A tutorial on trigonometric curve fitting

By Cedrick Collomb[1]

Abstract. This tutorial introduces the concept and usefulness of approximating a signal by a sum of sinusoidal functions hopefully in a simple manner intelligible to any reader with minimal mathematical and engineering skills. The Prony algorithm for trigonometric curve fitting is explained in full detail, formulas are derived and every step is purposely made unnecessarily detailed for ease of understanding. Source code is provided as a way to skip the potential formula obfuscation in order to help the reader implement and use the ideas behind this tutorial immediately after reading this document.

1. Introduction

Joseph Fourier changed forever the mathematics and science landscape when he introduced to the world the fact that periodic functions could be expressed as trigonometric series, his discovery was so astonishing at the time that even some the most brilliant mathematicians of that period had some difficulties accepting his surprising results. Fourier's theory generated interest to further attempt to break down a function as a sum of simple sinusoidal waves and thus gave rise to the field of Harmonic Analysis, which has major implications in a wide range of areas from audio analysis and synthesis to stock market analysis and prediction [1][2].

The discrete Fourier transform, a process to transform series of sample data to frequencies, is unfortunately tightly linked to its data window, which raise the issue of resolution constraints. New methods called high resolution methods have to be used in order to decompose signals with arbitrary frequencies. There exists a method invented by Prony [3] prior to Fourier's discovery that can address this challenge, and that method is presented in this document.

The rest of the tutorial is organized as follow: Section 2 introduces the basic concept of trigonometric fitting. Section 3 explains how to solve for frequencies using Prony's method. Section 4 explains how to find the amplitudes and phases. Section 5 concludes this tutorial.

2. Trigonometric fitting

Trigonometric fitting is a method to approximate a function f by series of trigonometric functions. The approximation g of f can be written

$$g(x) = \sum_{i=1}^{m} \rho_i \cos(\omega_i x + \varphi_i) \qquad (1)$$

Where m is the number of terms we want to approximate f with, ρ_i is the amplitude, ω_i is the frequency and φ_i is the amplitude of the ith cosine function.

[1] ccollomb@yahoo.com / http://ccollomb.free.fr/

To transform the previous equation we need to use the following trigonometric identity.

$$\cos(a+b)=\cos(a)\cos(b)-\sin(a)\sin(b) \tag{2}$$

Using identity (2) we can rewrite (1) as

$$\begin{aligned} g(x) &= \sum_{i=1}^{m}\rho_i\left[\cos(\omega_i x)\cos(\varphi_i)-\sin(\omega_i x)\sin(\varphi_i)\right] \\ &= \sum_{i=1}^{m}\rho_i\cos(\varphi_i)\cos(\omega_i x)-\sum_{i=1}^{m}\rho_i\sin(\varphi_i)\sin(\omega_i x) \end{aligned}$$

Therefore g can also be written interchangeably with (1) as follow:

$$g(x)=\sum_{i=1}^{m}\alpha_i\cos(\omega_i x)+\sum_{i=1}^{m}\beta_i\sin(\omega_i x) \tag{3}$$

And in this case $\alpha_i=\rho_i\cos(\varphi_i)$ and $\beta_i=-\rho_i\sin(\varphi_i)$.

It is to be noted that only f is known at this stage. A value of m has to be decided depending on the problem at hand or accuracy of approximation desired. If equation (1) is used $(\rho_i)_{i\in[\![1;m]\!]}$, $(\omega_i)_{i\in[\![1;m]\!]}$ and $(\varphi_i)_{i\in[\![1;m]\!]}$ have to be determined. If equation (3) is used $(\omega_i)_{i\in[\![1;m]\!]}$, $(\alpha_i)_{i\in[\![1;m]\!]}$ and $(\beta_i)_{i\in[\![1;m]\!]}$ have to be determined.

