \frac{1}{\sqrt{2}}|0\rangle|k\rangle + \frac{1}{\sqrt{2}}|1\rangle e^{-i\theta_k}|k\rangle = \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle e^{-i\theta_k}\right)|k\rangle$.

13 Hadamard Gate with Beam Splitters: Mach-Zehnder interferometer

This section introduces the *beam splitter*, as applied to the implementation of a Hadamard gate and phase shifter in an optical interferometer. These optical elements are important since any gate can be implemented just by using beam splitters, phase shifters, photodetectors and single photon sources, allowing for universal quantum computing, as shown by the KLM protocol introduced

by Knill, Laflamme and Milburn. The optical interferometer provides a very accessible way of analyzing gates and the 'magic' of quantum superposition states exploited by quantum computing.

A beam splitter is a crystal that splits an incoming beam of light into two outgoing beams of intensities $I_c = |E_c^{out}|^2$ and $I_d = |E_d^{out}|^2$, as shown in (Figure 31). The incoming electric fields

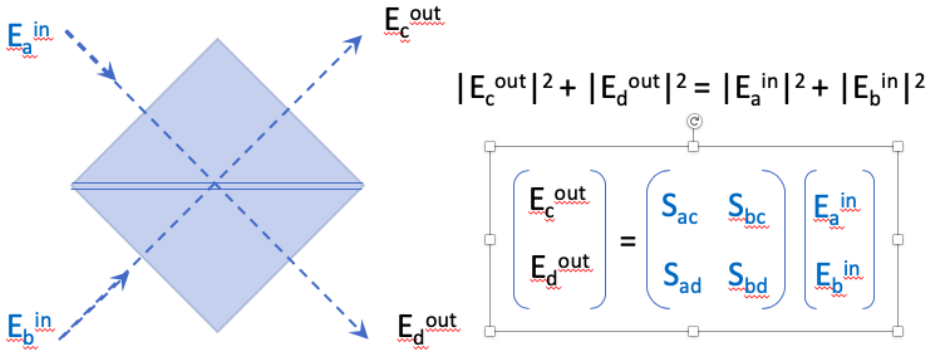


Figure 31: Beam splitter, a crystal that splits incoming beams with intensities $I_a = |E_a^{in}|^2$ and $I_b = |E_b^{in}|^2$ into outgoing beams of intensities $I_c = |E_c^{out}|^2$ and $I_d = |E_d^{out}|^2$.

E_a^{in} and E_b^{in} could correspond to two different beams, or two modes of a single beam (*e.g.*, two different states of polarization).

The outgoing beam E_c^{out} results from the partial reflection of incident beam E_a^{in} and partial transmission of beam E_b^{in}, with reflection and transmission coefficients given by the scattering matrix elements $S_{ac} = Re^{i\phi_{ac}}$ and $S_{bc} = Te^{i\phi_{bc}}$, respectively. Analogously, the outgoing beam E_d^{out} results from the partial reflection of incident beam E_b^{in} and partial transmission of beam E_a^{in} with reflection and transmission coefficients given by the scattering matrix elements $S_{bd} = Re^{i\phi_{bd}}$ and $S_{ad} = Te^{i\phi_{ad}}$, respectively. Energy conservation, $I_a + I_b = I_c + I_d$ (*i.e.*, conservation of photons), requires that $R^2 + T^2 = 1$ and $\phi_{ad} - \phi_{bd} + \phi_{bc} - \phi_{ac} = \pi$.

A simple example of a scattering matrix that splits incoming beams 50-50 into each of the possible outgoing directions with energy conservation is the matrix where $R = T = 1/\sqrt{2}$, with $\phi_{bd} = -\pi$ and $\phi_{ad} = \phi_{bc} = \phi_{ac} = 0$ (*i.e.*, the Hadamard matrix):

$$\mathrm{BS}_1 = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \tag{134}$$

Another S matrix that splits incoming beams in each of the possible outgoing directions with energy conservation has $R = T = 1/\sqrt{2}$, and $\phi_{ac} = -\pi$, where $\phi_{ad} = \phi_{bc} = \phi_{bd} = 0$,

$$\mathrm{BS}_2 = \begin{pmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \tag{135}$$

To illustrate how the beam splitters transform an incoming state, we consider first an incoming beam with $E_a^{in} = 0$ and $E_b^{in} = 1$, or $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, as shown in Fig. (32). The outgoing beams after BS_1 have $E_c^{in} = 1/\sqrt{2}$ and $E_d^{in} = -1/\sqrt{2}$, or $\begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$ with 50% intensity on each outgoing beam and a well-defined *relative phase* (i.e., $E_c^{out} = 1/\sqrt{2}$ and $E_d^{out} = 1/\sqrt{2}e^{i\pi}$). This example is very similar to the double-slit experiment, with the beam splitter functioning as a simplified version

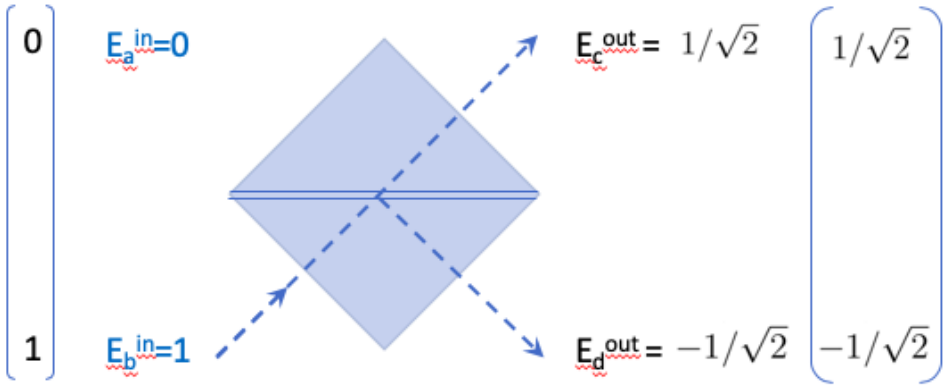


Figure 32: A beam splitter splits an incoming beam with $E_a^{in} = 0$ and $E_b^{in} = 1$ into outgoing beams with $E_c^{out} = 1/\sqrt{2}$ and $E_d = -1/\sqrt{2}$.

of a double-slit. The relative phases of the outgoing beams are important since they determine the interference phenomena that rules the behavior of the outgoing state, for example, as the beams pass through another BS_1, as shown in Fig. (33). Note that the second BS_1 transforms

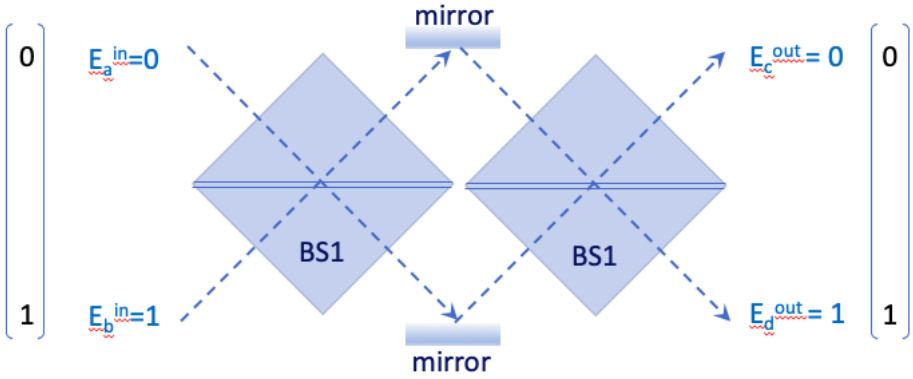


Figure 33: A beam splitter splits an incoming beam with $E_a^{in} = 0$ and $E_b^{in} = 1$ into outgoing beams with $E_c^{out} = 0$ and $E_d = 1$.

the state $\begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$ into the output state $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ which is identical to the initial state due to full constructive interference in the lower mode and no intensity in the top one (rather than getting a 50-50 distribution as one would expect in the absence of interference). This is the expected result since the Hadamard matrix is its own inverse.

Changing the second beamsplitter by BS_2, as shown in Fig. (34) provides another example that illustrates the essential role of the phases. Here, we obtain full constructive interference in c, and no intensity in d. These examples show that beam splitters can be used to generate a variety of states. In fact, when combined with phase shifters, we can generate arbitrary states, as shown below. But first, let us show what happens to the intensity of beam d if we block one of the branches, right before passing through the second beamsplitter, as shown in Fig. (35). Note that the lower branch of the state $\begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$ generated by BS_1 is absorbed, thus, generating

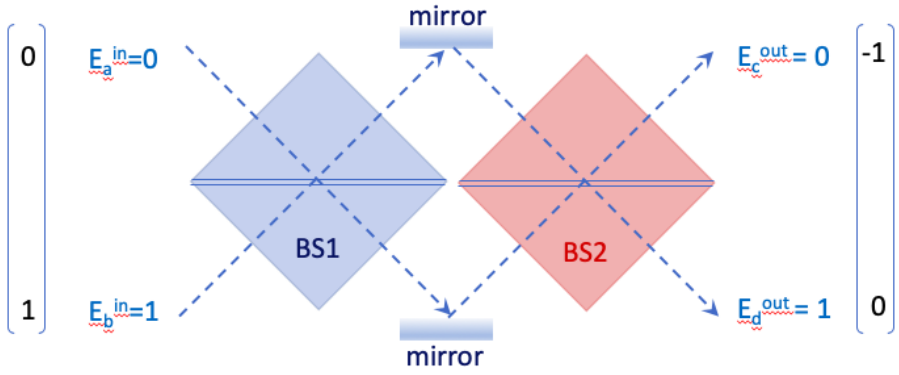


Figure 34: A beam splitter splits an incoming beam with $E_a^{in} = 0$ and $E_b^{in} = 1$ into outgoing beams with $E_c^{out} = -1$ and $E_d = 0$.

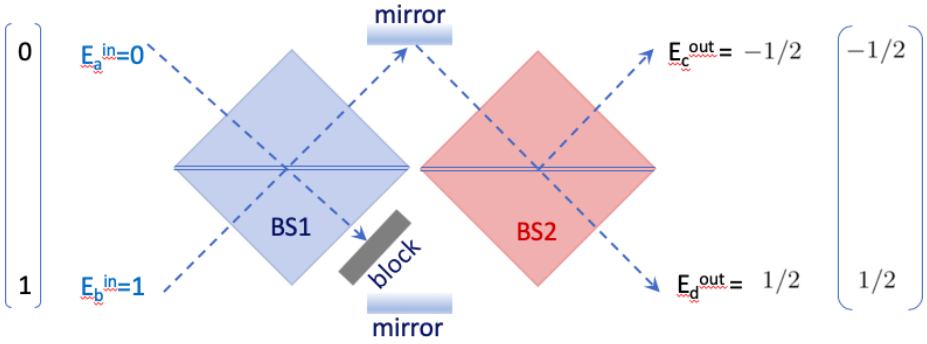


Figure 35: A beam splitter splits an incoming beam with $E_a^{in} = 0$ and $E_b^{in} = 1$ into outgoing beams with $E_c^{out} = -1$ and $E_d = 0$.

the state $\begin{pmatrix} 1/\sqrt{2} \\ 0 \end{pmatrix}$ that is transformed by BS_2 into $\begin{pmatrix} -1/2 \\ 1/2 \end{pmatrix}$. Remarkably, we increased the intensity of beam d by actually *blocking* one of the branches!

Elitzur-Vaidman test: The remarkable effect of the block, responsible for increasing the intensity in channel d, is the basis of the Elitzur-Vaidman thought experiment. Consider a fragile precious molecule that is destroyed when it absorbs a photon (and absorbs photons with 100 % quantum yield). In contrast, a defective state of the molecule does not absorb photons and is not destroyed. So, if the molecule is defective and is positioned like the block in Fig. 35 we get the intensities of Fig. 34 (*i.e*, no photons are ever detected in channel d). When the molecule is good and is placed as the block in Fig. (35), it is destroyed 50% of the times, since a photon from b has only 50% probability of hitting the block. If it is not destroyed, it is because the photon went through the other branch after BS_1. In that case, the probability of detecting a photon in channel d is equal to $1/4$, as shown in Fig. (35). So, if we detect a photon in channel d we know the molecule is good, as reported by a photon that never interacted with the molecule!

Hong-Ou-Mandel effect This effect is another manifestation of interference at a beam splitter demonstrated in 1987 by three physicists from the University of Rochester: Chung Ki Hong, Zhe Yu Ou and Leonard Mandel. The experiment shows that the incidence of two indistinguishable pho-

tons, one from a and the other from b as described by the incident state $|\psi^{in}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$, produces either both photons in c or both in d, with 50-50 probability, according to the outgoing entangled state $|\psi^{out}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ (Fig. 36). Never one photon coming out from c and the other from d.

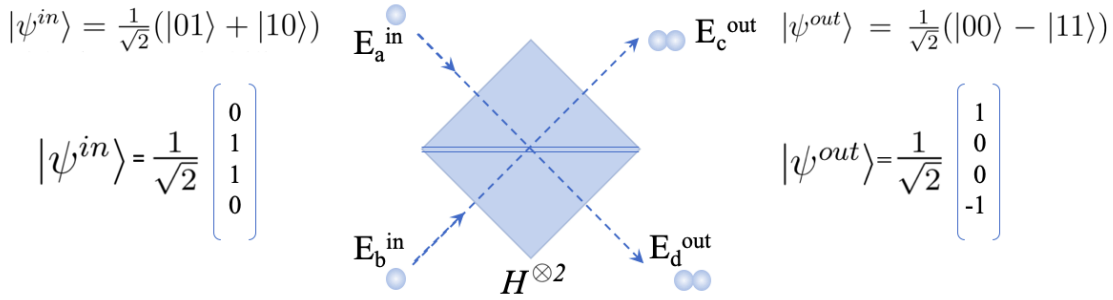


Figure 36: Experiment demonstrating the Hong-Ou-Mandel effect: when two indistinguishable photons strike a beam splitter, one from a and the other from b, they both come out of the same channel, either both from c, or both from d, with 50-50 probability. Never one from c and the other from d.

Mathematically, we can show this remarkable result, as follows. We build a symmetrized initial state with one photon in a (*i.e.*, $|1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$) and one in b (*i.e.*, $|0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$), as follows:

$|\psi^{in}\rangle = \frac{1}{\sqrt{2}}(|0\rangle|1\rangle + |1\rangle|0\rangle)$, where $|0\rangle|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$, and $|1\rangle|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$, defining the initial state $|\psi^{in}\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$.

Applying the Hadamard gate $H^{\otimes 2} = H \otimes H$ to $|\psi^{in}\rangle$, we obtain:

$$|\psi^{out}\rangle = \frac{1}{2^{3/2}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle).$$

Therefore, $|\langle 01|\psi^{out}\rangle|^2 = |\langle 10|\psi^{out}\rangle|^2 = 0$, while $|\langle 00|\psi^{out}\rangle|^2 = |\langle 11|\psi^{out}\rangle|^2 = \frac{1}{2}$.

Phase-Shift: Another interesting experiment is the effect of a piece of glass that introduces a phase-shift in one of the branches, as shown in Fig. 37. The phase shift is described by the gate $Z(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$. The phase-shift can be measured by counting the number of photons detected at c and d (*i.e.*, measuring the relative intensity of beams c and d).

Note that the lower branch of the state generated by BS_1, $\begin{pmatrix} 1/\sqrt{2} \\ e^{-i\pi}/\sqrt{2} \end{pmatrix}$ gets a phase-shift and becomes $\begin{pmatrix} 1/\sqrt{2} \\ e^{-i(\pi-\phi)}/\sqrt{2} \end{pmatrix}$, so it is transformed by BS_2 into $\begin{pmatrix} (-e^{i\phi}-1)/2 \\ (-e^{i\phi}+1)/2 \end{pmatrix}$. Therefore, the intensity ratio $I_c/I_d = (1+cos(\phi))/(1-cos(\phi))$ is a simple function of the phase shift.

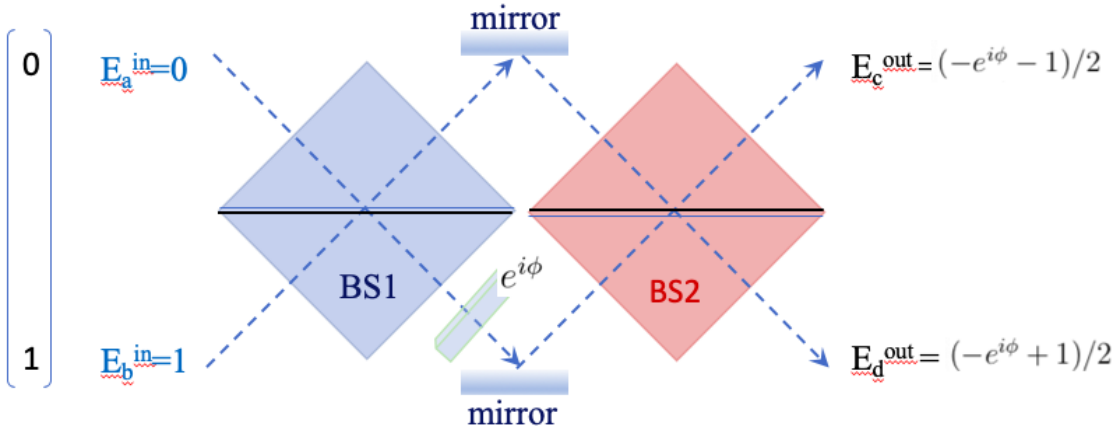


Figure 37: Mach-Zehnder interferometer demonstrating the effect of a phase shift on the detected field amplitudes E_c^{out} and E_d^{out}.

This method of measuring the phase-shift introduced on a beam of light by a sample, as determined by mixing the signal with a reference beam that comes from the same source as the beam that went through the sample, is called optical *homodyne detection*. It has the main advantage of using the reference beam (the so-called *local oscillator*) to compensate for any fluctuations in the light source.

14 Deutsch Algorithm

The goal of this section is to explain the so-called *Deutsch algorithm* that exploits the phase kick-back trick to determine if a function $f : \{0,1\} \to \{0,1\}$ is constant (i.e., $f(0) = f(1)$, and thus $f(0) \oplus f(1) = 0$), or not constant (*i.e.*, 'balanced' since $f(0) \neq f(1)$, so $f(0) \oplus f(1) = 1$) by making a *single* query to the function f, as shown in the circuit below. This is a remarkable result since any classical algorithm would have to call f twice to determine whether it is constant, or not. To

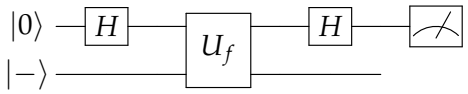


Figure 38: Quantum circuit for implementing the Deutsch algorithm.

see how the algorithm works, we note that the initial state $|\psi_0\rangle = |0\rangle|-\rangle$ is transformed into state $|\psi_1\rangle$ by the first Hadamard gate, as follows:

$$\begin{aligned} |\psi_1\rangle &= \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)|-\rangle, \\ &= \frac{1}{\sqrt{2}}|0\rangle|-\rangle + \frac{1}{\sqrt{2}}|1\rangle|-\rangle. \end{aligned} \tag{136}$$

When applying the function f to ψ_1, we obtain:

$$\begin{aligned} |\psi_2\rangle &= \hat{U}_f|\psi_1\rangle, \\ &= \frac{1}{2}|0\rangle\left(|0\oplus f(0)\rangle - |1\oplus f(0)\rangle\right) + \frac{1}{2}|1\rangle\left(|0\oplus f(1)\rangle - |1\oplus f(1)\rangle\right), \end{aligned} \tag{137}$$

We note that $(|0 \oplus f(0)\rangle - |1 \oplus f(0)\rangle) = (-1)^{f(0)}(|0\rangle - |1\rangle)$ regardless of whether $f(0) = 0$ or $f(0) = 1$ since $1 \oplus 1 = 0$ and $1 \oplus 0 = 1$. Therefore,

$$\begin{aligned} |\psi_2\rangle &= \frac{(-1)^{f(0)}}{2}|0\rangle(|0\rangle - |1\rangle) + \frac{(-1)^{f(1)}}{2}|1\rangle(|0\rangle - |1\rangle), \\ &= (-1)^{f(0)}\frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle\right)|-\rangle, \end{aligned} \tag{138}$$

Factorizing $(-1)^{-f(0)}$ which is equal to $(-1)^{f(0)}$, we obtain:

$$|\psi_2\rangle = (-1)^{f(0)}\frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle\right)|-\rangle. \tag{139}$$

When $f(0) \oplus f(1) = 0$, the first qubit is in the state $|+\rangle$, so a Hadamard transformation makes it $|0\rangle$, and when $f(0) \oplus f(1) = 1$, the first qubit is $|-\rangle$, so the Hadamrad transformation makes it $|1\rangle$. Therefore, a measurement of the first qubit can determine whether the function is constant, or not.

15 Simon's Algorithm

The goal of this section is to explain Simon's algorithm, one of the earliest quantum algorithms that was able to demonstrate exponential speedup relative to classical computing.

Simon's algorithm solves the following problem. We are given a function f that transforms n bits into n bits (e.g., a black box executable program), such that $f(x) = f(x \oplus s)$, with s a secret string. So, the resulting n string is the same for x and $x \oplus s$. The problem is to discover the bits of $|s\rangle$ by making the fewest possible calls to f.

Solving the problem by classical computing requires multiple calls to $f(x)$, at least until finding two strings for which the output is the same. So, considering that there are 2^n equally probable strings, one can show that classical computing requires of the order of $\sqrt{2^n}$ evaluations – i.e., an exponential number of evaluations of the function, analogously to the 'birthday problem'.[5] Remarkably, solving the problem by quantum computing with Simon's algorithm requires only a polynomial in n number of evaluations for $f(x)$, rather than an exponential number.

Simon's algorithm requires preparation of the particular superposition $\frac{1}{\sqrt{2}}|r\rangle + \frac{1}{\sqrt{2}}|r \oplus s\rangle$ where $|r\rangle$ is a random n-bit string. That particular superposition state can be generated by initializing $2n$ qubits as $|0\rangle$, applying a Hadamard transform $H^{\otimes n}$ to the first n qubits to generate a uniform superposition of all possible 2^n random strings, as shown in Eq. (127), and then transforming the second set of n qubits initialized as $|0\rangle$ into $|f(r)\rangle$. Upon measuring the second set of n qubits (as shown in Fig. 39), we collapse the first set of n qubits into the desired superposition since a given value of $f(r)$ is fulfilled by both r and $s \oplus r$.

Finally, we apply a Hadamard transform to the first set of n qubits (already prepared in the particular superposition $\frac{1}{\sqrt{2}}|r\rangle + \frac{1}{\sqrt{2}}|r \oplus s\rangle$, as described above) and we measure. It is important

[5] **Birthday Problem:** We want to find out the probability that out of 30 people two of them have the same birthday. Person 1, Person 2: prob=364/365 of no overlap with the first; Person 3: prob=363/365 of no overlap with 1 and 2; Person 4: prob=362/365 of no overlap with 1, 2 and 3; ... Person 30: prob=336/365 of no overlap with any person above; The probability of having no shared birthdays is then $(364/365) \times (363/365) \times (362/365) \cdots (336/365) = 0.293684$. So the probability of having at least one pair of people having the same birthday is 71%. Analogously, we find a probability $> 96\%$ when the group has more than 48 people $(2.5\sqrt{365})$. Let us find this probability with the Monte Carlo Approach: 1) Pick 30 random numbers in the range [1,365]. 2) Check to see if any of the thirty are equal. 3) Go back to step 1 and repeat 10000 times. 4) Report the fraction of trial that have matching birthdays and use it to compare with the result above.

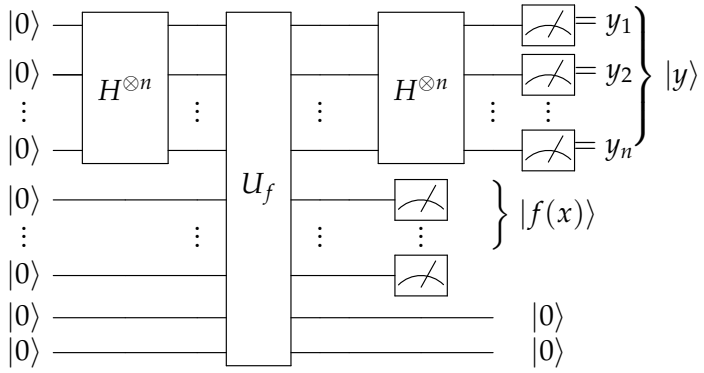


Figure 39: Quantum circuit for Simon's algorithm based on Fourier sampling (ı,e., application of a Hadamard transform and measurement) of the superposition state $\frac{1}{\sqrt{2}}|r\rangle + \frac{1}{\sqrt{2}}|r \oplus s\rangle$ prepared upon measurement of the ancilla qubits in the $f(x)$ state when the first n working qubits are in a uniform superposition state x.

to note that the strings y sampled by measuring the first n qubits, after the Hadamard transform, satisfy the condition that $y \cdot s = 0$ (i.e., $y \in s^{\perp}$) since (as shown below) all other strings have coefficients equal to zero. This can be shown, as follows:

$$\begin{aligned} H^{\otimes n}\frac{1}{\sqrt{2}}|r\rangle + H^{\otimes n}\frac{1}{\sqrt{2}}|r \oplus s\rangle &= \sum_{y\in\{0,1\}^n}\left(\frac{(-1)^{r\cdot y}}{2^{(n+1)/2}} + \frac{(-1)^{(r\oplus s)\cdot y}}{2^{(n+1)/2}}\right)|y\rangle, \\ &= \sum_{y\in\{0,1\}^n}\frac{(-1)^{r\cdot y}}{2^{(n+1)/2}}\left(1 + (-1)^{s\cdot y}\right)|y\rangle, \\ &= \sum_{y\in s^{\perp}}\frac{(-1)^{r\cdot y}}{2^{(n-1)/2}}|y\rangle, \end{aligned} \tag{140}$$

where $y \in s^{\perp}$ fulfill the condition $s \cdot y = 0$. We note that the probability of each sampled string is given by the square of the expansion coefficient $|\frac{1}{2^{(n-1)/2}}|^2 = \frac{1}{2^n/2}$. So, the number of strings that are orthogonal to s ($y \in s^{\perp}$) is equal to $2^n/2 = 2^{n-1}$ (*i.e.*, half the total number of n-bit strings !).

Therefore, by repeating the preparation and measurement $(n-1)$ times, we get a set of $(n-1)$ equations of the form:

$$\begin{aligned} y_1^{(1)}s_1 + y_2^{(1)}s_2 + \cdots + y_n^{(1)}s_n &= 0 \\ y_1^{(1)}s_1 + y_2^{(1)}s_2 + \cdots + y_n^{(1)}s_n &= 0 \\ &\cdots \\ y_1^{(n-1)}s_1 + y_2^{(n-1)}s_2 + \cdots + y_n^{(n-1)}s_n &= 0 \end{aligned} \tag{141}$$

that can be solved for $s_1, s_2, \cdots s_n$ to get two possible solutions, including the trivial solution with all $s_j = 0$ and the non-trivial solution of interest.

The algorithm works only when the $(n-1)$ equations are linearly independent, which can be shown to happen at least with 25% probability (and exact probability of success close to 29%), as follows.

The $\geq 25\%$ probability of success is obtained by considering that there are 2^{n-1} equally probable strings y that are sampled. Sampling a linearly dependent string in the $(n-1)$ step, after

having sampled $(n-2)$ linearly independent strings in the previous steps would lead to failure. The probability of that event is equal to the probability of a string $1/2^{n-1}$ times the number of strings that are linearly dependent on of the previously sampled strings. Writing a linear combination of all $(n-2)$ previously selected strings, with expansion coefficients equal to 1 or 0 for each string, shows that there are 2^{n-2} possible strings that would be linearly dependent on the previously sampled strings. So, the probability of failure in that last step is $2^{n-2} * 1/2^{n-1} = 1/2$. Analogously, we find that failing in the previous step would have probability $1/4$, in the previous one $1/8$, until the very first step for which the probability of failing is $1/2^{n-1}$ because only the string with all zeros would lead to failure.

Summing the probabilities of failing at the first $(n-2)$ steps (instead of multiplying them) gives us an upper bound to the probability that the algorithm fails in the first $(n-2)$ steps. Therefore, multiplying that probability by the probability of independently failing in the last step gives an upper bound to the total probability of failing, as follows. Considering that the geometric series is $\sum_{j=0}^{n-1} x^j = (1-x^n)/(1-x)$, we obtain:

$$\begin{aligned} 1/4 + 1/8 + \cdots + 1/2^{n-1} &= \sum_{j=1}^{n-1} 0.5^j - 1/2 \\ &= \sum_{j=0}^{n-1} 0.5^j - 1 - 1/2 \\ &= \frac{(1-0.5^n)}{(1-0.5)} - 1 - 1/2 \\ &= \frac{(0.5-0.5^n)}{0.5} - 1/2 = 1/2 - \frac{1}{2^{n-1}} \leq 1/2. \end{aligned} \tag{142}$$

Therefore, the probability of success in the first $(n-2)$ steps is at least $1/2$ and since the probability of success in the last step is $1/2$, the probability of success is at least $1/4$.

Exercise: Let $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ and let $\mathbf{s} = \mathbf{x} \oplus \mathbf{y}$. Show that

$$H^{\otimes n} \frac{1}{\sqrt{2}} |\mathbf{x}\rangle + H^{\otimes n} \frac{1}{\sqrt{2}} |\mathbf{y}\rangle = \sum_{\mathbf{z} \in \{\mathbf{s}\}^\perp} \frac{(-1)^{\mathbf{x}\cdot\mathbf{z}}}{2^{(n-1)/2}} |\mathbf{z}\rangle. \tag{143}$$

Solution:

$$\begin{aligned}
H^{\otimes n}\frac{1}{\sqrt{2}}|\mathbf{x}\rangle + H^{\otimes n}\frac{1}{\sqrt{2}}|\mathbf{y}\rangle &= \frac{1}{2^{(n+1)/2}}\sum_{\mathbf{z}\in\{0,1\}^n}\left((-1)^{\mathbf{x}\cdot\mathbf{z}}+(-1)^{\mathbf{y}\cdot\mathbf{z}}\right)|\mathbf{z}\rangle, \\
&= \frac{1}{2^{(n+1)/2}}\sum_{\mathbf{z}\in\{0,1\}^n}(-1)^{\mathbf{x}\cdot\mathbf{z}}\left(1+(-1)^{(\mathbf{y}-\mathbf{x})\cdot\mathbf{z}}\right)|\mathbf{z}\rangle, \\
&= \frac{1}{2^{(n+1)/2}}\sum_{\mathbf{z}\in\{0,1\}^n}(-1)^{\mathbf{x}\cdot\mathbf{z}}\left(1+(-1)^{(\mathbf{s}-2\mathbf{x})\cdot\mathbf{z}}\right)|\mathbf{z}\rangle, \\
&= \frac{1}{2^{(n+1)/2}}\sum_{\mathbf{z}\in\{0,1\}^n}(-1)^{\mathbf{x}\cdot\mathbf{z}}\left(1+(-1)^{\mathbf{s}\cdot\mathbf{z}}\right)|\mathbf{z}\rangle, \\
&= \frac{1}{2^{(n+1)/2}}\sum_{\mathbf{z}\in\{s\}^\perp}(-1)^{\mathbf{x}\cdot\mathbf{z}}2|\mathbf{z}\rangle, \\
&= \frac{1}{2^{(n-1)/2}}\sum_{\mathbf{z}\in\{s\}^\perp}(-1)^{\mathbf{x}\cdot\mathbf{z}}|\mathbf{z}\rangle,
\end{aligned} \tag{144}$$

Exercise:

We have defined $\mathbf{s}^\perp$, but more generally we can let S be a vector subspace of Z_2^n, and define $S^\perp = \{\mathbf{t} \in Z_2^n | \mathbf{t}\cdot\mathbf{s} = 0 \text{ for all } \mathbf{s} \in S\}$. So our previously defined $\mathbf{s}^\perp$ corresponds to $\mathbf{S}^\perp$ where $S = \{\mathbf{0}, \mathbf{s}\}$ is the 2-dimensional vector space spanned by $\mathbf{s}$.

- (a) Define $|S\rangle = \sum_{\mathbf{s}\in S}\frac{1}{\sqrt{2^m}}|\mathbf{s}\rangle$. Prove that $H^{\otimes n}|S\rangle = \sum_{\mathbf{w}\in S^\perp}\frac{1}{s^{(n-m)/2}}|\mathbf{w}\rangle$.
- (b) For any $\mathbf{y} \in \{0,1\}^n$ define $|\mathbf{y}+S\rangle = \sum_{\mathbf{s}\in S}\frac{1}{\sqrt{2^m}}|\mathbf{y}+\mathbf{s}\rangle$. What is $H^{\otimes n}|\mathbf{y}+S\rangle$?

Solution:

(a)

$$\begin{aligned}
H^{\otimes n}|S\rangle &= \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{s}\in S}\sum_{\mathbf{w}\in\{0,1\}^n}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle, \\
&= \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{s}\in S}\sum_{\mathbf{w}\in S^\perp}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle + \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{s}\in S}\sum_{\mathbf{w}\notin S^\perp}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle, \\
&= \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{s}\in S}\sum_{\mathbf{w}\in S^\perp}|\mathbf{w}\rangle + \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{s}\in S}\sum_{\mathbf{w}\notin S^\perp}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle, \\
&= \frac{2^m}{2^{(n+m)/2}}\sum_{\mathbf{w}\in S^\perp}|\mathbf{w}\rangle + \frac{1}{2^{(n+m)/2}}\sum_{\mathbf{w}\notin S^\perp}\sum_{\mathbf{s}\in S}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle, \\
&= \frac{1}{2^{(n-m)/2}}\sum_{\mathbf{w}\in S^\perp}|\mathbf{w}\rangle,
\end{aligned} \tag{145}$$

since $\sum_{\mathbf{s}\in S}(-1)^{\mathbf{s}\cdot\mathbf{w}} = 0$.

