
# Spectrum Analysis in Microsoft Excel

**Nattaya Hanrattanasakul**

**This internship project is a part of education course in Bachelor of Engineering,**

**Computer Engineering**

**Faculty of Engineering**

**Thai-Nichi Institute of Technology**

**Year 2011**

**Spectrum Analysis in Microsoft Excel**

**Nattaya Hanrattanasakul**

**Cooperative Education Report submitted in partial fulfillment for the degree of**

**Bachelor of Engineering Program in Computer Engineering**

**Faculty of Engineering**

**Thai-Nichi Institute of Technology**

**Academic Year 2011**

**Cooperative Education Report Examination Committee:**

**..... Commission Chairman**

**(Prawet Ueatrongchit)**

**.....Committee and Advisor**

**(Warakorn Srichavengsup, Ph.D.)**

**.....Committee**

**(Adna Sengto)**

# Conclusion

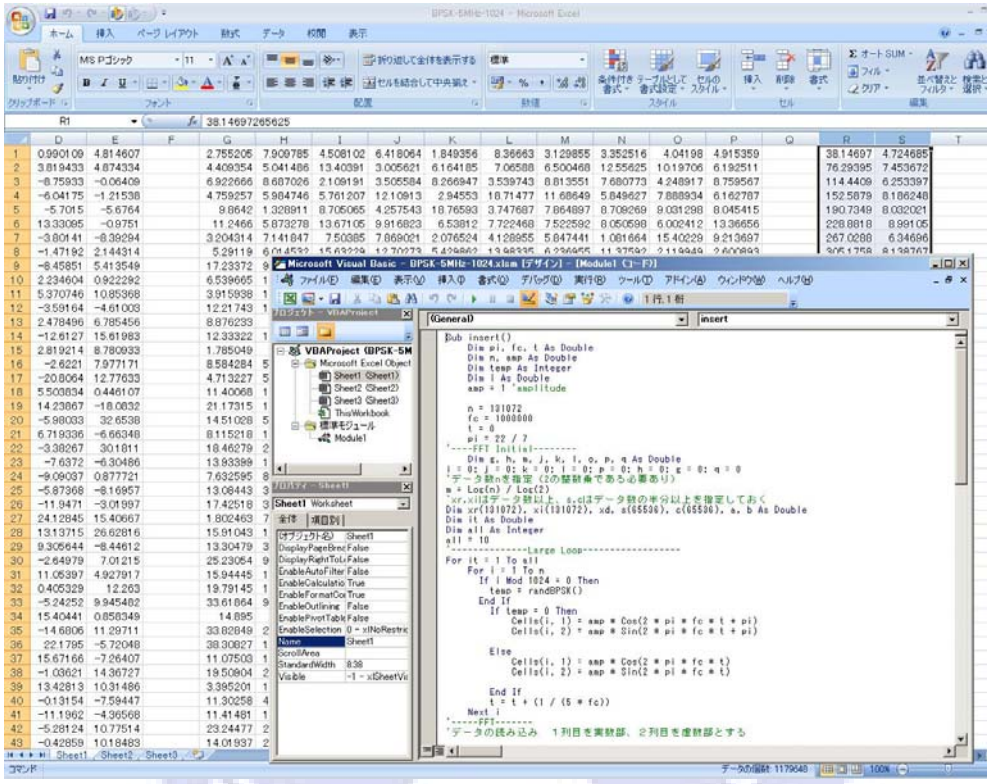
| | |
|---|---|
| Project Name | Spectrum Analysis in Microsoft Excel |
| Writer | Nattaya Hanrattanasakul |
| Faculty | Engineering |
| | Major Computer Engineering |
| Advisor Teacher | Warakorn Srichavengsup, Ph.D. |
| Advisor Staff | Takahashi Yurie |
| Organization Name | NEC |
| Bussiness Type | Electronic and Communication Developer |

Job Description

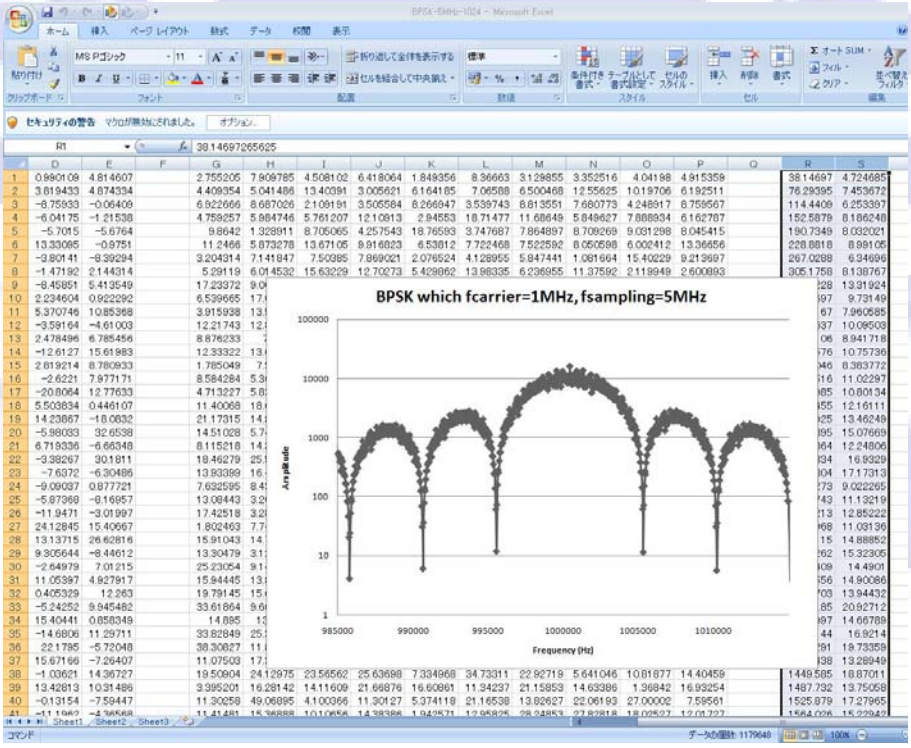
This project aims to calculate theoretical 100Gbps-QPSK signal and plot spectrum graph which needed for design ROADM optical channel.

Expected results

To effectively analyze spectrum from data by using Microsoft Excel and make it more convenient for learning and testing about spectrum in the future.



Visual Basic for Applications Coding




Spectrum Graph Plotting

**Acknowledgement**

I would like to express my sincere thanks to my advisor staff, Takahashi Yurie and all Optical Division Network group staffs at NEC Abiko Plant for their invaluable help and constant encouragement throughout this internship. I am most grateful for their teaching and advice, not only the research methodologies but also many other methodologies in life. I would not have achieved this far and this thesis would not have been completed without all the support that I have always received from them.

Nattaya Hanrattanasakul

Internship student

10 October 2011

# Contents

## Contents (Continue)

# List of Tables



## List of Figures

# Chapter 1

# Introduction

## 1.1 Organization Name and Location

NEC Organization

NEC Empowered by Innovation

**Figure 1.1** NEC Logo

**Abiko Plant**

1131, Hinode, Abiko, Chiba 270-1198, Japan


**Figure 1.2** Abiko Plant Map

## 1.2 Business type of the organization

On 17 July 1899, Nippon Electric Company, Limited (renamed NEC Corporation, effective April, 1983, both expressed as NEC hereafter) Japan's first joint venture with foreign capital, was established by Kunihiko Iwadare in association with the U.S. firm Western Electric Company (presently Alcatel-Lucent).

The basic aim of the new company, expressed in the slogan “Better Products, Better Service,” was to carry out the promise to provide its customers with world-class products and dependable follow-up service. The notion of follow-up service didn't take root among Japanese businesses until a full half-century later, whereas NEC had from the beginning embraced a concept that developed into what we now call Customer Satisfaction (CS).

World and domestic firsts in technology and research development, made possible by managerial innovation and backed by establishment, improvement and reform of its various personnel systems, as well as the early mounting of environmental projects, make it possible to say that NEC's history has been marked by constant innovation for more than a hundred years. NEC is empowered by the DNA of innovation.

## 1.3 Organization Structure

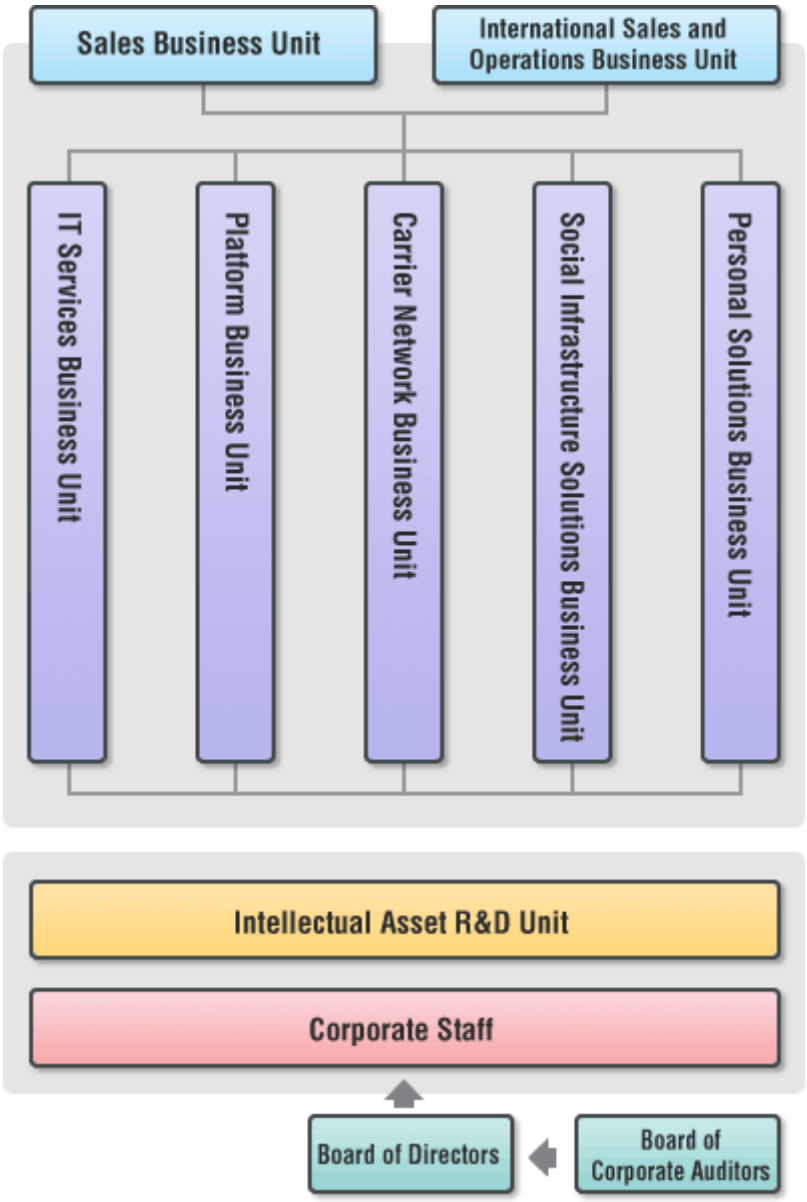



**Figure 1.3** Organization Structure

## 1.4 Position and Work

Position : Internship Student of Optical Networks Division Development Group (Lambda Network System 2).

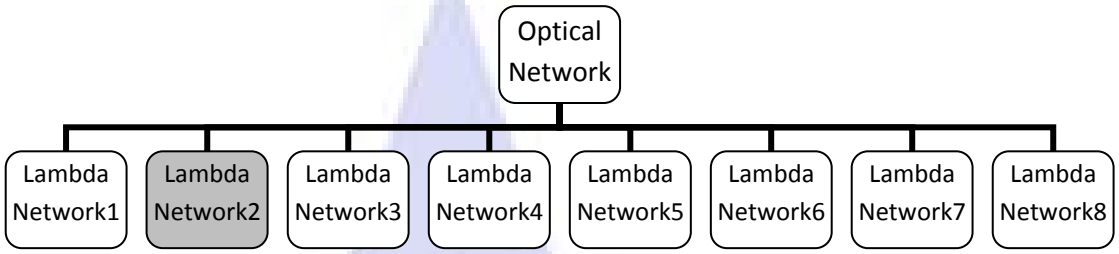



**Figure 1.4** Given Position

Work : Coding Visual Basic for Application Language to analyse spectrum from real baud rate digital signal in Microsoft Excel.