3. Finding the frequencies $(\omega_i)_{i\in[\![1;m]\!]}$

a. Prony's reformulation of the problem

Let $p=2m$, the Prony method starts by replacing (3) by the following equation:

$$g(x)=\sum_{j=1}^{p}\gamma_j z_j^x \tag{4}$$

Where γ_j are complex numbers, $z_j=e^{i\theta_j}$, with θ_j real numbers and $i^2=-1$.

If equation (4) can be solved and constrained so that to get couples of conjugate γ_j and z_j, equation (4) can be transformed back into equation (3).

For all index j let there exists an index k so that $\gamma_k=\overline{\gamma_j}$ and $z_k=\overline{z_j}$ then it results that $\gamma_j z_j^x+\gamma_k z_k^x=\gamma_j z_j^x+\overline{\gamma_j}\,\overline{z_j}^x=\gamma_j z_j^x+\overline{\gamma_j z_j^x}=2\operatorname{Re}\left(\gamma_j z_j^x\right)=2\operatorname{Re}\left(\gamma_j e^{ix\theta_j}\right)$ and therefore

$$\gamma_j z_j^x+\gamma_k z_k^x=2\left(\operatorname{Re}(\gamma_j)\cos(x\theta_j)-\operatorname{Im}(\gamma_j)\sin(x\theta_j)\right) \tag{5}$$

b. Prony's polynomial

Prony's original idea to solve (4) is to introduce the polynomial $P(z)$ that has all the $(z_j)_{j\in[\![1;p]\!]}$ for roots.

$$P(z)=\prod_{j=1}^{p}(z-z_j) \tag{6}$$

There exists $(a_k)_{k\in[\![0;p]\!]}$ so that $$P(z)=\sum_{k=0}^{p}a_k z^{p-k} \tag{7}$$

By definition $P(z_j)=0$ therefore $$\sum_{k=0}^{p}a_k z_j^{p-k}=0 \tag{8}$$

Also by definition $z_j=e^{i\theta_j}$ therefore $\overline{z_j}=e^{-i\theta_j}=\frac{1}{z_j}$, and thus $P\left(\frac{1}{z_j}\right)=0$. Which leads to the following equation: $\sum_{k=0}^{p}a_k\frac{1}{z_j^{p-k}}=\sum_{k=0}^{p}a_k\frac{z_j^k}{z_j^p}=\frac{1}{z_j^p}\sum_{k=0}^{p}a_k z_j^k=0$.

Therefore $$\sum_{k=0}^{p}a_k z_j^k=0 \tag{9}$$

The polynomial $Q(z)=\sum_{k=0}^{p}a_k z^k$ in (9) has p roots as has $P(z)$, moreover they are all equal to the roots of $P(z)$. Therefore $Q(z)=\lambda P(z)$. Rewriting $P(z)=\sum_{k=0}^{p}a_{p-k}z^k$ gives:

For all $k\in[\![0;p]\!]$ $$a_k=\lambda a_{p-k} \tag{10}$$

Considering that $(z_j)_{j\in[\![1;p]\!]}$ contain couples of roots of the form $z_j=e^{i\theta_j}$, $P(z)$ can be transformed as follow: $P(z)=\prod_{j=1}^{m}\left(z-e^{i\theta_j}\right)\left(z-e^{-i\theta_j}\right)=\prod_{j=1}^{m}\left(z^2-\left(e^{i\theta_j}+e^{-i\theta_j}\right)z+e^{i\theta_j}e^{-i\theta_j}\right)$.

Therefore $$P(z)=\prod_{j=1}^{m}\left(z^2-2\cos(\theta_j)z+1\right) \tag{11}$$

It follows that the $(a_k)_{k\in[\![0;p]\!]}$ are real and that $a_0=1$ and $a_p=1$, therefore that $\lambda=1$.

Thus $\forall k\in[\![0;p]\!]$ $$a_k=a_{p-k} \tag{12}$$

c. Identifying Prony's polynomial coefficients

The next step is to find an equation that will help determine the $\left(a_k\right)_{k\in[\![0;p]\!]}$ coefficients.