(b)

$$
\begin{aligned}
H^{\otimes n}|\mathbf{y}+S\rangle &= \sum_{\mathbf{s}\in S}\frac{1}{\sqrt{2^{(m+n)}}}\sum_{\mathbf{w}\in\{0,1\}^n}(-1)^{(\mathbf{y}+\mathbf{s})\cdot\mathbf{w}}|\mathbf{w}\rangle,\\
&= \frac{1}{\sqrt{2^{(m+n)}}}\sum_{\mathbf{w}\in\{0,1\}^n}(-1)^{\mathbf{y}\cdot\mathbf{w}}\sum_{\mathbf{s}\in S}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle,\\
&= \frac{1}{\sqrt{2^{(m+n)}}}\sum_{\mathbf{w}\in S^{\perp}}(-1)^{\mathbf{y}\cdot\mathbf{w}}\sum_{\mathbf{s}\in S}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle + \frac{1}{\sqrt{2^{(m+n)}}}\sum_{\mathbf{w}\notin S^{\perp}}(-1)^{\mathbf{y}\cdot\mathbf{w}}\sum_{\mathbf{s}\in S}(-1)^{\mathbf{s}\cdot\mathbf{w}}|\mathbf{w}\rangle,\\
&= \frac{1}{\sqrt{2^{(m+n)}}}\sum_{\mathbf{w}\in S^{\perp}}(-1)^{\mathbf{y}\cdot\mathbf{w}}2^m|\mathbf{w}\rangle,\\
&= \frac{1}{\sqrt{2^{(n-m)}}}\sum_{\mathbf{w}\in S^{\perp}}(-1)^{\mathbf{y}\cdot\mathbf{w}}|\mathbf{w}\rangle,
\end{aligned}
\tag{146}
$$

16 Quantum Fourier Transform

The goal of this section is to introduce the quantum Fourier transform, the quantum circuit of the classical discrete Fourier transform, and its comparison to the Hadamard transform.

As we discussed earlier, the Hadamard gate $H^{\otimes n} = H \otimes H \otimes \cdots \otimes H$, with

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{147}$$

transforms the N-bit string $|s\rangle = |s_1 s_2 \cdots s_N\rangle$, with $N = 2^n$, as follows:

$$H^{\otimes n}|s\rangle = \sum_{x \in \{0,1\}^N} \frac{(-1)^{s \cdot x}}{N^{1/2}} |x\rangle, \tag{148}$$

with matrix elements $\langle j|H^{\otimes n}|k\rangle = N^{-1/2}(-1)^{jk}$. Analogously, the quantum Fourier transform of the N-bit string $|s\rangle = |s_1 s_2 \cdots s_N\rangle$ is defined, as follows:

$$QFT_N|s\rangle = \sum_{x \in \{0,1\}^N} \frac{w^{s \cdot x}}{N^{1/2}} |x\rangle, \tag{149}$$

where $w = e^{i2\pi/N}$ is the N-th root of unity since $w^N = 1$. Therefore, the matrix elements of the N-dimensional quantum Fourier transform are $\langle j|QFT_N|k\rangle = N^{-1/2} w^{jk}$, defining the QFT in matrix form, as follows:

$$QFT_N = \frac{1}{N^{1/2}} \begin{bmatrix} w^0 & w^0 & \cdots & w^0 \\ w^0 & w^1 & \cdots & w^{N-1} \\ w^0 & w^2 & \cdots & w^{2(N-1)} \\ \cdots & \cdots & \cdots & \cdots \\ w^0 & w^{N-1} & \cdots & w^{(N-1)^2} \end{bmatrix}. \tag{150}$$

We note that the Fourier transform of a single qubit coincides with the Hadamard transform, shown in Eq. (147), since $w = -1$ when $N = 2$. Also, $QFT_N|0 \cdots 0\rangle = H^{\otimes n}|0 \cdots 0\rangle$, since $(-1)^{0 \cdot x} = (w)^{0 \cdot x} = 1$ in both Eq. (148) and Eq. (149). We also note that each of the matrix elements $w^{jk} = e^{i\frac{2\pi}{N}jk}$ are simple phase shifters.

Inverse Fourier transform: The inverse Fourier transform is defined, as follows:

$$QFT_N^{-1}|x\rangle = \sum_{r \in \{0,1\}^N} \frac{w^{-r \cdot x}}{N^{1/2}} |r\rangle. \tag{151}$$

Therefore, the matrix elements of the N-dimensional inverse Fourier transform are $\langle j|QFT_N|k\rangle = N^{-1/2} w^{-jk}$

Note that according to Eqs. (149) and (151), $QFT_N^{-1} QFT_N|s\rangle = |s\rangle$:

$$\begin{aligned} QFT_N^{-1} QFT_N|s\rangle &= \sum_{x \in \{0,1\}^N} \frac{w^{s \cdot x}}{N^{1/2}} \sum_{r \in \{0,1\}^N} \frac{w^{-r \cdot x}}{N^{1/2}} |r\rangle, \\ &= \sum_{r \in \{0,1\}^N} \sum_{x \in \{0,1\}^N} \frac{w^{(s-r) \cdot x}}{N} |r\rangle, \\ &= |s\rangle + \frac{1}{N} \sum_{r \neq s} \left(\sum_{x=0}^{N-1} w^{(s-r) \cdot x} \right) |r\rangle, \end{aligned} \tag{152}$$

with $\sum_{x=0}^{N-1} w^{(s-r) \cdot x} = 0$ when $s - r \neq 0$.

16.1 Properties of the Fourier transform

Property 1: Shift Invariant. Fourier sampling is unaffected by a constant shift since

$$\begin{aligned} QFT_n|s+\Delta\rangle &= \sum_{x\in\{0,1\}^n} \frac{w^{x\cdot(s+\Delta)}}{n^{1/2}}|x\rangle, \\ &= \sum_{x\in\{0,1\}^n} w^{x\Delta}\frac{w^{s\cdot x}}{n^{1/2}}|x\rangle, \end{aligned} \tag{153}$$

with $\left|w^{x\Delta}\frac{w^{s\cdot x}}{n^{1/2}}\right|^2 = \left|\frac{w^{s\cdot x}}{n^{1/2}}\right|^2$, since $w = e^{i2\pi/n}$ and thus $\left|w^{x\Delta}\right|^2 = 1$. Therefore, the probability of sampling string $|x\rangle$ is the same before and after the shift.

Property 2: Sum of roots. The sum of the n-th roots of unity is equal to zero, which can be shown by using the geometric series $\sum_{j=0}^{n-1} x^j = (1-x^n)/(1-x)$, as follows:

$$\sum_{j=0}^{n-1} w^j = \frac{(1-w^n)}{(1-w)} = 0, \tag{154}$$

since $w^n = 1$. Note that the same result is obtained by replacing w by any power of w.

Property 3: Orthogonality. The columns of the Fourier transform are orthonormal vectors that define a coordinate transformation (*i.e.*, a rotation in Hilbert space). Computing the scalar product of columns l and k (with $\bar{w} = e^{-i2\pi/n}$ the conjugate of $w = e^{i2\pi/n}$, we obtain:

$$\begin{aligned} \langle l|k\rangle &= \frac{1}{n}\sum_{j=0}^{n-1} \bar{w}^{lj}w^{jk}, \\ &= \frac{1}{n}\sum_{j=0}^{n-1} w^{(k-l)j}, \\ &= \frac{1}{n}\delta_{lk}\sum_{j=0}^{n-1} 1 + \frac{1}{n}(1-\delta_{lk})\sum_{j=0}^{n-1} w^{(k-l)j}, \\ &= \delta_{lk} + \frac{1}{n}(1-\delta_{lk})\frac{(1-w^{n(k-l)})}{(1-w^{(k-l)})}, \\ &= \delta_{lk}. \end{aligned} \tag{155}$$

where the fourth line was obtained by using Property 2, according to the geometric series $\sum_{j=0}^{n-1} x^j = (1-x^n)/(1-x)$, while the fifth line by using that w is the n-th root of unity since $w^n = 1$).

Property 4: Efficient Quantum Circuit. The discrete Fourier transform (FT_N) is defined according to Eq. (150) but without the normalization factor and can be efficiently computed by rearranging the columns of the matrix to have the even numbered columns as the first $N/2$ columns followed by the $N/2$ odd numbered columns (labeled with index $k = 0-(N/2)$), and rows labeled with

index $j = 0 - (N/2)$, as follows:

$$FT^{(N)} = \left[\begin{array}{c|c} w^{j2k} & w^{j(2k+1)} \\ \hline w^{(j+N/2)2k} & w^{(j+N/2)(2k+1)} \end{array}\right]. \tag{156}$$

We note that $w^N = 1$, and $w^{N/2} = -1$, so $w^{(j+N/2)(2k)} = w^{(2kj+Nk)} = w^{2kj}$ and $w^{(j+N/2)(2k+1)} = w^{(2kj+j+Nk+N/2)} = -w^{(2k+1)j}$, giving

$$FT_N = \left[\begin{array}{c|c} w^{j2k} & w^j w^{j2k} \\ \hline w^{j2k} & -w^j w^{j2k} \end{array}\right]. \tag{157}$$

Therefore, the FT_N can be applied to an N-bit string X_N by applying the $FT_{N/2}$ to $N/2$-bit strings, as follows:

$$FT_N X_N = \left[\begin{array}{c|c} FT_{N/2} & w^j FT_{N/2} \\ \hline FT_{N/2} & -w^j FT_{N/2} \end{array}\right] \left[\begin{array}{c} x_0 \\ x_2 \\ \cdots \\ x_{N-2} \\ \hline x_1 \\ x_3 \\ \cdots \\ x_{N-1} \end{array}\right], \tag{158}$$

where $FT_{N/2}$ is applied to the top $N/2$ even-numbered bits, and $w_j FT_{N/2}$ is applied to the bottom $N/2$ odd-numbered bits. The top $N/2$ bits are updated by summing the two contributions while the bottom $N/2$ bits are updated by subtracting the two contributions, according to the following 'butterfly' circuit:

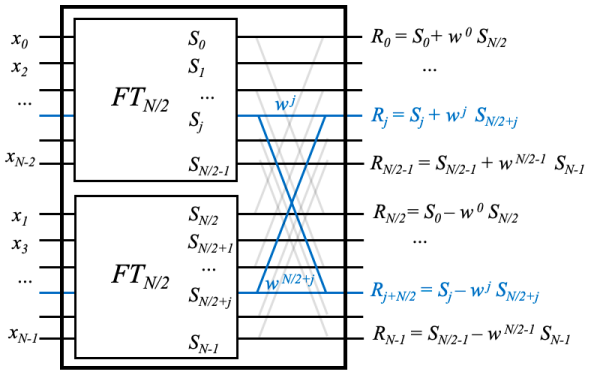


Figure 40: Classical 'butterfly' circuit for implementation of the FFT algorithm.

Therefore, the computational cost $C(N)$ of FT_N is $C(N) = 2C(N/2) + \mathcal{O}(N)$. If we recurse $\log(N)$ times, we obtain: $C(N/2) = 2C(N/4) + \mathcal{O}(N/2), \cdots, C(4) = 2C(2) + \mathcal{O}(4)$, and, $C(2) =$

$2C(1) + \mathcal{O}(2)$. So, the total cost of the discrete Fourier transform FT_N implemented recursively is $\mathcal{O}(N\log N)$, and is therefore called *Fast Fourier Transform* (FFT) algorithm [Cooley and Tukey *Math. Comp.* **19** (1965), 297-301]. By making the Fourier transform $\mathcal{O}(N\log N)$ (*i.e.*, much faster that the naive implementation with cost $\mathcal{O}(N^2)$), the FFT algorithm has enabled a wide range of revolutionary technologies.

Here, we show that the *quantum Fourier transform algorithm* implements the FT in $\mathcal{O}(\log^2 N)$ elementary operations (*i.e.,* in fewer steps that the number of operations it would take to input the initial N-bit state to be transformed (!)). For example, the QFT_N of an input with $N = 2^{1000}$ would take only 10^6 operations. While this might sound impossible, it is in fact achievable because the N-bit input can be encoded in a superposition of states of $n = \log N$ qubits (*i.e.*, $n = 1000$).

Using $n = \log_2(N/2) + 1$ qubits, the quantum circuit that performs the QFT_N looks, as follows:

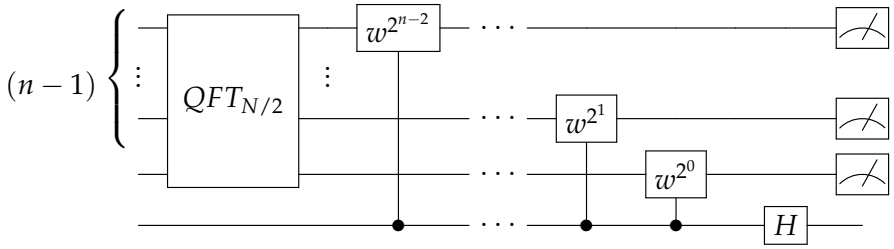


Figure 41: Quantum circuit for implementation of the QFT_N, with $w = e^{i2\pi/N}$

Note that the quantum circuit, introduced by Fig. 41, has only one $QFT_{N/2}$ unit, in stark contrast with the classical circuit (Fig. 40), since the odd and even numbered bits can be encoded as superposition states of the same set of $n-1$ qubits, using the n-th qubit in state $|0\rangle$, or $|1\rangle$, for encoding even or odd numbered bits, respectively. Therefore, the outcome of the $QFT_{N/2}$ resulting from the odd numbered bits (*i.e.*, $S_{j+N/2}$, with $j = 0 - (N/2 - 1)$), is also encoded as a superposition state of the first $n-1$ qubits when the n-th qubit is in state $|1\rangle$. To multiply each state $S_{j+N/2}$ of the superposition by w^j, we encode j in binary, in terms of $(n-1)$ bits that can be either one or zero $(j_0, \cdots, j_{n-2})$, corresponding to the $n-1$ qubits, as follows: $j = \sum_{k=0}^{n-2} j_k 2^k$, so $w^j = \prod_{k=0}^{n-2} w^{j_k 2^k}$ and can be applied by applying a phase shift defined by w^{2^k} to the outcome of the k-th wire when that k-th bit is $|1\rangle$ (*i.e.*, when $j_k = 1$) and the n-th qubit is also $|1\rangle$.

Remarkably, the butterfly sums and differences of S_j and $S_{j+N/2}$, described in Fig. 40, can be efficiently performed by applying a Hadamard gate to the n-th qubit. To understand that 'magic' step, consider the effect of the Hadamard gate on the last qubit when the outcome of the $QFT_{N/2}$ is written as a superposition of all possible $n-1$ strings $|\mathbf{x}\rangle$, as follows:

$$\begin{aligned} |\psi\rangle &= \sum_{y\in\{0,1\}} \sum_{\mathbf{x}\in\{0,1\}^{n-1}} c_{y,\mathbf{x}} |y,\mathbf{x}\rangle, \\ &= |0\rangle \sum_{\mathbf{x}\in\{0,1\}^{n-1}} c_{0,\mathbf{x}} |\mathbf{x}\rangle + |1\rangle \sum_{\mathbf{x}\in\{0,1\}^{n-1}} c_{1,\mathbf{x}} |\mathbf{x}\rangle. \end{aligned} \tag{159}$$

where the coefficients $c_{0,\mathbf{x}}$ correspond to the superposition of outcomes S_j from even numbered bits, and $c_{1,\mathbf{x}}$ correspond to the superposition of $w^j S_{j+N/2}$ resulting from odd numbered bits. Ap-

plying the Hadamard gate to the ancilla n-th qubit, we obtain:

$$\begin{aligned} H|\psi\rangle &= \sum_{\mathbf{x}\in\{0,1\}^{n-1}} (c_{0,\mathbf{x}}|+\rangle + c_{1,\mathbf{x}}|-\rangle)|\mathbf{x}\rangle, \\ &= \frac{1}{\sqrt{2}} \sum_{\mathbf{x}\in\{0,1\}^{n-1}} (c_{0,\mathbf{x}} + c_{1,\mathbf{x}})|0\rangle + (c_{0,\mathbf{x}} - c_{1,\mathbf{x}})|1\rangle|\mathbf{x}\rangle. \end{aligned} \tag{160}$$

Therefore, the sum and differences correspond to the superposition of the first $n-1$ qubits when the n-th qubit is $|0\rangle$ and $|1\rangle$, respectively, as provided by the Hadamard transformation with the correct normalization factor.

Note that the computational cost of the QFT is $S(n) = S(n-1) + \mathcal{O}(n)$. So, the total computational cost over n recursion layers is $n + (n-1) + \cdots + 1 = n(n+1)/2$, or $S(n) = \mathcal{O}(n^2) = \mathcal{O}(\log^2 N)$ for an input of size $N = 2^n$. The enhanced version of QFT improves it to $\mathcal{O}(n \log n)$ [L. Hales and S. Hallgren, An improved quantum Fourier transform algorithm and applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 515 (2000)]. So, the QFT is exponentially faster than the classical FFT algorithm. However, there is caveat! While performing the QFT would be exponentially fast, reading the outcome for a very large N would still be challenging.

The simplest QFT circuit (beyond the circuit with $N = 2$ (*i.e.*, $n = 1$ for which the QFT is the Hadamard gate) is the Fourier transform of a state with $N = 4$ (*i.e.*, with $n = 2$), which is implemented as follows:

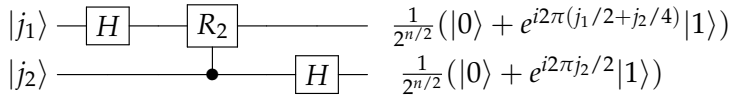


Figure 42: Quantum circuit for QFT_N, with $N = 4$, where $R_k = e^{i2\pi/2^k}$.

Analogously, the circuit for $N = 8$ (*i.e.*, with $n = 3$), is implemented as follows:

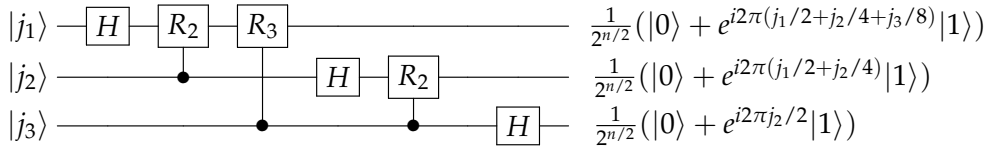


Figure 43: Quantum circuit for QFT_N, with $N = 8$, where $R_k = e^{i2\pi/2^k}$.

To obtain the implementation, shown in Fig. 43, we consider $QFT_N|j\rangle = 2^{-n/2}\sum_k w^{jk}|k\rangle$.

Writing k in binary, $k = \sum_{l=1}^{n} 2^{n-l} k_l$, we obtain:

$$\begin{aligned}
QFT_N|j\rangle &= 2^{-n/2} \sum_k w^{jk} |k\rangle, \\
&= 2^{-n/2} \sum_k e^{i2\pi jk/2^n} |k\rangle, \\
&= 2^{-n/2} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} e^{i2\pi j \sum_{l=1}^{n} 2^{-l} k_l} |k_1\rangle \cdots |k_n\rangle, \\
&= 2^{-n/2} \bigotimes_{l=1}^{n} \sum_{k_l=0}^{1} e^{i2\pi j 2^{-l} k_l} |k_l\rangle, \\
&= 2^{-n/2} \bigotimes_{l=1}^{n} \left(|0\rangle + e^{i2\pi j 2^{-l}} |1\rangle \right), \\
&= 2^{-n/2} \bigotimes_{l=1}^{n} \left(|0\rangle + e^{i2\pi \sum_{m=n-l+1}^{n} j_m 2^{n-m-l}} |1\rangle \right), \\
&= 2^{-n/2} \left(|0\rangle + e^{i2\pi 0.j_n} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi 0.j_{n-1} j_n} |1\rangle \right) \cdots \otimes \left(|0\rangle + e^{i2\pi 0.j_1 j_2 \cdots j_n} |1\rangle \right),
\end{aligned} \tag{161}$$

where the line before the last one is obtained by writing j in binary, $j = \sum_{m=1}^{n} j_m 2^{n-m}$, so $2^{-l} j = \sum_{m=1}^{n-l} j_m 2^{n-l-m} + \sum_{m=n-l+1}^{n} j_m 2^{n-m-l}$, and for all m of the first sum $(n-l-m)$ is an integer, so $e^{i2\pi j 2^{-l}} = \left(\prod_{m=n-l+1}^{n} e^{i2\pi j_m 2^{n-m-l}} \right) e^{i2\pi \sum_{m=n-l+1}^{n} j_m 2^{n-m-l}}$, with $\prod_{m=n-l+1}^{n} e^{i2\pi j_m 2^{n-m-l}} = 1$. The last line is obtained by introducing the popular notation $0.j_l j_{l+1} \cdots j_n = j_l 2^{-1} + j_{l+1} 2^{-2} + \cdots + j_n 2^{-(1+n-l)}$.

When $n = 3$, we obtain as shown in Fig. 43,

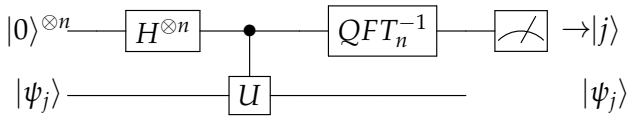
$$\begin{aligned}
QFT_8|j\rangle &= 2^{-n/2} \left(|0\rangle + e^{i2\pi j 2^{-1}} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi j 2^{-2}} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi j 2^{-3}} |1\rangle \right). \\
&= 2^{-3/2} \left(|0\rangle + e^{i2\pi j_3 2^{-1}} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi (j_2 2^{-1} + j_3 2^{-2})} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi (j_1 2^{-1} + j_2 2^{-2} + j_3 2^{-3})} |1\rangle \right), \\
&= 2^{-3/2} \left(|0\rangle + e^{i2\pi 0.j_3} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi 0.j_2 j_3} |1\rangle \right) \otimes \left(|0\rangle + e^{i2\pi 0.j_1 j_2 j_3} |1\rangle \right).
\end{aligned} \tag{162}$$

.

16.2 Quantum Phase Estimation

Fourier sampling can be used for phase estimation. A simple example is given by the unitary that performs the following transformation $U|\psi_j\rangle = e^{i2\pi\theta}|\psi_j\rangle$ where $\theta = j/2^n$, with j an *integer* between 0 and $(2^n - 1)$ (later we will discuss what happens when j is not an integer but rather a non-integer *real* number between 0 and $(2^n - 1)$). When U is applied conditionally, it operates on $|\psi_j\rangle$, as follows: c-U$|k\rangle|\psi_j\rangle = |k\rangle e^{ik2\pi\theta}|\psi_j\rangle$, where $|k\rangle$ is an n-bit string that defines the k number of times that the unitary U is applied on $|\psi_j\rangle$ (with $k = 0, \cdots, (2^n - 1)$).

In the rest of this subsection we show that the phase θ can be estimated by first applying c-U to a uniform superposition of states $|k\rangle$ (prepared, as follows: $H^{\otimes n}|0\rangle^{\otimes n} = QFT_n|0\rangle^{\otimes n}$). Then, we exploit the phase kickback algorithm, we compute the inverse Fourier transform, and we measure, according to the following circuit:



(163)

We note that the state after the Hadamard gate, right before the conditional unitary, is the uniform superposition $\frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle|\psi_j\rangle$. Therefore, the state after the unitary is $\frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle(U^k|\psi_j\rangle) = \frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle e^{ik2\pi\theta}|\psi_j\rangle$. Introducing the substitutions $\theta = j/2^n$ and $w = e^{i2\pi/2^n}$, and invoking the phase kickback trick, the resulting state is

$$\left(\frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle w^{jk}\right)|\psi_j\rangle, \tag{164}$$

which, according to Eq. (149) is equal to the $(QFT_{2^n}|j\rangle)|\psi_j\rangle$. Therefore, we can obtain $|j\rangle$ by computing the inverse Fourier transform over the first n qubits and measuring. Also, note that $|\psi_j\rangle$ is left unchanged so the process can be repeated multiple times using only one copy of $|\psi_j\rangle$.

The rest of this section shows how to apply the c-U gate in the *computational basis* to transform the state $|k\rangle|\psi_j\rangle$ into the state $|k\rangle e^{ik2\pi\theta}|\psi_j\rangle$, as shown in the circuit of Eq. (163). First, we write k in binary ($k = \sum_{l=1}^{n} 2^{n-l}k_l$). So, we obtain:

$$\begin{aligned}
\frac{1}{2^{n/2}}\sum_{k=1}^{2^n}|k\rangle e^{i2\pi k\theta}|\psi_j\rangle &= \frac{1}{2^{n/2}}\sum_{k_1=0}^{1}\cdots\sum_{k_n=0}^{1}|k_1\cdots k_n\rangle e^{i2\pi\sum_{l=1}^{n}k_l2^{n-l}\theta}|\psi_j\rangle, \\
&= \frac{1}{2^{n/2}}\bigotimes_{l=1}^{n}\sum_{k_l=0}^{1}e^{i2\pi jk_l2^{-l}}|k_l\rangle|\psi_j\rangle, \\
&= \frac{1}{2^{n/2}}\bigotimes_{l=1}^{n}(|0\rangle|\psi_j\rangle + |1\rangle e^{i2\pi j2^{-l}}|\psi_j\rangle), \\
&= \frac{1}{2^{n/2}}(|0\rangle|\psi_j\rangle + |1\rangle e^{i2\pi j2^{-1}}|\psi_j\rangle)\otimes\cdots\otimes(|0\rangle|\psi_j\rangle + |1\rangle e^{i2\pi j2^{-n}}|\psi_j\rangle), \\
&= \frac{1}{2^{n/2}}(|0\rangle|\psi_j\rangle + |1\rangle e^{i2\pi\theta2^{n-1}}|\psi_j\rangle)\otimes\cdots\otimes(|0\rangle|\psi_j\rangle + |1\rangle e^{i2\pi\theta2^{0}}|\psi_j\rangle).
\end{aligned} \tag{165}$$

According to Eq. (165), the circuit to implement the c-U gate in the computational basis is, as follows:

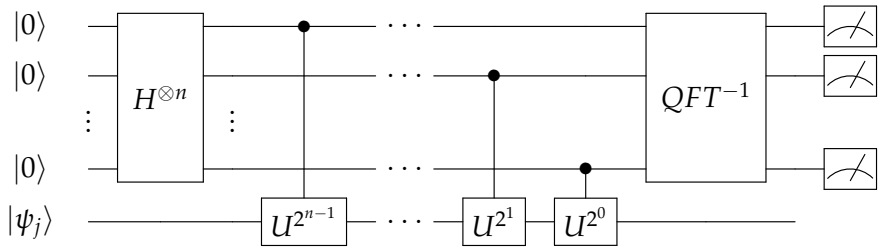


Figure 44: Circuit for quantum phase estimation.

Now, substituting $j = \sum_{m=1}^{n} j_m 2^{n-m}$, we obtain:

$$\frac{1}{2^{n/2}}\sum_{k=1}^{2^n}|k\rangle e^{i2\pi k\theta}|\psi_j\rangle = \frac{1}{2^{n/2}}\otimes_{l=1}^{n}e^{i2\pi k_l\sum_{m=1}^{n}j_m2^{(n-l-m)}}|k_l\rangle|\psi_j\rangle. \tag{166}$$

Therefore,

$$\begin{aligned}
\frac{1}{2^{n/2}}\sum_{k=1}^{2^n}|k\rangle e^{i2\pi k\theta}|\psi_j\rangle &= \frac{1}{2^{n/2}}\otimes_{l=1}^{n}\left(|0\rangle + |1\rangle e^{i2\pi\sum_{m=n-l+1}^{n}j_m2^{(n-l-m)}}\right)|\psi_j\rangle, \\
&= QFT_{2^n}|j\rangle|\psi_j\rangle,
\end{aligned} \tag{167}$$

where the second line of Eq. (167) is obtained by comparing the first line to Eq. (161).

Probability of Success in Phase Estimation

For an arbitrary value of θ –*i.e.*, regardless of whether θ is a multiple of $1/2^n$) or not, we obtain that the state after the unitary is $\frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle(U^k|\psi_j\rangle) = \frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1}|k\rangle e^{i2\pi k\theta}|\psi_j\rangle$, and the inverse Fourier transform applied to the first n-qubts gives the output state,

$$\frac{1}{2^n}\sum_{k=0}^{2^n-1}\sum_{l=0}^{2^n-1}|l\rangle e^{i2\pi k(\theta-l/2^n)}|\psi_j\rangle. \tag{168}$$

Therefore, the probability of measuring the first n-qubits in state $|l\rangle = |j\rangle$ is

$$P_j = \left|\frac{1}{2^n}\sum_{k=0}^{2^n-1} e^{i2\pi k(\theta-j/2^n)}\right|^2. \tag{169}$$

Clearly, when $\theta = j'/2^n$, $P_j = \delta_{jj'}$. More generally (*i.e.*, when θ is not a multiple of $1/2^n$), using the geometric series $\sum_{k=0}^{2^n-1}x^k = (x^{2^n}-1)/(x-1)$, we obtain:

$$\begin{aligned} P_j &= \left|\frac{1}{2^n}\frac{(e^{i2\pi 2^n(\theta-j/2^n)}-1)}{(e^{i2\pi(\theta-j/2^n)}-1)}\right|^2, \\ &= \frac{1}{2^{2n}}\left|\frac{(e^{i2\pi(2^n\theta-j)}-1)}{(e^{i2\pi(\theta-j/2^n)}-1)}\right|^2, \\ &= \frac{1}{2^{2n}}\frac{a^2}{b^2}, \end{aligned} \tag{170}$$

where $\theta = j/2^n + \epsilon$, so $a = e^{i2\pi 2^n\epsilon} - 1$, and $b = e^{i2\pi\epsilon} - 1$. Here, ϵ is the difference between the phase θ and the best possible n-bit approximation $j/2^n$, so $0 < \epsilon < 1/2^n$.

Representing $e^{i2\pi 2^n\epsilon}$ and $e^{i2\pi\epsilon}$ in the complex plane, as shown in Fig. 45, we find that $2\pi|\epsilon|2^n/a \leq \pi/2$ since that ratio is maximum for $2\pi|\epsilon|2^n = \pi$, for which $a = 2$ (Fig. 45, left). In addition, we find that $2\pi|\epsilon| \geq b$ (Fig. 45, right). Substituting these inequalities into Eq. (170), we obtain:

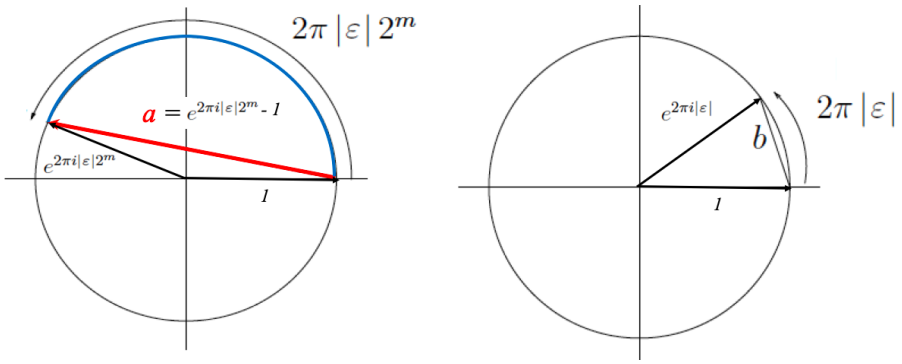


Figure 45: Left: The ratio of the arc length $2\pi|\epsilon|2^n$ (in blue) to the chord length a (in red) is maximum and equal to $\pi/2$ for $2\pi|\epsilon|2^n = \pi$, for which the chord $a = 2$. Right: The ratio of the arc length $2\pi|\epsilon|$ to the chord length b is always larger than 1.

$$P_j = \frac{1}{2^{2n}}\frac{a^2}{b^2}, \\ \geq \frac{1}{2^{2n}}\frac{(4|\epsilon|2^n)^2}{(2\pi|\epsilon|)^2} = \frac{4}{\pi^2} > 0.4. \tag{171}$$

Therefore, the probability of measuring all n bits of j correctly (such that $|j/2^n - \theta| \leq 2^{-n}$) is higher the 40%. In addition, since both the first and second closest multiple of 2^{-n} are within 2^{-n} of the actual value to be determined, the phase is estimated within an error of $1/2^n$ with probability $\frac{8}{\pi^2} > 0.8$.