## 1.5 Advisor staff's name and position

Advisor staff : Takahashi Yurie

Position : Engineer

## 1.6 Work Period

15 August 2011 – 30 September 2011

## 1.7 Aimed result from project

To calculate theoretical 100Gbps-DP-QPSK signal and plot spectrum graph which needed for design ROADM optical channel.

## 1.8 Expected result from project

Be able to make frequency-amplitude graph from digital data as ideal as posible by use Quadrature Phase-Shift Keying Modulation and 100Gbps-DP-QPSK signal.

# Chapter 2

# Literature review

## 2.1 Visual Basic for Applications

Visual Basic for Applications (VBA) is an implementation of Microsoft's event-driven programming language Visual Basic 6 and its associated integrated development environment (IDE), which are built into most Microsoft Office applications. VBA enables building user defined functions, automating processes and accessing Windows API and other low-level functionality through dynamic-link libraries (DLLs). It was also built into Office applications apart from version 2008 for Apple's Mac OS X, other Microsoft applications such as Microsoft MapPoint and Microsoft Visio; as well as being at least partially implemented in some other applications such as AutoCAD, WordPerfect and ArcGIS. It supersedes and expands on the abilities of earlier application-specific macro programming languages such as Word's WordBasic. It can be used to control many aspects of the host application, including manipulating user interface features, such as menus and toolbars, and working with custom user forms or dialog boxes. VBA can also be used to create import and export filters for various file formats, such as OpenDocument (ODF).

As its name suggests, VBA is closely related to Visual Basic and uses the Visual Basic Runtime, but can normally only run code within a host application rather than as a standalone application. It can, however, be used to control one application from another via OLE Automation. For example, it is used automatically to create a Word report from Excel data, in turn automatically collected by Excel from polled observation sensors. The VBA IDE is reached from within an Office document by pressing the key sequence Alt+F11.

VBA is functionally rich and flexible but it does have some important limitations, such as restricted support for function pointers which are used as callback functions in the Windows API. It has the ability to use (but not create) (ActiveX/COM) DLLs, and later versions add support for class modules.

**Language**

Code written in VBA is compiled to a proprietary intermediate language called *P-code* (packed code), which the hosting applications (Access, Excel, Word and PowerPoint) store as a separate stream in COM Structured Storage files (e.g., .doc or .xls) independent of the document streams. The intermediate code is then executed by a virtual machine (hosted by the hosting application). Despite its resemblance to many old BASIC dialects (particularly Microsoft BASIC, from which it is indirectly derived), VBA is incompatible with any of them except Visual Basic, where source-code of VBA modules and classes can be directly imported, and which shares the same library and virtual machine. Compatibility ends with Visual Basic version 6; VBA is incompatible with Visual Basic .NET (VB.NET). VBA is proprietary to Microsoft and, apart from the COM interface, is not an open standard.

**Object models**

To use VBA with an application such as Access, Word or Excel, terminology and language constructions are needed to interact with the application. This portion of VBA is called the *Object Model* for the application. A map of the object model is online for Excel and for Word. A listing of the object model is found by opening the Macro/VBA editor in the target application and then using "View" to open the "Object Browser" (F2).

Much of the difficulty in using VBA is related to learning the object model, which uses names invented by the originators of the model that may be less than transparent to a new user. One way to learn the terms and syntax of the object model is to use the *macro recorder* to record the steps taken to achieve a desired result using the mouse and menus of the application. Once this is done, the VBA code constructed by the recorder can be viewed in the VBA editor, and often greatly streamlined or generalized with only a modicum of understanding of VBA itself. The macro recorder does not always record everything (particularly for graphs), and some applications employing VBA do not provide a recorder at all. Use of debugging tools to discover VBA constructs for some cases where the macro recorder does not work are described by Jelen and Syrstad, but some steps may remain obscure.

A more complete description of the strengths and weaknesses of the Visual Basic language is found in Visual Basic.

**Automation**

Interaction with the host application uses OLE Automation. Typically, the host application provides a type library and application programming interface (API) documentation which document how VBA programs can interact with the application. This documentation can be examined from inside the VBA development environment using its *Object Browser*.

VBA programs which are written to use the OLE Automation interface of one application cannot be used to automate a different application, even if that application hosts the Visual Basic runtime, because the OLE Automation interfaces will be different. For example, a VBA program written to automate Microsoft Word cannot be used with a different word processor, even if that word processor hosts VBA.

Conversely, multiple applications can be automated from the one host by creating Application objects within the VBA code. References to the different libraries must be created within the VBA client before any of the methods, objects, etc. become available to use in the application. These application objects create the OLE link to the application when they are first created. Commands to the different applications must be done explicitly through these application objects in order to work correctly.

For example: In Microsoft Access, users automatically have access to the Access library. References to the Excel, Word and Microsoft Outlook libraries can also be created. This will allow creating an application that runs a query in Access, exports the results to Excel, formats the text, then writes a mail merge document in Word that it automatically e-mails to each member of the original query through Outlook. (In this example, Microsoft Outlook contains a security feature that forces a user to allow, disallow, or cancel an e-mail being sent through an automated process with a forced 5 second wait. Information on this can be found at the Microsoft website.)

VBA programs can be attached to a menu button, a *macro*, a keyboard shortcut, or an OLE/COM event, such as the opening of a document in the application. The language also provides a user interface in the form of UserForms, which can host ActiveX controls for added functionality.

**Security concerns**

Like any common programming language, VBA macros can be created with a malicious intent. Using VBA, most of the security features lie in the hands of the user, not the author. The VBA 'host-application' options are accessible to the user. The user who runs any document containing VBA macros can preset the software with user preferences, much like those for web browsers. End-users can protect themselves from attack by disabling macros from running in an application if they do not intend to use documents containing them, or only grant permission for a document to run VBA code if they are sure the source of the document can be trusted. However, if the author is known VBA code is no more dangerous than any other

**Named variables and user-defined functions**

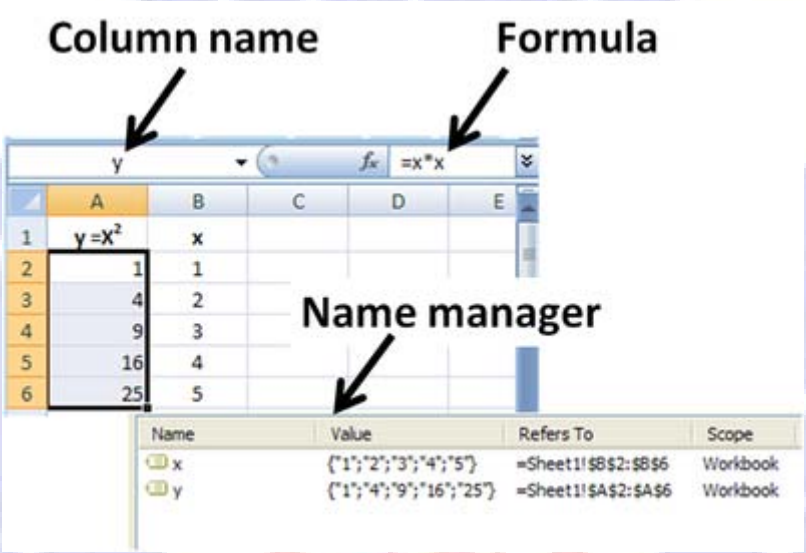



**Figure 2.1** Named variables

Use of named column variables x & y in Microsoft Excel; y = x*x is calculated using the formula displayed in the formula box, which is copied down the entire y-column.

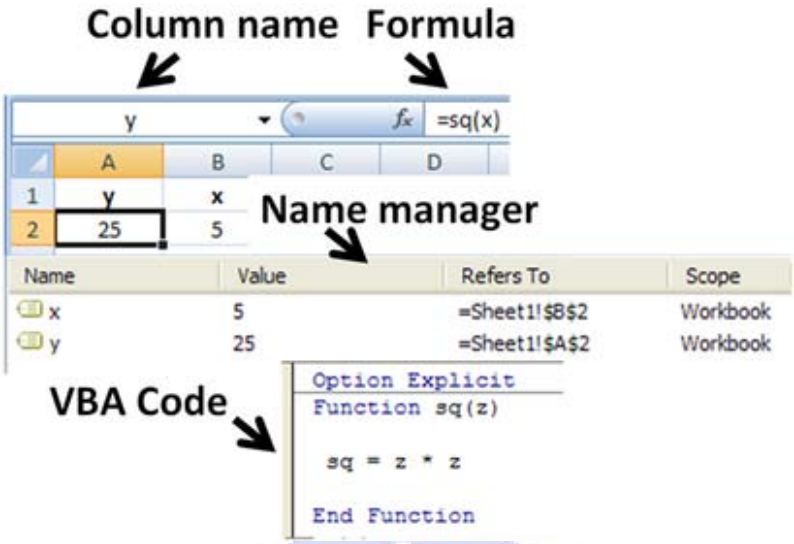



**Figure 2.2** User-defined functions

Use of a user-defined function sq(x) of named variable x in Microsoft Excel. Function supplied automatically from the code in the Visual Basic for Applications editor.

A common use of VBA is to add functionality that may be missing from the standard user interface. Use of VBA is made much easier by using *named variables* on the spreadsheet, as shown at the left. The formula for $y=x^2$ resembles Fortran or BASIC, and the *Name Manager* shows the definitions of column variables *y* and *x*.

Using VBA, the user can add their own functions and subroutines that refer to these named ranges. In the figure at the right, the function *sq* is created in the *Visual Basic* editor supplied with Excel, and *x* & *y* are named variables in the spreadsheet.

**Subroutines**

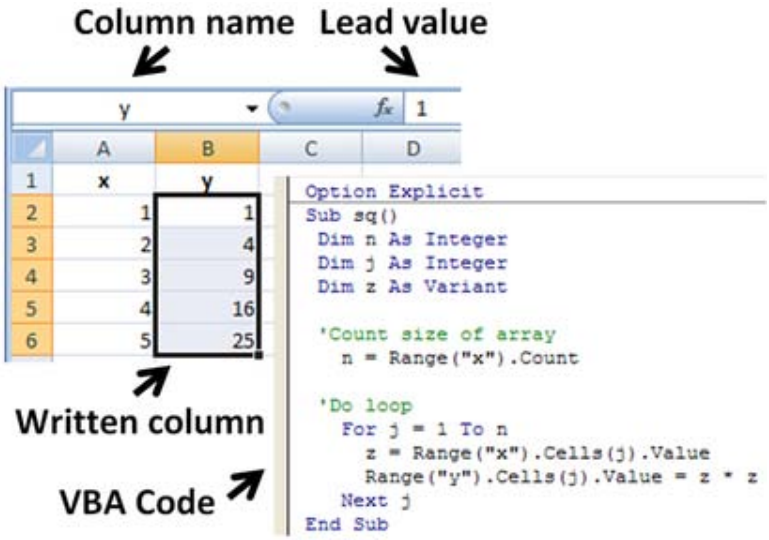



**Figure 2.3** Subroutines

Subroutine in Excel calculates the square of named column variable x read from the spreadsheet, and writes it into the named column variable y.

Functions themselves cannot write into the worksheet, but simply return their evaluation. However, in Microsoft Excel, subroutines can write values or text found within the subroutine directly to the spreadsheet. The figure shows the Visual Basic code for a subroutine that reads each member of the *x*-column (named column variable *x*), calculates its square, and writes this value into the corresponding *y*-column (also a named column variable). The *y*-column contains no formula because its values are calculated in the subroutine and simply written in.