Using (4), (9) and the fact that $Q\left(z_j\right)=0$ gives:

$$\sum_{k=0}^{p} a_k g\left(x+k\right)=\sum_{k=0}^{p} a_k \sum_{j=1}^{p} \gamma_j z_j^{x+k}=\sum_{j=1}^{p} \gamma_j z_j^{x} \sum_{k=0}^{p} a_k z_j^{k}=\sum_{j=1}^{p} \gamma_j z_j^{x} Q\left(z_j\right)$$

Therefore
$$\sum_{k=0}^{p} a_k g\left(x+k\right)=0 \tag{13}$$

Using (12) and (13) gets $\sum_{k=0}^{m-1} a_k\left(g\left(x+k\right)+g\left(x+p-k\right)\right)+a_m g\left(x+m\right)=0$ and since $a_0=1$, the following result defining $\left(a_k\right)_{k\in[\![1;m]\!]}$ can be reached:

$$a_m g\left(x+m\right)+\sum_{k=1}^{m-1} a_k\left(g\left(x+k\right)+g\left(x+p-k\right)\right)=-g\left(x\right)-g\left(x+p\right) \tag{14}$$

The approximation g of f is done at N regularly spaced samples, with $p\le N$. Most of the time when N is large enough, equation (14) defines an over determined, over constrained linear equation, because it defines $N-p$ equations for only m unknown variables. There are several different ways to solve those linear systems. A simple and common method that is described in this tutorial is the Least-Squares (LS) method.

d. Least-Squares method to identify the $\left(a_k\right)_{k\in[\![1;m]\!]}$ coefficients

Since (14) gives too much data for not enough unknowns, an error function which is the sum of the square of residuals between the left and right sides of equation (14), allows to regroup everything under one sum and one line.

$$e\left(a_{1,\cdots,}a_m\right)=\sum_{x=1}^{N-p}\left(a_m g\left(x+m\right)+\sum_{k=0}^{m-1} a_k\left(g\left(x+k\right)+g\left(x+p-k\right)\right)\right)^2 \tag{15}$$

To minimize (15), the point $\left(a_{1,\cdots,}a_m\right)$ at which $\nabla e\left(a_{1,\cdots,}a_m\right)=0$ need to be found. It is equivalent to solve for all j the following partial derivative equations:

$$\frac{\partial e\left(a_{1,\cdots,}a_m\right)}{\partial a_j}=0 \tag{16}$$

To simplify the derivations lets define $S_{g,k}(x)=g(x+k)+g(x+p-k)$, equation (15) can therefore be rewritten as $e(a_{1,\cdots,}a_m)=\sum_{x=1}^{N-p}\left(a_m g(x+m)+\sum_{k=0}^{m-1}a_k S_{g,k}(x)\right)^2$

It results that
$$\frac{\partial e}{\partial a_j}=2\sum_{x=1}^{N-p}S_{g,j}(x)\left(a_m g(x+m)+\sum_{k=0}^{m-1}a_k S_{g,k}(x)\right) \tag{17}$$

And that
$$\frac{\partial e}{\partial a_m}=2\sum_{x=1}^{N-p}g(x+m)\left(a_m g(x+m)+\sum_{k=0}^{m-1}a_k S_{g,k}(x)\right) \tag{18}$$

Using (16) and (17) the following linear equations for $j\in[\![1;m-1]\!]$ can be found

$$\sum_{x=1}^{N-p}S_{g,j}(x)\left(a_m g(x+m)+\sum_{k=1}^{m-1}a_k S_{g,k}(x)\right)=-\sum_{x=1}^{N-p}S_{g,0}(x)S_{g,j}(x) \tag{19}$$

Using (16) and (18) the following linear equations for $j=m$ can be found

$$\sum_{x=1}^{N-p}g(x+m)\left(a_m g(x+m)+\sum_{k=1}^{m-1}a_k S_{g,k}(x)\right)=-\sum_{x=1}^{N-p}S_{g,0}(x)g(x+m) \tag{20}$$