Next, we address the probability $pr(|m-j| > k)$ of measuring a value m beyond the $k2^{-n}$ accuracy, as defined by set of $2k$ multiples of 2^{-n} closer to j, shown in Fig. 46:

$$pr(|m-j| > k) = \sum_{l=-2^{n-1}+1}^{-k} P_{j+l} + \sum_{l=k+1}^{2^{n-1}} P_{j+l}, \tag{172}$$

with P_{l+j} defined according to Eq. (170), with $\theta = j/2^n + \epsilon$:

$$P_{l+j} = \frac{1}{2^{2n}}\left|\frac{(e^{i2\pi(2^n\epsilon - l)} - 1)}{(e^{i2\pi(\epsilon - l/2^n)} - 1)}\right|^2. \tag{173}$$

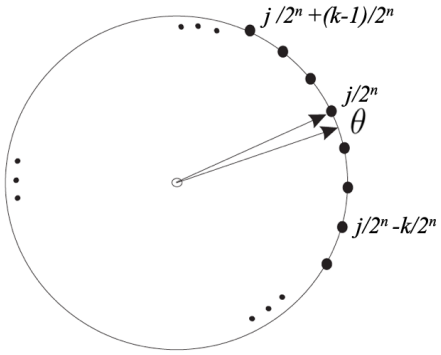


Figure 46: $\frac{j}{2^n}$ is the multiple of $\frac{1}{2^n}$ nearest to θ, surrounded by the nearer $2k$ values $\frac{m}{2^n}$ such that $|m-j| \leq k$.

Bounds: An expression for pr based on Eq. (172) has been reported [J. M. Chappell *et al.*]. Here, we use an upper bound of Eq. (173), according to the standard approach (Nielsen and Chuang, p. 224), by considering that the numerator $|e^{i2\pi(2^n\epsilon - l)} - 1| \leq 2$, while for the denominator we note that $|e^{i\phi} - 1| = |e^{i\phi/2} - e^{-i\phi/2}| = 2|\sin(\phi/2)| \geq 2|\phi|/\pi$, when $0 < \phi < \pi$ (since $\left|\sin\left(\frac{\phi}{2}\right)\right| \geq \left|\frac{\phi}{2}\right| \frac{2}{\pi}$). In particular, when $\phi = 2\pi(\epsilon - l/2^n)$, we find $|e^{i\phi} - 1| \geq 4|(\epsilon - l/2^n)|$. We note that $-\pi \geq \phi = 2\pi(\epsilon - l/2^n) \geq \pi$ since $-2^{n-1} + 1 \leq l \leq 2^{n-1}$. So, substituting l by its upper bound 2^{n-1} we find $2\pi(\epsilon - 2^{-1}) \leq 2\pi(\epsilon - l/2^n)$ –*i.e.,* $-\pi \leq 2\pi(\epsilon - l/2^n)$. Analogously, substituting l by its lower bound $-2^{n-1} + 1$ we find $2\pi(\epsilon - l/2^n) \leq 2\pi(\epsilon + 2^{-1} - 1/2^n)$, and since $\epsilon < 2^{-n}$, we obtain $2\pi(\epsilon - l/2^n) \leq \pi$.

Substituting $|e^{i2\pi(2^n\epsilon - l)} - 1| \leq 2$ and $|e^{i2\pi(\epsilon - l/2^n)} - 1| > 4(\epsilon - l/2^n)$ into Eq. (174), we obtain:

$$P_{l+j} < \frac{1}{2^{2n}}\left|\frac{1}{2(\epsilon - l/2^n)}\right|^2. \tag{174}$$

Substituting Eq. (174) into Eq. (172), we obtain:

$$\begin{aligned} pr(|m-j|>k) &\leq \frac{1}{4}\left(\sum_{l=-2^{n-1}+1}^{-k}\frac{1}{(2^n\epsilon-l)^2}+\sum_{l=k+1}^{2^{n-1}}\frac{1}{(2^n\epsilon-l)^2}\right), \\ &< \frac{1}{4}\left(\sum_{l=-2^{n-1}+1}^{-k}\frac{1}{l^2}+\sum_{l=k+1}^{2^{n-1}}\frac{1}{(l-1)^2}\right), \\ &< \frac{1}{4}\left(\sum_{\tilde{l}=2^{n-1}-1}^{k}\frac{1}{\tilde{l}^2}+\sum_{\tilde{l}=k}^{2^{n-1}-1}\frac{1}{\tilde{l}^2}\right), \end{aligned} \tag{175}$$

where the second line is obtained by using $2^n\epsilon < 1$. In the third line, we introduced the variable transformation $\tilde{l} = -l$ for the first sum, and $\tilde{l} = (l-1)$ for the second sum. Therefore,

$$\begin{aligned} pr(|m-j|>k) &< \frac{1}{2}\sum_{l=k}^{2^{n-1}-1}\frac{1}{l^2}, \\ &< \frac{1}{2}\int_{k-1}^{2^{n-1}-1} dl\frac{1}{l^2} < \frac{1}{2}\int_{k-1}^{\infty} dl\frac{1}{l^2}, \\ &< \frac{1}{2(k-1)}. \end{aligned} \tag{176}$$

According to Eq. (176), it is clear that the phase is estimated within an error of $k/2^n$ with probability at least as high as $pr(|m-j|<k) = 1-\frac{1}{2(k-1)}$.

Therefore, to estimate θ up to the first r bits (with $r < n$) –*i.e.*, with an accuracy of 2^{-r}, we choose $k \leq 2^{n-r}-1$. Using $n = r+p$ qubits in the first register, the probability of estimating θ within the desired error margin is at least $1-\frac{1}{2(k-1)} = 1-\frac{1}{2(2^p-2)} = 1-pr$. Solving for p in terms of pr, we obtain: $p = \log_2(2+\frac{1}{2pr})$. Therefore, the number of qubits necessary to estimate θ with accuracy of 2^{-r} and probability of success higher than $1-pr$ is $n = r+\log_2(2+\frac{1}{2pr})$.

Probabilistic Algorithm: As shown above, the probability of measuring all n bits of j correctly (such that $|j/2^n-\theta| \leq 2^{-n}$) is higher the 40%. Here, we show that a probabilistic algorithm can increase that probability by running the calculations with more bits, taking the most commonly occurring outcome and round it to n bits of precision. In fact, the probability of getting the correct n bits with that algorithm approaches 100 % exponentially fast with the number of times the procedure is repeated. The tools necessary to show the capabilities of probabilistic algorithms are the following bounds, or inequalities, of the tails of probability distributions.

Markov's inequality: The Markov's inequality is based on the *first moment* $E(x)$ of a distribution of a random variable x, as follows:

$$P(x \geq \epsilon) \leq \frac{E(x)}{\epsilon}, \tag{177}$$

indicating that the probability $P(x \geq \epsilon)$ of sampling $x \geq \epsilon$ in the *upper tail* of the distribution is always less or equal than the average value $E(x)$ divided by ϵ. To prove Eq. (177), we consider that $E(x) = P(x \geq \epsilon)E(x \geq \epsilon) + P(x \leq \epsilon)E(x \leq \epsilon)$, with $E(x \leq \epsilon) \geq 0$ and $E(x \geq \epsilon) \geq \epsilon$. Therefore, $E(x) \geq P(x \geq \epsilon)\epsilon$, as shown in Eq. (177).

Substituting $\epsilon = \alpha E(x)$, into Eq. (177), we obtain another equivalent form of Markov's inequality:

$$P(x \geq \alpha E(x)) \leq \frac{1}{\alpha}. \tag{178}$$

Chebyshev inequality: Introducing into Eq. (177) the substitution $x = (y - E(y))^2$ with $E(x) = E((y - E(y))^2) = \sigma^2(y)$, and $\epsilon = \alpha^2$, we obtain:

$$P((y - E(y))^2 \geq \alpha^2) \leq \frac{\sigma^2(y)}{\alpha^2}, \tag{179}$$

or

$$P(|y - E(y)| \geq \alpha) \leq \frac{\sigma^2(y)}{\alpha^2}, \tag{180}$$

so

$$P\left(|y - E(y)| \geq \alpha\sigma(y)\right) \leq \frac{1}{\alpha^2}, \tag{181}$$

which is known as the *Chebyshev inequality* based on the *second moment* $\sigma(y)$ of the distribution of the random variable y.

Chernoff Bounds: Here, we obtain sharper bounds (specifically, *exponential bounds* rather than polynomial bounds as those based on the first and second moments of the distribution) by addressing the particular case of a random variable x defined as the sum of independent random bits $x_j = \{0, 1\}$ with equal probabilities $p_j = p$ that $x_j = 1$. A simple example is a sequence of measurements $j = 1, \cdots n$, with probability p_j that the j-th measurement provides the correct output (*e.g.*, the result within a given range). So, $x = \sum_j x_j$ and $E(x) = \sum_j p_j = np$.

According to Eq. (177), $P(y \geq \epsilon) \leq \frac{E(y)}{\epsilon}$, so introducing the variable transformation $y = e^{tx}$ with $t > 0$ and $\epsilon = e^{ta}$, we obtain:

$$\begin{aligned} P(e^{tx} \geq e^{ta}) &\leq \frac{E(e^{tx})}{e^{ta}}, \\ P(x \geq a) &\leq e^{-ta} \prod_j E(e^{tx_j}), \end{aligned} \tag{182}$$

giving a bound for the *upper tail* of the probability distribution.

When all independent measurements j have the same probability of success (*i.e.*, $x_j = 1$), the variables are called Bernoulli random variables $x_1, \cdots, x_n$ with $p_j = p \geq 0$, and we obtain:

$$\begin{aligned} E[e^{tx_j}] &= pe^t + (1 - p), \\ &= 1 + p(e^t - 1), \\ &\leq e^{p(e^t - 1)} = 1 + p(e^t - 1) + \cdots, \end{aligned} \tag{183}$$

with $e^t > 1$.

For any $\delta > 0$, defining $t = \ln(1 + \delta) > 0$, we obtain:

$$E[e^{tx_j}] < e^{p\delta}, \tag{184}$$

and

$$e^{-ta} = \frac{1}{(1 + \delta)^a}. \tag{185}$$

Substituting Eqs. (184) and (185) into Eq. (182), with $a = (1 + \delta)np$, where n is an arbitrary integer, we obtain:

$$P(x \geq (1 + \delta)np) < \frac{e^{np\delta}}{(1 + \delta)^{(1+\delta)np}} = \left[\frac{e^{\delta}}{(1 + \delta)^{(1+\delta)}}\right]^{np}. \tag{186}$$

Considering that for this case $E(x) = np$, we obtain the *upper tail* Chernoff bound (Theorem A.1.4 in Kaye-Laflamme-Mosca *Introduction to Quantum Computing*):

$$P(x \geq (1+\delta)E(x)) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{E(x)}. \tag{187}$$

Another form of the *upper tail* Chernoff bound can be found by taking the logarithm of the r.h.s. of Eq. (187) and using the inequality $\log(1+\delta) \geq \delta/(1+\delta/2)$, we obtain:

$$E(x)(\delta - (1+\delta)\log(1+\delta)) \leq -\frac{E(x)\delta^2}{(2+\delta)}, \tag{188}$$

which allows us to obtain a more convenient (although looser bound) than Eq. (187) by exponentiating, as follows:

$$P(x \geq (1+\delta)E(x)) < e^{-\frac{E(x)\delta^2}{(2+\delta)}}. \tag{189}$$

Analogously, we can obtain a Chernoff bound for the *lower tail* of the probability distribution by introducing the random variable $y = e^{-tx}$ with $t > 0$ and substituting into the Markov's inequality $P(y \geq \epsilon) \leq \frac{E(y)}{\epsilon}$ given by Eq. (182), with $\epsilon = e^{-ta}$:

$$\begin{aligned} P(e^{-tx} \geq e^{-ta}) = P(x \leq a) &\leq \frac{E(e^{-tx})}{e^{-ta}}, \\ &\leq e^{ta}\prod_j E(e^{-tx_j}), \end{aligned} \tag{190}$$

Defining $t = -\ln(1-\delta) > 0$, we obtain:

$$e^{-ta} = (1-\delta)^a. \tag{191}$$

and

$$\begin{aligned} E[e^{-tx_j}] &= pe^{-t} + (1-p), \\ &= 1 + p(e^{-t} - 1), \\ &\leq e^{p(e^{-t}-1)}. \end{aligned} \tag{192}$$

Substituting Eqs. (191) and (192) into Eq. (190), with $a = (1-\delta)np$, we obtain:

$$\begin{aligned} P(x \leq (1-\delta)np) &\leq \frac{E(e^{-tx})}{e^{-ta}}, \\ &\leq \frac{e^{np(e^{-t}-1)}}{(1-\delta)^{(1-\delta)np}}, \\ &\leq \left[\frac{e^{(1-\delta)-1}}{(1-\delta)^{(1-\delta)}}\right]^{np}, \\ &\leq \left[\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right]^{np}. \end{aligned} \tag{193}$$

Another form of the Chernoff bound for the *lower tail* can be obtained by taking the log of the r.h.s. of Eq. (193) and using the inequality $\ln(1-\delta) \geq \delta(\delta/2-1)/(1-\delta)$, when $0 \leq \delta \leq 1$, we obtain:

$$\begin{aligned} np(-\delta - (1-\delta)\ln(1-\delta)) &\leq np(-\delta - \delta(1-\delta)(\delta/2-1)/(1-\delta)), \\ &\leq -np\delta^2/2. \end{aligned} \tag{194}$$

Exponentiating both sides of Eq. (194), we obtain:

$$\left[\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right]^{np} \leq \left[e^{-\frac{\delta^2}{2}}\right]^{np}, \tag{195}$$

and substituting into Eq. (193), we obtain (Theorem A.1.3 in Kaye-Laflamme-Mosca *Introduction to Quantum Computing*):

$$\boxed{P(x \leq (1-\delta)np) \leq \left[e^{-\frac{\delta^2}{2}}\right]^{np}} \tag{196}$$

Practical Application: The Chernoff bound, introduced by Eq. (196), can be used to show that the probability of failure (*i.e.*, the probability that the algorithm gives an answer outside the desired error range from the expected value) is reduced exponentially (*i.e.*, as γ^n with $0 < \gamma < 1$, and n the number of times the algorithm is repeated with the same input). Therefore, we can amplify the success probability of a bounded-error algorithm simply by repeating the algorithm.

Considering that the probability of success of a single trial is $p = 1/2 + \beta/2$, with $0 < \beta < 1$, the number of times the algorithm is expected to provide the correct answer in n trials is $E(x) = np = \frac{n}{2}(1+\beta)$. Defining $\delta = \beta/(1+\beta)$ and substituting into Eq. (196), we obtain:

$$P(x \leq \frac{n}{2}) \leq e^{-\frac{\beta^2 n}{4(1+\beta)}} = \gamma^n, \tag{197}$$

with $\gamma = e^{-\frac{\beta^2}{4(1+\beta)}}$. Eq. (197) gives the probability that the correct answer is obtained in fewer than 50 % of the times after n trials. Therefore, obtaining the correct answer more that 50 % of the times with probability higher than $1-\epsilon$ requires n to be sufficiently large so that $\gamma^n < \epsilon$, as follows:

$$\begin{aligned} e^{-\frac{n\beta^2}{4(1+\beta)}} &< \epsilon, \\ n &> \frac{4(1+\beta)}{\beta^2} \ln\left(\frac{1}{\epsilon}\right). \end{aligned} \tag{198}$$

Coin bias estimation: The Chernoff bounds are useful for all kinds of probabilistic calculations, beyond the problem of phase estimation. Perhaps the simplest example is the problem of estimating the bias of a coin, as determined by a slight curvature (Fig. 47), so it lands with heads facing up more often than 50 % of the times (*i.e.*, with probability $p = 1/2 + \beta/2$, with $0 < \beta < 1$). So the problem is: how many times n do we need to toss the coin to determine the bias within δ with probability higher than $1-\epsilon$?

Figure 47 shows that, after tossing the coin n times, j is the integer nearest to the expectation value $\bar{n}_h = n(1+\beta)/2$. Therefore, j gives the best possible estimator of the bias β, as follows: $E(\beta) = (2j-n)/n$, with probability that j of the n times shows heads facing up:

$$P_j^{(n)} = \left(\begin{array}{c} n \\ j \end{array}\right) p^j (1-p)^{n-j}. \tag{199}$$

According to Eq. (198), when $n > \frac{4(1+\beta)}{\beta^2}\ln\left(\frac{1}{\epsilon}\right)$, we will obtain heads up for more than half of the times (*i.e.*, $j > n/2$), with probability at least $1-\epsilon$.

Figure 47: Top: Bent penny with probability of landing with heads up $> 50\%$. Bottom: Possible number of heads up $n_h = 0 - n$, upon tossing the penny n times, with $\bar{n}_h$ the expectation value of n_h, determined by the bias β, and j the integer nearest to $\bar{n}_h$.

16.3 Period Finding

The Fourier transform of a periodic superposition state $|f\rangle = \sum_{j=0}^{n-1} \alpha_j |j\rangle$, with period r, is a periodic superposition $|\tilde{f}\rangle = \sum_{j=0}^{n-1} \beta_j |j\rangle$ with period n/r. A particular example is the function with only one non-zero coefficient per period (*i.e.*, a total of n/r non-zero coefficients), $\alpha_j = \sqrt{\frac{r}{n}}$, for $j = k_j r$, with $k_j = 0, 1, 2, \cdots, n/r - 1$, so that $|f\rangle = \sqrt{\frac{r}{n}} \sum_{k_j=0}^{n/r-1} |k_j r\rangle$. The discrete Fourier transform of $|f\rangle$ is

$$\begin{aligned} QFT^{(n)} \sum_{k=0}^{\frac{n}{r}-1} \sqrt{\frac{r}{n}} |kr\rangle &= \sum_{k=0}^{\frac{n}{r}-1} \sqrt{\frac{r}{n}} \frac{1}{\sqrt{n}} \sum_{l=0}^{n-1} w^{lkr} |l\rangle, \\ &= \sum_{k=0}^{\frac{n}{r}-1} \sqrt{\frac{r}{n}} \frac{1}{\sqrt{n}} \sum_{j=0}^{r-1} w^{njk} |j\frac{n}{r}\rangle + \sum_{k=0}^{\frac{n}{r}-1} \sqrt{\frac{r}{n}} \frac{1}{\sqrt{n}} \sum_{l'} w^{l'kr} |l'\rangle, \end{aligned} \tag{200}$$

where $j = lr/n$ while l' is not a multiple of n/r. Therefore, the second term is zero since $\sum_{k=0}^{\frac{n}{r}-1} w^{l'rk} = 0$. Considering that $w^n = 1$, we obtain:

$$QFT^{(n)} \sum_{k=0}^{\frac{n}{r}-1} \sqrt{\frac{r}{n}} |kr\rangle = \sqrt{\frac{1}{r}} \sum_{l=0}^{r-1} |l\frac{n}{r}\rangle. \tag{201}$$

Therefore, measurements of the resulting superposition state after applying the Fourier transform provide equally probable indices ln/r with $l = 0, 1, 2, \cdots, r-1$. The minimum common factor divisor of all outcomes is n/r which provides the period r since n is known.

We note that the particular state discussed in this subsection (*i.e.*, a superposition state with only one non-zero coefficient per period) could be prepared by measuring the ancilla bits prepared

in state $|f(x)\rangle$ when the state of the circuit is in the superposition state $|\psi\rangle = 2^{-n/2}\sum_{x\in\{0,1\}^n}|x\rangle|f(x)\rangle$, as right after applying the unitary of the function in the following circuit:

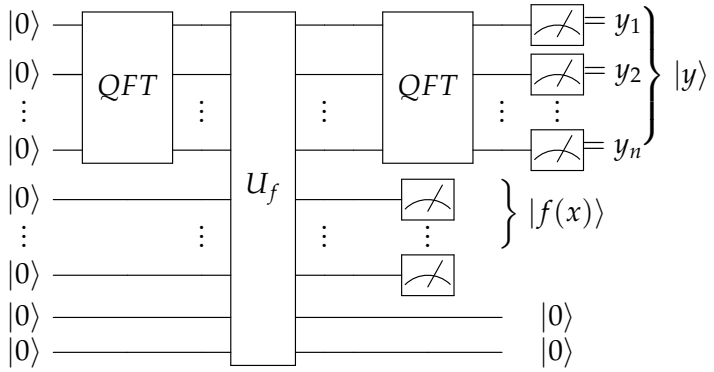


Figure 48: Quantum circuit for determining the period of $f(x)$ by first preparing a superposition state with only one non-zero coefficient per period in the first n qubits, by measuring the n ancilla bits prepared in state $f(x)$, when the first n working bits are in a uniform superposition. The subsequent QFT then generates a state with non-zero coefficients for those strings with indices that are n/r periodic.

We note that a measurement reporting a value of the function $f(x_j)$ would leave the state in the superposition,

$$|\psi\rangle = \sqrt{\frac{r}{n}}\sum_{k=0}^{\frac{n}{r}-1}|x_j + kr\rangle|f(x_j)\rangle, \tag{202}$$

for which the Fourier transform of the first n-bits is shift-invariant (Property 1). Therefore,

$$\begin{aligned} QFT^{(n)}\sqrt{\frac{r}{n}}\sum_{k=0}^{\frac{n}{r}-1}|x_j + kr\rangle|f(x_j)\rangle &= QFT^{(n)}\sqrt{\frac{r}{n}}\sum_{k=0}^{\frac{n}{r}-1}|kr\rangle|f(x_j)\rangle, \\ &= \sqrt{\frac{1}{r}}\sum_{l=0}^{r-1}|l\frac{n}{r}\rangle|f(x_j)\rangle. \end{aligned} \tag{203}$$

At this point we must mention that, according to the so-called *principle of deferred measurement*, it is not even necessary to measure $f(x)$ since measuring commutes with conditioning (*i.e.,* a measurement can be replaced by a CNOT with with an acilla that is measured at the end).

16.4 Shor's Algorithm

The goal of this section is to explain Shor's algorithm for finding the prime factors of an integer N, as applied to finding the prime factors 7 and 3 of $N = 21$.

Shor's algorithm is essentially the period finding algorithm, discussed in the previous section, applied for the specific case of the function $f(x) = m^x (\text{mod } N)$, where m is a random integer $m < N$. As an example, we choose $m = 2$ and we evaluate $f(x)$ for $x = 1, 2, 3, \cdots$, as shown in the table below, by considering that $f(x)$ is the remainder of $\frac{m^x}{N}$, or $m^x = Nq + f(x)$, where q is an integer.

x	2^x	$f(x)$
0	1	1
1	2	2
2	4	4
3	8	8
4	16	16
5	32	11
6	64	1
7	128	2
8	256	4
9	512	8
10	1024	16
11	2048	11
12	4096	1
	...	

We note that $f(x)$ is a single-valued periodic function, with period $r = 6$ that could be efficiently resolved by the period finding algorithm described in the previous section (Fig. 48). Having found the period r, we know that $1 = 2^r (\text{mod } 21)$, or $1 = y^2 (\text{mod } 21)$ with $y = 2^{r/2} = 2^3 = 8$. Therefore, $1 = 8^2 (\text{mod } 21)$, or $8^2 = 1 (\text{mod } 21)$. So, by period finding we found that $y = 8$ is a *non-trivial root* of unity $(\text{mod } N)$, since $8^2 = 1 (\text{mod } 21)$ and $(8+1) \cdot (8-1) = 0 (\text{mod } 21)$. So, we find that 21 divides $(8+1) \cdot (8-1)$, although it is not divided either $(8+1)$ or $(8-1)$. How could this be possible? Well, by writing 21 in terms of its prime factors, we find that $21 = 3 \cdot 7$, so 3 divides $(8+1)$ and 7 divides $(8-1)$ (*i.e.*, the prime factors of N are the greatest common divisors $\gcd(N, y+1)$ and $\gcd(N, y-1)$ which can be found very efficiently by Euclid's algorithm. The algorithm has a success rate of at least 50% since r is even number half of the times and occasionally, we find that $\gcd(N, m) > 1$ (*i.e.*, we find a prime factor for free through the Euclid's algorithm). If r is odd, we fail, since we cannot find y, so we need to pick another random number m and try again.

17 Appendix I: Golden Rule

The goal of this section is to introduce the so-called *Fermi Golden Rule* expression,

$$\Gamma = \frac{2\pi}{\hbar}\int_{-\infty}^{\infty} dE_f \rho(E_f)|\langle f|\hat{A}|i\rangle|^2 \delta(E_f - (E_i + \hbar w)), \tag{204}$$

giving the rate of decay of a system initially prepared in state $|i\rangle$ due to coupling with states $|f\rangle$, with density of states $\rho(E_f)$, as described by first-order time dependent perturbation theory.

We consider a system initially prepared in state $|i\rangle$. At time $t = 0$, we turn on the perturbation $W(t)$ and we analyze the decay to the final state $|f\rangle$, as described by first order time-dependent perturbation theory:

$$c_f(t) = -\frac{i}{\hbar}\int_0^t dt' \langle f|\hat{W}(t')|i\rangle e^{\frac{i}{\hbar}(E_f - E_i)t'}, \tag{205}$$

Therefore, the probability of observing the system in the final state is

$$P_{fi}(t) = \frac{1}{\hbar^2}\int_0^t dt'' \int_0^t dt' \langle i|\hat{W}^*(t'')|f\rangle\langle f|\hat{W}(t')|i\rangle e^{\frac{i}{\hbar}(E_f - E_i)(t'-t'')}, \tag{206}$$

17.1 Monochromatic Plane Wave

Assuming that the perturbation involves a single frequency component, $\hat{W}(t') = \hat{A}e^{-iwt'}$, we obtain:

$$\begin{aligned} c_f(t) &= \langle f|\hat{A}|i\rangle \frac{[1 - e^{i(w_{fi} - w)t}]}{\hbar(w_{fi} - \omega)}, \\ &= -\frac{i}{\hbar} t \langle f|\hat{A}|i\rangle e^{i(w_{fi} - w)t/2} \frac{\sin[(w_{fi} - w)t/2]}{(w_{fi} - \omega)t/2}. \end{aligned} \tag{207}$$

Therefore, the probability of observing the system in the final state is

$$P_{fi}(t) = \frac{t^2}{\hbar^2}|\langle f|\hat{A}|i\rangle|^2 \frac{\sin^2[(w_{fi} - w)t/2]}{[(w_{fi} - \omega)t/2]^2}. \tag{208}$$

To compute the survival probability that the system remains in the initial state, we must add up the probability over all possible final states,

$$\begin{aligned} P(t) &= 1 - \frac{t^2}{\hbar^2}\sum_f |\langle f|\hat{A}|i\rangle|^2 \frac{\sin^2[(w_{fi} - w)t/2]}{[(w_{fi} - \omega)t/2]^2} \\ &= 1 - \frac{t^2}{\hbar^2}\int_{-\infty}^{\infty} dE_f \rho(E_f)|\langle f|\hat{A}|i\rangle|^2 \frac{\sin^2[(w_{fi} - w)t/2]}{[(w_{fi} - \omega)t/2]^2} \end{aligned} \tag{209}$$

If the very short time limit, $P(t) = exp(-\alpha t^2) \approx 1 - \alpha t^2 + \cdots$, where

$$\begin{aligned} \alpha &= \lim_{t\to 0}\frac{1}{\hbar^2}\int_{-\infty}^{\infty} dE_f |\langle f|\hat{A}|i\rangle|^2 \rho(E_f) \frac{\sin^2[(E_f - E_i - \hbar w)t/(2\hbar)]}{[(E_f - E_i - \hbar w)t/(2\hbar)]^2}, \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dE_f |\langle f|\hat{A}|i\rangle|^2 \rho(E_f), \end{aligned} \tag{210}$$

In the longer time limit, the kernel of Eq. (209) is approximated as the delta function to obtain:

$$\begin{aligned} P(t) &= 1 - \frac{t}{\hbar^2}\int_{-\infty}^{\infty} d(tE_f)\rho(E_f)|\langle f|\hat{A}|i\rangle|^2\pi\delta((E_f t - (E_i+\hbar w)t)/(2\hbar)) \\ &= 1 - t\frac{2\pi}{\hbar}\int_{-\infty}^{\infty} d\xi\rho(\xi 2\hbar/t)|\langle f|\hat{A}|i\rangle|^2\delta(\xi - (E_i+\hbar w)t/(2\hbar)) \\ &= 1 - t\frac{2\pi}{\hbar}\rho(E_i+\hbar w)|\langle E_i+\hbar w|\hat{A}|i\rangle|^2 \end{aligned} \tag{211}$$

so $P(t) = exp(-\Gamma t) \approx 1 - \Gamma t + \cdots$, where

$$\begin{aligned} \Gamma &= \frac{2\pi}{\hbar}\rho(E_i+\hbar w)|\langle E_i+\hbar w|\hat{A}|i\rangle|^2, \\ &= \frac{2\pi}{\hbar}\int_{-\infty}^{\infty} dE_f\rho(E_f)|\langle f|\hat{A}|i\rangle|^2\delta(E_f - (E_i+\hbar w)), \end{aligned} \tag{212}$$

or as a discrete sum over states,

$$\Gamma = \frac{2\pi}{\hbar}\sum_f |\langle f|\hat{A}|i\rangle|^2\delta(E_f - E_i - \hbar w), \tag{213}$$

which is known as Fermi's Golden rule.

Without introducing the approximation of the kernel of Eq. (209), we obtain:

$$\begin{aligned} P(t) &= 1 - \frac{t^2}{\hbar^2}\int_{-\infty}^{\infty} dE_f\rho(E_f)|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2[(w_{fi}-w)t/2]}{[(w_{fi}-\omega)t/2]^2} \\ &= 1 - t\frac{2}{\hbar}\int_{-\infty}^{\infty} dE_f\rho(E_f)\frac{t}{2\hbar}|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2((E_f-E_i-\hbar w)t/(2\hbar))}{((E_f-E_i-\hbar\omega)t/(2\hbar))^2} \\ &= 1 - t\frac{2}{\hbar}\int_{-\infty}^{\infty} d\xi\rho(\xi 2\hbar/t)|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2(\xi-(E_i+\hbar w)t/(2\hbar))}{(\xi-(E_i+\hbar\omega)t/(2\hbar))^2} \\ &= 1 - t\frac{2}{\hbar}\int_{-\infty}^{\infty} d\xi\rho(\xi 2\hbar/t)|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2(\xi)}{\xi^2} \end{aligned} \tag{214}$$

which gives, in the time-range when the decay is exponential (i.e., $P(t) = exp(-\Gamma t) \approx 1 - \Gamma t$),

$$\begin{aligned} \Gamma &= \frac{2}{\hbar}\int_{-\infty}^{\infty} d\xi\rho(\xi 2\hbar/t)|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2(\xi)}{\xi^2}, \\ &= \frac{2}{\hbar}\int_{-\infty}^{\infty} dE_f\rho(E_f)|\langle f|\hat{A}|i\rangle|^2\frac{\sin^2((E_f-E_i-\hbar w)t/(2\hbar))}{(E_f-E_i-\hbar\omega)^2 t/(2\hbar)}. \end{aligned} \tag{215}$$

Substituting the delta function in Eq. (213) by its integral form, we obtain:

$$\begin{aligned} \Gamma_{fi} &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt\langle f|\hat{A}|i\rangle\langle i|\hat{A}|f\rangle e^{\frac{i}{\hbar}(E_f-E_i-\hbar w)t}, \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt\langle f|e^{\frac{i}{\hbar}\hat{H}t}\hat{A}e^{-\frac{i}{\hbar}\hat{H}t}|i\rangle\langle i|\hat{A}|f\rangle e^{-iwt}, \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}A_{fi}(t)A_{if}(0), \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}A_{fi}(t)A_{if}(0). \end{aligned} \tag{216}$$

The equilibrium ensemble average is

$$\begin{aligned}\langle\Gamma_{fi}\rangle &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\sum_{\alpha} p_\alpha \langle\alpha|A_{fi}(t)A_{if}(0)|\alpha\rangle, \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\langle A_{fi}(t)A_{if}(0)\rangle, \\ &= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\langle[A_{fi}(t)A_{if}(0)+A_{fi}(0),A_{if}(t)]\rangle,\end{aligned} \tag{217}$$

where $p_\alpha = Z^{-1}e^{-\beta E_\alpha}$ with $Z = \sum_\alpha e^{-\beta E_\alpha}$.

The rest of this subsection shows that, according to Eq. (217), $\langle\Gamma_{fi}\rangle$ can be written as follows:

$$\langle\Gamma_{fi}\rangle = \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\langle[A_{fi}(t)A_{if}(0)+A_{fi}(0),A_{if}(t)]\rangle, \tag{218}$$

where the symmetrized correlation function $C(t) = A_{fi}(t)A_{if}(0)+A_{fi}(0),A_{if}(t)$ is real, and is an even function of time just like its classical analogue correlation function. Therefore, Eq. (218) has often been used for estimations of $\langle\Gamma_{fi}\rangle$ based on classical simulations. However, it has been pointed out by Berne and co-workers that the classical version of $C(t)$ underestimates $\langle\Gamma_{fi}\rangle$ by a factor of $(\beta\hbar\omega/2)\coth(\beta\hbar\omega/2)$ [J. Chem. Phys. (1994) 100: 8359-8366].