**Examples**

This macro provides a shortcut for entering the current date in Word:

```
Sub EnterCurrentDate()
    Selection.InsertDateTime DateTimeFormat:="dd-MM-yy", InsertAsField:=False, _
        DateLanguage:=wdEnglishAUS, CalendarType:=wdCalendarWestern, _
        InsertAsFullWidth:=False
End Sub
```

VBA is useful for automating database tasks such as traversing a table:

```
Sub LoopTableExample
   Dim db As DAO.Database
   Dim rs As DAO.Recordset
   Set db = CurrentDb
   Set rs = db.OpenRecordset("select columnA, columnB from tableA")
   Do Until rs.EOF
      MsgBox rs!columnA & " " & rs!columnB
      rs.MoveNext
   Loop
    rs.Close
   Set db = Nothing
End Sub
```

VBA is useful for automating repeated actions in rows of a spreadsheet. For example, using the following code example, the built-in iterative solver Goal Seek is applied automatically to each row in a column array, avoiding repeated use of manual menu entry. Below a column variable "C_M" determines the values of another column variable "Target" in some nonlinear fashion. The built-in nonlinear solver Goal Seek is called to find the value of "C_M" that brings "Target" to value one. The subroutine is inserted into the workbook using the VBA editor and command Insert Module. It is called directly from the VBA editor, or by using a "hot key" or keyboard shortcut. Values on the spreadsheet automatically update as the rows are scanned.

It is useful to note that subroutines have the power to update variables on the spreadsheet; functions do not - they simply report their evaluation.

Line Option Explicit is not part of the subroutine: it sets a compiler option that forces identification of all variables that have not been specified in Dim statements, which avoids difficult to detect debugging problems caused by typos. Notation ( ' ) in the following code denotes a comment, and ( _) line continuation. The code uses NAMED variables: a form of cell reference in which cells are assigned names of user choice, rather than the standard cell designation denoting specific row and column numbers. Naming is done on the worksheet via the Excel "Name Manager", or menu Insert Name: Create.

```
Option Explicit
Sub SetTarget()
' SetTarget Macro
  Dim J As Integer
  Dim Size As Integer
' On the spreadsheet, array "C_M" is a NAMED column variable
'    Its members use a row index taken as J
' Built-in function COUNT determines size of array "C_M"
  Size = Range("C_M").Cells.Count
' Set initial value of all members of array
'   C_M to 1E-06; J = row index
     For J = 1 To Size
        Range("C_M").Cells(J) = 0.000001
     Next J
' "Target" is another NAMED array on the spreadsheet of
'     dimension "Size"; the same size as array "C_M"
' Each "Target" entry in each row depends in a
'     specified way upon the value of "C_M" in that row,
'     for example, by a function such as: Target = C_M*C_M
' GOAL SEEK is a built-in iterative solver in Excel
' Call GOAL SEEK to set each "Target" member to unity: for example,
'   taking J = row index, in row J the cell named "C_M" is changed
'   by GOAL SEEK until "Target" in row J is one
' Syntax (aside from "for-next" details) found with macro recorder;
'     underscore "_" is line continuation
     For J = 1 To Size
        Range("Target").Cells(J).GoalSeek Goal := 1, _
          ChangingCell := Range("C_M").Cells(J)
     Next J
End Sub
```

In the example below VBA is used to get an array from a cell range, manipulate the array, and then set the values back in a different range. This works many times faster than directly setting the cell values one-by-one.

```
Sub CalculateSquares(rinput As Range, routput As Range)
  'Variable specifications
   Dim values() As Variant
   Dim i As Integer, N As Integer
   'Count the rows to compute
   N = rinput.Rows.Count
   'Set values array from input range.
   'Expected shape of array is (1 to N, 1 to 1)
   values = rinput.Value
   'Iterate through rows and set values
   For i = 1 To N
      values(i, 1) = values(i, 1) ^ 2
   Next i
   'Export values back into the spreadsheet by setting the value property of the output range.
   routput.Value = values
End Sub
```

VBA can be used to create a user defined function for use in a Microsoft Excel workbook:

```
Public Function BusinessDayPrior(dt As Date) As Date
   Select Case Weekday(dt, vbMonday)
      Case 1
         BusinessDayPrior = dt - 3      'Monday becomes Friday
      Case 7
         BusinessDayPrior = dt - 2      'Sunday becomes Friday
      Case Else
         BusinessDayPrior = dt - 1      'All other days become previous day
   End Select
End Function
```

Example of how to add an external application object (the user must have the application library referenced in the application before this):

```
Public Sub Example()
   Dim XLApp As Excel.Application
   Dim WDApp As Word.Application
   Set XLApp = CreateObject("Excel.Application")
   Set WDApp = CreateObject("Word.Application")
   ' ...your code here...
   XLApp.Quit
   WDApp.Quit
   Set XLApp = Nothing
   Set WDApp = Nothing
End Sub
```

## 2.2 Fast Fourier Transform

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. "The FFT has been called the most important numerical algorithm of our lifetime (Strang, 1994)." (Kent & Read 2002, 61) There are many distinct FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory; this article gives an overview of the available techniques and some of their general properties, while the specific algorithms are described in subsidiary articles linked below.

A DFT decomposes a sequence of values into components of different frequencies. This operation is useful in many fields (see discrete Fourier transform for properties and applications of the transform) but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing a DFT of N points in the naive way, using the definition, takes O(N2) arithmetical operations, while an FFT can compute the same result in only O(N log N) operations. The difference in speed can be substantial, especially for long data sets where N may be in the thousands or millions—in practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly

proportional to N / log(N). This huge improvement made many DFT-based algorithms practical; FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The most well known FFT algorithms depend upon the factorization of N, but (contrary to popular misconception) there are FFTs with O(N log N) complexity for all N, even for prime N. Many FFT algorithms only depend on the fact that $e^{-\frac{2\pi i}{N}}$ is an Nth primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms.

Since the inverse DFT is the same as the DFT, but with the opposite sign in the exponent and a 1/N factor, any FFT algorithm can easily be adapted for it.

**Definition and speed**

An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the only difference is that an FFT is much faster. (In the presence of round-off error, many FFT algorithms are also much more accurate than evaluating the DFT definition directly, as discussed below.)

Let x0, ...., xN-1 be complex numbers. The DFT is defined by the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1.$$

Evaluating this definition directly requires $O(N^2)$ operations: there are N outputs $X_k$, and each output requires a sum of N terms. An FFT is any method to compute the same results in O(N log N) operations. More precisely, all known FFT algorithms require O(N log N) operations (technically, O only denotes an upper bound), although there is no known proof that better complexity is impossible.

To illustrate the savings of an FFT, consider the count of complex multiplications and additions. Evaluating the DFT's sums directly involves N2 complex multiplications and N(N-1) complex additions [of which O(N) operations can be saved by eliminating trivial operations such as multiplications by 1]. The well-known radix-2 Cooley–Tukey algorithm, for N a power of 2, can compute the same result with only $(N/2)\log_2 N$ complex multiplies (again, ignoring simplifications of multiplications by 1 and similar) and N log2N complex additions. In practice,

actual performance on modern computers is usually dominated by factors other than arithmetic and is a complicated subject (see, e.g., Frigo & Johnson, 2005), but the overall improvement from O(N2) to O(N log N) remains.

## Algorithms

### Cooley–Tukey algorithm

By far the most common FFT is the Cooley–Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1N_2$ into many smaller DFTs of sizes $N_1$ and $N_2$, along with O(N) multiplications by complex roots of unity traditionally called twiddle factors (after Gentleman and Sande, 1966).

This method (and the general idea of an FFT) was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965, but it was later discovered (Heideman & Burrus, 1984) that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805 (and subsequently rediscovered several times in limited forms).

The most well-known use of the Cooley–Tukey algorithm is to divide the transform into two pieces of size N / 2 at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey). These are called the radix-2 and mixed-radix cases, respectively (and other variants such as the split-radix FFT have their own names as well). Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion. Also, because the Cooley–Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT, such as those described below.

### Other FFT algorithms

There are other FFT algorithms distinct from Cooley–Tukey. For $N = N_1N_2$ with coprime $N_1$ and $N_2$, one can use the Prime-Factor (Good-Thomas) algorithm (PFA), based on the Chinese Remainder Theorem, to factorize the DFT similarly to Cooley–Tukey but without the twiddle factors. The Rader-Brenner algorithm (1976) is a Cooley–Tukey-like factorization but with purely imaginary twiddle factors, reducing multiplications at the cost of increased additions and reduced numerical stability; it was later superseded by the split-radix variant of Cooley–Tukey (which achieves the same multiplication count but with fewer additions and without sacrificing accuracy).

Algorithms that recursively factorize the DFT into smaller operations other than DFTs include the Bruun and QFT algorithms. (The Rader-Brenner and QFT algorithms were proposed for power-of-two sizes, but it is possible that they could be adapted to general composite n. Bruun's algorithm applies to arbitrary even composite sizes.) Bruun's algorithm, in particular, is based on interpreting the FFT as a recursive factorization of the polynomial $z^{N-1}$, here into real-coefficient polynomials of the form $z^{M-1}$ and $z^{2M} + az^{M+1}$.

Another polynomial viewpoint is exploited by the Winograd algorithm, which factorizes $z^{N-1}$ into cyclotomic polynomials—these often have coefficients of 1, 0, or -1, and therefore require few (if any) multiplications, so Winograd can be used to obtain minimal-multiplication FFTs and is often used to find efficient algorithms for small factors. Indeed, Winograd showed that the DFT can be computed with only O(N) irrational multiplications, leading to a proven achievable lower bound on the number of multiplications for power-of-two sizes; unfortunately, this comes at the cost of many more additions, a tradeoff no longer favorable on modern processors with hardware multipliers. In particular, Winograd also makes use of the PFA as well as an algorithm by Rader for FFTs of prime sizes.

Rader's algorithm, exploiting the existence of a generator for the multiplicative group modulo prime N, expresses a DFT of prime size n as a cyclic convolution of (composite) size N-1, which can then be computed by a pair of ordinary FFTs via the convolution theorem (although Winograd uses other convolution methods). Another prime-size FFT is due to L. I. Bluestein, and is sometimes called the chirp-z algorithm; it also re-expresses a DFT as a convolution, but this time of the same size (which can be zero-padded to a power of two and evaluated by radix-2 Cooley–Tukey FFTs, for example).

**FFT algorithms specialized for real and/or symmetric data**

In many applications, the input data for the DFT are purely real, in which case the outputs satisfy the symmetry

$$X_{N-k} = X_k^*,$$

and efficient FFT algorithms have been designed for this situation (see e.g. Sorensen, 1987). One approach consists of taking an ordinary algorithm (e.g. Cooley–Tukey) and removing the redundant parts of the computation, saving roughly a factor of two in time and memory. Alternatively, it is possible to express an even-length real-input DFT as a complex DFT of half

the length (whose real and imaginary parts are the even/odd elements of the original real data), followed by O(N) post-processing operations.

It was once believed that real-input DFTs could be more efficiently computed by means of the discrete Hartley transform (DHT), but it was subsequently argued that a specialized real-input DFT algorithm (FFT) can typically be found that requires fewer operations than the corresponding DHT algorithm (FHT) for the same number of inputs. Bruun's algorithm (above) is another method that was initially proposed to take advantage of real inputs, but it has not proved popular.

There are further FFT specializations for the cases of real data that have even/odd symmetry, in which case one can gain another factor of (roughly) two in time and memory and the DFT becomes the discrete cosine/sine transform(s) (DCT/DST). Instead of directly modifying an FFT algorithm for these cases, DCTs/DSTs can also be computed via FFTs of real data combined with O(N) pre/post processing.

**Computational issues**

A fundamental question of longstanding theoretical interest is to prove lower bounds on the complexity and exact operation counts of fast Fourier transforms, and many open problems remain. It is not even rigorously proved whether DFTs truly require O(NlogN) (i.e., order NlogN or greater) operations, even for the simple case of power of two sizes, although no algorithms with lower complexity are known. In particular, the count of arithmetic operations is usually the focus of such questions, although actual performance on modern-day computers is determined by many other factors such as cache or CPU pipeline optimization.

Following pioneering work by Winograd (1978), a tight O(N) lower bound is known for the number of real multiplications required by an FFT. It can be shown that only $4N - 2\log_2^2 N - 2\log_2 N - 4$ irrational real multiplications are required to compute a DFT of power-of-two length N = 2m. Moreover, explicit algorithms that achieve this count are known (Heideman & Burrus, 1986; Duhamel, 1990). Unfortunately, these algorithms require too many additions to be practical, at least on modern computers with hardware multipliers.