Finally equations (19) and (20) can be put in matrix form.

$$M\begin{bmatrix}a_1\\ \vdots\\ a_{m-1}\\ a_m\end{bmatrix}=-\begin{bmatrix}\sum_{x=1}^{N-p}S_{g,0}(x)S_{g,1}(x)\\ \vdots\\ \sum_{x=1}^{N-p}S_{g,0}(x)S_{g,m-1}(x)\\ \sum_{x=1}^{N-p}S_{g,0}(x)g(x+m)\end{bmatrix} \tag{21}$$

With $M=\begin{bmatrix}\sum_{x=1}^{N-p}S_{g,1}(x)S_{g,1}(x) & \cdots & \sum_{x=1}^{N-p}S_{g,m-1}(x)S_{g,1}(x) & \sum_{x=1}^{N-p}g(x+m)S_{g,1}(x)\\ \vdots & \ddots & \vdots & \vdots\\ \sum_{x=1}^{N-p}S_{g,1}(x)S_{g,m-1}(x) & \cdots & \sum_{x=1}^{N-p}S_{g,m-1}(x)S_{g,m-1}(x) & \sum_{x=1}^{N-p}g(x+m)S_{g,m-1}(x)\\ \sum_{x=1}^{N-p}S_{g,1}(x)g(x+m) & \cdots & \sum_{x=1}^{N-p}S_{g,m-1}(x)g(x+m) & \sum_{x=1}^{N-p}g(x+m)g(x+m)\end{bmatrix}$

Although some source code is provided in the appendix, solving equation (21) is outside the scope of this tutorial but should not be a blocker for the reader since solving linear systems is well addressed in the literature; see [4], [5] and [6] for example.

e. Solving the polynomial for frequencies

This is the last part of the puzzle. At this stage the polynomial and its coefficients can be determined, however nothing guarantees that the roots will take the form $z_j = e^{i\theta_j}$, nor that their conjugate will also be roots as desired initially. The necessary conditions to establish constraints on the $(a_k)_{k\in[\![1;m]\!]}$ have not been shown to be sufficient. Instead of doing so, stronger necessary conditions are going to be derived by transforming the polynomial in an alternative form.

Decomposing (8) gives $P(z) = \sum_{k=0}^{m-1} a_k z^{p-k} + a_m z^m + \sum_{k=m+1}^{p} a_k z^{p-k}$, and using (12), it becomes $P(z) = \sum_{k=0}^{m-1} a_k z^{p-k} + a_m z^m + \sum_{k=m+1}^{p} a_{p-k} z^{p-k}$. Changing variables with $k' = p-k$, gives $k = p-k'$ and k' goes from $p-p=0$ to $p-(m+1) = 2m-m-1 = m-1$. The equation can therefore be rewritten $P(z) = \sum_{k=0}^{m-1} a_k z^{p-k} + a_m z^m + \sum_{k'=0}^{m-1} a_{k'} z^{k'}$. The variable k' can be renamed k and the two sums can be regrouped since now they have similar bounds.

$$P(z) = \sum_{k=0}^{m-1} a_k \left(z^{p-k} + z^k\right) + a_m z^m \tag{22}$$

Since the roots are to be of the form $z_j = e^{i\theta_j}$, none of the roots will be zero and equation (22) can be factored by z^m which gives:

$$P(z) = z^m \left(\sum_{k=0}^{m-1} a_k \left(z^{m-k} + z^{k-m}\right) + a_m \right) = z^m \left(\sum_{k=0}^{m-1} a_k \left(z^{m-k} + \frac{1}{z^{m-k}} \right) + a_m \right)$$

Moreover given that $z_j^{m-k} + \frac{1}{z_j^{m-k}} = 2\cos\left((m-k)\theta_j\right)$, instead of solving for z in $P(z)$, the equation $R(\cos(\theta)) = \sum_{k=0}^{m-1} 2a_k \cos\left((m-k)\theta\right) + a_m$ can be used as a replacement.