The derivation of the last line of Eq. (218) is as follows:

$$\begin{aligned}\langle\Gamma_{fi}\rangle &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle, \\ &= \frac{1}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\frac{[1+e^{-\beta\hbar\omega}]}{[1+e^{-\beta\hbar\omega}]}\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle, \\ &= \frac{2\pi}{\hbar}\frac{1}{[1+e^{-\beta\hbar\omega}]}\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(0)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle\delta(E_\gamma-E_\alpha-\hbar\omega)[1+e^{-\beta\hbar\omega}], \\ &= \frac{2\pi}{\hbar}\frac{1}{[1+e^{-\beta\hbar\omega}]}\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(0)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle\delta(E_\gamma-E_\alpha-\hbar\omega)[1+e^{-\beta(E_\gamma-E_\alpha)}], \\ &= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle[1+e^{-\beta(E_\gamma-E_\alpha)}], \\ &= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\left[\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle\right. \\ &\qquad \left.+\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle e^{-\beta(E_\gamma-E_\alpha)}\right], \\ &= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2}\int_{-\infty}^{\infty} dt e^{-iwt}\left[\sum_{\alpha,\gamma} p_\alpha \langle\alpha|A_{fi}(t)|\gamma\rangle\langle\gamma|A_{if}(0)|\alpha\rangle\right. \\ &\qquad \left.+\sum_{\alpha,\gamma} p_\gamma \frac{p_\alpha}{p_\gamma}\langle\gamma|A_{if}(0)|\alpha\rangle\langle\alpha|A_{fi}(t)|\gamma\rangle e^{-\beta(E_\gamma-E_\alpha)}\right],\end{aligned} \tag{219}$$

$$
\begin{aligned}
\langle \Gamma_{fi} \rangle &= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2} \int_{-\infty}^{\infty} dt e^{-iwt} \left[\sum_{\alpha,\gamma} p_\alpha \langle \alpha | A_{fi}(t) | \gamma \rangle \langle \gamma | A_{if}(0) | \alpha \rangle + \sum_{\alpha,\gamma} p_\gamma \langle \gamma | A_{if}(0) | \alpha \rangle \langle \alpha | A_{fi}(t) | \gamma \rangle \right], \\
&= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2} \int_{-\infty}^{\infty} dt e^{-iwt} \sum_{\alpha,\gamma} p_\alpha \left[\langle \alpha | A_{fi}(t) | \gamma \rangle \langle \gamma | A_{if}(0) | \alpha \rangle + \langle \alpha | A_{if}(0) | \gamma \rangle \langle \gamma | A_{fi}(t) | \alpha \rangle \right], \\
&= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2} \int_{-\infty}^{\infty} dt e^{-iwt} \sum_{\alpha} p_\alpha \left[\langle \alpha | A_{fi}(t) A_{if}(0) + A_{if}(0) A_{fi}(t) | \alpha \rangle \right], \\
&= \frac{[1+e^{-\beta\hbar\omega}]^{-1}}{\hbar^2} \int_{-\infty}^{\infty} dt e^{-iwt} \langle [A_{fi}(t) A_{if}(0) + A_{if}(0) A_{fi}(t)] \rangle, \\
&= \frac{[1+e^{-\beta\hbar\omega_{fi}}]^{-1}}{\hbar^2} \int_{-\infty}^{\infty} dt e^{-iwt} \langle [A_{fi}(t) A_{if}(0) + A_{if}(0) A_{fi}(t)] \rangle,
\end{aligned}
\tag{220}
$$

18 Appendix II: Coherent States

The goal of this section is to introduce coherent states $|\alpha\rangle$, as discussed in the quantum optics community for descriptions of states of coherent light as eigenstates of the anihilation operator:

$$\hat{a}|\alpha\rangle = \alpha|\alpha\rangle, \tag{221}$$

where α is a complex number. A coherent state simply has a precise phase, defined by the complex amplitude α, although an indefinite number of photons as in a laser beam. In contrast, a state with a fixed number of photons usually has completely arbitrary (random) phase.

The rest of this section shows that we can create coherent states, as follows:

$$\hat{D}(\alpha)|0\rangle = |\alpha\rangle, \tag{222}$$

where $|0\rangle$ is the vacuum state defined as the ground state of the harmonic oscillator, and $\hat{D}$ is the displacement operator, defined as follows:

$$\begin{aligned} \hat{D}(\alpha) &= e^{\alpha\hat{a}^\dagger - \alpha^*\hat{a}}, \\ &= e^{\alpha\hat{a}^\dagger} e^{-\alpha^*\hat{a}} e^{-\frac{1}{2}|\alpha|^2}, \end{aligned} \tag{223}$$

The second row is obtained from the first one, making use of the Hausdorff formula $e^{A+B} = e^A e^B e^{-\frac{1}{2}[A,B]}$, with $A = \alpha\hat{a}^\dagger$ and $B = -\alpha^*\hat{a}$, which is valid if $[A,[A,B]] = [B,[A,B]] = 0$ as in this case. Note that $[A,B] = -|\alpha|^2[\hat{a}^\dagger,\hat{a}] = |\alpha|^2$ since $[\hat{a}^\dagger,\hat{a}] = -1$.

We also note that the inverse must be

$$\begin{aligned} \hat{D}(\alpha)^{-1} &= e^{\frac{1}{2}|\alpha|^2} e^{\alpha^*\hat{a}} e^{-\alpha\hat{a}^\dagger}, \\ &= e^{-\frac{1}{2}|\alpha|^2} e^{-\alpha\hat{a}^\dagger} e^{\alpha^*\hat{a}} = \hat{D}(-\alpha), \end{aligned} \tag{224}$$

since $\hat{D}(\alpha)^{-1}\hat{D}(\alpha) = 1$. The second row of Eq. (224) is obtained from the first one since

$$e^{\alpha^*\hat{a}} e^{-\alpha\hat{a}^\dagger} = e^{-\alpha\hat{a}^\dagger} e^{\alpha^*\hat{a}} e^{-|\alpha|^2}. \tag{225}$$

Note that multiplying both sides of Eq. (225) by $e^{\alpha\hat{a}^\dagger} e^{-\alpha^*\hat{a}} = \hat{D}(\alpha) e^{\frac{1}{2}|\alpha|^2}$, we obtain:

$$\begin{aligned} 1 &= \hat{D}(\alpha) e^{\frac{1}{2}|\alpha|^2} \hat{D}(-\alpha) e^{\frac{1}{2}|\alpha|^2} e^{-|\alpha|^2}. \\ 1 &= 1. \end{aligned} \tag{226}$$

Therefore, according to Eq. (224),

$$\hat{D}(\alpha)^{-1} = \hat{D}(-\alpha) = \hat{D}(\alpha)^\dagger. \tag{227}$$

The Backer Campbell Hausdorff relation,

$$e^A B e^{-A} = B + [A,B] + \frac{1}{2}[A,[A,B]] + \cdots, \tag{228}$$

can be used with $A = -\alpha\hat{a}^\dagger + \alpha^*\hat{a}$, and $B = \hat{a}$ to show that

$$\hat{D}(\alpha)^\dagger \hat{a} \hat{D}(\alpha) = \hat{a} + \alpha, \tag{229}$$

since $[A, B] = [-\alpha\hat{a}^\dagger + \alpha^*\hat{a}, \hat{a}] = \alpha$, and therefore $[A, [A, B]] = 0$. Applying Eq. (229) to the vacuum state $|0\rangle$, we obtain:

$$\hat{D}(\alpha)^\dagger \hat{a} \hat{D}(\alpha)|0\rangle = \alpha|0\rangle, \tag{230}$$

since $\hat{a}|0\rangle = 0$, or

$$\hat{a}\hat{D}(\alpha)|0\rangle = \alpha\hat{D}(\alpha)|0\rangle, \tag{231}$$

since $\hat{D}(\alpha)^\dagger = \hat{D}(\alpha)^{-1}$. Therefore, according to Eq. (221),

$$\hat{D}(\alpha)|0\rangle = |\alpha\rangle, \tag{232}$$

which is Eq. (222).

Substituting Eq. (223) into Eq. (232), we obtain:

$$|\alpha\rangle = e^{-\frac{1}{2}|\alpha|^2} e^{\alpha\hat{a}^\dagger} e^{-\alpha^*\hat{a}}|0\rangle, \tag{233}$$

and expanding the exponentials in Taylor series (*i.e.*, $e^A = \sum_{n=0}^{\infty} \frac{1}{n\,!} A^n$, with $\hat{a}|0\rangle = 0$), we obtain:

$$\begin{aligned} |\alpha\rangle &= e^{-\frac{1}{2}|\alpha|^2} \sum_{n=0}^{\infty} \frac{\alpha^n}{n!} (\hat{a}^\dagger)^n |0\rangle, \\ &= e^{-\frac{1}{2}|\alpha|^2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle, \end{aligned} \tag{234}$$

where in the second row, we used $\hat{a}|n\rangle = \sqrt{n+1}|n+1\rangle$.

18.1 Overlap

Here, we show that coherent states are not orthogonal since according to Eq. (234), we obtain:

$$\begin{aligned} \langle\beta|\alpha\rangle &= e^{-\frac{1}{2}|\alpha|^2} e^{-\frac{1}{2}|\beta|^2} \sum_{n=0}^{\infty} \sum_{m=0}^{\infty} \frac{(\beta^*)^m \alpha^n}{\sqrt{m!n!}} \langle m|n\rangle, \\ &= e^{-\frac{1}{2}|\alpha|^2} e^{-\frac{1}{2}|\beta|^2} \sum_{n=0}^{\infty} \frac{(\beta^*)^n \alpha^n}{n!} \\ &= e^{-\frac{1}{2}|\alpha|^2} e^{-\frac{1}{2}|\beta|^2} e^{\beta^*\alpha}, \\ &= e^{-\frac{1}{2}|\beta-\alpha|^2} e^{\frac{1}{2}(\beta^*\alpha - \beta\alpha^*)} \end{aligned} \tag{235}$$

18.2 Closure

Here, we show that

$$\frac{1}{\pi} \int |\alpha\rangle\langle\alpha|\, d^2\alpha = 1 \tag{236}$$

by substituting $|\alpha\rangle$ according to Eq. (234), as follows:

$$\begin{aligned} \int |\alpha\rangle\langle\alpha|\, d^2\alpha &= \int e^{-|\alpha|^2} \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \frac{(\alpha^*)^m}{\sqrt{m!}} \langle m|\, d^2\alpha, \\ &= \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} |n\rangle\langle m| \int e^{-|\alpha|^2} \frac{\alpha^n}{\sqrt{n!}} \frac{(\alpha^*)^m}{\sqrt{m!}} d^2\alpha \end{aligned} \tag{237}$$

Now, we transform to polar coordinates: $\alpha = re^{i\theta}$, with $d^2\alpha = rdrd\theta$:

$$\begin{aligned}\int |\alpha\rangle\langle\alpha|\, d^2\alpha &= \frac{1}{\sqrt{m!n!}} \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} |n\rangle\langle m| \int_0^{\infty} dr\; e^{-r^2}\; r^{m+n+1} \int_0^{2\pi} d\theta e^{i(n-m)\theta} \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} |n\rangle\langle n|\; 2\pi\; \int_0^{\infty} dr\; e^{-r^2}\; r^{2n+1}\end{aligned} \tag{238}$$

where the second row is obtained considering that

$$\int_0^{2\pi} d\theta e^{i(n-m)\theta} = 2\pi\delta_{nm} \tag{239}$$

Introducing the variable substitution $y = r^2$ in Eq. (238), with $dy = 2rdr$, we obtain:

$$\int |\alpha\rangle\langle\alpha|\, d^2\alpha = \sum_{n=0}^{\infty} \frac{1}{n!} |n\rangle\langle n|\; \pi\; \int_0^{\infty} dy\; e^{-y}\; y^n \tag{240}$$

and considering that

$$\int_0^{\infty} dy\; e^{-y}\; y^n = n! \tag{241}$$

we obtain

$$\begin{aligned}\int |\alpha\rangle\langle\alpha|\, d^2\alpha &= \sum_{n=0}^{\infty} |n\rangle\langle n|\; \pi \\ &= \pi.\end{aligned} \tag{242}$$

18.3 Wavefunctions

The wavefunctions can be obtain by substituting into Eq. (234) the eigenfunctions of the Harmonic oscillator,

$$\langle x|n\rangle = (2^n n!)^{-1/2} \left(\frac{\omega}{\pi\hbar}\right)^{1/4} \exp(-\xi^2/2) H_n(\xi), \tag{243}$$

where $\xi = x\sqrt{\omega/\hbar}$, with H_n the n-Hermite polynomial, giving

$$\begin{aligned}\langle x|\alpha\rangle &= \left(\frac{\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{1}{2}|\alpha|^2} \sum_{n=0}^{\infty} \frac{(\alpha/\sqrt{2})^n}{n!} H_n(\xi), \\ &= \left(\frac{\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{1}{2}|\alpha|^2} e^{\frac{1}{2}\xi^2} e^{-(\xi-\alpha/\sqrt{2})^2}.\end{aligned} \tag{244}$$

Clearly, Eq. (244) shows that the vacuum state, corresponding to $n = 0$ photons has a wavefunction $\langle x|0\rangle$ that is a coherent state with $\alpha = 0$ (*i.e.*, the ground state of a harmonic oscillator with mass $m = 1$ and frequency ω).

18.4 Expectation Values

We note that the average number of photons in a coherent state $|\alpha\rangle$ is given by the square of the complex amplitude $|\alpha|^2$ (the intensity of the wave) since

$$\begin{aligned}\langle N\rangle &= \langle\alpha|\hat{a}^\dagger\hat{a}|\alpha\rangle, \\ &= \langle\alpha|\alpha^*\alpha|\alpha\rangle, \\ &= \alpha^*\alpha.\end{aligned} \tag{245}$$

The probability $P(n)$ of finding n photons is

$$\begin{aligned} P(n) &= |\langle n|\alpha\rangle|^2, \\ &= e^{-|\alpha|^2}\frac{|\alpha^n|^2}{n!}, \end{aligned} \tag{246}$$

defined by the Poisson distribution,

$$P(n) = \frac{\lambda^n e^{-\lambda}}{n!}, \tag{247}$$

with $\lambda = |\alpha|^2$.

18.4.1 Optical Equivalence Theorem

Another way of computing expectation values isby using the so-called *optical equivalence theorem*, as follows:

$$\begin{aligned} \langle\hat{O}\rangle &= \mathsf{Tr}[\hat{O}\hat{\rho}] \\ &= \int P(\alpha)\; O(\alpha^*,\alpha)\; d^2\alpha, \end{aligned} \tag{248}$$

where $O(\alpha^*,\alpha) = \langle\alpha|\hat{O}|\alpha\rangle = \sum_n\sum_m C_{nm}(\hat{a}^\dagger)^n\hat{a}^m$ is the so-called Q-representation of the *normally ordered* operator $\hat{O} = \sum_n\sum_m C_{nm}(\hat{a}^\dagger)^n\hat{a}^m$ (*i.e.*, where the annihilation operators stand to the right of the creation operators) where we replaced $\hat{a}$ and $\hat{a}^\dagger$ by a and a^*, respectively. $P(\alpha)$, is the Glauber-Sudarshan P-function, defined as follows:

$$\hat{\rho} \equiv \int P(\alpha)|\alpha\rangle\langle\alpha|\; d^2\alpha. \tag{249}$$

The derivation of the second line of Eq. (248) is, as follows:

$$\begin{aligned} \langle\hat{O}\rangle &= \mathsf{Tr}\left[\int P(\alpha)\;\hat{O}\;|\alpha\rangle\langle\alpha|\; d^2\alpha\right], \\ &= \sum_n\langle n|\int P(\alpha)|\hat{O}|\alpha\rangle\langle\alpha|n\rangle\; d^2\alpha, \\ &= \int P(\alpha)\langle\alpha|\hat{O}|\alpha\rangle\; d^2\alpha, \\ &= \sum_n\sum_m C_{nm}\int P(\alpha)\langle\alpha|(\hat{a}^\dagger)^n\hat{a}^m|\alpha\rangle\; d^2\alpha, \\ &= \int P(\alpha)\sum_n\sum_m C_{nm}(\alpha^*)^n\alpha^m\langle\alpha|\alpha\rangle\; d^2\alpha, \\ &= \int P(\alpha)\; O(\alpha^*,\alpha)\; d^2\alpha. \end{aligned} \tag{250}$$

18.4.2 P-representation of the density operator

The P-representation of the density operator $\hat{\rho}$ can be obtained (as shown by Mehta) by computing $\langle -u|\hat{\rho}|u\rangle$ according to Eq. (249), as follows:

$$\langle -u|\hat{\rho}|u\rangle = \int P(\alpha)\langle -u|\alpha\rangle\langle\alpha|u\rangle d^2\alpha, \tag{251}$$

and substituting $\langle \alpha | u \rangle$ according to Eq. (235), we obtain:

$$\begin{aligned} \langle -u|\hat{\rho}|u\rangle &= \int P(\alpha) e^{-\frac{1}{2}|u|^2 - \frac{1}{2}|\alpha|^2 - u^*\alpha} e^{-\frac{1}{2}|u|^2 - \frac{1}{2}|\alpha|^2 + \alpha^* u} d^2\alpha, \\ &= e^{-|u|^2} \int P(\alpha) e^{-|\alpha|^2} e^{\alpha^* u - u^* \alpha} d^2\alpha. \end{aligned} \tag{252}$$

Introducing the variable substitution $\alpha = x + iy$ and $u = x' + iy'$, we obtain

$$\begin{aligned} \langle -u(x',y')|\hat{\rho}|u(x',y')\rangle e^{|u(x',y')|^2} &= \int P(\alpha(x,y)) e^{-x^2-y^2} e^{(x-iy)(x'+iy') - (x'-iy')(x+iy)} dxdy \\ &= \int P(\alpha(x,y)) e^{-x^2-y^2} e^{-i2yx' + i2y'x} dxdy \end{aligned} \tag{253}$$

Therefore,

$$\begin{aligned} I(\tilde{x},\tilde{y}) &= \frac{1}{\pi^2} \int dx'dy' e^{i2x'\tilde{y} - i2y'\tilde{x}} \langle -u(x',y')|\hat{\rho}|u(x',y')\rangle e^{|u(x',y')|^2}, \\ &= \frac{1}{\pi^2} \int P(\alpha(x,y)) e^{-x^2-y^2} \int dx'dy' e^{-i2(y-\tilde{y})x' + iy'2(x-\tilde{x})} dxdy \\ &= \frac{1}{(2\pi)^2} \int P(\alpha(x,y)) e^{-x^2-y^2} \int dx''dy'' e^{-i(y-\tilde{y})x'' + iy''(x-\tilde{x})} dxdy \\ &= \int P(\alpha(x,y)) e^{-x^2-y^2} \delta(y-\tilde{y})\delta(x-\tilde{x}) dxdy \\ &= P(\alpha(\tilde{x},\tilde{y})) e^{-\tilde{x}^2-\tilde{y}^2} \end{aligned} \tag{254}$$

giving

$$\begin{aligned} P(\alpha(x,y)) &= \frac{e^{x^2+y^2}}{\pi^2} \int dx'dy' e^{ix'y - iy'x} \langle -u(x',y')|\hat{\rho}|u(x',y')\rangle e^{|u(x',y')|^2}, \\ P(\alpha) &= \frac{e^{|\alpha|^2}}{\pi^2} \int e^{-\alpha^* u + u^*\alpha} \langle -u|\hat{\rho}|u\rangle e^{|u|^2} \, d^2u. \end{aligned} \tag{255}$$

18.4.2.1 Pure coherent state For the pure coherent state $\hat{\rho} = |\beta\rangle\langle\beta|$, we have that according to Eq. (235):

$$\begin{aligned} \langle -u|\hat{\rho}|u\rangle &= e^{-\frac{1}{2}|-u-\beta|^2} e^{\frac{1}{2}(-u^*\beta + u\beta^*)} e^{-\frac{1}{2}|\beta-u|^2} e^{\frac{1}{2}(\beta^* u - \beta u^*)} \\ &= e^{-|u|^2} e^{-|\beta|^2} e^{u\beta^* - u^*\beta} \end{aligned} \tag{256}$$

Therefore, substituting Eq. (256) into Eq. (255), we obtain:

$$\begin{aligned} P(\alpha) &= \frac{e^{|\alpha|^2}}{\pi^2} \int e^{-\alpha^* u + u^*\alpha} e^{-|u|^2} e^{-|\beta|^2} e^{u\beta^* - u^*\beta} e^{|u|^2} \, d^2u, \\ &= \frac{e^{|\alpha|^2} e^{-|\beta|^2}}{\pi^2} \int e^{-\alpha^* u + u^*\alpha} e^{u\beta^* - u^*\beta} \, d^2u, \\ &= \frac{e^{|\alpha|^2} e^{-|\beta|^2}}{\pi^2} \int e^{-u(\alpha-\beta)^* + u^*(\alpha-\beta)} \, d^2u, \end{aligned} \tag{257}$$

and considering that $u = x' + iy'$, we obtain:

$$\begin{aligned} P(\alpha) &= \frac{e^{|\alpha|^2} e^{-|\beta|^2}}{\pi^2} \int e^{i\ 2x'\ \mathcal{I}(\alpha-\beta) - i\ 2y'\ \mathcal{R}(\alpha-\beta)}\ dx'\ dy', \\ &= \frac{e^{|\alpha|^2} e^{-|\beta|^2}}{(2\pi)^2} \int e^{i\ x''\ \mathcal{I}(\alpha-\beta) - i\ y''\ \mathcal{R}(\alpha-\beta)}\ dx''\ dy'', \\ &= e^{|\alpha|^2} e^{-|\beta|^2}\ \delta(\mathcal{I}(\alpha-\beta))\ \delta(\mathcal{R}(\alpha-\beta)), \\ &= \delta(\mathcal{I}(\alpha-\beta))\ \delta(\mathcal{R}(\alpha-\beta)) = \delta^2(\alpha-\beta), \end{aligned} \tag{258}$$

Note that $P(\alpha)$ of a pure coherent state coincides with the classical density of a pure state. Therefore, coherent states are classical-like quantum states.

18.4.2.2 Pure number state The P-representation of a pure number state, $\hat{\rho} = |n\rangle\langle n|$, is obtained as follows:

$$\begin{aligned} \langle -u|\hat{\rho}|u\rangle &= \langle -u|n\rangle\langle n|u\rangle, \\ &= e^{-|u|^2} \frac{u^n}{n!} (-u^*)^n, \end{aligned} \tag{259}$$

where in the second row we substituted $\langle n|u\rangle$ according to Eq. (234), as follows:

$$\langle n|u\rangle = e^{-\frac{1}{2}|u|^2} \frac{u^n}{\sqrt{n!}}, \tag{260}$$

Therefore, substituting Eq. (259) into Eq. (255), we obtain:

$$\begin{aligned} P(\alpha) &= \frac{e^{|\alpha|^2}}{\pi^2} \int e^{-\alpha^* u + u^* \alpha} \langle -u|\hat{\rho}|u\rangle e^{|u|^2}\ d^2u, \\ &= \frac{e^{|\alpha|^2}}{\pi^2} \int e^{-\alpha^* u + u^* \alpha} e^{-|u|^2} \frac{(-uu^*)^n}{n!} e^{|u|^2}\ d^2u, \\ &= \frac{e^{|\alpha|^2}}{n!\ \pi^2} \int e^{-\alpha^* u + u^* \alpha} (-uu^*)^n\ d^2u, \\ &= \frac{e^{|\alpha|^2}}{n!\ \pi^2} \frac{\partial^{2n}}{\partial^n \alpha \partial^n \alpha^*} \int e^{-\alpha^* u + u^* \alpha}\ d^2u, \end{aligned} \tag{261}$$

which according to Eq. (258) gives

$$P(\alpha) = \frac{e^{|\alpha|^2}}{n!} \frac{\partial^{2n}}{\partial^n \alpha \partial^n \alpha^*}\ \delta^2\alpha, \tag{262}$$

that is the so-called *tempered distribution* function which operates only as the argument of an integral, as follows:

$$\int\ d^2\alpha\ F(\alpha^*, \alpha) \frac{\partial^{2n}}{\partial^n \alpha \partial^n \alpha^*}\ \delta^2\alpha = \left. \frac{\partial^{2n} F(\alpha^*, \alpha)}{\partial \alpha^n \partial^n \alpha^*} \right|_{\alpha=0, \alpha^*=0} \tag{263}$$

18.4.3 P-representation of operators

The P-representation of an operator $\hat{O}(\hat{a}^\dagger, \hat{a})$, analogous to the diagonal coherent-state representation of the density operator introduced by Eq. (249), involves the P-function $P_O(\alpha^*, \alpha)$ which is defined, as follows:

$$\hat{O} \equiv \int P_{\hat{O}}(\alpha^*, \alpha)\ |\alpha\rangle\langle\alpha|\ d^2\alpha \tag{264}$$

The expectation value $\langle\hat{O}\rangle$ can then be calculated, as follows:

$$\begin{aligned}
\langle\hat{O}\rangle &= \mathsf{Tr}[\hat{O}\hat{\rho}], \\
&= \sum_n \int P_{\hat{O}}(\alpha^*, \alpha)\ \langle n|\alpha\rangle\langle\alpha|\hat{\rho}|n\rangle\ d^2\alpha, \\
&= \int P_{\hat{O}}(\alpha^*, \alpha)\ \sum_n\ \langle\alpha|\hat{\rho}|n\rangle\langle n|\alpha\rangle\ d^2\alpha, \\
&= \int P_{\hat{O}}(\alpha^*, \alpha)\ \langle\alpha|\hat{\rho}|\alpha\rangle\ d^2\alpha, \\
&= \int Q_{\hat{\rho}}(\alpha)\ P_{\hat{O}}(\alpha^*, \alpha)\ d^2\alpha,
\end{aligned} \tag{265}$$

where $Q_{\hat{\rho}}(\alpha) = \langle\alpha|\hat{\rho}|\alpha\rangle$ is the Q-representation of $\hat{\rho}$, which can be defined in terms of the Husimi function $Q(\alpha) = \pi^{-1}\langle\alpha|\hat{\rho}|\alpha\rangle$, which is $Q(\alpha) = \pi^{-1}|\langle\psi|\alpha\rangle|^2$ for a pure state $\hat{\rho} = |\psi\rangle\langle\psi|$.

In particular, when $\hat{O} = |n\rangle\langle n|$, we obtain:

$$\mathsf{Tr}[\hat{\rho}\ |n\rangle\langle n|] = \int Q_{\hat{\rho}}(\alpha)\ P_{|n\rangle\langle n|}(\alpha^*, \alpha)\ d^2\alpha, \tag{266}$$

where $P_{|n\rangle\langle n|}(\alpha^*, \alpha)$ is defined according to the tempered function introduced by Eq. (262),

$$P_{|n\rangle\langle n|}(\alpha^*, \alpha) = \frac{e^{|\alpha|^2}}{n!}\frac{\partial^{2n}}{\partial^n\alpha\partial^n\alpha^*}\ \delta^2\alpha \tag{267}$$

giving

$$\begin{aligned}
\mathsf{Tr}[|n\rangle\langle n|\hat{\rho}] &= \int\ Q_{\hat{\rho}}(\alpha)\ \frac{e^{|\alpha|^2}}{n!}\frac{\partial^{2n}}{\partial^n\alpha\partial^n\alpha^*}\ \delta^2\alpha\ d^2\alpha, \\
&= \left.\frac{\partial^{2n}F(\alpha^*, \alpha)}{\partial\alpha^n\partial^n\alpha^*}\right|_{\alpha=0,\alpha^*=0}
\end{aligned} \tag{268}$$

with $F(\alpha^*, \alpha)$ defined according to Eqs. (263) and (268), as follows:

$$F(\alpha^*, \alpha) = \ Q_{\hat{\rho}}(\alpha)\ \frac{e^{|\alpha|^2}}{n!} \tag{269}$$

.

18.5 Dynamics

The dynamical properties of a coherent-state in a harmonic well of frequency ω can be described by the time-dependent correlation function,

$$\langle\alpha|\hat{a}(t)^\dagger\hat{a}|\alpha\rangle = |\alpha|^2 e^{i\omega t}, \tag{270}$$

where $\hat{a}(t) = e^{\frac{i}{\hbar}\hat{H}t}\hat{a}e^{-\frac{i}{\hbar}\hat{H}t}$, and $|\alpha\rangle = \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}} \frac{\alpha^n}{\sqrt{n!}}|n\rangle$.

To confirm Eq. (270), we note that $\hat{H}|n\rangle = E_n|n\rangle$, with $E_n = \hbar\omega(n+1/2)$, so $e^{-\frac{i}{\hbar}\hat{H}t}|\alpha\rangle = \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}} \frac{\alpha^n}{\sqrt{n!}} e^{-\frac{i}{\hbar}E_n t}|n\rangle$. Further, $\hat{a}|n\rangle = \sqrt{n}|n-1\rangle$, so we obtain

$$\begin{aligned}\hat{a}(t)|\alpha\rangle = e^{\frac{i}{\hbar}\hat{H}t}\hat{a}e^{-\frac{i}{\hbar}\hat{H}t}|\alpha\rangle &= \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}} \frac{\alpha^n}{\sqrt{n!}} e^{-\frac{i}{\hbar}(E_n - E_{n-1})t}\sqrt{n}|n-1\rangle, \\ &= \alpha \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}} \frac{\alpha^n}{\sqrt{n!}} e^{-i\omega t}|n\rangle, \\ &= \alpha e^{-i\omega t}|\alpha\rangle.\end{aligned} \tag{271}$$

Therefore, $\hat{a}|\alpha\rangle = \alpha|\alpha\rangle$, and computing the adjoint we obtain $\langle\alpha|\hat{a}(t)^\dagger = \alpha^*\langle\alpha|e^{i\omega t}$, giving $\langle\alpha|\hat{a}(t)^\dagger\hat{a}|\alpha\rangle = |\alpha|^2 e^{i\omega t}$, which is Eq. (270).

Numerical Simulation: The dynamics of a harmonic oscillator and its absorption spectrum can be simulated by the following python script. Analogously, the simulation of a 2-dimensional harmonic oscillator in a 2-level system can be simulated with qutip with the following or python script or notebook python script.

Note: Install QuTiP by following the installation instructions:

```
conda create -n qutip-env python=3
conda install numpy scipy cython matplotlib pytest pytest-cov jupyter notebook spyder
conda config --append channels conda-forge
conda install qutip
```

Remember to work with QuTiP in that environment by running

```
conda activate qutip-env
```

18.6 Parity Operator

We define the parity operator, as follows:

$$\begin{aligned}\hat{\Pi} &= \sum_{j=0}^{\infty} |2j\rangle\langle 2j| - \sum_{j=0}^{\infty} |2j+1\rangle\langle 2j+1|, \\ &= e^{i\pi\hat{a}^\dagger\hat{a}},\end{aligned} \tag{272}$$

where states $|2j\rangle$ and $|2j+1\rangle$ in the first row of Eq. (272) are eigenstates of the number operator $\hat{N} = \hat{a}^\dagger\hat{a}$ with eigenvalues $2j$ and $2j+1$, respectively. Note that both expressions of $\hat{\Pi}$, introduced by Eq. (272), change the sign of an eigenstate of the number operator when the occupation number is odd, and leave the eigenstate unchanged if the occupation is even. Consequently,

when applied to a coherent-states,

$$\begin{aligned}\hat{\Pi}|\alpha\rangle &= e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} \hat{\Pi}|n\rangle \\ &= e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} (-1)^n |n\rangle \\ &= e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{(-\alpha)^n}{\sqrt{n!}} |n\rangle \\ &= |-\alpha\rangle.\end{aligned} \tag{273}$$

19 Appendix III: Second Quantization Mapping

The goal of this section is to introduce the single-particle basis $\{\psi_{\nu_1}(\mathbf{r}), \psi_{\nu_2}(\mathbf{r}), \psi_{\nu_3}(\mathbf{r}), \cdots\}$ for representation of the N-particle state $\Psi(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N)$ in terms of symmetrized product states $\hat{S}_{\pm} \prod_{j=1}^{N} \psi_{\nu_j}(\mathbf{r}_j)$, and its correspondence to the occupation number representation $|n_{\nu_1}, n_{\nu_2}, n_{\nu_3}, \cdots\rangle$, where n_{ν_j} is the number of particles in state $\psi_{\nu_j}(\mathbf{r})$ in the product state representation. Furthermore, we introduce the creation $\hat{a}_j^{\dagger}$ and anihilation $\hat{a}_j$ operators (i.e., operators that raise or lower the occupation numbers n_{ν_j} by one unit) and we show that any single particle operator $\hat{A}$ can be expressed in terms of $\hat{a}_j^{\dagger}$ and $\hat{a}_j$, as follows: $\hat{A} = \sum_{\nu_j,\nu_k} A_{\nu_j,\nu_k} \hat{a}_j^{\dagger} \hat{a}_k$, with $A_{\nu_j,\nu_k} = \langle \nu_j | \hat{A} | \nu_k \rangle$.