A tight lower bound is not known on the number of required additions, although lower bounds have been proved under some restrictive assumptions on the algorithms. In 1973, Morgenstern proved an O(NlogN) lower bound on the addition count for algorithms where the multiplicative constants have bounded magnitudes (which is true for most but not all FFT algorithms). Pan (1986) proved an O(NlogN) lower bound assuming a bound on a measure of the FFT algorithm's "asynchronicity", but the generality of this assumption is unclear. For the case of power-of-two N, Papadimitriou (1979) argued that the number $N\log_2 N$ of complex-number additions achieved by Cooley–Tukey algorithms is optimal under certain assumptions on the graph of the algorithm (his assumptions imply, among other things, that no additive identities in the roots of unity are exploited). (This argument would imply that at least $2N\log_2 N$ real additions are required, although this is not a tight bound because extra additions are required as part of complex-number multiplications.) Thus far, no published FFT algorithm has achieved fewer than $N\log_2 N$ complex-number additions (or their equivalent) for power-of-two N.

A third problem is to minimize the total number of real multiplications and additions, sometimes called the "arithmetic complexity" (although in this context it is the exact count and not the asymptotic complexity that is being considered). Again, no tight lower bound has been proven. Since 1968, however, the lowest published count for power-of-two N was long achieved by the split-radix FFT algorithm, which requires 4Nlog2N - 6N + 8 real multiplications and additions for N > 1. This was recently reduced to $\sim \frac{34}{9} N \log_2 N$ (Johnson and Frigo, 2007; Lundy and Van Buskirk, 2007).

Most of the attempts to lower or prove the complexity of FFT algorithms have focused on the ordinary complex-data case, because it is the simplest. However, complex-data FFTs are so closely related to algorithms for related problems such as real-data FFTs, discrete cosine transforms, discrete Hartley transforms, and so on, that any improvement in one of these would immediately lead to improvements in the others (Duhamel & Vetterli, 1990).

**Accuracy and approximations**

All of the FFT algorithms discussed below compute the DFT exactly (in exact arithmetic, i.e. neglecting floating-point errors). A few "FFT" algorithms have been proposed, however, that compute the DFT approximately, with an error that can be made arbitrarily small at the expense

of increased computations. Such algorithms trade the approximation error for increased speed or other properties. For example, an approximate FFT algorithm by Edelman et al. (1999) achieves lower communication requirements for parallel computing with the help of a fast multipole method. A wavelet-based approximate FFT by Guo and Burrus (1996) takes sparse inputs/outputs (time/frequency localization) into account more efficiently than is possible with an exact FFT. Another algorithm for approximate computation of a subset of the DFT outputs is due to Shentov et al. (1995). Only the Edelman algorithm works equally well for sparse and non-sparse data, however, since it is based on the compressibility (rank deficiency) of the Fourier matrix itself rather than the compressibility (sparsity) of the data.

In fixed-point arithmetic, the finite-precision errors accumulated by FFT algorithms are worse, with rms errors growing as O( N) for the Cooley–Tukey algorithm (Welch, 1969). Moreover, even achieving this accuracy requires careful attention to scaling in order to minimize the loss of precision, and fixed-point FFT algorithms involve rescaling at each intermediate stage of decompositions like Cooley–Tukey.

To verify the correctness of an FFT implementation, rigorous guarantees can be obtained in O(N log N) time by a simple procedure checking the linearity, impulse-response, and time-shift properties of the transform on random inputs (Ergün, 1995).

**Multidimensional FFTs**

As defined in the multidimensional DFT article, the multidimensional DFT

$$X_{\mathbf{k}} = \sum_{\mathbf{n}=0}^{\mathbf{N}-1} e^{-2\pi i \mathbf{k}\cdot(\mathbf{n}/\mathbf{N})} x_{\mathbf{n}}$$

transforms an array $x_{\mathbf{n}}$ with a d-dimensional vector of indices $\mathbf{n} = (n_1, n_2, \ldots, n_d)$ by a set of d nested summations (over $n_j = 0 \ldots N_j - 1$ for each j), where the division $\mathbf{n}/\mathbf{N}$, defined as $\mathbf{n}/\mathbf{N} = (n_1/N_1, \ldots, n_d/N_d)$, is performed element-wise. Equivalently, it is simply the composition of a sequence of d sets of one-dimensional DFTs, performed along one dimension at a time (in any order).

This compositional viewpoint immediately provides the simplest and most common multidimensional DFT algorithm, known as the row-column algorithm (after the two-dimensional case, below). That is, one simply performs a sequence of d one-dimensional FFTs (by any of the above algorithms): first you transform along the n1 dimension, then along the n2 dimension, and so on (or actually, any ordering will work). This method is easily shown to have the usual O(NlogN) complexity, where $N = N_1 N_2 \cdots N_d$ is the total number of data points transformed. In particular, there are N / $N_1$ transforms of size $N_1$, etcetera, so the complexity of the sequence of FFTs is:

$$\begin{aligned} &\frac{N}{N_1} O(N_1 \log N_1) + \cdots + \frac{N}{N_d} O(N_d \log N_d) \\ &= O\left(N \left[\log N_1 + \cdots + \log N_d\right]\right) = O(N \log N). \end{aligned}$$

In two dimensions, the $x_{\mathbf{k}}$ can be viewed as an $n_1 \times n_2$ matrix, and this algorithm corresponds to first performing the FFT of all the rows and then of all the columns (or vice versa), hence the name.

In more than two dimensions, it is often advantageous for cache locality to group the dimensions recursively. For example, a three-dimensional FFT might first perform two-dimensional FFTs of each planar "slice" for each fixed n1, and then perform the one-dimensional FFTs along the n1 direction. More generally, an asymptotically optimal cache-oblivious

algorithm consists of recursively dividing the dimensions into two groups $(n_1, \ldots, n_{d/2})$ and $(n_{d/2+1}, \ldots, n_d)$ that are transformed recursively (rounding if d is not even) (see Frigo and Johnson, 2005). Still, this remains a straightforward variation of the row-column algorithm that ultimately requires only a one-dimensional FFT algorithm as the base case, and still has O(NlogN) complexity. Yet another variation is to perform matrix transpositions in between transforming subsequent dimensions, so that the transforms operate on contiguous data; this is especially important for out-of-core and distributed memory situations where accessing non-contiguous data is extremely time-consuming.

There are other multidimensional FFT algorithms that are distinct from the row-column algorithm, although all of them have O(NlogN) complexity. Perhaps the simplest non-row-column FFT is the vector-radix FFT algorithm, which is a generalization of the ordinary Cooley–Tukey algorithm where one divides the transform dimensions by a vector $\mathbf{r} = (r_1, r_2, \ldots, r_d)$ of radices at each step. (This may also have cache benefits.) The simplest case of vector-radix is where all of the radices are equal (e.g. vector-radix-2 divides all of the dimensions by two), but this is not necessary. Vector radix with only a single non-unit radix at a time, i.e. $\mathbf{r} = (1, \ldots, 1, r, 1, \ldots, 1)$, is essentially a row-column algorithm. Other, more complicated, methods include polynomial transform algorithms due to Nussbaumer (1977), which view the transform in terms of convolutions and polynomial products. See Duhamel and Vetterli (1990) for more information and references.

## 2.3 Phase-shift keying

Phase-shift keying (PSK) is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal (the carrier wave).

Any digital modulation scheme uses a finite number of distinct signals to represent digital data. PSK uses a finite number of phases, each assigned a unique pattern of binary digits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. This requires the receiver to be able to

compare the phase of the received signal to a reference signal — such a system is termed coherent (and referred to as CPSK).

Alternatively, instead of using the bit patterns to set the phase of the wave, it can instead be used to change it by a specified amount. The demodulator then determines the changes in the phase of the received signal rather than the phase itself. Since this scheme depends on the difference between successive phases, it is termed differential phase-shift keying (DPSK). DPSK can be significantly simpler to implement than ordinary PSK since there is no need for the demodulator to have a copy of the reference signal to determine the exact phase of the received signal (it is a non-coherent scheme). In exchange, it produces more erroneous demodulations. The exact requirements of the particular scenario under consideration determine which scheme is used.

### 2.3.1 Binary Phase-shift keying

BPSK (also sometimes called PRK, Phase Reversal Keying, or 2PSK) is the simplest form of phase shift keying (PSK). It uses two phases which are separated by 180° and so can also be termed 2-PSK. It does not particularly matter exactly where the constellation points are positioned, and in this figure they are shown on the real axis, at 0° and 180°. This modulation is the most robust of all the PSKs since it takes the highest level of noise or distortion to make the demodulator reach an incorrect decision. It is, however, only able to modulate at 1 bit/symbol (as seen in the figure) and so is unsuitable for high data-rate applications when bandwidth is limited.

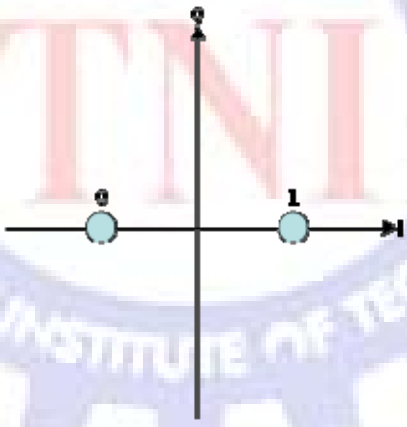



**Figure 2.4** BPSK Phases

In the presence of an arbitrary phase-shift introduced by the communications channel, the demodulator is unable to tell which constellation point is which. As a result, the data is often differentially encoded prior to modulation.

**Implementation**

The general form for BPSK follows the equation:

$$s_n(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi(1-n)), n = 0, 1.$$

This yields two phases, 0 and 1. In the specific form, binary data is often conveyed with the following signals:

$$s_0(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t + \pi) = -\sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t)$$ for binary "0"

$$s_1(t) = \sqrt{\frac{2E_b}{T_b}} \cos(2\pi f_c t)$$ for binary "1"

where $f_c$ is the frequency of the carrier-wave.

Hence, the signal-space can be represented by the single basis function

$$\phi(t) = \sqrt{\frac{2}{T_b}} \cos(2\pi f_c t)$$

where 1 is represented by $\sqrt{E_b}\phi(t)$ and 0 is represented by $-\sqrt{E_b}\phi(t)$. This assignment is, of course, arbitrary.

This use of this basis function is shown at the end of the next section in a signal timing diagram. The topmost signal is a BPSK-modulated cosine wave that the BPSK modulator would produce. The bit-stream that causes this output is shown above the signal (the other parts of this figure are relevant only to QPSK).

**Bit error rate**

The bit error rate (BER) of BPSK in AWGN can be calculated as :

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \quad \text{or} \quad P_b = \frac{1}{2}\operatorname{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right)$$

Since there is only one bit per symbol, this is also the symbol error rate.

### 2.3.2 Quadrature Phase-shift keying

Sometimes this is known as quaternary PSK, quadriphase PSK, 4-PSK, or 4-QAM. (Although the root concepts of QPSK and 4-QAM are different, the resulting modulated radio waves are exactly the same.) QPSK uses four points on the constellation diagram, equispaced around a circle. With four phases, QPSK can encode two bits per symbol, shown in the diagram with gray coding to minimize the bit error rate (BER) — sometimes misperceived as twice the BER of BPSK.

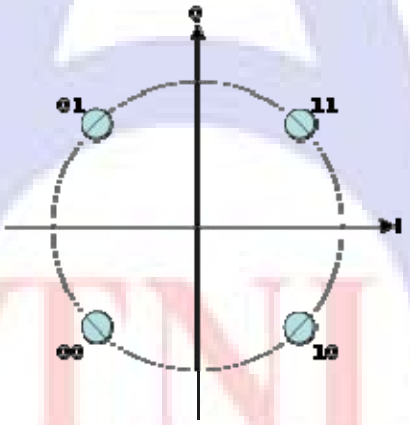



**Figure 2.5** QPSK Phases

The mathematical analysis shows that QPSK can be used either to double the data rate compared with a BPSK system while maintaining the same bandwidth of the signal, or to maintain the data-rate of BPSK but halving the bandwidth needed. In this latter case, the BER of QPSK is exactly the same as the BER of BPSK - and deciding differently is a common confusion when considering or describing QPSK.

Given that radio communication channels are allocated by agencies such as the Federal Communication Commission giving a prescribed (maximum) bandwidth, the advantage of QPSK over BPSK becomes evident: QPSK transmits twice the data rate in a given bandwidth compared to BPSK - at the same BER. The engineering penalty that is paid is that QPSK transmitters and receivers are more complicated than the ones for BPSK. However, with modern electronics technology, the penalty in cost is very moderate.