First $R(\cos(\theta))$ need to be expressed as a polynomial of $\cos(\theta)$ using Chebyshev's polynomials $T_n(x)$, which are defined so that $T_n(\cos(x)) = \cos(nx)$. The recursive formulation of Chebyshev's polynomials is $T_0(x) = 1$, $T_1(x) = x$ and $T_{n+2}(x) = 2xT_{n+1}(x) - T_n(x)$.

Therefore $R(\cos(\theta)) = \sum_{k=0}^{m-1} 2a_k T_{m-k}(\cos(\theta)) + a_m$ and a change of variable $y = \cos(\theta)$ can be applied to obtain the final form of the polynomial to solve.

$$R(y) = \sum_{k=0}^{m-1} 2a_k T_{m-k}(y) + a_m \tag{23}$$

It is interesting to note that Prony's method first expanded from m unknown frequencies to $p = 2m$ complex unknowns, but that (23) has brought it back to m unknowns.

The procedure to solve (23) is to expand the Chebyshev polynomials with the recursive formulas, multiply them by their respective a_k coefficients, and sum all the results to get the final polynomial $R(y)$. When this is done any polynomial root solver can be used to get the y values, finally the frequencies $(\omega_i)_{i\in[\![1;m]\!]}$, are extracted by $w_i = \cos^{-1}(y_i)$.

Again, although some source code is provided in the appendix, solving equation (23) is outside the scope of this tutorial but should not be a blocker for the reader since solving polynomials is well addressed in the literature; see [5], [7], [8] and [8] for example.

4. Finding the amplitudes and phases

a. Finding the amplitudes $(\alpha_i)_{i\in[\![1;m]\!]}$ and $(\beta_i)_{i\in[\![1;m]\!]}$

Looking back at equation (3) ($g(x) = \sum_{i=1}^{m} \alpha_i \cos(\omega_i x) + \sum_{i=1}^{m} \beta_i \sin(\omega_i x)$), at this stage the frequencies that were required have been determined. Only the amplitudes for the cosine and sine series are left to be found.

The Least-Squares method is again going to be used so that to minimize the sum of the square errors between f and g on the N data samples. As in section 3.d an error function to be minimized is defined as follow:

$$e(\alpha,\beta) = \sum_{x=1}^{N} \left(g(x) - f(x)\right)^2 \tag{24}$$

And using (3) $$e(\alpha,\beta) = \sum_{x=1}^{N} \left(\sum_{i=1}^{m} \alpha_i \cos(\omega_i x) + \sum_{i=1}^{m} \beta_i \sin(\omega_i x) - f(x) \right)^2 \tag{25}$$

In order to minimize equation (25), a point $(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m)$ need to be found at which $\nabla e(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m) = 0$, which is equivalent as to solve for all j:

$$\frac{\partial e(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m)}{\partial \alpha_j}=0 \tag{26}$$

and
$$\frac{\partial e(\alpha_1,\ldots,\alpha_m,\beta_1,\ldots,\beta_m)}{\partial \beta_j}=0 \tag{27}$$

From (25)
$$\frac{\partial e}{\partial \alpha_j}=2\sum_{x=1}^{N}\cos(\omega_j x)\left(\sum_{i=1}^{m}\alpha_i\cos(\omega_i x)+\sum_{i=1}^{m}\beta_i\sin(\omega_i x)-f(x)\right) \tag{28}$$

and
$$\frac{\partial e}{\partial \beta_j}=2\sum_{x=1}^{N}\sin(\omega_j x)\left(\sum_{i=1}^{m}\alpha_i\cos(\omega_i x)+\sum_{i=1}^{m}\beta_i\sin(\omega_i x)-f(x)\right) \tag{29}$$

Therefore the following equations that are obtained by swapping the sums signs and by moving all the terms with $f(x)$ on the right hand side of the equation need to be solved.