19.1 Single-Particle Basis

The state of the N-particle system $\Psi(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N)$ can be represented in a complete orthonormal basis composed of single-particle states $\{\psi_{\nu_j}(\mathbf{r})\}$, satisfying that

$$\sum_{\nu_j} \psi_{\nu_j}(\mathbf{r}')^* \psi_{\nu_j}(\mathbf{r}) = \delta(\mathbf{r}' - \mathbf{r}), \tag{274}$$

and

$$\int d\mathbf{r}\, \psi_{\nu_j}(\mathbf{r})^* \psi_{\nu_k}(\mathbf{r}) = \delta_{\nu_j \nu_k}. \tag{275}$$

To represent $\Psi(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N)$, we first project the state along the basis set of $\mathbf{r}_1$, as follows:

$$\Psi(\mathbf{r}_1, \mathbf{r}'_2, \cdots, \mathbf{r}'_N) = \sum_{\nu_1} \psi_{\nu_1}(\mathbf{r}_1) \int d\mathbf{r}'_1 \psi_{\nu_1}(\mathbf{r}'_1)^* \Psi(\mathbf{r}'_1, \mathbf{r}'_2, \cdots, \mathbf{r}'_N), \tag{276}$$

and then we proceed analogously with the other coordinates, so we obtain:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N) = \sum_{\nu_1, \cdots, \nu_N} c_{\nu_1, \cdots, \nu_N} \prod_{j=1}^{N} \psi_{\nu_j}(\mathbf{r}_j), \tag{277}$$

with

$$c_{\nu_1, \cdots, \nu_N} = \int d\mathbf{r}'_1 \psi_{\nu_1}(\mathbf{r}'_1)^* \cdots \int d\mathbf{r}'_N \psi_{\nu_N}(\mathbf{r}'_N)^* \Psi(\mathbf{r}'_1, \mathbf{r}'_2, \cdots, \mathbf{r}'_N). \tag{278}$$

While the product states $\prod_{j=1}^{N} \psi_{\nu_j}(\mathbf{r}_j)$ form a complete basis for the N-particle Hilbert space, they do not necessarily fulfill the indistinguishability requirement of bosons (or fermions) so they need to be symmetrized (or anti-symmetrized). Applying the bosonic symmetrization $\hat{S}_+$ (or the fermionic anti-symmetrization $\hat{S}_-$) operator, we obtain linear combinations of product states with the proper symmetry to describe systems of N-bosons (or fermions), according to the following normalized *permanents* (or *Slater determinants*):

$$\begin{aligned} \hat{S}_{\pm} \prod_{j=1}^{N} \psi_{\nu_j}(\mathbf{r}_j) &= \frac{1}{\prod_{\nu} \sqrt{n_{\nu}!}} \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_{\nu_1}(\mathbf{r}_1) & \psi_{\nu_1}(\mathbf{r}_2) & \cdots & \psi_{\nu_1}(\mathbf{r}_N) \\ \psi_{\nu_2}(\mathbf{r}_1) & \psi_{\nu_2}(\mathbf{r}_2) & \cdots & \psi_{\nu_2}(\mathbf{r}_N) \\ \cdots & \cdots & \cdots & \cdots \\ \psi_{\nu_N}(\mathbf{r}_1) & \psi_{\nu_N}(\mathbf{r}_2) & \cdots & \psi_{\nu_N}(\mathbf{r}_N) \end{vmatrix}_{\pm}, \\ &= \langle \mathbf{r} | \psi_{\nu_1} \psi_{\nu_2} \cdots \psi_{\nu_N} \rangle, \end{aligned} \tag{279}$$

which are linear combinations of product states corresponding to all possible permutation on the set of N coordinates. Each term of the Slater determinant has a sign $(-1)^p$, corresponding to the number of permutations p, while the bosonic permanent terms are all sign-less.

19.2 Occupation Number Basis

The product states, introduced by Eq. (279), are linear combinations of occupied single-particle states. The occupation number representation $|n_{\nu_1}, n_{\nu_2}, n_{\nu_3}, \cdots\rangle$, simply lists the number of particles n_{ν_j} in each occupied state ν_j, with $\sum_j n_{\nu_j} = N$. Such states are eigenstates of the number operators,

$$\hat{n}_{\nu_k}|n_{\nu_1}, n_{\nu_2}, n_{\nu_3}, \cdots\rangle = n_{\nu_k}|n_{\nu_1}, n_{\nu_2}, n_{\nu_3}, \cdots\rangle. \tag{280}$$

For fermions, $n_{\nu_k} = 0, 1$ while for bosons $n_{\nu_k} = 0, 1, 2, \cdots$ is a positive integer.

19.3 Creation and Anihilation Operators

Bosons: The creation and anihilation operators of bosons, $\hat{b}_j^\dagger$ and $\hat{b}_j$, are defined to ensure that the number operator $\hat{n}_{\nu_j} = \hat{b}_j^\dagger \hat{b}_j$ gives the number of bosons in state ν_j as follows:

$$\hat{n}_{\nu_j}|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle = n_{\nu_j}|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle, \tag{281}$$

and raise or lower the occupation of that state, as follows:

$$\begin{aligned}
\hat{b}_j^\dagger|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle &= B_+(n_{\nu_j})|n_{\nu_1}, n_{\nu_2}, \cdots (n_{\nu_j}+1), \cdots\rangle, \\
\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle &= B_-(n_{\nu_j})|n_{\nu_1}, n_{\nu_2}, \cdots (n_{\nu_j}-1), \cdots\rangle,
\end{aligned} \tag{282}$$

where $B_+(n_{\nu_j})$ and $B_-(n_{\nu_j})$ are normalization constants. We further demand that the occupation number of an unoccupied state (e.g., $n_{\nu_j} = 0$) cannot be further reduced, which is equivalent to demand that $\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle = 0$. Furthermore, we define the normalization constants $B_+(0) = 1$ and $B_-(1) = 1$ so that

$$\begin{aligned}
\hat{b}_j^\dagger|n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle &= |n_{\nu_1}, n_{\nu_2}, \cdots 1, \cdots\rangle, \\
\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots 1, \cdots\rangle &= |n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle.
\end{aligned} \tag{283}$$

Therefore,

$$\begin{aligned}
\hat{b}_j\hat{b}_j^\dagger|n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle &= |n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle, \\
\hat{b}_j^\dagger\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle &= 0,
\end{aligned} \tag{284}$$

which can be summarized as $\hat{b}_j\hat{b}_j^\dagger = \hat{n}_{\nu_j} + 1$ and $[\hat{b}_j, \hat{b}_j^\dagger] = 1$. When $j \neq k$, however, $[\hat{b}_j, \hat{b}_k^\dagger] = 0$, or generally $[\hat{b}_j, \hat{b}_k^\dagger] = \delta_{jk}$. Also, $[\hat{b}_j, \hat{b}_k] = [\hat{b}_j^\dagger, \hat{b}_k^\dagger] = 0$. The normalization constants for other states are found from Eq. (281), as follows:

$$\begin{aligned}
\langle n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots|\hat{b}_j^\dagger\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle &= n_{\nu_j}, \\
\langle n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots|\hat{b}_j^\dagger\hat{b}_j|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle &= B_-(n_{\nu_j})^2,
\end{aligned} \tag{285}$$

so $B_-(n_{\nu_j}) = \sqrt{n_{\nu_j}}$. Analogously, we obtain

$$\begin{aligned}
\langle n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots|\hat{b}_j\hat{b}_j^\dagger|n_{\nu_1}, n_{\nu_2}, \cdots n_{\nu_j}, \cdots\rangle &= B_+(n_{\nu_j})^2, \\
(n_{\nu_j}+1) &= B_+(n_{\nu_j})^2,
\end{aligned} \tag{286}$$

$B_+(n_{\nu_j}) = \sqrt{n_{\nu_j}+1}$. Therefore,

$$(\hat{b}_j^\dagger)^{n_\nu}|n_{\nu_1}, n_{\nu_2}, \cdots 0, \cdots\rangle = \sqrt{n_\nu!}|n_{\nu_1}, n_{\nu_2}, \cdots n_\nu, \cdots\rangle. \tag{287}$$

or

$$|n_{\nu_1}, n_{\nu_2}, n_{\nu_3}, \cdots\rangle = \prod_j \frac{(\hat{b}_j^\dagger)^{n_{\nu_j}}}{\sqrt{n_\nu!}}|0,0,0,\cdots\rangle. \tag{288}$$

Fermions: The creation and anihilation operators of fermions, $\hat{c}_j^\dagger$ and $\hat{c}_j$, are defined to ensure that

$$\begin{aligned}
\hat{c}_k|0\rangle &= 0, \\
\hat{c}_k^\dagger|0\rangle &= |k\rangle, \\
\hat{c}_k^\dagger\hat{c}_k|k_1\ldots k_N\rangle &= N|k_1\ldots k_N\rangle, \\
\hat{c}_k^\dagger|k_1\ldots k_N\rangle &= \sqrt{N+1}|kk_1\ldots k_N\rangle, \\
\hat{c}_j|k_1\ldots k_N\rangle &= \frac{1}{\sqrt{N}}\sum_{n=1}^{N}\delta_{jn}(-1)^{n-1}|k_1\ldots k_{n-1}k_{n+1}\ldots k_N\rangle,
\end{aligned} \tag{289}$$

where the factor $(-1)^{n-1}$ results from the $n-1$ permutations that are necessary to bring k_n to the front so the operator $\delta_{jn}\hat{c}_j$ can destroy it. Note that the only possible eigenvalues of the number operator are $n_j = \{0,1\}$ since

$$\begin{aligned}
\hat{N}_j|k_1\ldots k_N\rangle &= \hat{c}_j^\dagger\hat{c}_j|k_1\ldots k_N\rangle, \\
&= \hat{c}_j^\dagger\frac{1}{\sqrt{N}}\sum_{n=1}^{N}\delta_{jn}(-1)^{n-1}|k_1\ldots k_{n-1}k_{n+1}\ldots k_N\rangle, \\
&= \sum_{n=1}^{N}\delta_{jn}(-1)^{n-1}|k_jk_1\ldots k_{n-1}k_{n+1}\ldots k_N\rangle, \\
&= \sum_{n=1}^{N}\delta_{jn}|k_1\ldots k_N\rangle,
\end{aligned} \tag{290}$$

when $j \in \{1, N\}$. Otherwise, $\hat{c}_j^\dagger\hat{c}_j|k_1\ldots k_N\rangle = \hat{c}_j^\dagger 0 = 0$.

Furthermore, the fermionic states must be antisymmetric (*i.e.*, Slater determinants) since they change sign upon permutation of two fermions, as follows: $|k_lk_j\rangle = -|k_jk_l\rangle$. Therefore, the creation operators of fermions anticommute,

$$\begin{aligned}
[\hat{c}_j, \hat{c}_l]_+ &= 0, \qquad &\text{for } j \neq l. \\
[\hat{c}_j^\dagger, \hat{c}_l^\dagger]_+ &= 0, \qquad &\text{for } j \neq l,
\end{aligned} \tag{291}$$

where $[\hat{A}, \hat{B}]_+ = \hat{A}\hat{B} + \hat{B}\hat{A}$, since $|k_jk_l\rangle = \hat{c}_j^\dagger\hat{c}_l^\dagger|0\rangle = -\hat{c}_l^\dagger\hat{c}_j^\dagger|0\rangle = -|k_lk_j\rangle$.

In addition, we show that $[\hat{c}_j, \hat{c}_l^\dagger]_+ = \delta_{jl}$, as follows:

$$\begin{aligned}
\hat{c}_j\hat{c}_l^\dagger|k_1\ldots k_N\rangle &= \hat{c}_j\sqrt{N+1}|lk_1\ldots k_N\rangle, \\
&= \frac{1}{\sqrt{N+1}}\left[\sqrt{N+1}\delta_{jl}|k_1\ldots k_N\rangle + \sqrt{N+1}\sum_{n=1}^{N}\delta_{jn}(-1)^n|lk_1\ldots k_{n-1}k_{n+1}k_N\rangle\right],
\end{aligned} \tag{292}$$

Now, we change the order of the creation and annihilation operators, as follows:

$$\begin{aligned}\hat{c}_l^\dagger \hat{c}_j |k_1 \dots k_N\rangle &= \hat{c}_l^\dagger \frac{1}{\sqrt{N}} \sum_{n=1}^{N} (-1)^{n-1} \delta_{jn} |k_1 \dots k_{n-1} k_{n+1} \dots k_N\rangle, \\ &= -\sum_{n=1}^{N} (-1)^{n} \delta_{jn} |l k_1 \dots k_{n-1} k_{n+1} \dots k_N\rangle,\end{aligned} \tag{293}$$

so adding Eqs. (292) and (293), we obtain:

$$(\hat{c}_l^\dagger \hat{c}_k + \hat{c}_k \hat{c}_l^\dagger)|k_1 \dots k_N\rangle = \delta_{kl} |k_1 \dots k_N\rangle. \tag{294}$$

Therefore,

$$[\hat{c}_l^\dagger, \hat{c}_k]_+ = \delta_{jk}. \tag{295}$$

19.3.0.1 Comparison to spin-1/2 ladder operators: The spin-1/2 operators acting on a site j of a 1-dimensional spin chain are defined, as follows:

$$\begin{aligned}\sigma_j^+ &= \sigma_j^x + i\sigma_j^y, \\ \sigma_j^- &= \sigma_j^x - i\sigma_j^y,\end{aligned} \tag{296}$$

where

$$\sigma_j^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad \sigma_j^y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \tag{297}$$

so

$$\sigma_j^+ = \begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}, \qquad \sigma_j^- = \begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix}, \tag{298}$$

and, therefore, $S_j^\pm = \frac{1}{2}\sigma_j^\pm$ transform the spinors α and β, as follows:

$$S_j^+ \alpha = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \beta, \qquad S_j^+ \beta = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0, \tag{299}$$

and

$$S_j^- \alpha = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0, \qquad S_j^- \beta = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \alpha, \tag{300}$$

So, the commutation relations of the operators S_j^+ and S_j^- of spin j in the 1-dimensional lattice resemble those of fermions $\hat{c}_j^\dagger$ and $\hat{c}_j$ since

$$\begin{aligned}[S_j^-, S_j^+]_+ &= I, \\ [S_j^\pm, S_j^\pm]_+ &= 0,\end{aligned} \tag{301}$$

just like $[\hat{c}_j, \hat{c}_j^\dagger]_+ = I$, $[\hat{c}_j, \hat{c}_j]_+ = 0$, and $[\hat{c}_j^\dagger, \hat{c}_j^\dagger]_+ = 0$. However, in contrast to fermions, spins at different sites of the 1-dimensional lattice are independent of each other, so they *commute* with each other $[S_j^-, S_k^+] = 0$, when $j \neq k$, while fermions *anti-commute*: $[\hat{c}_j, \hat{c}_k^\dagger]_+ = 0$.

19.3.0.2 Jordan-Wigner fermionization: The Jordan-Wigner transformation is essentially a transformation of the operators of 1/2-spins (qubits) in a 1-dimensional lattice into operators that fulfill the commutation relations of fermions by multiplication of the spin-1/2 operators by the so-called *string operator*, $e^{\pm i\hat{\phi}_j}$, with $\hat{\phi}_j = \pi \sum_{k=1}^{j-1} S_k^+ S_k^-$, as follows:

$$\begin{aligned} \tilde{S}_j^\dagger &= e^{i\hat{\phi}_j} S_j^+, \\ \tilde{S}_j &= e^{-i\hat{\phi}_j} S_j^-, \\ \tilde{S}_j^z &= S_j^+ S_j^- - \frac{1}{2}. \end{aligned} \tag{302}$$

Note that

$$e^{\pm i\hat{\phi}_j} = \Pi_{k=1}^{j-1} e^{\pm i\pi S_k^+ S_k^-} = \Pi_{k=1}^{j-1}(1 - 2S_k^+ S_k^-) = \Pi_{k=1}^{j-1}(-\sigma_k^z), \tag{303}$$

since, according to Eq. (298), $S_j^z = \sigma_j^z/2 = S_j^+ S_j^- - 1/2$. We can also invert Eq. (302) to write the spin operators in terms of fermionic operators (considering that $\tilde{S}_j^\dagger \tilde{S}_j = S_j^+ S_j^-$), as follows:

$$\begin{aligned} S_j^+ &= \tilde{S}_j^\dagger e^{-i\pi \sum_{k=1}^{j-1} \tilde{S}_k^\dagger \tilde{S}_k}, \\ S_j^- &= \tilde{S}_j e^{i\pi \sum_{k=1}^{j-1} \tilde{S}_k^\dagger \tilde{S}_k}, \\ S_j^z &= \tilde{S}_j^\dagger \tilde{S}_j - \frac{1}{2}. \end{aligned} \tag{304}$$

Next, we show that the fermionized spin-1/2 operators $\tilde{S}_j^\dagger$ and $\tilde{S}_j$ fulfill the anti-commutation relations of fermions:

$$\begin{aligned} [\tilde{S}_j^\dagger, \tilde{S}_l^\dagger]_+ &= 0, \\ [\tilde{S}_j, \tilde{S}_l]_+ &= 0, \\ [\tilde{S}_j^\dagger, \tilde{S}_l]_+ &= \delta_{jl}. \end{aligned} \tag{305}$$

First, we analyze the commutation properties of the string operator.

Note that $e^{i\pi S_j^+ S_j^-} = 1 - 2S_j^+ S_j^-$ commutes with any $e^{i\pi S_k^+ S_k^-} = 1 - 2S_k^+ S_k^-$. Therefore,

$$[e^{i\hat{\phi}_j}, e^{i\hat{\phi}_k}] = 0, \qquad \text{for all } k, j. \tag{306}$$

Furthermore, $e^{i\pi S_j^+ S_j^-}$ *anti-commutes* with S_j^- and S_j^+,

$$\begin{aligned} [e^{i\pi S_j^+ S_j^-}, S_j^-]_+ &= 0, \\ [e^{i\pi S_j^+ S_j^-}, S_j^+]_+ &= 0, \end{aligned} \tag{307}$$

since $[S_j^+, S_j^-]_+ = 1$, so $S_j^+(1 - 2S_j^+ S_j^-) + (1 - 2S_j^+ S_j^-)S_j^+ = S_j^+((1 - 2S_j^+ S_j^-) - (1 - 2S_j^+ S_j^-)) = 0$, and $S_j^-(1 - 2S_j^+ S_j^-) + (1 - 2S_j^+ S_j^-)S_j^- = (1 - 2(1 - S_j^+ S_j^-))S_j^- + (1 - 2S_j^+ S_j^-)S_j^- = -(1 - 2S_j^+ S_j^-)S_j^- + (1 - 2S_j^+ S_j^-)S_j^- = 0$.

In addition, the phase factor $e^{i\pi S_j^+ S_j^-} = 1 - 2S_j^+ S_j^-$ *commutes* with S_k^+ and S_k^- for $k \neq j$,

$$[e^{i\pi S_j^+ S_j^-}, S_k^\pm] = 0, \qquad \text{for } k \neq j. \tag{308}$$

since $[S_j^{\pm}, S_k^{\pm}] = 0$ when $k \neq j$.

Therefore, combining the commutation relations of Eqs. (307) and (308), we show below that the string operator $e^{i\hat{\phi}_j} = \prod_{l=1}^{j-1}(1 - 2S_l^+ S_l^-)$ *anti-commutes* with any S_k^+ and S_k^- with $k < j$:

$$[e^{i\hat{\phi}_j}, S_k^{\pm}]_+ = 0, \qquad \text{for } k < j. \tag{309}$$

For S_k^+, consider that $e^{i\hat{\phi}_j} = \hat{A}\hat{B}$, with $\hat{A} = \prod_{l\neq k}^{j-1}(1 - 2S_l^+ S_l^-)$ and $\hat{B} = (1 - 2S_k^+ S_k^-)$, with $[\hat{A}, S_k^{\pm}] = 0$ and $[\hat{B}, S_k^{\pm}]_+ = 0$. Therefore, $[\hat{A}\hat{B}, S_k^+]_+ = \hat{A}\hat{B}S_k^+ + S_k^+\hat{A}\hat{B} = \hat{A}\hat{B}S_k^+ + \hat{A}S_k^+\hat{B} = \hat{A}[\hat{B}, S_k^+]_+ = 0$. Analogously, considering that $[\hat{A}, S_k^-] = 0$ and $[\hat{B}, S_k^-]_+ = 0$, we obtain $[\hat{A}\hat{B}, S_k^-]_+ = \hat{A}\hat{B}S_k + S_k^-\hat{A}\hat{B} = \hat{A}\hat{B}S_k^- + \hat{A}S_k^-\hat{B} = \hat{A}[\hat{B}, S_k^-]_+ = 0$.

Also, according to Eq. (308), it is clear that the string operator $e^{i\hat{\phi}_j} = \prod_{l=1}^{j-1}(1 - 2S_l^+ S_l^-)$ *commutes* with any S_k^+ and S_k^- with $k \geq j$, since $l \neq k$ for all $l = (1, j-1)$:

$$[e^{i\hat{\phi}_j}, S_k^{\pm}] = 0, \qquad \text{for } k \geq j. \tag{310}$$

Now, we can show that $[\tilde{S}_j^{\dagger}, \tilde{S}_k^{\dagger}]_+ = 0$, when $j \neq k$. First, we consider the case $k > j$ ($[e^{i\hat{\phi}_j}, S_k^{\pm}] = 0$, and $[e^{i\hat{\phi}_k}, S_j^{\pm}]_+ = 0$),

$$\begin{aligned} [\tilde{S}_j^{\dagger}, \tilde{S}_k^{\dagger}]_+ &= e^{i\hat{\phi}_j} S_j^+ e^{i\hat{\phi}_k} S_k^+ + e^{i\hat{\phi}_k} S_k^+ e^{i\hat{\phi}_j} S_j^+, \\ &= e^{i\hat{\phi}_j} e^{i\hat{\phi}_k} (-S_j^+ S_k^+ + S_k^+ S_j^+) = 0, \end{aligned} \tag{311}$$

since $[S_k^+, S_j^+] = 0$. Next, for $k < j$ ($[e^{i\hat{\phi}_j}, S_k^{\pm}]_+ = 0$, and $[e^{i\hat{\phi}_k}, S_j^{\pm}] = 0$),

$$\begin{aligned} [\tilde{S}_j^{\dagger}, \tilde{S}_k^{\dagger}]_+ &= e^{i\hat{\phi}_j} S_j^+ e^{i\hat{\phi}_k} S_k^+ + e^{i\hat{\phi}_k} S_k^+ e^{i\hat{\phi}_j} S_j^+, \\ &= e^{i\hat{\phi}_j} e^{i\hat{\phi}_k} (S_j^+ S_k^+ - S_k^+ S_j^+) = 0. \end{aligned} \tag{312}$$

Finally, we show that $[\tilde{S}_j, \tilde{S}_k^{\dagger}]_+ = \delta_{jk}$, as follows. First, when $j = k$, the phase factors cancel, so $[\tilde{S}_j, \tilde{S}_j^{\dagger}]_+ = [S_j^-, S_j^+]_+ = I$. Furthermore, for $k < j$ ($[e^{-i\hat{\phi}_j}, S_k^{\pm}]_+ = 0$, and $[e^{i\hat{\phi}_k}, S_j^{\pm}] = 0$),

$$\begin{aligned} [\tilde{S}_j, \tilde{S}_k^{\dagger}]_+ &= e^{-i\hat{\phi}_j} S_j^- e^{i\hat{\phi}_k} S_k^+ + e^{i\hat{\phi}_k} S_k^+ e^{-i\hat{\phi}_j} S_j^-, \\ &= e^{-i\hat{\phi}_j} e^{i\hat{\phi}_k} (S_j^+ S_k^+ - S_k^+ S_j^+) = 0. \end{aligned} \tag{313}$$

Analogously, for $k > j$ ($[e^{-i\hat{\phi}_j}, S_k^{\pm}] = 0$, and $[e^{i\hat{\phi}_k}, S_j^{\pm}]_+ = 0$),

$$\begin{aligned} [\tilde{S}_j, \tilde{S}_k^{\dagger}]_+ &= e^{-i\hat{\phi}_j} S_j^- e^{i\hat{\phi}_k} S_k^+ + e^{i\hat{\phi}_k} S_k^+ e^{-i\hat{\phi}_j} S_j^-, \\ &= e^{-i\hat{\phi}_j} e^{i\hat{\phi}_k} (-S_j^+ S_k^+ + S_k^+ S_j^+) = 0. \end{aligned} \tag{314}$$

19.4 Operators in Second Quantization

In this subsection we show that any single particle operator $\hat{A}$ can be expressed in terms of $\hat{b}_j^{\dagger}$ and $\hat{b}_j$, as follows: $\hat{A} = \sum_{\nu_j,\nu_k} A_{\nu_j,\nu_k} \hat{b}_j^{\dagger}\hat{b}_k$, with $A_{\nu_j,\nu_k} = \langle \nu_j | \hat{A} | \nu_k \rangle$. As an example of a single particle

operator, we consider the kinetic energy $\hat{T} = \sum_{k=1}^{N} \hat{T}_k$, with $\hat{T}_k = \frac{\hat{p}_k^2}{2m_k}$:

$$\begin{aligned}
\langle \mathbf{r}|\hat{T}|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle &= \sum_{\nu_j}\langle \mathbf{r}|\psi_{\nu_j}\rangle\langle\psi_{\nu_j}|\hat{T}|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle \\
&= \sum_{\nu_j}\langle \mathbf{r}|\psi_{\nu_j}\rangle \sum_{k=1}^{N}\langle\psi_{\nu_j}|\hat{T}_k|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle \\
&= \sum_{\nu_j}\langle \mathbf{r}|\psi_{\nu_j}\rangle \sum_{k=1}^{N}\langle\psi_{\nu_j}|\hat{T}_k|\psi_{\nu_k}\rangle\langle\mathbf{r}|\hat{b}_{\nu_k}|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle \\
&= \sum_{k=1}^{N}\sum_{\nu_j,\nu_l}\langle \mathbf{r}|\psi_{\nu_j}\rangle\delta_{\nu_l,\nu_k}T_{\nu_j,\nu_l}\langle\mathbf{r}|\hat{b}_{\nu_k}|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle \\
&= \sum_{k=1}^{N}\sum_{\nu_j,\nu_l}\delta_{\nu_l,\nu_k}T_{\nu_j,\nu_l}\langle\mathbf{r}|\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_k}|\psi_{\nu_1}\psi_{\nu_2}\cdots\psi_{\nu_N}\rangle
\end{aligned} \tag{315}$$

Therefore,

$$\begin{aligned}
\hat{T}\left[\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right] &= \sum_{k=1}^{N}\sum_{\nu_j,\nu_l}\delta_{\nu_l,\nu_k}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_k}\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle \\
&= \sum_{k=1}^{N}\sum_{\nu_j,\nu_l}\delta_{\nu_l,\nu_k}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\frac{\hat{n}_{\nu_k}}{n_{\nu_k}}\hat{b}_{\nu_k}\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle \\
&= \sum_{k=1}^{N}\sum_{\nu_j,\nu_l}\delta_{\nu_l,\nu_k}T_{\nu_j,\nu_l}\frac{\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_k}}{n_{\nu_k}}\left[\hat{b}_{\nu_k}^{\dagger}\hat{b}_{\nu_k}\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right] \\
&= \sum_{\nu_j,\nu_l}T_{\nu_j,\nu_l}\sum_{k=1}^{N}\delta_{\nu_l,\nu_k}\frac{\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_k}}{n_{\nu_k}}\left[\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right] \\
&= \sum_{\nu_j,\nu_l}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_l}\sum_{k=1}^{N}\frac{\delta_{\nu_l,\nu_k}}{n_{\nu_k}}\left[\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right] \\
&= \sum_{\nu_j,\nu_l}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_l}\frac{1}{n_{\nu_l}}\sum_{k=1}^{N}\delta_{\nu_l,\nu_k}\left[\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right] \\
&= \sum_{\nu_j,\nu_l}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_l}\left[\hat{b}_{\nu_1}^{\dagger}\cdots\hat{b}_{\nu_N}^{\dagger}|0\rangle\right]
\end{aligned} \tag{316}$$

Therefore,

$$\hat{T} = \sum_{\nu_j,\nu_l}T_{\nu_j,\nu_l}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_l}. \tag{317}$$

Analogously, any 2-particle operator $\hat{V}$ such as the pair-wise additive potential,

$$\hat{V} = \frac{1}{2}\sum_{j=1}^{N}\sum_{k\neq j}V(x_j, x_k), \tag{318}$$

can be written in second quantization, as follows:

$$\hat{V} = \sum_{\nu_j,\nu_i,\nu_l,\nu_k}V_{\nu_j,\nu_i,\nu_l,\nu_k}\hat{b}_{\nu_j}^{\dagger}\hat{b}_{\nu_i}^{\dagger}\hat{b}_{\nu_l}\hat{b}_{\nu_k} \tag{319}$$

where $V_{\nu_j,\nu_i,\nu_l,\nu_k} = \langle\psi_{\nu_j}\psi_{\nu_i}|V(x_1,x_2)|\psi_{\nu_l}\psi_{\nu_k}\rangle$.

19.5 Change of basis in Second Quantization

We consider two different complete and ordered single-particle basis sets $\{|\psi_{\nu_j}\rangle\}$ and $\{|\psi_{\mu_j}\rangle\}$ with $j = 1$–N. Using the completeness relationship we can write any element of one basis set as a linear combination of elements of the other basis set, as follows:

$$|\psi_{\mu_j}\rangle = \sum_k |\psi_{\nu_k}\rangle\langle\psi_{\nu_k}|\psi_{\mu_j}\rangle, \tag{320}$$

where $|\psi_{\nu_k}\rangle = \hat{a}^\dagger_{\nu_k}|0\rangle$ and $|\psi_{\mu_j}\rangle = \hat{a}^\dagger_{\mu_j}|0\rangle$. Therefore,

$$\hat{a}_{\mu_j}|0\rangle = \sum_k \langle\psi_{\nu_k}|\psi_{\mu_j}\rangle\hat{a}_{\nu_k}|0\rangle, \tag{321}$$

or

$$\hat{a}_{\mu_j} = \sum_k \langle\psi_{\nu_k}|\psi_{\mu_j}\rangle\hat{a}_{\nu_k}, \tag{322}$$

and

$$\hat{a}^\dagger_{\mu_j} = \sum_k \langle\psi_{\nu_k}|\psi_{\mu_j}\rangle^*\hat{a}^\dagger_{\nu_k}, \tag{323}$$

19.6 Mapping into Cartesian Coordinates

Introducing the Cartesian operators $\tilde{x}_{\nu_j} = \frac{1}{\sqrt{2}}[\hat{b}^\dagger_{\nu_j} + \hat{b}_{\nu_j}]$ and $\tilde{p}_{\nu_j} = \frac{i}{\sqrt{2}}[\hat{b}^\dagger_{\nu_j} - \hat{b}_{\nu_j}]$, with $[\tilde{x}_{\nu_j}, \tilde{p}_{\nu_j}] = i$, since $\tilde{x}_{\nu_j} = \hat{x}\sqrt{\frac{m\omega}{\hbar}}$, $\tilde{p}_{\nu_j} = \hat{p}/\sqrt{m\omega\hbar}$ and $[\hat{x}_{\nu_j}, \hat{p}_{\nu_j}] = i\hbar$, for the harmonic oscillator Hamiltonian

$$\begin{aligned} H &= \frac{\hat{p_{\nu_j}}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2, \\ &= \frac{\tilde{p_{\nu_j}}^2}{2m}m\omega\hbar + \frac{1}{2}m\omega^2\frac{\hbar}{m\omega}\tilde{x_{\nu_j}}^2, \\ &= \frac{\hbar\omega}{2}\left[\tilde{p}^2_{\nu_j} + \tilde{x}^2_{\nu_j}\right]. \end{aligned} \tag{324}$$

Considering that

$$\begin{aligned} \hat{n}_{\nu_j} &= \hat{b}^\dagger_{\nu_j}\hat{b}_{\nu_j}, \\ \hat{b}^\dagger_{\nu_j} &= \frac{1}{\sqrt{2}}\left[\tilde{x}_{\nu_j} - i\tilde{p}_{\nu_j}\right], \\ \hat{b}_{\nu_j} &= \frac{1}{\sqrt{2}}\left[\tilde{x}_{\nu_j} + i\tilde{p}_{\nu_j}\right]. \end{aligned} \tag{325}$$

we obtain,

$$\begin{aligned} \hat{n}_{\nu_j} &= \frac{1}{2}(\tilde{x}_{\nu_j} - i\tilde{p}_{\nu_j})(\tilde{x}_{\nu_j} + i\tilde{p}_{\nu_j}) \\ &= \frac{1}{2}(\tilde{x}^2_{\nu_j} + i[\tilde{x}_{\nu_j}, \tilde{p}_{\nu_j}] + \tilde{p}^2_{\nu_j}) \\ &= \frac{1}{2}(\tilde{x}^2_{\nu_j} + \tilde{p}^2_{\nu_j} - 1) \end{aligned} \tag{326}$$

and

$$H = \hbar\omega \left(\hat{n}_{\nu_j} + \frac{1}{2} \right). \tag{327}$$

Substituting the Cartesian expressions of b_{ν_j} and $b^\dagger_{\nu_j}$ into Eq. (317), we obtain:

$$\begin{aligned} \hat{T} &= \frac{1}{2} \sum_{\nu_j,\nu_l} T_{\nu_j,\nu_l} \left[\tilde{x}_{\nu_j} - i\tilde{p}_{\nu_j} \right] \left[\tilde{x}_{\nu_l} + i\tilde{p}_{\nu_l} \right], \\ &= \frac{1}{2} \sum_{\nu_j} T_{\nu_j,\nu_j} (\tilde{x}^2_{\nu_j} + \tilde{p}^2_{\nu_j} - 1) + \frac{1}{2} \sum_{\nu_j} \sum_{\nu_l \neq \nu_j} T_{\nu_j,\nu_l} \left[\tilde{x}_{\nu_j} - i\tilde{p}_{\nu_j} \right] \left[\tilde{x}_{\nu_l} + i\tilde{p}_{\nu_l} \right] \\ &= \frac{1}{2} \sum_{\nu_j} T_{\nu_j,\nu_j} (\tilde{x}^2_{\nu_j} + \tilde{p}^2_{\nu_j} - 1) + \frac{1}{2} \sum_{\nu_j} \sum_{\nu_l \neq \nu_j} T_{\nu_j,\nu_l} \left[\tilde{x}_{\nu_j} \tilde{x}_{\nu_l} + \tilde{p}_{\nu_j} \tilde{p}_{\nu_l} \right] \end{aligned} \tag{328}$$

since $[\tilde{x}_{\nu_j}, \tilde{p}_{\nu_l}] = i\delta_{\nu_l,\nu_j}$ while $[\tilde{x}_{\nu_j}, \tilde{x}_{\nu_l}] = 0$ and $[\tilde{p}_{\nu_j}, \tilde{p}_{\nu_l}] = 0$.