As with BPSK, there are phase ambiguity problems at the receiving end, and differentially encoded QPSK is often used in practice.

**Implementation**

The implementation of QPSK is more general than that of BPSK and also indicates the implementation of higher-order PSK. Writing the symbols in the constellation diagram in terms of the sine and cosine waves used to transmit them:

$$s_n(t) = \sqrt{\frac{2E_s}{T_s}} \cos\left(2\pi f_c t + (2n-1)\frac{\pi}{4}\right), \quad n = 1,2,3,4.$$

This yields the four phases $\pi/4$, 3 $\pi$ /4, 5 $\pi$ /4 and 7 $\pi$ /4 as needed.

This results in a two-dimensional signal space with unit basis functions

$$\phi_1(t) = \sqrt{\frac{2}{T_s}} \cos(2\pi f_c t)$$

$$\phi_2(t) = \sqrt{\frac{2}{T_s}} \sin(2\pi f_c t)$$

The first basis function is used as the in-phase component of the signal and the second as the quadrature component of the signal.

Hence, the signal constellation consists of the signal-space 4 points

$$\left(\pm\sqrt{E_s/2}, \pm\sqrt{E_s/2}\right).$$

The factors of 1/2 indicate that the total power is split equally between the two carriers.

Comparing these basis functions with that for BPSK shows clearly how QPSK can be viewed as two independent BPSK signals. Note that the signal-space points for BPSK do not need to split the symbol (bit) energy over the two carriers in the scheme shown in the BPSK constellation diagram.

QPSK systems can be implemented in a number of ways. An illustration of the major components of the transmitter and receiver structure are shown below.

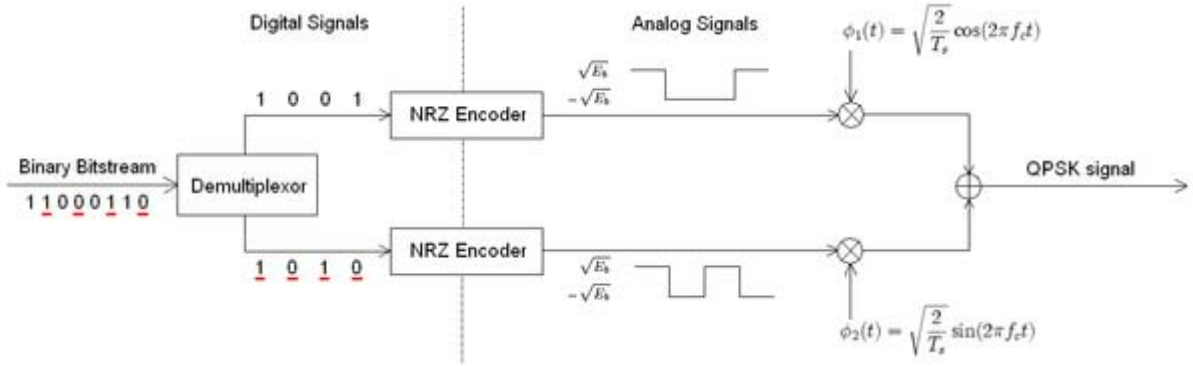



**Figure 2.6** QPSK systems 1

Conceptual transmitter structure for QPSK. The binary data stream is split into the in-phase and quadrature-phase components. These are then separately modulated onto two orthogonal basis functions. In this implementation, two sinusoids are used. Afterwards, the two signals are superimposed, and the resulting signal is the QPSK signal. Note the use of polar non-return-to-zero encoding. These encoders can be placed before for binary data source, but have been placed after to illustrate the conceptual difference between digital and analog signals involved with digital modulation.

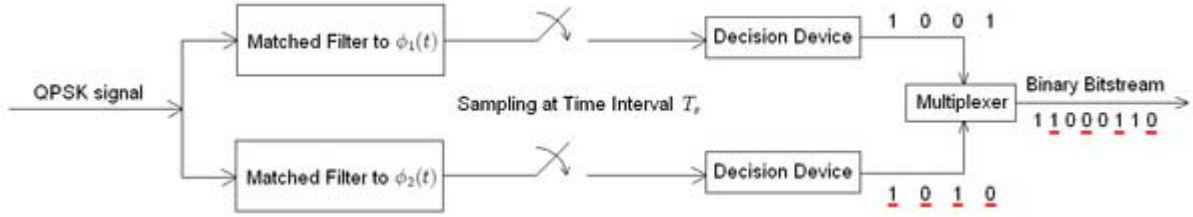



**Figure 2.7** QPSK systems 2

Receiver structure for QPSK. The matched filters can be replaced with correlators. Each detection device uses a reference threshold value to determine whether a 1 or 0 is detected.

**Bit error rate**

Although QPSK can be viewed as a quaternary modulation, it is easier to see it as two independently modulated quadrature carriers. With this interpretation, the even (or odd) bits are used to modulate the in-phase component of the carrier, while the odd (or even) bits are used to modulate the quadrature-phase component of the carrier. BPSK is used on both carriers and they can be independently demodulated.

As a result, the probability of bit-error for QPSK is the same as for BPSK:

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right).$$

However, in order to achieve the same bit-error probability as BPSK, QPSK uses twice the power (since two bits are transmitted simultaneously).

The symbol error rate is given by:

$$\begin{aligned} P_s &= 1 - (1 - P_b)^2 \\ &= 2Q\left(\sqrt{\frac{E_s}{N_0}}\right) - Q^2\left(\sqrt{\frac{E_s}{N_0}}\right)^2 \end{aligned}$$

If the signal-to-noise ratio is high (as is necessary for practical QPSK systems) the probability of symbol error may be approximated:

$$P_s \approx 2Q\left(\sqrt{\frac{E_s}{N_0}}\right)$$

# Chapter 3

# Methodology

## 3.1 Plan of the project

**Table 3.1** plan for project

| Detail | Month 1 | | | | Month 2 | | | |
|---|---|---|---|---|---|---|---|---|
| Introduce to work in workgroup and seeking for project | | | | | | | | |
| | | | | | | | | |
| Learning about theorem and technology in project | | | | | | | | |
| | | | | | | | | |
| Doing and testing project | | | | | | | | |
| | | | | | | | | |

Blue color represents expected time

Red color represents usage time

## 3.2 Work details

This project aims to calculate theoretical 100Gbps-QPSK signal and plot spectrum graph which needed for design ROADM optical channel by using Visual Basic for Application (VBA) language in Microsoft Excel.

## 3.3 Project Workflow

### 3.3.1 Learning the basics of Visual Basic for Application in Excel

First, we learn how to work with cells in Microsoft Excel by trying to compile some simples codes.

### 3.3.2 Learning about Binary Phase-shift keying (BPSK)

In this step, we use BPSK equation on coding to implement result and see graph from BPSK result to make sure the code is really work. Then put the result of BPSK into array of real part and array of imaginary part and use them for executing result from Fourier transform. Plot graph from the result of Fourier transform to make sure it work.

### 3.3.3 Learning about Quadrature Phase-shift keying (QPSK)

After using BPSK for calculation, now we use QPSK equation on coding to implement result and see graph from QPSK result to make sure the code is really work. Then put the result of QPSK into array of real part and array of imaginary part and use them for executing result from Fourier transform. Plot graph from the result of Fourier transform to make sure it work.

### 3.3.4 Use real data frequency

In this phase, we use the real baud rate and try to make spectrum graph as ideal as possible by testing in many condition.

# Chapter 4

# Results and Discussions

## 4.1 Workflow Conclusions

During the internship period at the organization for 1 month 15 days the workflow conclusions are

- Introduce to work in workgroup and seek for the project.
- Learning about Visual Basic for Application in Microsoft Excel , Binary Phase-Shift Keying, Quadrature Phase-Shift Keying.
- Trying to implement result from Fourier transform and draw a graph in many condition start from Binary Phase-Shift Keying by use unreal frequency and unreal baud rate then use Quadrature Phase-Shift Keying.
- Trying to use real baud rate in Quadrature Phase-Shift Keying and try to make a graph as ideal as possible.

## 4.2 Workflow Discussions

### 4.2.1 Use Binary Phase-shift keying (BPSK ) equation to modulate signal

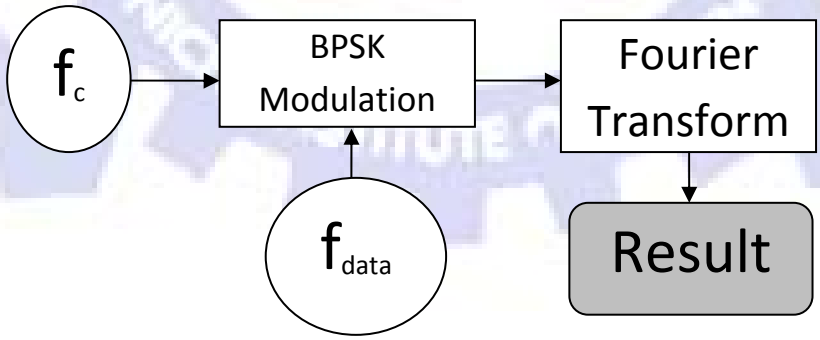



**Figure 4.1** Instruction of calculation (for BPSK)

The goal of this project is QPSK used for modulating but firstly we start with BPSK because it easier than QPSK. After trying Visual Basic for Application by coding some simple programs, BPSK modulation method is used to convert digital input signal to analog signal. Then the BPSK graph is drawn by Microsoft Excel for make sure the program is really work

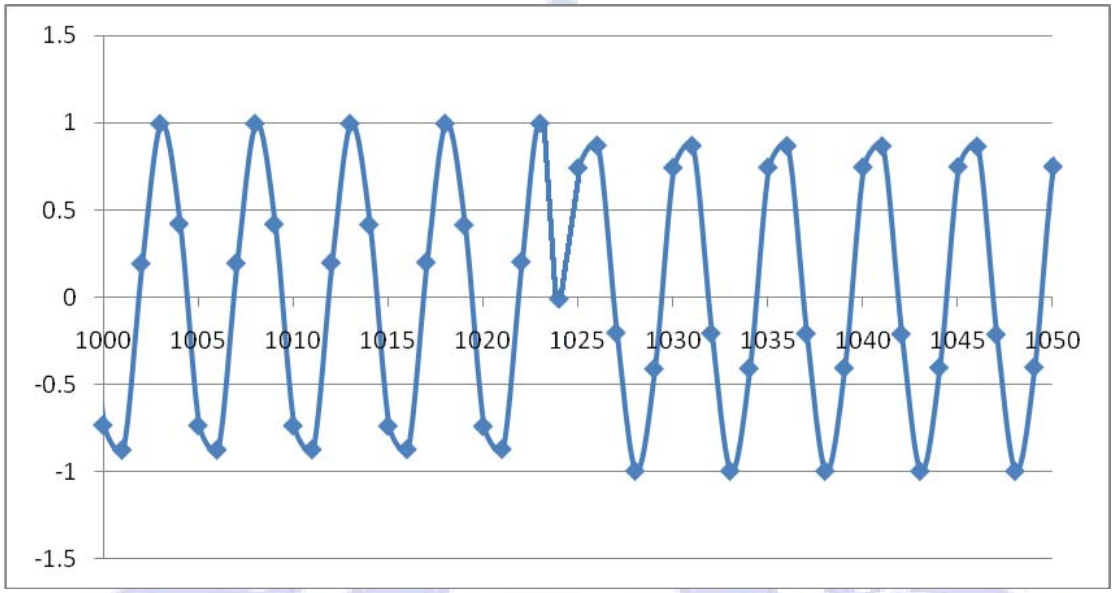



**Figure 4.2** BPSK result

From Figure 4.2, the carrier frequency ($f_c$) is set to 1 MHz, the sampling frequency ($f_s$) is set to 5 MHz ($5f_c$) and the baud rate is set to 4883 bps (datarate=1024) which let program random new bit (for BPSK is 0 or 1) when n pass 1024

The equation between sampling frequency ($f_s$), baud rate and data frequency ($f_d$) is

$$baud\ rate = f_s/f_d$$

After that, the result of modulation is put into Fourier transform which implemented by Fast Fourier Transform Algorithm. The result can be plotted in graph like Figure 4.3.

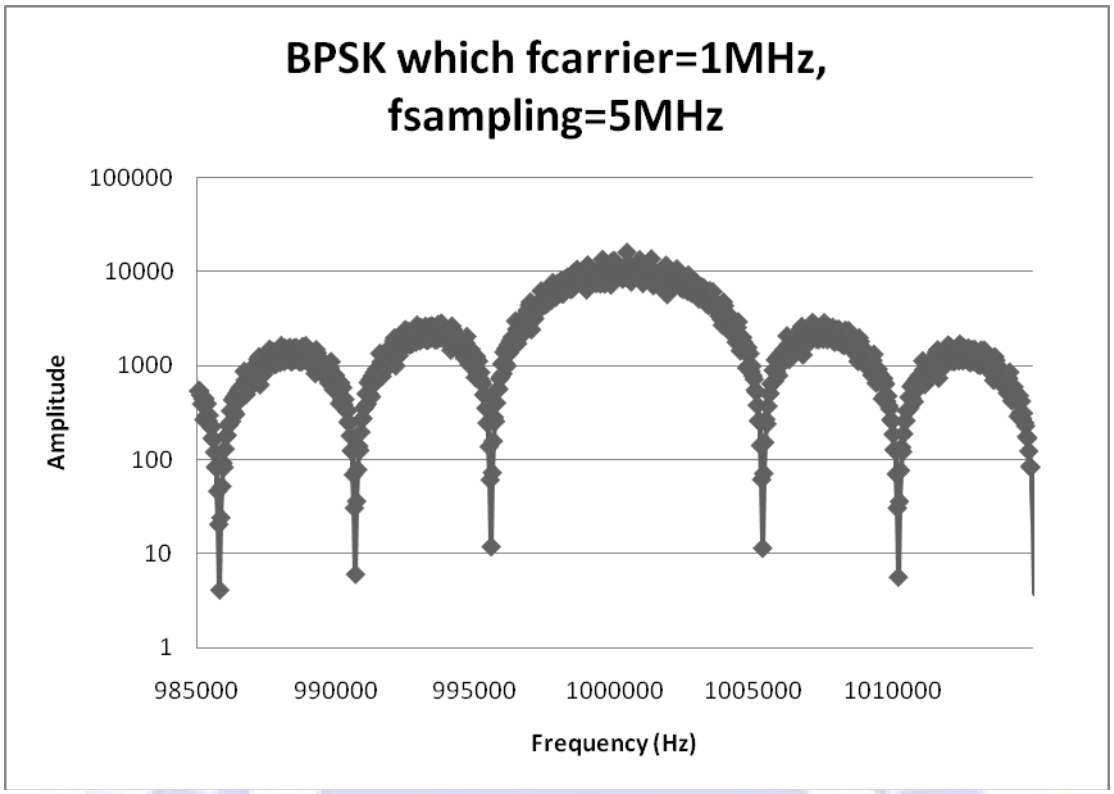



**Figure 4.3** Result from fourier transform (BPSK)

Because carrier frequency (fc) is 1 MHz so the peak of graph will be about 1 MHz in X-Axis.

If baud rate is decreased to 2441.5 bps (by increase datarate to 2048) the result frequency will decrease half like orange line in Figure 4.4

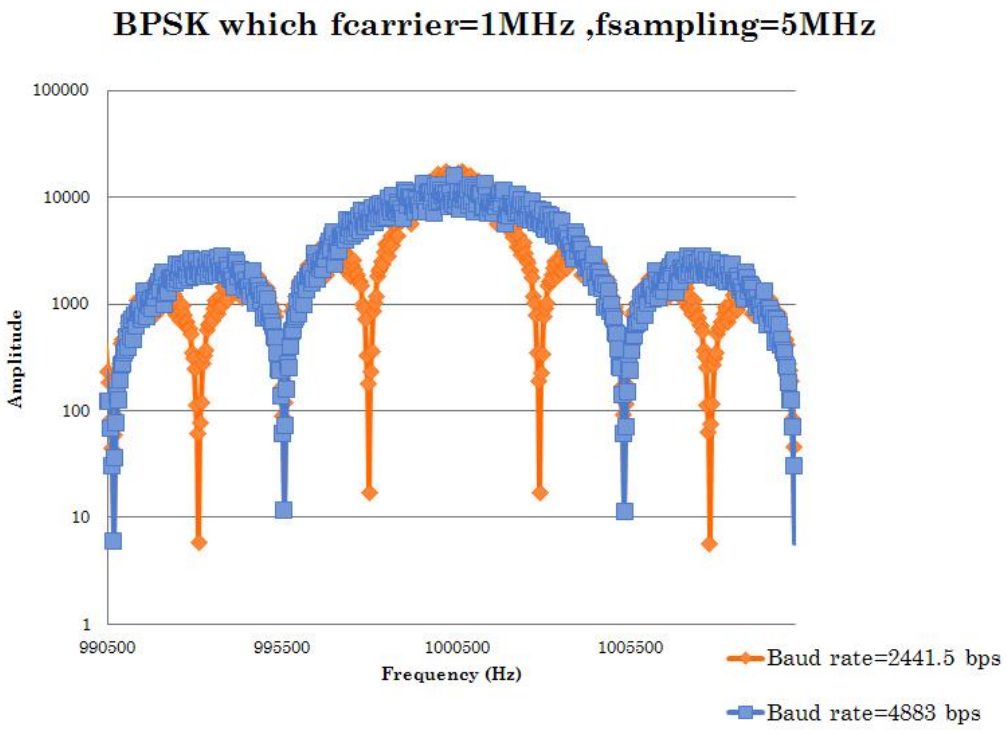



**Figure 4.4** Compare 2 Baud rate (BSPK)

So the Figure 4.4 show that bandwidth of graph equal to baud rate.

### 4.2.2 Use Quadrature Phase-shift keying (QPSK) equation to modulate signal

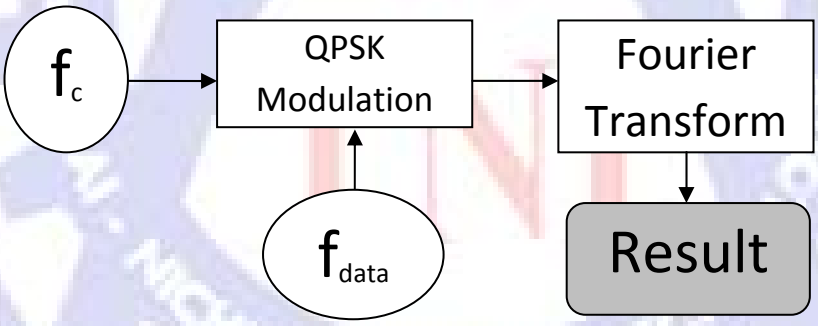



**Figure 4.5** Instruction of calculation (for QPSK)

The next step is using QPSK equation for modulating data signal by change data random from 0 and 1 to be 1 to 4 instead.

The result of QPSK can be drawn like Figure 4.6. (Phase in QPSK will be 0, $\pi$ /2, $\pi$ and 3$\pi$ /2 when phase in BPSK will be only 0 and $\pi$.)

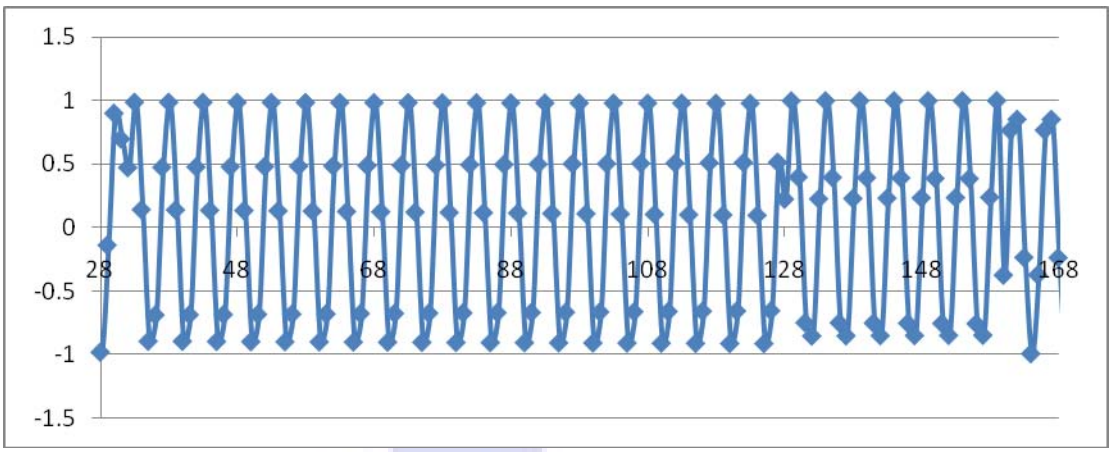



**Figure 4.6** QPSK result

The results of QPSK are calculated by Fourier transform when the conditions is similar to BPSK conditions. The result from Fourier transform was drawn like Figure 4.7

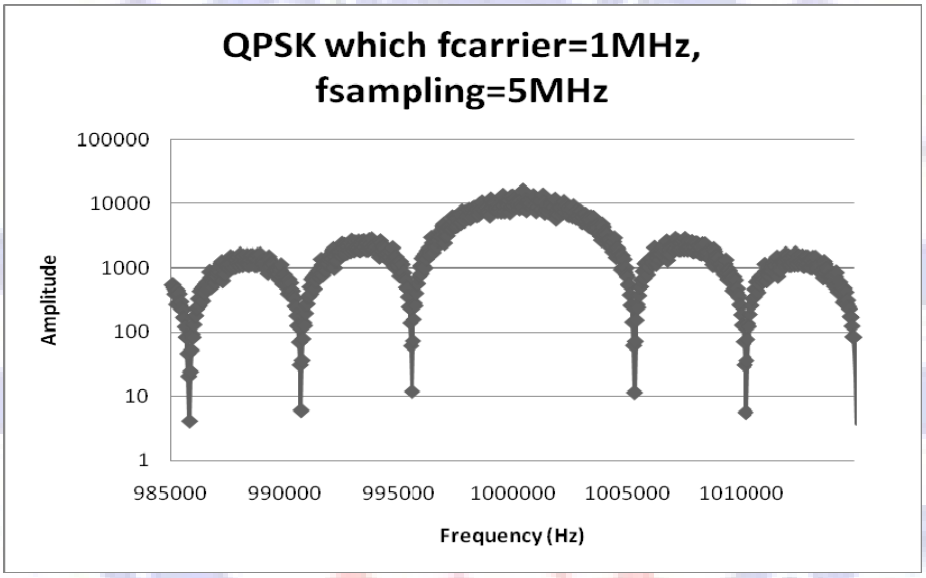



**Figure 4.7** Result from fourier transform (QPSK)

If baud rate is decreased to 2441.5 bps (by increase datarate to 2048) the result frequency will decrease half like orange line in Figure 4.8

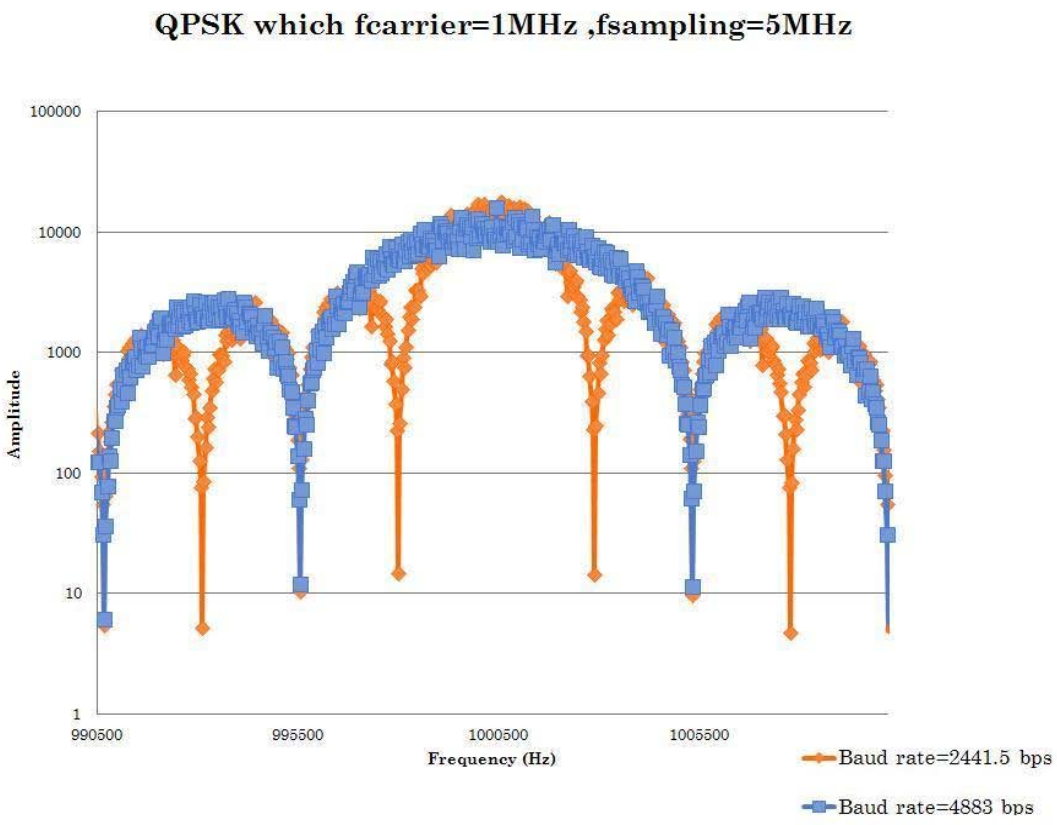



**Figure 4.8** Compare 2 Baud rate (QPSK)

### 4.2.3 Use real baud rate in QPSK

In this step, we implement the result from QPSK modulation and Fourier transform by use real signal frequency and real baud rate.