20 Appendix IV: Superconducting Circuits: IBM Quantum Computer

The LC-circuit (or resonant circuit), shown in Fig. 49, is an electrical circuit that consists of an inductor, L, connected in series with a capacitor, C (*i.e.*, and idealized model of the RLC- circuit where the resistance is assumed to be zero, thus no energy dissipation). As the capacitor starts discharging, a current is generated and as it passes through the inductor it generates a magnetic flux, described by the Biot-Savart law. As the current increases, the magnetic flux increases, and that change in the magnetic flux induces a voltage across the inductor, as described by Faraday's law: $V_{ind} = -d\Phi/dt$, that reverts the current and recharges the capacitor. So, the charge sloshes back and forth like a harmonic oscillator with all of the electrons behaving as a single coordinate charge carrier.

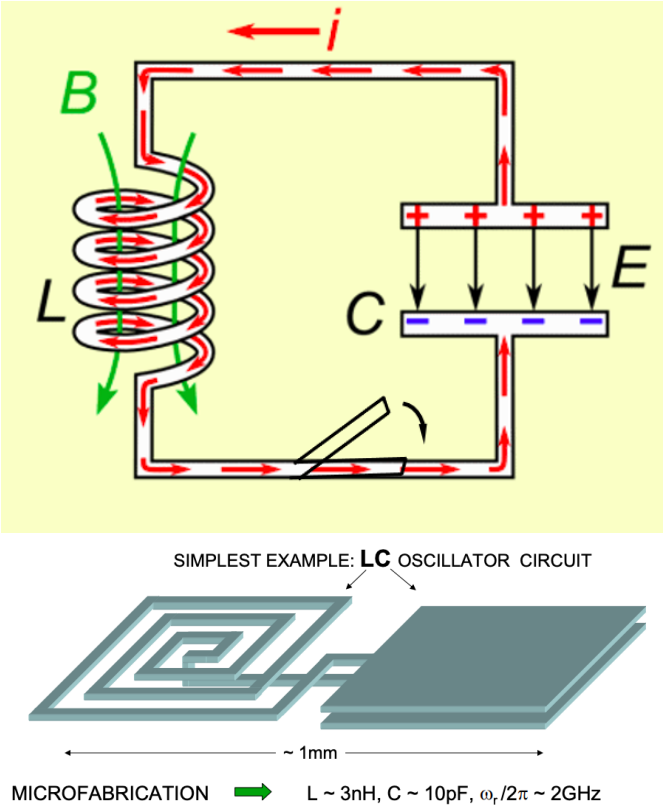


Figure 49: Top: Schematic diagram of an LC-circuit. Bottom: Simplest LC-circuit.

Starting with the open circuit, the capacitor can be charged with an applied voltage V. The initial charge of the capacitor is $Q = CV$, where $C = \epsilon a/d$ is its capacitance defined by the dielectric constant ϵ of the material in between the plates, the area of the plates a and the distance d between the plates.

Upon closing the circuit, the voltage of the capacitor $V = Q/C$ equals the voltage across the inductor determined by $V_{ind} = -d\Phi/dt$, where the time-dependent flux

$$\begin{aligned} \Phi(t) &= \int_{-\infty}^{t} dt' V(t') \\ &= L\, i(t) \end{aligned} \tag{329}$$

is defined by the time-dependent current $i(t)$, and the inductance $L = \mu\ N^2\ A\ /l$ of the solenoid with core magnetic permeability μ, length l, number of turns N, and area A.

We obtain the equation of motion of the time-dependent charge $Q(t) = \int_{-\infty}^{t} dt' i(t')$ by using Kirchhoff's law of voltages (*i.e.*, the sum of voltages around a closed loop is zero, $\sum_i V_i = 0$):

$$\begin{aligned} V &= V_{ind} \\ \frac{Q}{C} + L\frac{d}{dt} i(t) &= 0 \\ \frac{Q}{C} + L\frac{d^2}{dt^2} Q(t) &= 0 \end{aligned} \tag{330}$$

and solving for $Q(t)$, we obtain:

$$Q(t) = Q(0)\,\cos(\omega\, t). \tag{331}$$

Equation (331) shows that the $Q(t)$ behaves like a harmonic oscillator with resonant frequency $\omega = 1/\sqrt{L\,C}$. Equation (330) is essentially Newton's second law: $F = m\,a$, with $F = -Q/C$, $m = L$ and $a = \ddot{Q}$. Therefore, the effective momentum is $p = m\,v = L\,\dot{Q}$. So, the kinetic energy is $T = \frac{p^2}{2\,m} = \frac{L}{2}\dot{Q}^2$ and since $F = -dU/dQ$, the potential energy is $U = Q^2/(2\,C)$, giving the Hamiltonian,

$$H = T + V = \frac{L}{2}\dot{Q}^2 + \frac{Q^2}{2\,C}, \tag{332}$$

which is essentially the Hamiltonian of a harmonic oscillator with coordinate Q and mass C. Therefore, the energy levels of the circuit are quantized (Fig. 50, top panel), just like the energy levels of a harmonic oscillator (or energy levels of an atom, or molecule), so these superconducting circuits are often called *artificial atoms*. The resonance frequency $\omega_0 = 1/\sqrt{L\,C}$ defines the energy level spacing $\Delta E = \hbar\omega$ and can be engineered during fabrication of the device by the macroscopic dimension of the device since, as mentioned above, $L = \mu\,N^2\,A\,/l$ and $C = \epsilon a/d$. Note that at low temperatures (10-15 mK), achievable with a dilution refrigerator, the population of the first excited state is negligible. For example, when $\omega = 2GHz$, the Boltzmann population of the first excited state at 10 mK is $e^{-\hbar\omega/(k_B T)} = e^{-10}$. Furthermore, the quantum state of the circuit can undergo transitions upon interaction with an external field (usually in the microwave frequency range).

Another representation of the circuit Hamiltonian is in terms of the flux $\Phi(t) = L\,i(t)$, obtained by taking the time derivative of both sides of Eq. (330), as follows:

$$\begin{aligned} \frac{\dot{Q}}{C} + L\frac{d^2}{dt^2}\,i(t) &= 0, \\ \dot{Q} + C\frac{d^2}{dt^2}\Phi(t) &= 0, \\ \frac{\Phi}{L} + C\frac{d^2}{dt^2}\Phi(t) &= 0, \end{aligned} \tag{333}$$

which is essentially Kirchhoff's law of currents (*i.e.*, the sum of input and output currents at any node of the circuit is zero, $\sum_k i_k = 0$), with the first term giving the current from the inductor $i_L = \Phi/L$ and the second term the current to the capacitor $i_C = \dot{Q} = C\dot{V} = C\ddot{V}$.

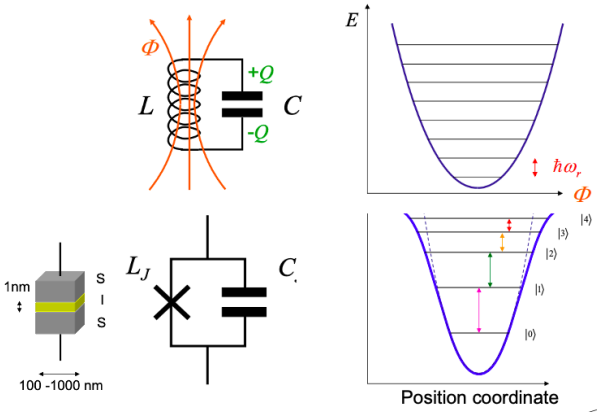


Figure 50: Schematic energy diagram of the harmonic LC-circuit (top), compared to the anharmonic version where the solenoid is replaced by a Josephson junction (bottom).

Therefore, the Hamiltonian can be written in terms of the flux with kinetic energy $T = \frac{C}{2}\dot{\Phi}^2$ and potential energy $U = \frac{\Phi^2}{2L}$, since Eq. (333) is essentially $F = m\,a$, with $m = C$, $a = \frac{d^2}{dt^2}\Phi(t)$, $F = -dU/d\Phi = -\Phi/L$, and momentum $p = m\,v = C\,\dot{\Phi} = C\,V = Q$:

$$\begin{aligned} H &= \frac{1}{2C}(C\dot{\Phi})^2 + \frac{1}{2L}\Phi^2, \\ &= \frac{1}{2C}Q^2 + \frac{1}{2L}\Phi^2. \end{aligned} \tag{334}$$

Therefore, the coordinate Φ and the momentum Q are conjugate variables with the usual commutator of coordinates and momenta ($[\hat{\phi}, \hat{Q}] = i\hbar$).

The Hamiltonian introduced by Eq. (334) can also be written, as follows:

$$H = 4E_c\hat{n}^2 + \frac{1}{2}E_L\hat{\phi}^2, \tag{335}$$

where we have introduced the reduced charge $\hat{n} = Q/(2e)$, and the phase $\hat{\phi} = 2\pi\Phi/\Phi_0$, with $\Phi_0 = h/(2e)$ the flux quanta, corresponding to the operators for the Josephson junction introduced below, including the number of Cooper pairs and phase across, as well as the charging energy $E_c = \frac{e^2}{2C}$ and the inductive energy $E_L = \frac{\Phi_0^2}{4\pi^2 L}$. In fact, replacing the solenoid by a Josephson junction (a sandwich of superconducting layers with an insulating layer in between, such as two pieces of aluminum with a thin layer of aluminum oxide in between) makes the so-called transmon (*i.e.*, a single Josephson junction shunted by a **large** capacitance) which as we show below is essentially an anharmonic version of the LC-circuit (Figure 50, lower panel). An advantage of the anharmonic circuit is that it allows for selective transitions between energy levels (*e.g.*, transition between the ground state and the first excited state) by interaction with an external field with the characteristic frequency of the ground to the first excited state transition, while the harmonic-LC circuit is less controllable since the same frequency induces $\Delta\nu = \pm 1$ transitions for all energy levels.

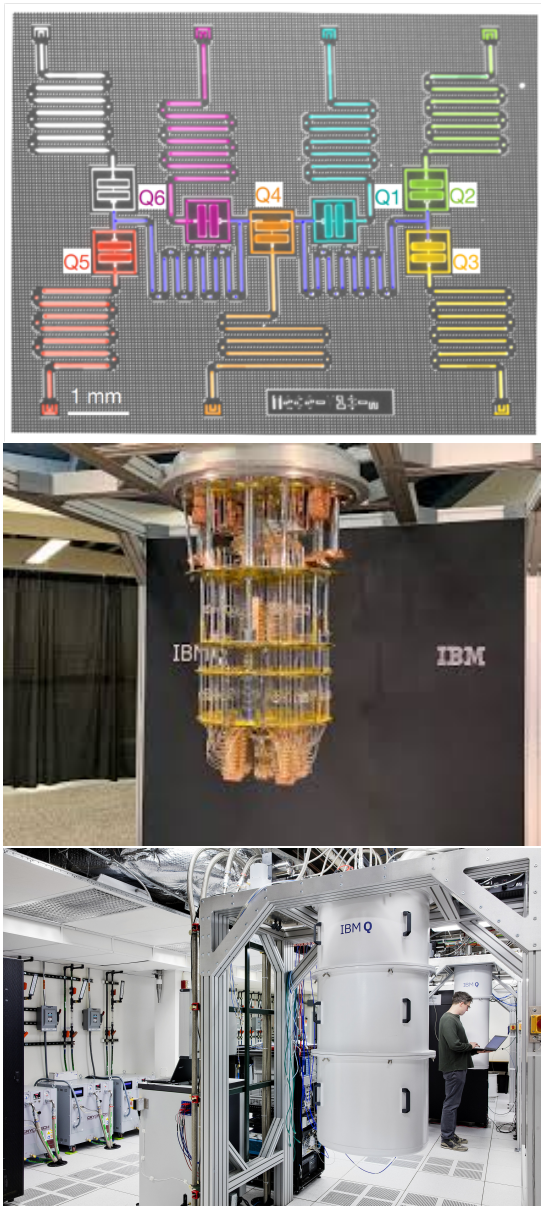


Figure 51: Top: IBM 7-transmon processor. Middle: pulsing system. Bottom: Refrigerator.

20.1 Transmon: Capacitively Shunted Junction

The transmon qubit introduced by Schoelkopf, Girvin and Devoret has been adopted by companies such as IBM (Fig. 51), Google and Rigetti as the so-called *circuit quantum electrodynamics* (cQED) technology by analogy to the QED experiments that manipulate transitions in real atoms by interaction with photons in optical cavities. In fact, the transmon is a modified version of the conceptually simpler Cooper pair box circuit, introduced by Devoret (*i.e.*, a superconducting box ('island') connected to a superconductor reservoir by Josephson junctions) as discussed in Sec. 20.4, where the Cooper pair charge states in the box are analogous to the states of photons in an optical cavity of QED experiments. The phase conjugate to the number of electrons is analogous to the phase of the field in the cavity.

Some advantages of cQED when compared to QED are:

1. the parameters of the artificial atoms can be engineered with great flexibility

2. many qubits can be placed in a transmission line resonator to make them all interact with each other (Fig. 51, top panel)

3. qubits and coupling parameters are highly tunable and allow for strong coupling

We note that although the Josephson junction is made up of millions of atoms and electrons, the electrons form Cooper pairs that condensed into a single collective ground state, below the critical temperature of the superconductor. The Cooper pairs can also coherently tunnel through the thin insulating barrier, leading to a phase difference, ϕ, between the macroscopic superconducting condensate wavefunctions on each side of the barrier.

The tunneling current yields a new current-flux relationship, described by the Josephson relation,

$$i(t) = I_0 \sin \phi(t), \tag{336}$$

where $\phi(t) = 2\pi\Phi(t)/\Phi_0$, with $\Phi_0 = h/2e$ the flux quanta and I_0 the critical current of the superconductor set by the fabrication parameters of the junction. Note that the relationship between $i(t)$ and Φ is non-linear, so the transmon is simply a non-linear inductor. That non-linearity leads to a Hamiltonian for the transmon that is the Hamiltonian of an anharmonic oscillator (i.e., a Harmonic oscillator with an additional Kerr-type nonlinear term introduced by Eq. (339)). So, the transmon is essentially a Kerr Non-linear Resonator (KNR), as discussed in Sec. 20.2.

Transmon Hamiltonian: The equation of motion $F = m\ a$ for the transmon flux, analogous to Eq. (333), can be obtained by using Kirchhoff's law of currents (*i.e.*, the sum of input and output currents at any node of a circuit is zero, $\sum_k i_k = 0$):

$$I_0 \sin(2\pi\ \Phi/\Phi_0) + C\ \ddot{\Phi} = 0, \tag{337}$$

where the first term is the current from the transmon and the second term is the current to the capacitor. Therefore, the potential energy $U = -I_0\Phi_0\cos(2\pi\ \Phi/\Phi_0)$ is anharmonic (Fig. 50, lower panel), and the kinetic energy is $T = C\dot{\Phi}^2/2 = Q^2/(2C)$, giving the transmon Hamiltonian:

$$\begin{aligned} H &= \frac{Q^2}{2C} - I_0\Phi_0 \cos(2\pi\ \Phi/\Phi_0), \\ &= 4\ E_c\ \hat{n}^2 - E_J \cos\hat{\phi}, \end{aligned} \tag{338}$$

with $E_c = e^2/(2C)$, $\hat{n} = Q/(2e)$, and $E_J = I_0\Phi_0$ the Josephson energy replacing the inductive energy of the LC-circuit. Equation 338 shows that a small ratio E_J/E_c provides high anharmonicity, while increasing E_J/E_c makes the cos $\hat{\phi}$ term dominate.

The comparison of eigenstates of the transmon to those of the LC circuit is provided in the notebook vic_transmon.ipynb and vic_transmon.pdf, where the Hamiltonian is written in terms of the annihilation operator $\hat{c}$. To obtain the expression of the Hamiltonian, we note that according to Eq. (338), which is analogous to Eq. 334, the coordinate and momentum of the transmon correspond to the phase $\hat{\phi}$ and reduced charge $\hat{n}$ operators, respectively. So, we can rewrite them in terms of the annihilation $\hat{c} = \sum_j \sqrt{j+1}|j\rangle\langle j+1|$ and creation $\hat{c}^\dagger$ operators, as follows: $\hat{\phi} = \phi_{zpf}(\hat{c} + \hat{c}^\dagger)$ and $\hat{n} = i\ n_{zpf}(\hat{c}^\dagger - \hat{c})$ where $\phi_{zpf} = \left(\frac{2E_c}{E_J}\right)^{1/4}$ and $n_{zpf} = \left(\frac{E_J}{32E_c}\right)^{1/4}$.

Considering that $E_J/E_c \gg 1$, we have $\phi \ll 1$, so we can expand the potential energy in Taylor series, as follows: $\cos\hat{\phi} = 1 - \frac{1}{2}\hat{\phi}^2 + \frac{1}{4}\hat{\phi}^4 + \cdots$ to obtain:

$$\begin{aligned} H &= -4\, E_c\, n_{zpf}^2(\hat{c}^\dagger - \hat{c})^2 - E_J\, \left(1 - \frac{1}{2!}\phi_{zpf}^2(\hat{c} + \hat{c}^\dagger)^2 + \frac{1}{4!}\phi_{zpf}^4(\hat{c} + \hat{c}^\dagger)^4 + \cdots\right), \\ &\approx \sqrt{8E_cE_J}\left(\hat{c}^\dagger\hat{c} + \frac{1}{2}\right) - E_J - \frac{E_c}{12}(\hat{c}^\dagger - \hat{c})^4. \end{aligned} \tag{339}$$

The Hamiltonian, introduced by Eq. (339) can be further simplified, as follows: $(\hat{c}^\dagger - \hat{c})^2 = (\hat{c}^\dagger)^2 + \hat{c}^2 - \hat{c}^\dagger\hat{c} - \hat{c}\hat{c}^\dagger$, with $[\hat{c}, c^\dagger] = 1$. Therefore, in normal order, $(\hat{c}^\dagger - \hat{c})^2 = (\hat{c}^\dagger)^2 + \hat{c}^2 - 2\hat{c}^\dagger\hat{c} - 1$ and $(\hat{c}^\dagger + \hat{c})^4 = ((\hat{c}^\dagger)^2 + \hat{c}^2 - 2\hat{c}^\dagger\hat{c} - 1)((\hat{c}^\dagger)^2 + \hat{c}^2 - 2\hat{c}^\dagger\hat{c} - 1)$. Keeping terms with the same power of $\hat{c}$ and $\hat{c}^\dagger$ (*i.e.*, rotating wave approximation (RWA)), we obtain $(\hat{c}^\dagger - \hat{c})^4 \approx (\hat{c}^\dagger)^2\hat{c}^2 + \hat{c}^2(\hat{c}^\dagger)^2 + 4\hat{c}^\dagger\hat{c}\hat{c}^\dagger\hat{c} + 4\hat{c}^\dagger\hat{c} + 1 = (\hat{c}^\dagger)^2\hat{c}^2 + (1 + \hat{c}^\dagger\hat{c})(1 + (1 + \hat{c}^\dagger\hat{c})) + 4\hat{c}^\dagger(1 + \hat{c}^\dagger\hat{c})\hat{c} + 4\hat{c}^\dagger\hat{c} + 1 = (\hat{c}^\dagger)^2\hat{c}^2 + 2 + \hat{c}^\dagger\hat{c} + \hat{c}^\dagger\hat{c}(2 + \hat{c}^\dagger\hat{c}) + 4\hat{c}^\dagger\hat{c} + 4(\hat{c}^\dagger)^2(\hat{c})^2 + 4\hat{c}^\dagger\hat{c} + 1 = (\hat{c}^\dagger)^2\hat{c}^2 + 2 + \hat{c}^\dagger\hat{c} + 2\hat{c}^\dagger\hat{c} + \hat{c}^\dagger\hat{c} + (\hat{c}^\dagger)^2\hat{c}^2 + 4\hat{c}^\dagger\hat{c} + 4(\hat{c}^\dagger)^2(\hat{c})^2 + 4\hat{c}^\dagger\hat{c} + 1 = 6(\hat{c}^\dagger)^2\hat{c}^2 + 12\hat{c}^\dagger\hat{c} + 3$:

$$H \approx \sqrt{8E_cE_J}\left(\hat{c}^\dagger\hat{c} + \frac{1}{2}\right) - E_J - \frac{E_c}{2}(\hat{c}^\dagger)^2\hat{c}^2 - E_c\hat{c}^\dagger\hat{c} - \frac{E_c}{4}. \tag{340}$$

Defining $\omega_0 = \sqrt{8E_cE_J}$, $\delta = -E_c$ and neglecting the constants that have no influence on the transmon dynamics, we obtain:

$$\begin{aligned} H &\approx (\omega_0 + \delta)\hat{c}^\dagger\hat{c} + \frac{\delta}{2}(\hat{c}^\dagger)^2\hat{c}^2, \\ &\approx (\omega_0 + \delta)\hat{c}^\dagger\hat{c} + \frac{\delta}{2}\hat{c}^\dagger(-1 + \hat{c}\hat{c}^\dagger)\hat{c} \\ &= (\omega_0 + \frac{\delta}{2})\hat{c}^\dagger\hat{c} + \frac{\delta}{2}(\hat{c}^\dagger\hat{c})^2 \end{aligned} \tag{341}$$

which is the Hamiltonian of a Duffing oscillator (Korsch, chapter 8).

Considering that the number operator is

$$\hat{N} = \hat{c}^\dagger\hat{c} = \sum_{n=0}^{\infty} n|n\rangle\langle n|, \tag{342}$$

we obtain

$$\begin{aligned} H &= (\omega - \frac{\delta}{2})\sum_{n=0}^{\infty} n|n\rangle\langle n| + \frac{\delta}{2}\sum_{n=0}^{\infty} n^2|n\rangle\langle n|, \\ &= \sum_{n=0}^{\infty}((\omega - \frac{\delta}{2})n + \frac{\delta}{2}n^2)|n\rangle\langle n|, \end{aligned} \tag{343}$$

where $\omega = \omega_0 + \delta$. So, the eigenvalues of the transmon are spaced, as follows:

$$E_n = (\omega - \frac{\delta}{2})n + \frac{\delta}{2}n^2. \tag{344}$$

20.2 Kerr Hamiltonian

The goal of this section is to show that the transmon Hamiltonian, introduced by Eq. (341) under the RWA, is essentially the Hamiltonian of the anharmonic oscillator,

$$\hat{H}_s = \hbar\omega\left(\hat{N} + \frac{1}{2}\right) + K\hat{N}^2, \tag{345}$$

with the number operator $\hat{N} = \hat{c}^\dagger\hat{c}$. Here, we show it can be written simply as the Kerr Hamiltonian,

$$\hat{H}_R = K(\hat{c}^\dagger)^2\hat{c}^2, \tag{346}$$

with $K = \delta/2$ –*i.e.* when represented in the reference frame rotating with frequency ω.

The second term in Eq. (345) introduces the anharmonicity, while the first term corresponds to the Hamiltonian of the harmonic oscillator,

$$\begin{aligned}\hat{H}_h &= \hbar\omega\left(\hat{N} + \frac{1}{2}\right), \\ &= \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2,\end{aligned} \tag{347}$$

where the annihilation $\hat{c} = \frac{1}{\sqrt{2}}(\tilde{x} + i\tilde{p})$ and creation $\hat{c}^\dagger = \frac{1}{\sqrt{2}}(\tilde{x} - i\tilde{p})$ operators are defined in terms of the reduced coordinate and momentum operators, $\tilde{x} = \hat{x}\sqrt{\frac{m\omega}{\hbar}}$, and $\tilde{p} = \hat{p}\sqrt{\frac{1}{m\hbar\omega}}$, of a Harmonic oscillator of mass m and frequency ω.

We rewrite $\hat{H}_s$ with the so-called *normal order* of creation and annihilation operators (i.e., where each term has powers of $\hat{c}^\dagger$ followed by powers of $\hat{c}$) by using the commutator $[\hat{c}, \hat{c}^\dagger] = 1$, as follows:

$$\begin{aligned}\hat{H}_s &= \hbar\omega\left(\hat{c}^\dagger\hat{c} + \frac{1}{2}\right) + K\hat{c}^\dagger\hat{c}\hat{c}^\dagger\hat{c}, \\ &= \hbar\omega\left(\hat{c}^\dagger\hat{c} + \frac{1}{2}\right) + K\hat{c}^\dagger(1 + \hat{c}^\dagger\hat{c})\hat{c}, \\ &= \hbar\omega\left(\hat{c}^\dagger\hat{c} + \frac{1}{2}\right) + K(\hat{c}^\dagger\hat{c} + \hat{c}^\dagger\hat{c}^\dagger\hat{c}\hat{c}), \\ &= (\hbar\omega + K)\hat{c}^\dagger\hat{c} + \frac{\hbar\omega}{2} + K(\hat{c}^\dagger)^2\hat{c}^2, \\ &= \hat{H}_v + K(\hat{c}^\dagger)^2\hat{c}^2, \\ &= \hat{H}_v + K(\hat{N}^2 - \hat{N}).\end{aligned} \tag{348}$$

To obtain the anharmonic Hamiltonian $\hat{H}_R$ in the rotating frame, we consider an initial state $|\Psi(0)\rangle$ evolving according to the time-evolution operator $\hat{U}_R = e^{-\frac{i}{\hbar}\hat{H}_R t}$, as follows: $|\Psi_R(t)\rangle = \hat{U}_R|\Psi(0)\rangle$. The resulting state $|\Psi_R(t)\rangle$ can also be generated by first propagating the system in the static frame $|\Psi_s(t)\rangle = \hat{U}_s|\Psi(0)\rangle$ using the time-evolution operator $\hat{U}_s = e^{-\frac{i}{\hbar}\hat{H}_s t}$ defined by the Hamiltonian in the static frame $\hat{H}_s$, and then transform the propagated state to the rotated frame by applying the transformation $\hat{U} = e^{\frac{i}{\hbar}\hat{H}_v t}$, where $\hat{H}_v = (\hbar\omega + K)\hat{N} + \frac{\hbar\omega}{2}$ is the Hamiltonian that defines the unitary 'rotation' $\hat{U}$ with frequency ω, from the static to the rotated frame. Therefore,

$$\begin{aligned}|\Psi_R(t)\rangle &= \hat{U}_R|\Psi(0)\rangle, \\ &= \hat{U}\hat{U}_s|\Psi(0)\rangle.\end{aligned} \tag{349}$$

The Hamiltonian $\hat{H}_R$ can now be obtained by computing the time-derivative of $\frac{\partial \hat{U}_R}{\partial t} = -\frac{i}{\hbar}\hat{H}_R\hat{U}_R$, and solving for $\hat{H}_R$, as follows:

$$\begin{aligned}
\hat{H}_R &= i\hbar\frac{\partial \hat{U}_R}{\partial t}\hat{U}_R^\dagger, \\
&= i\hbar\frac{\partial \hat{U}\hat{U}_s}{\partial t}\hat{U}_s^\dagger\hat{U}^\dagger, \\
&= i\hbar\frac{\partial \hat{U}}{\partial t}\hat{U}_s\hat{U}_s^\dagger\hat{U}^\dagger + i\hbar\hat{U}\frac{\partial \hat{U}_s}{\partial t}\hat{U}_s^\dagger\hat{U}^\dagger, \\
&= i\hbar\frac{\partial \hat{U}}{\partial t}\hat{U}^\dagger + i\hbar\hat{U}\frac{\partial \hat{U}_s}{\partial t}\hat{U}_s^\dagger\hat{U}\dagger, \\
&= i\hbar\frac{\partial \hat{U}}{\partial t}\hat{U}^\dagger + \hat{U}\hat{H}_s\hat{U}^\dagger, \\
&= -\hat{H}_v + \hat{U}\hat{H}_s\hat{U}^\dagger.
\end{aligned} \tag{350}$$

Therefore, we obtain:

$$\begin{aligned}
\hat{H}_R &= -\hat{H}_v + e^{\frac{i}{\hbar}\hat{H}_v t}\left(\hat{H}_v + K(\hat{N}^2 - \hat{N})\right)e^{-\frac{i}{\hbar}\hat{H}_v t}, \\
&= K(\hat{N}^2 - \hat{N}), \\
&= K(\hat{c}^\dagger)^2\hat{c}^2,
\end{aligned} \tag{351}$$

where the second line was obtained considering that $e^{\frac{i}{\hbar}\hat{H}_v t}$ commutes with both $\hat{H}_v = \frac{1}{2}\hbar\omega + (\hbar\omega + K)\hat{N}$, and $K(\hat{N}^2 - \hat{N})$, and the third line considering that $(\hat{c}^\dagger)^2\hat{c}^2 = (\hat{N}^2 - \hat{N})$.

The evolution of a coherent state, according to the Kerr Hamiltonian $\hat{H}_R$ can be analytically computed, as follows:

$$\begin{aligned}
e^{-\frac{i}{\hbar}\hat{H}_R t}|\alpha\rangle &= \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}}\frac{\alpha^n}{\sqrt{n!}}e^{-\frac{i}{\hbar}K(\hat{N}^2-\hat{N})t}|n\rangle, \\
&= \sum_{n=0}^{\infty} e^{-\frac{|\alpha|^2}{2}}\frac{\alpha^n}{\sqrt{n!}}e^{-\frac{i}{\hbar}K(n^2-n)t}|n\rangle,
\end{aligned} \tag{352}$$

so both the number of photons and the Poisson probability distribution of photons on Fock states $|n\rangle$ remain unchanged.

20.3 SQUID: Tunable Junction

The transmon single Josephson junction is often replaced by a *Superconducting QUantum interference Device* (SQUID) with two Josephson junctions in a superconducting loop, as shown in Fig. 52, with an externally applied magnetic flux Φ_{ext} passing through the loop. As shown below, the Hamiltonian of the SQUID is equivalent to the Hamiltonian of a Josephson junction with $C_{J'} = 2\ C_J$ and energy $E'_J(\Phi_{ext}) = 2\ E_J\ cos(\Phi_{ext}/2)$, tunable *in situ* by simply changing the applied Φ_{ext}.

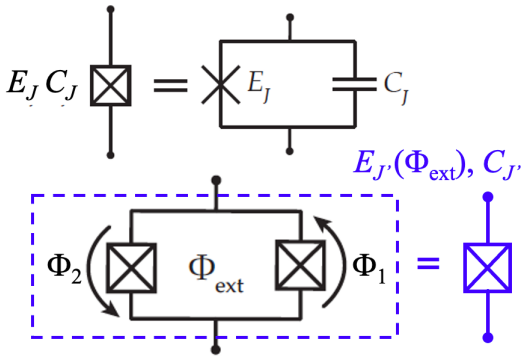


Figure 52: Top: Effective function whose Josephson energy $E_{J''}$ can be tuned by application of an external magnetic flux Φ.

SQUID Hamiltonian: In this subsection we show that the Hamiltonian of the SQUID is the same as the Hamiltonian of a Josephson junction with energy $E_J(\Phi)$ dependent on the externally applied magnetic flux Φ.

The Lagrangian of the SQUID is given by the difference of kinetic and potential energies, as follows:

$$\mathcal{L} = \frac{1}{2}C_J\left(\dot{\Phi}_1^2 + \dot{\Phi}_2^2\right) + E_J\left(\cos 2\pi\frac{\Phi_1}{\Phi_0} + \cos 2\pi\frac{\Phi_2}{\Phi_0}\right), \tag{353}$$

where the magnetic flux quantum $\Phi_0 = h/2e$ (for convenience, we choose units so that $\Phi_0 = 2\pi$). The flux quantization requires that $\Phi_2 - \Phi_1 = \Phi_{ext}$. Therefore, defining $\phi = (\Phi_1 + \Phi_2)/2$, we obtain $\Phi_2 = \phi + \Phi_{ext}/2$ and $\Phi_1 = \phi - \Phi_{ext}/2$. Therefore,

$$\begin{aligned}\mathcal{L} &= 2C_J\dot{\phi}^2 + E_J\left(\cos\Phi_1 + \cos\Phi_2\right), \\ &= 2C_J\dot{\phi}^2 + E_J\left(\cos(\phi - \Phi_{ext}/2) + \cos(\phi + \Phi_{ext}/2)\right), \\ &= C_{J'}\dot{\phi}^2 + E_J 2\cos(\phi)\cos(\Phi_{ext}/2),\end{aligned} \tag{354}$$

with $C_{J'} = 2C_J$. The Hamiltonian is obtained as usual,

$$H = Q\dot{\phi} - \mathcal{L}, \tag{355}$$

with momentum $Q = \frac{\partial\mathcal{L}}{\partial\dot{\phi}} = C_{J'}\dot{\phi}$, giving

$$\begin{aligned}H &= \frac{Q^2}{2C_{J'}} - E_{J'}(\Phi_{ext})\cos(\phi), \\ &= 4\ E_c\hat{n}^2 - E_{J'}(\Phi_{ext})\cos(\hat{\phi}),\end{aligned} \tag{356}$$

where $E_c = e^2/2C_{J'}$, $\hat{n} = Q/2e^2$, and $E_{J'}(\Phi_{ext}) = 2E_J\cos\frac{\Phi_{ext}}{2}$.