So the carrier frequency is set to 196 THz, baud rate is set to 25 Gbps when n is set to131072 or $2^{17}$.

Because n is too small, the graph is look like Figure 4.9.

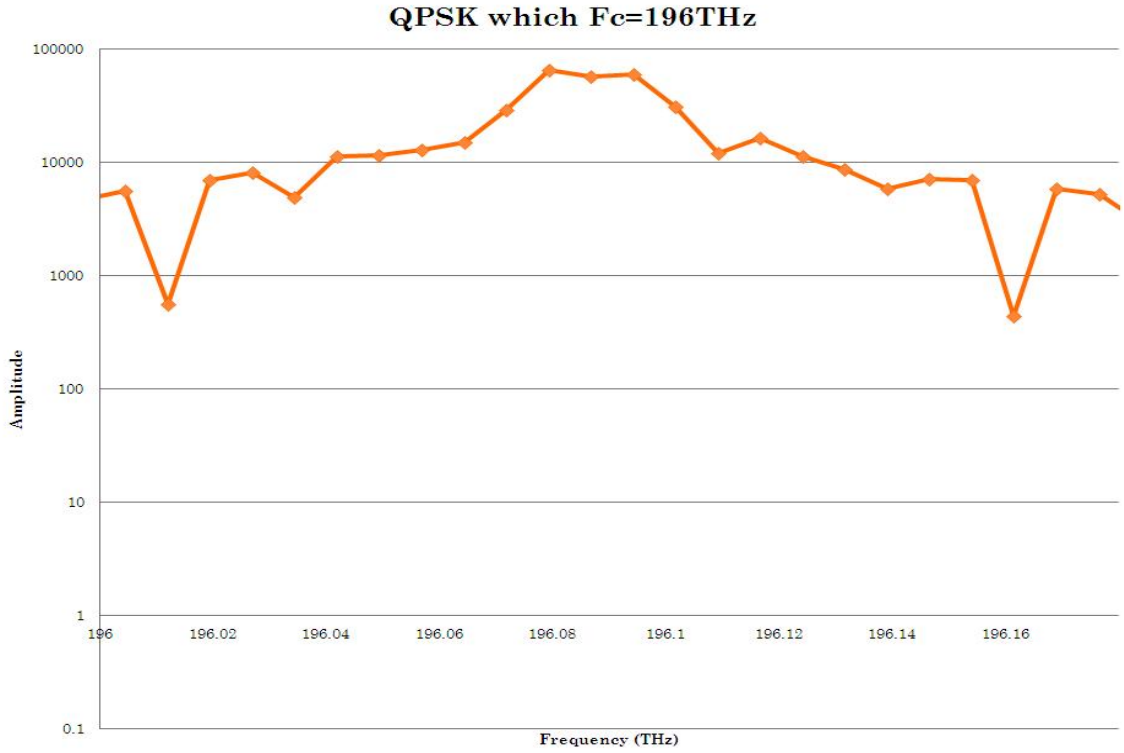



**Figure 4.9** real signal used 1

Now in this case, baud rate cannot be changed because it is the wanted real frequency and if n is bigger than 131072, the strange graph will be drawn because of Microsoft Excel limitation.

When the carrier frequency is reduced to 19.6 THz, the graph in Figure 4.10 which look much better than Figure 4.9 is plotted.

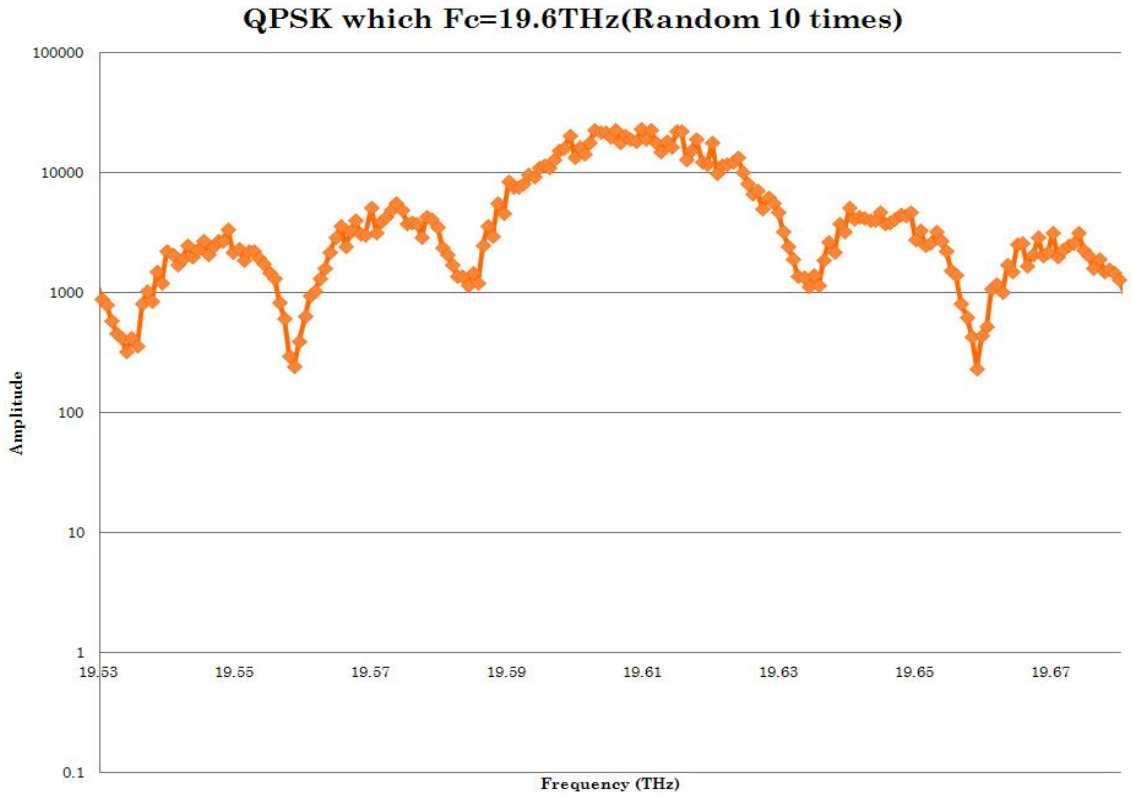



**Figure 4.10** real signal used 2

If the result from Figure 4.9 and Figure 4.10 have the same bandwidth, the carrier frequency may be set to 19.6 THz instead of 196 THz. The comparison of two frequencies in Figure 4.11 is created for investigating.

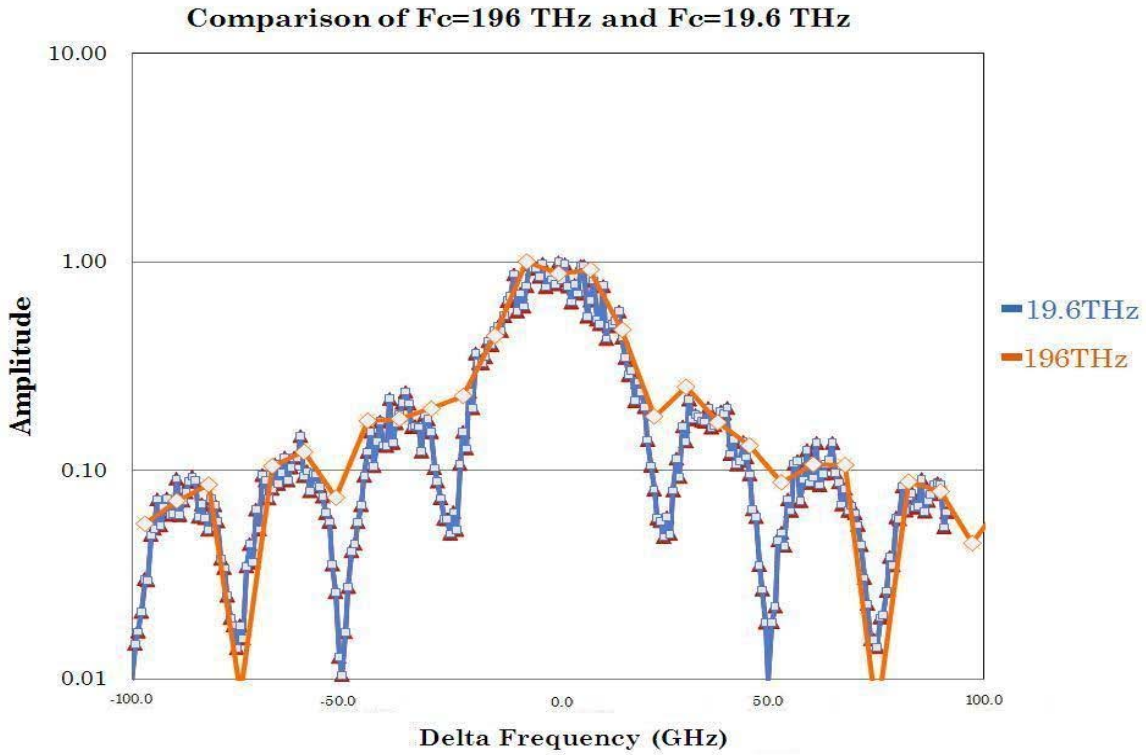



**Figure 4.11** real signal used 3

Figure 4.11 shows that carrier frequency 19.6 THz can be used instead of 196 THz because they almost to be the same bandwidth. So, carrier frequency 19.6THz is chosen to use instead of 196THz and the Figure 4.12 is the final result which use the average of 20 times calculated spectrum results .