20.4 Cooper Pair Box: Charge Qubit

In this subsection we introduce the Hamiltonian of the so-called charge qubit circuit, introduced by Devoret and co-workers (Fig. 53, top panel), also called Cooper pair box, which is a simplified version of the circuit of a junction capacitively coupled to a current source (Fig. 53, middle panel), introduced earlier by M. Buttiker to demonstrate it is possible to externally control the charge on the junction.

The Cooper pair box (CPB) consists of a superconducting 'island' box into which Cooper pairs may tunnel via a Josephson junction from a capacitively coupled biased reservoir. The Cooper pairs in the box are analogous to the states of photons in an optical cavity of QED experiments, where the phase conjugate to the number of electrons is analogous to the phase of the field in the cavity. The resulting circuit exhibits coherent quantum oscillations that can be used as a prototypical charge qubit where the charge degree of freedom is used for coupling and interaction.

Both circuits are described by the Hamiltonian:

$$\begin{aligned}\hat{H} &= 4E_C(\hat{N}-N_g)^2 - E_J\cos(\hat{\phi}) \\ &= 4E_C(\hat{N}-N_g)^2 - \frac{1}{2}E_J(e^{i\hat{\phi}}+e^{-i\hat{\phi}})\end{aligned} \tag{357}$$

where $N_g = C_g V_g/(2e)$ is the effective offset charge due to the applied voltage V_g. As already introduced in Sec. 20, $E_J = I_0\Phi_0$ is the Josephson inductive energy of the tunneling junction, where $\Phi_0 = h/(2e)$ is the flux quanta, and I_0 the critical current of the superconductor set by the fabrication parameters of the junction. These parameters define the nonlinear inductance relationship between current and induced flux $\Phi(t)$, as follows: $i(t) = I_0 \sin\phi(t)$, with $\phi(t) = 2\pi\,\Phi(t)/\Phi_0$. Furthermore, $E_C = e^2/(2C)$ is the capacitor charging energy, with $C = C_J + C_g$ and $\hat{\phi}$ is the phase difference between the macroscopic wavefunctions of Copper pairs on each side of the barrier. $\hat{N} = \frac{\hat{Q}}{2e}$ describes the number of Cooper pairs. Here, $\hat{\phi}$ and $\hat{N}$ are the generalized coordinates and momenta, so that $[\hat{\phi}, \hat{N}] = i\hbar$.

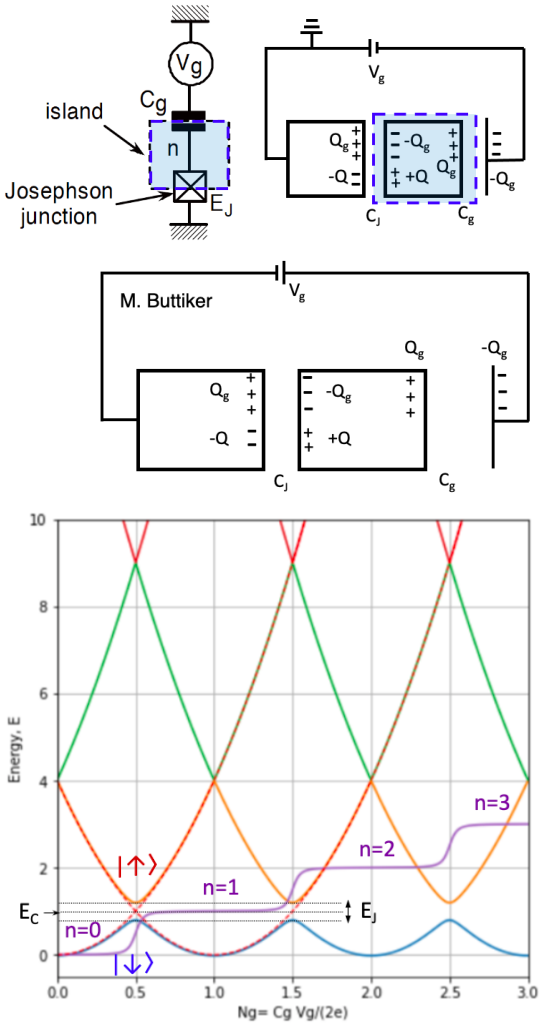


Figure 53: Top panel: Circuit of a Cooper pair box with a superconducting 'island' into which Cooper pairs may tunnel from a superconductor reservoir via a Josephson junction. Middle panel: Josephson junction capacitively coupled to a current source. Bottom panel: Energy levels and expectation value of the number of excess Cooper pairs in the grounds state of the island $\langle\hat{N}\rangle$ versus the effective offset charge N_g defined by the applied voltage V_g.

20.4.1 CPB Eigenstates

We obtain the eigenstates of the CPB as a function of the applied voltage V_g by diagonalization (Fig. 53,

bottom) of the Hamiltonian, introduced by Eq. (357), in the basis of eigenstates of $\hat{N}$ corresponding to the number of excess Cooper pairs in the island, which is the natural choice of basis when $E_C \gg E_J$. We expand the eigenstates of $\hat{\phi}$ in the basis of $\hat{N}$, as follows: $|\phi\rangle = \sum_N |N\rangle\langle N|\phi\rangle$ with $\langle N|\phi\rangle = \frac{1}{\sqrt{2\pi\hbar}} e^{\frac{i}{\hbar}N\phi}$ giving $\langle\phi'|\phi\rangle = \sum_N \langle\phi'|N\rangle\langle N|\phi\rangle = \frac{1}{2\pi\hbar}\sum_N e^{\frac{i}{\hbar}N(\phi-\phi')}$.

Furthermore, considering that $\hat{N}$ is the generalized momentum, the displacement operator gives

$$e^{\frac{i}{\hbar}\alpha\hat{N}}|\phi\rangle = |\phi+\alpha\rangle$$

consistent with $e^{\frac{i}{\hbar}\alpha N'}\langle N'|\phi\rangle = \langle N'|\phi+\alpha\rangle$. Analogously, $e^{-\frac{i}{\hbar}\alpha\hat{\phi}}|N\rangle = |N+\alpha\rangle$, giving $e^{-\frac{i}{\hbar}\alpha\phi'}\langle\phi'|N\rangle = \langle\phi'|N+\alpha\rangle$.

Therefore, we can write the Hamiltonian introduced by Eq. (357) in the basis of $|N\rangle$ by inserting closure, $1 = \sum_N |N\rangle\langle N|$, as follows:

$$\begin{aligned}
\hat{H} &= 4E_C\sum_N (N-N_g)^2|N\rangle\langle N| - \frac{E_J}{2}\sum_N (e^{i\hat{\phi}} + e^{-i\hat{\phi}})|N\rangle\langle N| \\
&= 4E_C\sum_N (N-N_g)^2|N\rangle\langle N| - \frac{E_J}{2}\sum_N (e^{i\hat{\phi}}|N\rangle\langle N| + e^{-i\hat{\phi}}|N\rangle\langle N|) \\
&= 4E_C\sum_N (N-N_g)^2|N\rangle\langle N| - \frac{E_J}{2}\sum_N (|N-1\rangle\langle N| + |N+1\rangle\langle N|)
\end{aligned} \tag{358}$$

Numerical Diagonalization: A turn-key tutorial on how to diagonalize the Hamiltonian introduced by Eq. (358) to obtained the eigenstates shown in Fig. (53, lower panel) on Colab or the IBM Quantum can be downloaded as a notebook:vic_cp.ipynb, and vic_cp.pdf.

20.4.2 NMR Hamiltonian

We can map the CPB to the NMR Hamiltonian of a sping-1/2 by truncating the basis to include only the states $|0\rangle$ and $|1\rangle$, with $0 \leq N_g \leq 1$, a valid approximation when $E_C \gg E_J$ and $E_C \gg k_B T$, since the other energy levels are far away (E_C away). The resulting Hamiltonian is

$$H = \begin{bmatrix} 4E_C N_g^2 & -E_J/2 \\ -E_J/2 & 4E_C(1-N_g)^2 \end{bmatrix}, \tag{359}$$

Factorizing half of the trace $E_0 = 4E_C(N_g^2 + (1-N_g)^2)/2 = 4E_C(N_g^2 + 1/2 - N_g) = 4E_C(1/2 - N_g)^2 + E_C$, we obtain:

$$H = E_0 + \frac{1}{2}\begin{bmatrix} -4E & -E_J \\ -E_J & 4E \end{bmatrix}, \tag{360}$$

with $E = E_C(1-2N_g)$. The 2×2 Hamiltonian introduced by Eq. (360) can be written in terms of the Pauli matrices $\sigma = (\sigma_x, \sigma_y, \sigma_z)$, as follows:

$$\begin{aligned}
H &= E_0 - \frac{1}{2}(4E\sigma_z + E_J\sigma_x) \\
&= E_0 - \mathbf{s}\cdot\mathbf{B}
\end{aligned} \tag{361}$$

where $\mathbf{s} = \sigma/2$, and $\mathbf{B} = (B_x, B_y, B_z)$, with $B_x = E_J$, $B_y = 0$ and $B_z = 4E$.

The NMR mapping Hamiltonian, introduced by Eq. (361), allows us to think about the parameters of the CPB circuit in terms of the parameters of an NMR experiment. For example, the number of Cooper pairs in the island can be obtained from $\langle s_z\rangle$, as follows: $\langle N\rangle = 1/2 + \langle s_z\rangle$.

20.4.3 State Preparation and Control

Just like in an NMR experiment, full control of the quantum dynamics of the 'spin' is possible if the parameters that defined the 'magnetic field' $\mathbf{B}(t)$ can be switched arbitrarily. Clearly, $B_z = 4E_C(1 - 2C_gV_g/(2e))$ can be switched by changing the applied voltage V_g. Furthermore, $B_x = E_J$ can be externally switched by replacing the junction by a squid and applying an external magnetic field, as described in Sec. 20.5.

For example, the initial state can be prepared in the state $|0\rangle$ by turning on a large value of $B_z(t) \gg k_BT$ with $N_g = 1$ and $B_y(t) = B_x(t) = 0$ while keeping the system at low temperature so it relaxes to the ground state. Switching $B_z(t)$ back to zero, with $N_g = 1/2$, leaves the system in the ground state with $H = 0$, so there is no evolution.

A spin-flip (NOT gate) can be applied by switching-on B_x for some time τ, letting the spin evolve according to the time-evolution operator

$$e^{-\frac{i}{\hbar}H\tau} = e^{\frac{i}{\hbar}s_xB_x\tau}, \tag{362}$$

with $s_x = \frac{1}{2}\sigma_x$, where

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{363}$$

to induce a rotation about the x axis, as discussed in Sec. 8.2. Note that

$$\sigma_x = \Gamma \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \Gamma^\dagger = \Gamma\sigma_z\Gamma^\dagger \tag{364}$$

with $\Gamma\Gamma^\dagger = I$ and

$$\Gamma = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \tag{365}$$

so

$$e^{\frac{i}{\hbar}s_xB_x\tau} = e^{\frac{i}{\hbar}\Gamma s_z\Gamma^\dagger B_x\tau} = \Gamma e^{\frac{i}{\hbar}s_zB_x\tau}\Gamma^\dagger, \tag{366}$$

or

$$e^{\frac{i}{\hbar}s_xB_x\tau} = \Gamma^\dagger \begin{pmatrix} e^{\frac{i}{2\hbar}B_x\tau} & 0 \\ 0 & e^{-\frac{i}{2\hbar}B_x\tau} \end{pmatrix} \Gamma = \begin{pmatrix} cos(\phi/2) & i\sin(\phi/2) \\ i\sin(\phi/2) & \cos(\phi/2) \end{pmatrix} = R_X(\phi), \tag{367}$$

with $\phi = B_x\,\tau/\hbar$.

Starting with the initial state $|0\rangle$, the unitary transformation introduced by Eq. (367) with $B_x\tau/\hbar = \pi$ produces a π rotation (a spin flip, NOT gate), $|0\rangle \rightarrow i|1\rangle$. Furthermore, when $B_x\tau/\hbar = \pi/2$, the evolution produces an equal-weight superposition: $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$.

Analogously, the evolution generated by switching-on the field B_z for some time τ produces a phase-shift between the components $|0\rangle$ and $|1\rangle$ in any superposition state $|\psi\rangle = c_0|0\rangle + c_1|1\rangle$, generating the state $|\tilde{\psi}\rangle = c_0e^{i\phi/2}|0\rangle + c_1e^{-i\phi/2}|1\rangle$ according to the rotation around the z axis:

$$e^{\frac{i}{\hbar}B_zs_z\tau} = e^{i\phi\sigma_z/2} = \begin{pmatrix} e^{i\phi/2} & 0 \\ 0 & e^{-i\phi/2} \end{pmatrix} = R_Z(\phi), \tag{368}$$

where, $\phi = B_z\tau/\hbar$.

Importantly, any unitary transformation of a single qubit state (*i.e.*, any single-qubit operation) can be performed with a sequence of these rotations around the x and z axes.

20.5 Split Cooper Pair Box

In this subsection we introduce the so-called split Cooper pair box circuit (Fig. (54)), where Cooper pairs tunnel into the island through a junction that is split into two identical junctions with energy $E_J/2$, forming a squid superconducting loop, as introduced in Sec. 20.3. The resulting circuit forms a flux qubit (also known as persistent current qubits).

The Hamiltonian of the split Cooper pair box is

$$\begin{aligned}\hat{H} &= 4E_C(\hat{N}-N_g)^2 - E_{J'}(\Phi_{ext})\cos(\hat{\phi}) \\ &= 4E_C(\hat{N}-N_g)^2 - \frac{1}{2}E_{J'}(\Phi_{ext})(e^{i\hat{\phi}}+e^{-i\hat{\phi}})\end{aligned} \tag{369}$$

where $\hat{\phi}$ and $\hat{N}=\hat{Q}/(2e)$ are the generalized coordinates and momenta, so that $[\hat{\phi},\hat{N}]=i\hbar$. Here, $\hat{\phi}=(\hat{\Phi}_1+\hat{\Phi}_2)/2$, with $\hat{\Phi}_1$ and $\hat{\Phi}_2$ the phase differences of the two junctions, and $E_{J'}(\Phi_{ext}) = 2E_J\cos\frac{\Phi_{ext}}{2}$. Therefore, we can write the Hamiltonian in the basis of $|N\rangle$, as follows:

$$\begin{aligned}\hat{H} &= 4E_C\sum_N (N-N_g)^2|N\rangle\langle N| \\ &- E_J\cos\frac{\Phi_{ext}}{2}\sum_N(|N-1\rangle\langle N| + |N+1\rangle\langle N|)\end{aligned} \tag{370}$$

The phase difference across the series combination of the two junctions is proportional to the flux through the loop. The persistent loop currents $i_0 = \frac{2e}{\hbar}\frac{\partial E_0}{\partial \Phi_{ext}}$ and $i_1 = \frac{2e}{\hbar}\frac{\partial E_1}{\partial \Phi_{ext}}$ in states $|0\rangle$ and $|1\rangle$, respectively, provide a readout for state discrimination, instead of measuring the charge. Protection from charge and phase noise during state manipulation is achieved by biasing the circuit with $N_g=\frac{1}{2}$ and $\Phi_{ext}=0$, where the transition frequency ν_{01} is stationary with respect to both parameters.

Numerical Diagonalization: A turn-key tutorial on how to diagonalize the Hamiltonian introduced by Eq. (370) to obtained the eigenstates shown in Fig. (54), (lower panel) on Colab or the IBM Quantum can be downloaded as a notebook: vic_scp.ipynb, and vic_scp.pdf.

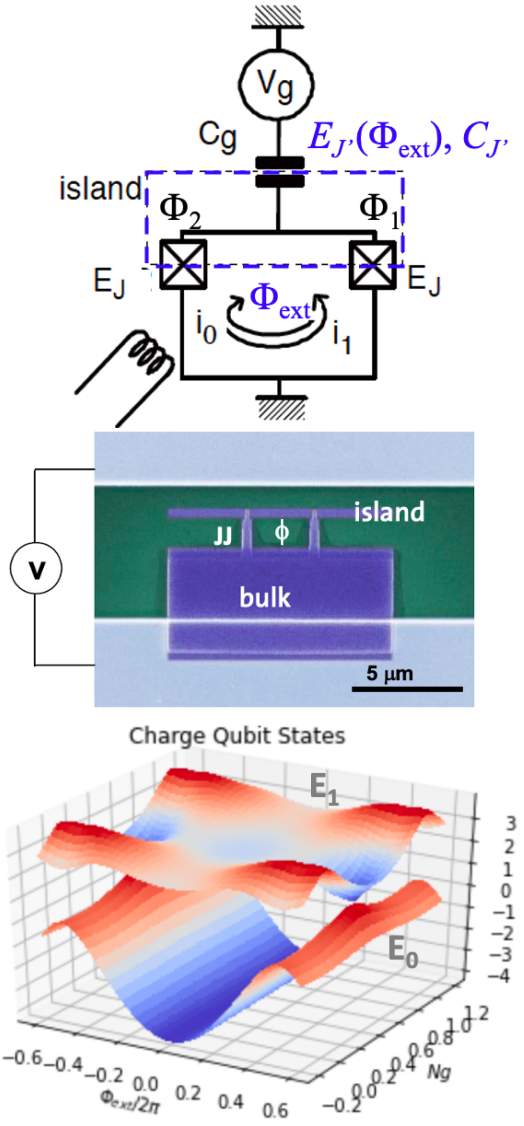


Figure 54: Top panel: Circuit of a split Cooper pair box with a superconducting 'island' into which Cooper pairs may tunnel from a superconductor reservoir via two identical Josephson junctions. Bottom panel: Energy levels versus N_g as defined by the applied voltage V_g.

20.6 Transmon Coupled to a Resonator

In this subsection we introduce the Hamiltonian of the transmon coupled to a harmonic resonator (Fig. 55), as introduced by Schoelkopf, Girvin and Devoret [PRA2004), PRA2007]. The kinetic energy T for charging the capacitors is

$$T = \frac{C_{in}}{2}\dot{\phi}_r^2 + \frac{C_r}{2}\dot{\phi}_r^2 + \frac{C_g}{2}(\dot{\phi}_J - \dot{\phi}_r)^2 + \frac{C_B + C_J}{2}\dot{\phi}_J^2. \tag{371}$$

The potential energy V includes the resonator inductance, the Josephson term introduced in Sec. 20.5, and the potential energy from the external current source, as follows:

$$V = \frac{1}{2L_r}\phi_r^2 - E_J \cos\left(2\pi\,\frac{\phi_J}{\Phi_0}\right) - V_g C_{in}\dot{\phi}_r. \tag{372}$$

The Lagrangian $\mathcal{L} = T - V$ defines the momenta:

$$\begin{aligned} Q_r &= \frac{\partial \mathcal{L}}{\partial \dot{\phi}_r} = (C_{in} + C_r + C_g)\dot{\phi}_r - C_g\dot{\phi}_J + V_g C_{in}, \\ Q_J &= \frac{\partial \mathcal{L}}{\partial \dot{\phi}_J} = (C_g + C_B + C_J)\dot{\phi}_J - C_g\dot{\phi}_r, \end{aligned} \tag{373}$$

The Hamiltonian, $H = Q_r\dot{\phi}_r + Q_J\dot{\phi}_J - \mathcal{L}$, is:

$$\begin{aligned} H = &\frac{\phi_r^2}{2L_r} + \frac{(C_B + C_J + C_g)Q_r^2}{2C^2} \\ &+ \frac{(C_g + C_{in} + C_r)Q_J^2}{2C^2} - E_J \cos\left(\frac{2\pi}{\hbar}\phi_J\right) \\ &+ \frac{C_g Q_r Q_J}{C^2} + \frac{(C_B + C_J + C_g)C_{in}Q_r V_g + C_g C_{in} Q_J V_g}{C^2} \end{aligned} \tag{374}$$

with $C^2 = (C_B + C_J)(C_g + C_{in}) + C_g(C_{in} + C_r) + (C_B + C_J)C_r$. The first line of Eq. (374) is the resonator term, the second line is the transmon, and the third line are the couplings. When C_r is much greater than the other capacitances, the Hamiltonian becomes

$$\begin{aligned} H = &\frac{\phi_r^2}{2L_r} + \frac{Q_r^2}{2C_r} \\ &+ \frac{(Q_J - C_g V_g)^2}{2C_s} - E_J \cos\left(\frac{2\pi}{\hbar}\phi_J\right) \\ &+ \frac{C_g}{C_s C_r} Q_r Q_J + \frac{C_{in} Q_r V_g}{C_r} \end{aligned} \tag{375}$$

with $C_s = C_g + C_B + C_J$, where C_g/C_s is the impedance divider ratio determining how much of the applied voltage is seen by the qubit.

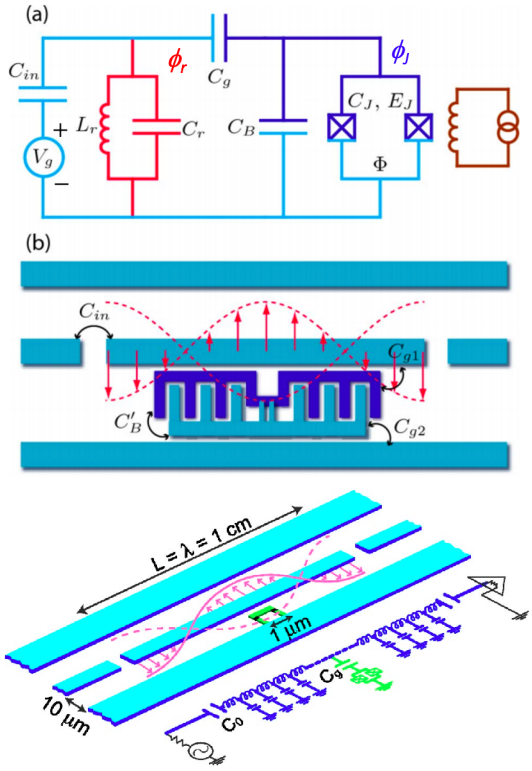


Figure 55: **Top panel:** Circuit of a transmon (capacitively shunted split Cooper pair box) capacitively coupled to an LC resonator from a waveguide and a current source. **Middle panel:** Out of scale diagram of the transmon -*i.e.*, a split Cooper pair box shunted by a short section of a twin-lead transmission line, formed by extending the superconducting islands of the qubit, leading to the increase in the capacitances C_{g1}, C_{g2}, and C_B, and hence in the effective capacitances C_B and C_g in the circuit. **Bottom panel:** Dimensions of the resonator formed by the capacitive gaps in the center trace of the transmission line. The outer two traces are ground. The qubit is placed at the middle of the resonator to couple to the strong electric fields at the antinode of the second mode.

20.6.1 Quantization

The first line of the Hamiltonian, introduced by Eq. (375), corresponds to the harmonic LC-resonator which can be quantized, as follows:

$$\frac{\hat{\phi}_r^2}{2L_r} + \frac{\hat{Q}_r^2}{2C_r} = \hbar\omega_r \frac{1}{2}(\tilde{Q}_r^2 + \tilde{\phi}_r^2) = \hbar\omega_r(\hat{c}^\dagger\hat{c} + \frac{1}{2}), \tag{376}$$

with $\omega_r = \frac{1}{\sqrt{C_r L_r}}$, $\hat{c}^\dagger = \frac{1}{\sqrt{2}}(\tilde{Q}_r - i\tilde{\phi}_r)$ and $\hat{c} = \frac{1}{\sqrt{2}}(\tilde{Q}_r + i\tilde{\phi}_r)$, where $\tilde{Q}_r = \hat{Q}_r \left(\frac{L_r}{\hbar^2 C_r}\right)^{1/4}$ and $\tilde{\phi}_r = \hat{\phi}_r/(\hbar^2 L_r/C_r)^{1/4}$. Therefore, $\hat{Q}_r = \left(\frac{\hbar^2 C_r}{4L_r}\right)^{1/4}(\hat{c} + \hat{c}^\dagger)$. The second line of the Hamiltonian, introduced by Eq. (375), is quantized according to Sec. 20.4.2,

$$\frac{(\hat{Q}_J - C_g V_g)^2}{2C_s} - E_J \cos\left(\frac{2\pi}{\Phi_0}\hat{\phi}_J\right) = E_0 - \frac{1}{2}(4E\sigma_z + E_J\sigma_x), \tag{377}$$

with $\hat{N} = \hat{Q}_J/(2e)$, $E = \frac{e^2}{2C_s}(1 - C_g V_g/e)$ and $E_0 = 2C_s E^2/e^2 + e^2/(2C_s)$. In particular, when the voltage is adjusted to make $N_g = C_g V_g/(2e) = 1/2$, then $E = 0$ and $E_0 = e^2/(2C_s)$, so

$$\frac{(\hat{Q}_J - C_g V_g)^2}{2C_s} - E_J \cos\left(\frac{2\pi}{\Phi_0}\hat{\phi}_J\right) = \frac{e^2}{2C_s} - \frac{E_J}{2}\sigma_x, \tag{378}$$

which can be rewritten in the *reverted* basis of eigenstates of σ_x (*i.e.*, $|\uparrow\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and $|\downarrow\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$), as follows:

$$\frac{(\hat{Q}_J - C_g V_g)^2}{2C_s} - E_J \cos\left(\frac{2\pi}{\Phi_0}\hat{\phi}_J\right) \to \frac{e^2}{2C_s} - \frac{E_J}{4}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix}\sigma_x\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix} = E_{C_s} + \frac{E_J}{2}\sigma_z, \tag{379}$$

where $E_{C_s} = \frac{e^2}{2C_s}$. The third line of the Hamiltonian, introduced by Eq. (375), is quantized using that $\hat{N} = \hat{Q}_J/(2e)$ and $\hat{Q}_r = \left(\frac{\hbar^2 C_r}{4L_r}\right)^{1/4}(\hat{c} + \hat{c}^\dagger)$. So, disregarding the coupling between the resonator and the source of current, we obtain:

$$\frac{C_g}{C_s C_r} Q_r Q_J = \frac{C_g}{C_s C_r} 2e\hat{N}\left(\frac{\hbar^2 C_r}{4L_r}\right)^{1/4}(\hat{c} + \hat{c}) = 2e\beta\hat{N}V_{rms}^0(\hat{c} + \hat{c}) \tag{380}$$

where $\beta = \frac{C_g}{C_s}$ and $V_{rms}^0 = \sqrt{\frac{\hbar\omega_r}{2C_r}}$. Writing $\hat{N}$ in the *reverted* basis of eigenstates of σ_x, as follows:

$$\hat{N} \to \frac{1}{2}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix}\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix} = I - (\sigma^+ + \sigma^-), \tag{381}$$

and applying the RWA (*i.e.*, dropping terms with unequal powers of raising and lowering operators), we obtain the coupling between the qubit and the resonator, as follows: $\frac{C_g}{C_s C_r} Q_r Q_J = -\hbar g(\hat{\sigma}^+\hat{c} + \hat{\sigma}^-\hat{c})$, where $g = \frac{\beta e}{\hbar} 2V_{rms}^0$,[6] giving the transmon-resonator Jaynes-Cummings Hamiltonian,

$$\hat{H} = \hbar\omega_r(\hat{c}^\dagger\hat{c} + \frac{1}{2}) + \frac{\hbar\omega_q}{2}\sigma_z - \hbar g(\hat{\sigma}^+\hat{c} + \hat{\sigma}^-\hat{c}), \tag{382}$$

with $\omega_q = \hbar^{-1}E_J$, where we have dropped constant terms that do not affect the dynamics of the transmon.

[6] This constant is different in Ref. [PRA2004]

20.6.2 Resonant and Dispersive Limits

In this subsection we discuss the resonant and dispersive limits of a two-level system coupled to a resonator, such as the transmon capacitively coupled to a coplanar resonator on a sapphire substrate (Fig. 56, top), a transmon in a 3D cavity resonator (Fig. 56, middle), and an atom passing through a cavity defined by two mirrors that confine a single mode of an electromagnetic field with frequency ω_c. As discussed in Sec. 21, all of these qubits coupled to an electromagnetic mode in the cavity resonator can be described by the same Jaynes-Cummings model Hamiltonian, introduced by Eq. (382).

We analyze the dynamics of the qubit when its frequency is resonant with the cavity (*i.e.*, $\omega_q = \omega_c$), and in the dispersive limit (*i.e.*, when the two frequencies are very different, for example, $\omega_q \ll \omega_c$).

20.6.2.1 Resonant Limit: When the frequency of the photon in the cavity matches the frequency of the qubit, the system exhibits Rabi oscillations due to coherent absorption of a single photon, followed by emission, reabsorption and emission mutliple times .

Simulation: A turn-key tutorial on how to simulate the dynamics of a qubit in resonance with a mode in the cavity on Colab or the IBM Quantum can be downloaded as a notebook:VacuumRabiOscillations.ipynb, and VacuumRabiOscillations.pdf.

20.6.2.2 Dispersive Limit: In the dispersive (off-resonance) limit, the resonator and the qubit are far off-resonance, $\Delta \gg g$, where $\Delta = |\omega_r - \omega_q|$ is the detuning between the resonator and the qubit (for example $\omega_r \gg \omega_q$), so only virtual photon exchange is allowed. Nevertheless, even a single off-resonance photon in the cavity produces a large effect on the effective frequency of the qubit (without ever being absorbed).

Here, we show how a photon affects the frequency of qubit. We transform the James-Cummings Hamiltonian, introduced by Eq. (382), $H = H_0 + V$, with $H_0 = \omega_r a^\dagger a + \frac{1}{2}\omega_q \sigma_z$ and $V = -\hbar g(a^\dagger + a)\sigma_x$, according to the similarity transformation $\tilde{H} = e^{\xi} H e^{-\xi}$, with $\xi = g(a\sigma^+ - a^\dagger\sigma^-)$. Using the Backer Campbell Hausdorff relation, $e^A B e^{-A} = B + [A,B] + \frac{1}{2}[A,[A,B]] + \cdots$, we obtain the perturbative expansion $\tilde{H} = H_0 + V +$

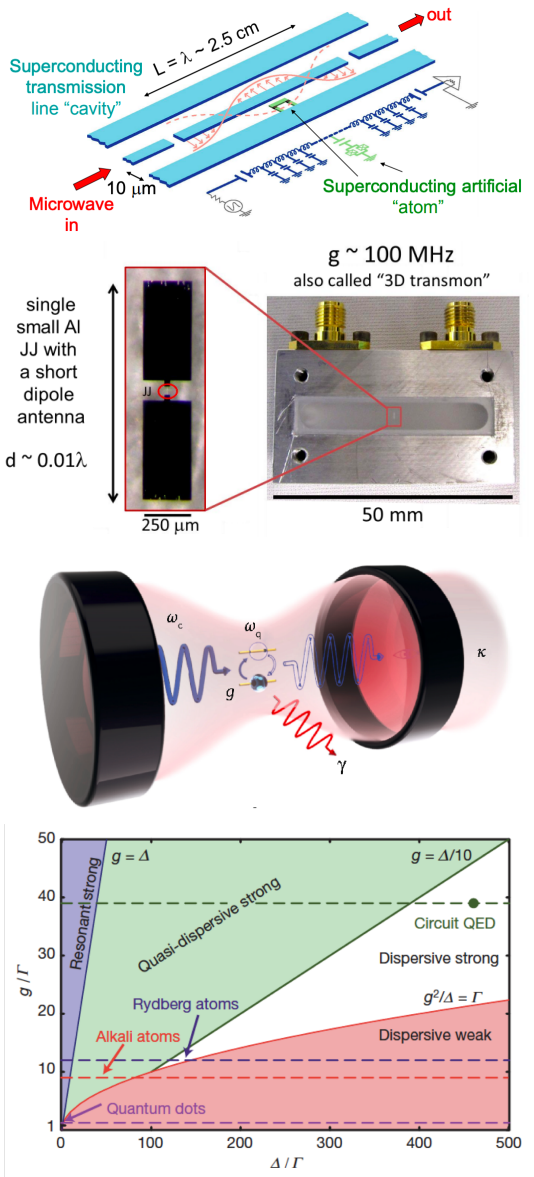


Figure 56: **Top:** Coplanar superconducting readout resonator on a sapphire substrate capacitively coupled to a transmon. **Middle:** Transmon in a 3D cavity resonator. **Bottom:** Two-level atom interacting with the electromagnetic field in a cavity, defined by two mirrors that confine a single mode of an oscillatory electromagnetic field with frequency ω_c. Parameter diagram for the Jaynes-Cummings model defined by the atom-photon coupling strength, g, and the detuning frequency $\Delta = \omega_q - \omega_c$, normalized to the decay rates $\Gamma = \max[\gamma, \kappa, 1/T]$.

$[\xi, H_0] + [\xi, V] + \frac{1}{2!}[\xi, [\xi, H_0]] + \cdots$, we find that $[\xi, H_0] = -V$, and keeping terms up to second order in g (*i.e.*, solving to second order after a dispersive approximation), we obtain that the effective Hamiltonian in the dispersive limit is approximately,

$$\begin{aligned}\tilde{H} &= \omega_r(a^\dagger a + 1/2) + \frac{1}{2}\omega_q\sigma_z + \chi(a^\dagger a + 1/2)\sigma_z \\ &= (\omega_r + \chi\sigma_z)(a^\dagger a + 1/2) + \frac{1}{2}\omega_q\sigma_z \\ &= \omega_r(a^\dagger a + 1/2) + \frac{1}{2}(\omega_q + \chi(2a^\dagger a + 1))\sigma_z\end{aligned} \tag{383}$$

where $\chi = g^2/\Delta$. We can view the last term of the first line of Eq. (383) as a correction of the resonator frequency that depends on the qubit state (*i.e.*, second line of Eq. (383)), or a correction to the qubit frequency that depends on the number of photons in the cavity (*i.e.*, third line of Eq. (383)). So, the dispersive limit allows us to resolve the number of photons in the cavity by monitoring the frequency spectrum of the qubit, as reported by D. I. Schuster et al. *Resolving photon number states in a superconducting circuit* Nature 445, 515 (2007).

Simulation: A turn-key tutorial on how to simulate the dynamics of a qubit coupled to a resonator in the dispersive limit by using qutip on Colab or the IBM Quantum can be downloaded as a notebook:lecture10.ipynb, and lecture10.pdf. The simulation shows that the qubit and the resonator do not exchange energy and therefore keep the expectation value of their own occupation number constant throughout the dynamics. However, when preparing the initial state of the qubit in the superposition $|\psi_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, the spectrum of the resonator exhibits a splitting of its resonance frequency (analogously to a Fermi resonance), showing that the qubit state can be determined by probing the resonator. Analogously, we can determine the state of the resonator by measuring the qubit since the frequency of the qubit is shifted by a different amount for each Fock state of the resonator, according to $\chi\langle(2a^\dagger a + 1)\rangle$. For example, when the resonator is prepared in a coherent state, it has a Poisson probability distribution in Fock space (as shown in Sec. 18). Therefore, the qubit exhibits a spectrum with a Poisson distribution of intensities for the peaks corresponding to the various Fock states of the resonator.

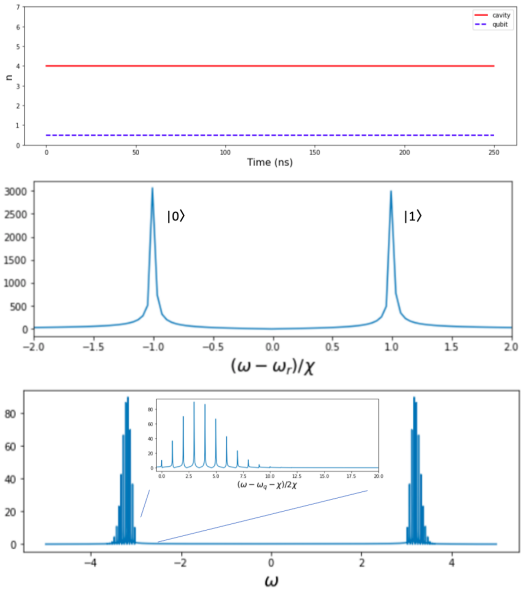


Figure 57: Results of a quantum dynamics simulation of a qubit-resonator system coupled in the dispersive limit. **Top:** Excitation numbers remain constant for the qubit and the cavity. **Middle:** Spectrum of the resonator with frequency ω_r, coupled to the qubit in a superposition state, thus exhibiting peaks splitted by $\chi\langle\sigma_z\rangle$ when the qubit is in state $|0\rangle$ and $|1\rangle$, respectively. **Bottom:** Spectrum of the qubit with frequency $\omega_q = 3.0$, exhibiting peaks shifted by $\chi\langle(2a^\dagger a + 1)\rangle$ for the resonator in a coherent state -i.e., with a Poisson distribution in Fock space.

21 Dicke Model and Jaynes-Cummings Hamiltonian

The *Jaynes Cummings* model Hamiltonian, derived in Sec. 20.6.1 for the description of the transmon-resonator circuit, was originally applied by the quantum optics community for the description of real atoms (qubits) in optical cavities. A typical experiment in *cavity quantum electrodynamics* (or cavity QED) involves a real atom in a cavity interacting with an electromagentic mode confined by two highly reflecting mirrors, as shown in Fig. 56 (bottom panel).

Without invoking the RWA, the qubit-resonator model system is described by the Dicke Hamiltonian (with $\hbar = 1$),

$$H = \omega_c(a^\dagger a) + \frac{1}{2}\omega_q\sigma_z + g(a^\dagger + a)\sigma_x, \tag{384}$$

where ω_c and ω_q are the bare frequencies of the resonator and qubit, respectively, and g is the strength of the dipole interaction coupling the atom and field, which is inversely proportional to the volume of the cavity. Neglecting the high frequency terms according to the rotating wave approximation (RWA), we obtain the Jaynes-Cummings model Hamiltonian,

$$H \approx \omega_c(a^\dagger a) + \frac{1}{2}\omega_q\sigma_z + \frac{g}{2}(a^\dagger\sigma^- + a\sigma^+), \tag{385}$$

In the weak-coupling limit, the rate of spontaneous emission due to coupling with modes in the cavity and surrounding environment can be described by the Golden Rule, discussed in Sec. 17. In resonance, the rate is enhanced when the cavity enhances the local density of states at the resonant frequency (Purcell effect). Furthermore, confinement can *suppress* spontaneous decay when the size of the cavity $d < \lambda/2$, with λ the wavelength of the atom electronic transition. Therefore, coupling of a qubit to a cavity resonator can stabilize the qubit prepared in the excited state by suppressing spontaneous decay.

The coupling g can be made much stronger than the coupling to all the other modes of the electromagnetic field in the surrounding environment simply by making a very small cavity (*e.g.*, with the mirrors very close to each other) so that the volume V of the cavity is small and thus the density of states in the cavity is much larger than the density of states in the environment (so the Purcell factor $F_p = \frac{3}{4\pi^2}\left(\frac{\lambda}{n}\right)^3\frac{Q}{V}$ is large). Here, Q is the quality factor, n is the refractive index of the medium and λ is the frequency of the free photon. So, the 'transverse' decay rate γ of the atom due to coupling with the surrounding field can be made very small, so the cavity effectively protects the atom from coupling to the surrounding environment. In addition, the photon decay described by the cavity decay κ can be made very small by making highly reflecting super-polished mirrors.

In the dispersive limit, the interaction of the atom with the resonator mode changes the atom level spacing proportionally to the number of photons in the cavity, as discussed in Sec. 20.6.1, a phenomenon initially shown by Cohen-Tannoudji, 1961. Therefore, the frequency of the atom electronic transition can be probed to determine the number of photons in the cavity (even when the photons are never absorbed or emitted by the atom).

Early Experiments: Experiments at Yale by Serge Haroche demonstrated for the first time suppression of spontaneous emission in the near infrared by preparing cesium atoms in the low-lying 5d level, passing them through a cavity that inhibited the $5d \rightarrow 6p$ at wavelength of 3.5 microns and propagated through the cavity for about 12 natural lifetimes without appreciable decay.

22 Appendix V: Python

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing. I expect that many of you will have some experience with Python and numpy; for the rest of you, this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images
- IPython: Creating notebooks, Typical workflows

22.1 A Brief Note on Python Versions

As of Janurary 1, 2020, Python has officially dropped support for `python2`. We'll be using Python 3.7 for this iteration of the course. You can check your Python version at the command line by running `python --version`. In Colab, we can enforce the Python version by clicking `Runtime -> Change Runtime Type` and selecting `python3`. Note that as of April 2020, Colab uses Python 3.6.9 which should run everything without any errors.

[6]:
```
!python --version
```

```
Python 3.6.9
```

22.1.1 Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

[7]:
```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

22.1.1.1 Basic data types

22.1.1.1.1 Numbers

Integers and floats work as you would expect from other languages:

[8]:
```
x = 3
print(x, type(x))
```

```
3 <class 'int'>
```

[9]:
```
print(x + 1)   # Addition
print(x - 1)   # Subtraction
print(x * 2)   # Multiplication
print(x ** 2)  # Exponentiation
```

```
4
2
6
9
```

[10]:
```
x += 1
print(x)
x *= 2
print(x)
```

```
4
8
```

[11]:
```
y = 2.5
print(type(y))
print(y, y + 1, y * 2, y ** 2)
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x–) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

22.1.1.1.2 Booleans

Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

[12]:
```
t, f = True, False
print(type(t))
```

```
<class 'bool'>
```

Now we let's look at the operations:

[13]:
```
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

22.1.1.1.3 Strings

[14]:
```
hello = 'hello'   # String literals can use single quotes
world = "world"   # or double quotes; it does not matter
print(hello, len(hello))
```

```
hello 5
```

[15]:
```
hw = hello + ' ' + world  # String concatenation
print(hw)
```

```
hello world
```

[16]:
```
hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
print(hw12)
```

```
hello world 12
```

String objects have a bunch of useful methods; for example:

[17]:
```
s = "hello"
print(s.capitalize())  # Capitalize a string
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces
print(s.center(7))     # Center a string, padding with spaces
print(s.replace('l', '(ell)'))  # Replace all instances of one␣
 ↪substring with another
print('  world '.strip())  # Strip leading and trailing whitespace
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

You can find a list of all string methods in the documentation.

22.1.1.2 Containers Python includes several **built-in container types**: lists, dictionaries, sets, and tuples.

1. List item
2. List item
3. List item
4. List item

22.1.1.2.1 Lists A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

[18]:
```
xs = [3, 1, 2]   # Create a list
print(xs, xs[2])
print(xs[-1])     # Negative indices count from the end of the list;␣
 ↪prints "2"
```

```
[3, 1, 2] 2
2
```

Lists can be generated from arrays, as follows:

[20]:
```
import numpy as np

int_list = [] # list initialization
int_list = [0,0,1,2,3] # list with commas
int_list.append(4) # add 4 to end of the list
int_list.pop(2) # remove element with index 2

int_list2 = list(range(5)) # make list [0,1,2,3,4]
int_array = np.array(int_list) # make array [] with no commas: [0 1 2␣
 ↪3 4]
int_array2 = np.arange(5)  # make array [] with no commas: [0 1 2 3 4]
int_list2 = int_array.tolist()  # convert array to list

first = 0
last = 4
float_array = np.linspace(first,last,num=5)

print('int_list=',int_list)
print('int_list2=',int_list2)
print('int_array=',int_array)
print('int_array2=',int_array2)
print('float_array=',float_array)
```

```
int_list= [0, 0, 2, 3, 4]
int_list2= [0, 0, 2, 3, 4]
int_array= [0 0 2 3 4]
```

```
int_array2= [0 1 2 3 4]
float_array= [0. 1. 2. 3. 4.]
```

[21]:
```
xs[2] = 'foo'    # Lists can contain elements of different types
print(xs)
```

```
[3, 1, 'foo']
```

Lists have methods, including append, insert, remove, sort

[22]:
```
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

```
[3, 1, 'foo', 'bar']
```

[23]:
```
x = xs.pop()     # Remove and return the last element of the list
print(x, xs)
```

```
bar [3, 1, 'foo']
```

As usual, you can find all the gory details about lists in the documentation.

22.1.1.2.2 Slicing In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

[24]:
```
nums = list(range(5))    # range is a built-in function that creates␣
 ↪a list of integers
print(nums)          # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])     # Get a slice from index 2 to 4 (exclusive);␣
 ↪prints "[2, 3]"
print(nums[2:])      # Get a slice from index 2 to the end; prints␣
 ↪"[2, 3, 4]"
print(nums[:2])      # Get a slice from the start to index 2␣
 ↪(exclusive); prints "[0, 1]"
print(nums[:])       # Get a slice of the whole list; prints ["0, 1,␣
 ↪2, 3, 4]"
print(nums[:-1])     # Slice indices can be negative; prints ["0, 1,␣
 ↪2, 3]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print(nums)          # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

22.1.1.2.3 Loops You can loop over the elements of a list like this:

[25]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

```
cat
dog
monkey
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

[26]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

```
#1: cat
#2: dog
#3: monkey
```

22.1.1.2.4 List comprehensions: When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

[27]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

[28]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

[29]:
```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

```
[0, 4, 16]
```

22.1.1.2.5 Dictionaries A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

[30]:
```
d = {}
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with␣
 ↪some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints␣
 ↪"True"
```

```
cute
True
```

[31]:
```
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
```

```
wet
```

[32]:
```
print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most␣
↪recent call last)
```

[33]:
```
print(d.get('monkey', 'N/A'))  # Get an element with a default;␣
 ↪prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default;␣
 ↪prints "wet"
```

```
N/A
wet
```

[34]:
```
del d['fish']         # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

You can find all you need to know about dictionaries in the documentation.
It is easy to iterate over the keys in a dictionary:

[35]:
```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A {} has {} legs'.format(animal, legs))
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Add pairs to the dictionary

[36]:
```
d['bird']=2
```

List keys

[37]:
```
d.keys()
```

[37]:
```
dict_keys(['person', 'cat', 'spider', 'bird'])
```

List Values

[38]:
```
d.values()
```

[38]:
```
dict_values([2, 4, 8, 2])
```

Query values from keys

[39]:
```
d['bird']
```

[39]:
```
2
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

[40]:
```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Convert array to list

22.1.1.2.6 Sets (like dictionaries but with no values, add & remove)

A set is an unordered collection of distinct elements. As a simple example, consider the following:

[41]:
```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints␣
 ↪"True"
print('fish' in animals)  # prints "False"
```

```
True
False
```

[42]:
```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

[43]:
```
animals.add('cat')        # Adding an element that is already in the
 ↪set does nothing
print(len(animals))
animals.remove('cat')     # Remove an element from a set
print(len(animals))
```

```
3
2
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

[44]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

```
#1: fish
#2: dog
#3: cat
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

[45]:
```
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

22.1.1.2.7 Tuples A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a simple example:

[46]:
```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with
 ↪tuple keys
print(d)

tt = ()  # initialization of empty tuple
t1 = (66,)  # initialization of tuple with a single value
t = (5, 6)       # Create a tuple
tt = tt+t1+t
print("tt=",tt)
print("tt[2]=",tt[2])
print("tt[1:3]=",tt[1:3])
print("66 in tt", 66 in tt)

print(type(t))
print(d[t])
print(d[(1, 2)])
```

```
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5, (6,␣
↪7): 6,
(7, 8): 7, (8, 9): 8, (9, 10): 9}
tt= (66, 5, 6)
tt[2]= 6
tt[1:3]= (5, 6)
66 in tt True
<class 'tuple'>
5
1
```

[47]:
```
t[0] = 1
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most␣
↪recent call last)

<ipython-input-47-c8aeb8cd20ae> in <module>()
----> 1 t[0] = 1

TypeError: 'tuple' object does not support item assignment
```

22.1.1.3 Functions

Python functions are defined using the `def` keyword. For example:

[48]:
```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments, like this:

[49]:
```
def hello(name, loud=False):
    if loud:
```

```
        print('HELLO, {}'.format(name.upper()))
    else:
        print('Hello, {}!'.format(name))

hello('Bob')
hello('Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

22.1.1.4 Classes A new class creates a new type of object, bounding data and functionality that allows new instances of the type made. Each class instance can have attributes attached to it, so we can make class instances as well as instances to variables and methods for maintaining the state of the class. Instances of the method can have attributes and can modify the state of the class, as clearly described by the documentation.

The syntax for defining classes in Python is straightforward:

[50]:
```
class Greeter:
    """ My greeter class """
    # Constructor (method of construction of class in a specific␣
 ↪state)
    v1 ='papa' # class variable shared by all instances
    def __init__(self, name_inp): # name_inp: argument given to␣
 ↪Greeter for class instantiation
        self.name = name_inp  # Create an instance variable␣
 ↪maintaining the state
                              # instance variables are unique to each␣
 ↪instance
    # Instance method
    # note that the first argument of the function method is the␣
 ↪instance object
    def greet(self, loud=False):
        if loud:
          print('HELLO, {}'.format(self.name.upper()))
          self.name = 'Haote'
        else:
          print('Hello, {}!'.format(self.name))
          self.name = 'Victor'

# Class instantiation (returning a new instance of the class assigned␣
 ↪to g):
# Constructs g of type Greeter & initialzes its state
# as defined by the class variables (does not execute methods)
g = Greeter('Fred')

# Call an instance method of the class in its current state:
```

```
# prints "Hello, Fred!" and updates state varible to 'Victor' since␣
 ↪loud=False
g.greet() # equivalent to Greeter.greet(g) since the first arg of␣
 ↪greet is g

# Call an instance method; prints "HELLO, VICTOR" and updates␣
 ↪variable to 'Haote'
g.greet(loud=True)    # equivalent to Greeter.greet(g,loud=True)
                      #since the first arg of greet is g
print(g.v1)
g.greet()             # Call an instance method; prints "Hello, Haote!"
                      # A method object is created by packing␣
 ↪(pointers to) the
                      # instance object g and the function object greet

g2 = Greeter('Lea')    # Class instance reinitializes variable to 'Lea'

g2.greet()             # Call an instance method; prints "Hello, Lea!"
g2.__doc__
g2.x=20                # Data attributes spring into existence upon␣
 ↪assignment
print(g2.x)
del g2.x               # Deletes attribute
g2.v1
```

```
Hello, Fred!
HELLO, VICTOR
papa
Hello, Haote!
Hello, Lea!
20
```

[50]:
```
'papa'
```

For loops (iterators). Behind the scenes, the for statement calls iter() on the container object.

[51]:
```
for element in [1,2,3]: # elements of list
  print(element)
for element in (1,2,3): # elements of tuple
  print(element)
for key in {'first':1, 'second':2, 'third':3}: # elements of␣
 ↪dictionary
  print('key=',key)
for char in '1234':
  print(char)
#for line in open(''myfile.txt)
#  print(line,end='')
```

```
1
2
3
1
2
3
key= first
key= second
key= third
1
2
3
4
```

22.1.1.5 Modules A module is a .py file containing Python definitions and statements that can be imported into a Python script, as described in the Python documentation.

As an example, after mounting your Google drive as described by the Navigating_tutorial.ipynb Jupyter notebook, use a text editor and write a module with the line:

[52]:
```
greeting = "Good Morning!"
```

Save the document with the name mymod.py

Next, go the the folder where you saved that file and open a notebook with the lines:

[ ]:
```
import mymod as my
print(my.greeting)
```

you will see that the notebook has imported the variable greetingfrom the module `mymod.py` and has invoked the variable as an attribute of the module mymod that was imported as my when printing `Good Morning!!`.

Modules are very convenient since they allow you to import variables, functions and classes that you might have developed for previous projects, without having to copy them into each program. So, you can build from previous projects, or split your work into several files for easier maintenance.

Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

22.1.2 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multi-dimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

To use Numpy, we first need to import the `numpy` package:

[54]:
```
import numpy as np
```

22.1.2.1 Arrays A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

[55]:
```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

[56]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b)
```

```
[[1 2 3]
 [4 5 6]]
```

[57]:
```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

[58]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

```
[[0. 0.]
 [0. 0.]]
```

[59]:
```
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

```
[[1. 1.]]
```

[60]:
```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

[61]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

[62]:
```
e = np.random.random((2,2)) # Create an array filled with random␣
 ↪values
print(e)
```

```
[[0.32071297 0.96986179]
 [0.32331846 0.50510489]]
```

22.1.2.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[63]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

[64]:
```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

[65]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

[66]:
```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
row_r3 = a[[1], :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

[67]:
```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

[68]:
```
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
[1 4 5]
```

[69]:
```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])
```

```
# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

[70]:
```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

[71]:
```
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[ 1  6  7 11]
```

[72]:
```
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

[73]:
```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the␣
 ↪same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.
```

```
print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

[74]:
```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

22.1.2.3 Datatypes Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

[75]:
```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

22.1.2.4 Array math Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[76]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

[77]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

[78]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

[79]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

[80]:
```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, $\star$ is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

[81]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

You can also use the `@` operator which is equivalent to numpy's `dot` operator.

[82]:
```
print(v @ w)
```

```
219
```

[83]:
```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

[84]:
```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

[85]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
```

```
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

[86]:
```
print(x)
print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

[87]:
```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

22.1.2.5 Broadcasting Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

[88]:
```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape␣
 ↪as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v
```

```
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

[89]:
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #         [1 0 1]
                         #         [1 0 1]
                         #         [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

[90]:
```
y = x + vv  # Add x and vv elementwise
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

[91]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

[92]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

[93]:
```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

[94]:
```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
```

```
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[95]:
```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[96]:
```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

22.1.3 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

[97]:
```
import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline:

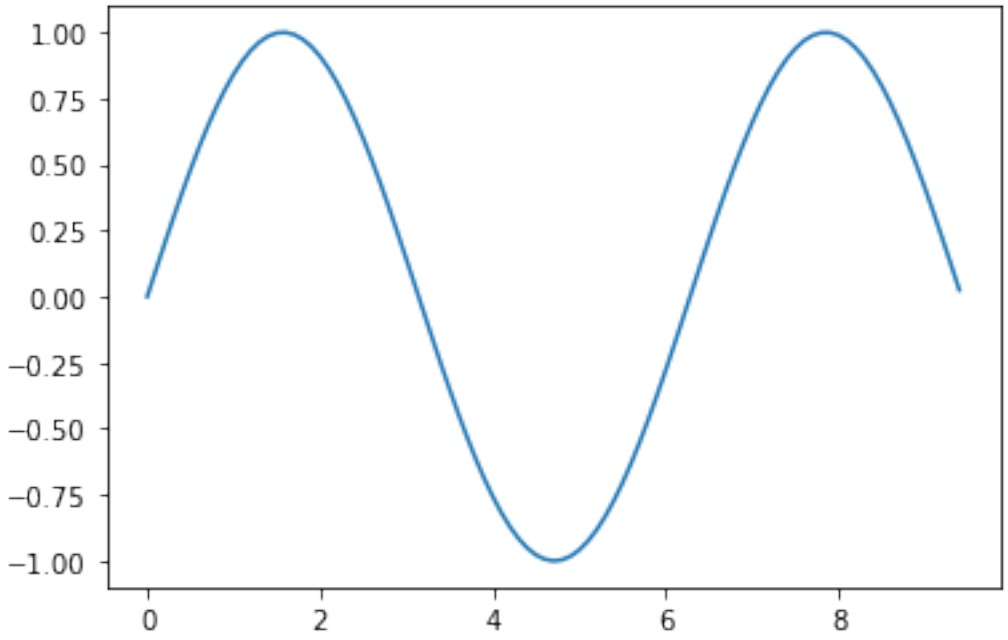
[98]:
```
%matplotlib inline
```

22.1.3.1 Plotting The most important function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

[99]:
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
```

```
# Plot the points using matplotlib
plt.plot(x, y)
```

[99]:
```
[<matplotlib.lines.Line2D at 0x7f78639a1748>]
```

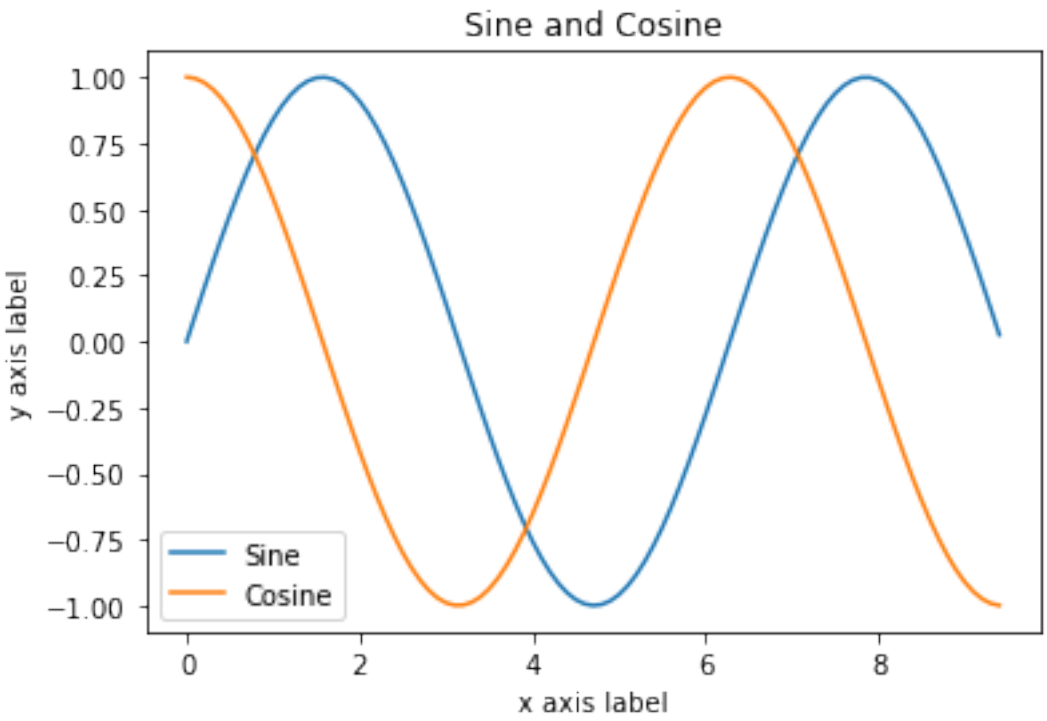


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

[100]:
```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

[100]:
```
<matplotlib.legend.Legend at 0x7f78634f2860>
```



22.1.3.2 Subplots You can plot different things in the same figure using the subplot function. Here is an example:

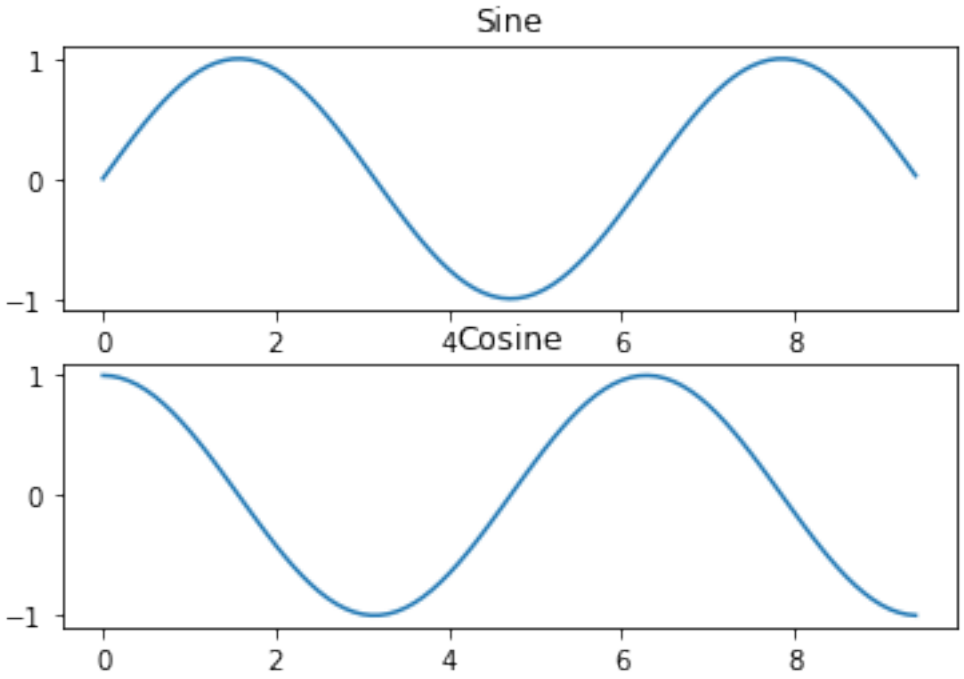
[101]:
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



You can read much more about the `subplot` function in the documentation.

22.2 Torch tensor

The pytorch tensor class involves two data attribute (*i.e.*, two data containers), including one analogous to a numpy multidimensional array with the elements of the tensor, and the other container with the gradients of an input function with respect to the tensor elements (Fig. 58). In addition, the tensor class involves the attribute *backward* method for the so-called *backward propagation* procedure applied for training neural networks that computes the gradients of the loss function with respect to the elements of the tensor. In addition, PyTorch has modules and functions summarized in Fig. 59.

As an example, we consider the following numpy multiarray

[102]:
```
nt=np.ones((2,2))
```

which can be used to build a corresponding torch tensor, with associated gradients, as follows:

[103]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

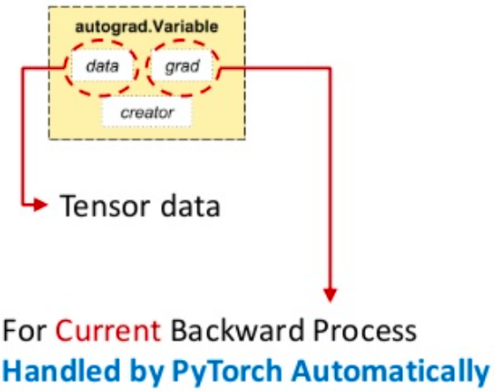


Figure 58: Schematic representation of a PyTorch tensor.

```
import torch.optim as optim
```

[104]:
```
r = torch.tensor(nt, requires_grad=True)
```

The resulting torch tensor can be combined with other torch tensors to define, for example, a function f as follows:

[105]:
```
p = torch.ones((2,2), requires_grad=True)
p2 = p+p
y=(r+2)+p2
z=y*y*3
f = z.mean()
print("r=",r)
print("p=",p)
print("f=",f)

print('before backward: r.grad=', r.grad)
```

```
r= tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64, requires_grad=True)
p= tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
f= tensor(75., dtype=torch.float64, grad_fn=<MeanBackward0>)
before backward: r.grad= None
```

Note that the torch tensor r does not have any gradients (i.e.,`before backward: r.grad= None`) since so far we have not invoked the method `backward` for any function of r.

Now we can compute the gradient of f with respect to the elments of r by instantiating the `backward` attribute of f, as follows:

[106]:
```
f.backward()
```

We can now check that the tensor r has the correct gradients of f with respect to the 4 elements of r, as follows:

[107]:
```
print('after backward: r.grad=', r.grad)
```

```
after backward: r.grad= tensor([[7.5000, 7.5000],
        [7.5000, 7.5000]], dtype=torch.float64)
```

We can also zero the gradients, as follows:

[108]:
```
r.grad.data.zero_()
print('after zero: r.grad=', r.grad)
```

```
after zero: r.grad= tensor([[0., 0.],
        [0., 0.]], dtype=torch.float64)
```

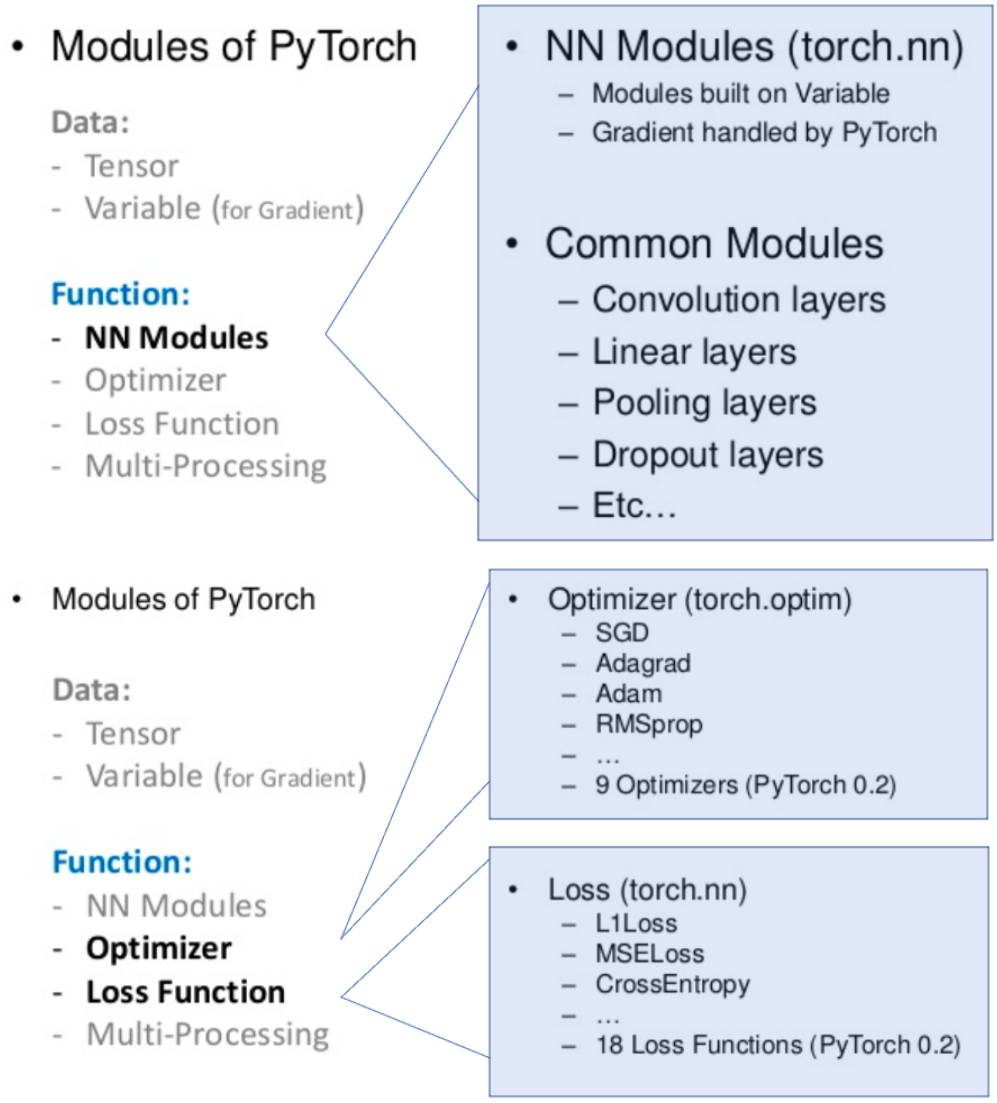


Figure 59: Summary of PyTorch modules and functions.