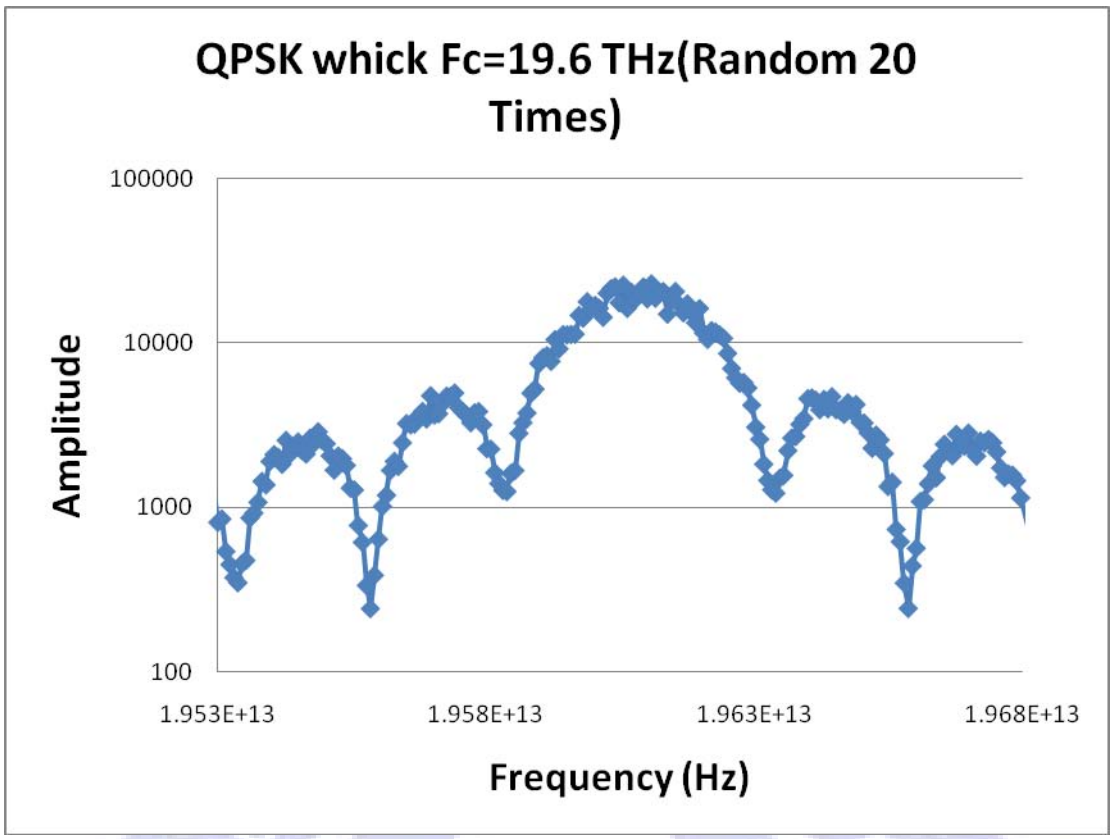



**Figure 4.12** real signal used 4

Now the spectrum values from real baud rate in Visual Basic for Application in Microsoft Excel is able to calculate under this condition

Modulation method is Quadrature Phase-Shift Keying

Data sampling number (n) = 131072 ($2^{17}$)

$f_{carrier}$ = 19.6 THz

Baud rate = 25 Gbps

Average of 20 times random

# Chapter 5

# Conclusions and Suggestions

## 5.1 Summary

This project aims to analysis and plot spectrum graph of real digital input signal in Microsoft Excel coding on Visual Basic for Application. At first we use unreal signal for simulation in Binary Phase Shift Keying modulation. Then, using unreal signal for Quadrature Phase-Shift Keying(QPSK) and finally we use real signal for Quadrature Phase-Shift Keying(QPSK) modulation and plot spectrum graph.

## 5.2 Numerical results

Firstly, Binary Phase-Shift Keying (BPSK) is used for modulating an unreal baud rate. Then, the results of BPSK are calculated by Fourier transform, implemented by Fast Fourier Transform Algorithm, to make a spectrum graph.

After that, Quadrature Phase-Shift Keying is used for modulating an unreal baud rate instead.

Then, Quadrature Phase-Shift Keying is used for modulating real baud rate which equal to 25Gbps and carrier frequency which equal 196 THz. But the problem occurred because n is too small and Microsoft Excel reach its limitation.

So the carrier frequency is changed to 19.6 THz. And the comparison between carrier frequency = 196THz and carrier frequency = 19.6THz show that graph shapes of 196 THz and 19.6 THz of carrier frequency are almost the same.

Finally , the 19.6 THz is chosen to be used as carrier frequency instead of 196 THz when n is set to 131072 ($2^{17}$) and baud rate is set to 25 Gbps. Average of 20 times calculation of Fourier transform results are used to plot graph.

The developed program is able to calculate values for plotting spectrum graph from digital data when modulation method is QPSK.

## 5.3 Problems

1. Program calculation takes too long time.
2. At first, Microsoft Excel version 2003 is used for calculation but the program can bear only 65536 cells ($2^{16}$ cells). So the program version is upgraded to version 2007.

## 5.4 Suggestions

If there is any faster algorithm for calculating, the program algorithm should be modified because it takes too long time for calculating.

# References


[1] Wikipedia, Visual Basic for Applications, **Wikipedia the free encyclopedia** [Online], Available : http://en.wikipedia.org/wiki/Visual_Basic_for_Applications. [2011, August 22].

[2] Wikipedia, Phase-shift keying, **Wikipedia the free encyclopedia** [Online], Available : http://en.wikipedia.org/wiki/Phase-shift_keying. [2011, August 22].

[3] Wikipedia, Fast Fourier transform, **Wikipedia the free encyclopedia** [Online], Available : http://en.wikipedia.org/wiki/Fast_Fourier_transform. [2011, August 22].

[4] Tiff, Pulse Code Modulation: Filtering and Sampling , **Atif's Blog** [Online], Available : http://atif-razzaq.blogspot.com/2011/03/pulse-code-modulation-filtering-and.html. [2011, August 25].

[5] Martin, Joao, How to implement the FFT algorithm, **CodeProject** [Online], Available : http://www.codeproject.com/KB/recipes/howtofft.aspx. [2011, August 25].

[6] David, Marshall, The Fast Fourier Transform Algorithm, **Cardiff School of Computer Science & Informatics** [Online], Available : http://www.cs.cf.ac.uk/Dave/Vision_lecture/node20.html. [2011, September 3].

[7] Katja, Bit Reversal and Permutation, **katja's homepage on sinusoids,complex numbers and modulation** [Online], Available : http://www.katjaas.nl/bitreversal/bitreversal.html. [2011, September 3].

# Appendix

## 1. Fourier Method

Let consider a 1D Fourier transform example:

Consider a complicated sound such as the noise of a car horn. We can describe this sound in two related ways:

- sample the amplitude of the sound many times a second, which gives an approximation to the sound as a function of time.
- analyze the sound in terms of the pitches of the notes, or frequencies, which make the sound up, recording the amplitude of each frequency.

Similarly brightness along a line can be recorded as a set of values measured at equally spaced distances apart, or equivalently, at a set of spatial frequency values.

Each of these frequency values is referred to as a ***frequency component***.

An image is a two-dimensional array of pixel measurements on a uniform grid.

This information be described in terms of a two-dimensional grid of spatial frequencies.

A given frequency component now specifies what contribution is made by data which is changing with specified x and y direction spatial frequencies.

**The Fast Fourier Transform Algorithm**

This is how the DFT may be computed efficiently.

**1D Case**

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-2\pi ixu/N} \qquad (23)$$

has to be evaluated for $N$ values of $u$, which if done in the obvious way clearly takes $N^2$ multiplications.

It is possible to calculate the DFT more efficiently than this, using the **fast Fourier transform** or FFT algorithm, which reduces the number of operations to $O(N \log_2 N)$.

We shall assume for simplicity that $N$ is a power of 2, $N = 2^n$.

If we define $\omega_N$ to be the $N^{th}$ root of unity given by $\omega_N = e^{-2\pi i/N}$, and set $M=N/2$, we have

$$F(u) = \frac{1}{2M}\sum_{x=0}^{2M-1} f(x)\omega_{2M}^{ux}. \qquad (24)$$

This can be split apart into two separate sums of alternate terms from the original sum,

$$F(u) = \frac{1}{2}\left(\frac{1}{M}\sum_{x=0}^{M-1} f(2x)\omega_{2M}^{(2x)u} + \frac{1}{M}\sum_{x=0}^{M-1} f(2x+1)\omega_{2M}^{(2x+1)u}\right). \qquad (25)$$

Now, since the square of a $2M^{th}$ root of unity is an $M^{th}$ root of unity, we have that

$$\omega_{2M}^{(2x)u} = \omega_M^{xu} \qquad (26)$$

and hence

$$F(u) = \frac{1}{2}\left(\underbrace{\frac{1}{M}\sum_{x=0}^{M-1} f(2x)\omega_M^{xu}}_{F_{even}(u)} + \underbrace{\frac{1}{M}\sum_{x=0}^{M-1} f(2x+1)\omega_M^{xu}}_{F_{odd}(u)}\omega_{2M}^{u}\right). \qquad (27)$$

If we call the two sums demarcated above $F_{even}(u)$ and $F_{odd}(u)$ respectively, then we have

$$F(u) = \frac{1}{2}\left(F_{even}(u) + F_{odd}(u)\omega_{2M}^{u}\right). \qquad (28)$$

Note that each of $F_{even}(u)$ and $F_{odd}(u)$ for $u = 0, \ldots, M-1$ is in itself a discrete Fourier transform over $N/2=M$ points. How does this help us? Well

$$\omega_M^{M+u} = \omega_M^u \quad \text{and} \quad \omega_{2M}^{M+u} = -\omega_{2M}^u, \qquad (29)$$

and we can also write

$$F(u+M) = \frac{1}{2}\left(F_{even}(u) - F_{odd}(u)\omega_{2M}^{u}\right). \qquad (30)$$

Thus, we can compute an $N$-point DFT by dividing it into two parts:

The first half of $F(u)$ for $u = 0, \ldots, M-1$ can be found from Eqn. 28,

The second half for $u = M, \ldots, 2M-1$ can be found simply be reusing the same terms differently as shown by Eqn. 30. This is obviously a divide and conquer method.

To show how many operations this requires, let $T(n)$ be the time taken to perform a transform of size $N = 2^n$, measured by the number of multiplications performed. The above analysis shows that

$$T(n) = 2T(n-1) + 2^{(n-1)}, \qquad (31)$$

the first term on the right hand side coming from the two transforms of half the original size, and the second term coming from the multiplications of $F_{odd}$ by $\omega_{2M}^{u}$. Induction can be used to prove

that

$$T(n) = 2^{(n-1)} \log_2 2^n = \frac{1}{2} N \log_2 N, \qquad (32)$$

A similar argument can also be applied to the number of additions required, to show that the algorithm as a whole takes time $O(N \log_2 N)$.

Also Note that the same algorithm can be used with a little modification to perform the inverse DFT too. Going back to the definitions of the DFT and its inverse,

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-2\pi i x u / N} \qquad (23)$$

and

$$f(x) = \sum_{u=0}^{N-1} F(u) e^{2\pi i x u / N}, \qquad (33)$$

If we take the complex conjugate of the second equation, we have that

$$f^*(x) = \sum_{u=0}^{N-1} F^*(u) e^{-2\pi i x u / N}. \qquad (34)$$

This now looks (apart from a factor of 1/*N*) like a forward DFT, rather than an inverse DFT. Thus to compute an inverse DFT, take the conjugate of the Fourier space data, put conjugate through a forward DFT algorithm, take the conjugate of the result, at the same time multiplying each value by *N*.

**2D Case**

The same fast Fourier transform algorithm can be used -- applying the separability property of the 2D transform. Rewrite the 2D DFT as

$$\begin{aligned} F(u, v) &= \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) e^{-2\pi i (xu + yv)/N} \\ &= \frac{1}{N} \sum_{x=0}^{N-1} e^{-2\pi i x u / N} \sum_{y=0}^{N-1} f(x, y) e^{-2\pi i y v / N}. \end{aligned} \qquad (35)$$

The right hand sum is basically just a one-dimensional DFT if *x* is held constant. The left hand sum is then another one-dimensional DFT performed with the numbers that come out of the first set of sums. So we can compute a two-dimensional DFT by

- performing a one-dimensional DFT for each value of *x*, ***i.e.*** for each column of *f*(*x*,*y*), then

- performing a one-dimensional DFT in the opposite direction (for each row) on the resulting values.

This requires a total of 2 $N$ one dimensional transforms, so the overall process takes time $O(N^2 \log_2 N)$.

# Profile

| | |
|---|---|
| **Name** | Ms. Nattaya Hanrattanasakul |
| **Birthday** | 12 January 1990 |
| **Education** | |
| Elementary School | Grade 1 - Grade 6 Year 1996<br>Anubarnpattani School |
| Junior School | Grade7 – Grade 12 Year 2002<br>Demonstration School Prince of Songkla University Pattani Campus |
| Undergraduate School | Faculty of Engineering , Computer Engineering Year 2007<br>Thai-Nichi Institute of Technology |
| **Scholarship** | Thai-Nichi Institute of Technology Scholarship Type 2 |
| **Training Profile** | - None - |
| **Published Book** | - None - |