$$\sum_{i=1}^{m}\alpha_i\sum_{x=1}^{N}\cos(\omega_j x)\cos(\omega_i x)+\sum_{i=1}^{m}\beta_i\sum_{x=1}^{N}\cos(\omega_j x)\sin(\omega_i x)=\sum_{x=1}^{N}\cos(\omega_j x)f(x) \tag{30}$$

and
$$\sum_{i=1}^{m}\alpha_i\sum_{x=1}^{N}\sin(\omega_j x)\cos(\omega_i x)+\sum_{i=1}^{m}\beta_i\sum_{x=1}^{N}\sin(\omega_j x)\sin(\omega_i x)=\sum_{x=1}^{N}\sin(\omega_j x)f(x) \tag{31}$$

Again both (30) and (31) define a linear system, and the algorithm to solve this system is strictly identical to the one used in section 3.d and requires the use of the same source code provided in the Appendix of this document.

$$M\begin{bmatrix}\alpha_1\\ \vdots\\ \alpha_m\\ \beta_1\\ \vdots\\ \beta_m\end{bmatrix}=\begin{bmatrix}\sum_{x=1}^{N}\cos(\omega_1 x)f(x)\\ \vdots\\ \sum_{x=1}^{N}\cos(\omega_m x)f(x)\\ \sum_{x=1}^{N}\sin(\omega_1 x)f(x)\\ \vdots\\ \sum_{x=1}^{N}\sin(\omega_m x)f(x)\end{bmatrix} \tag{32}$$

For space sake, let $c(x)=\cos(x)$ and $s(x)=\sin(x)$ and M is defined as below.

$$M=\begin{bmatrix}
\sum_{x=1}^{N}\mathrm{c}\left(\omega_1 x\right)\mathrm{c}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{c}\left(\omega_1 x\right)\mathrm{c}\left(\omega_m x\right) & \sum_{x=1}^{N}\mathrm{c}\left(\omega_1 x\right)\mathrm{s}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{c}\left(\omega_1 x\right)\mathrm{s}\left(\omega_m x\right) \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\sum_{x=1}^{N}\mathrm{c}\left(\omega_m x\right)\mathrm{c}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{c}\left(\omega_m x\right)\mathrm{c}\left(\omega_m x\right) & \sum_{x=1}^{N}\mathrm{c}\left(\omega_m x\right)\mathrm{s}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{c}\left(\omega_m x\right)\mathrm{s}\left(\omega_m x\right) \\
\sum_{x=1}^{N}\mathrm{s}\left(\omega_1 x\right)\mathrm{c}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{s}\left(\omega_1 x\right)\mathrm{c}\left(\omega_m x\right) & \sum_{x=1}^{N}\mathrm{s}\left(\omega_1 x\right)\mathrm{s}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{s}\left(\omega_1 x\right)\mathrm{s}\left(\omega_m x\right) \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\sum_{x=1}^{N}\mathrm{s}\left(\omega_m x\right)\mathrm{c}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{s}\left(\omega_m x\right)\mathrm{c}\left(\omega_m x\right) & \sum_{x=1}^{N}\mathrm{s}\left(\omega_m x\right)\mathrm{s}\left(\omega_1 x\right) & \cdots & \sum_{x=1}^{N}\mathrm{s}\left(\omega_m x\right)\mathrm{s}\left(\omega_m x\right)
\end{bmatrix}$$

b. Finding the amplitudes $\left(\rho_i\right)_{i\in[\![1;m]\!]}$ and phases $\left(\varphi_i\right)_{i\in[\![1;m]\!]}$

This is the final and easiest step in this whole tutorial. Values of $\left(\alpha_i\right)_{i\in[\![1;m]\!]}$, $\left(\beta_i\right)_{i\in[\![1;m]\!]}$ have been determined, and from section 2: $\alpha_i=\rho_i\cos\left(\varphi_i\right)$ and $\beta_i=-\rho_i\sin\left(\varphi_i\right)$.

Therefore $\rho_i=\sqrt{\alpha_i^2+\beta_i^2}$ and $\varphi_i=\tan^{-1}\left(\dfrac{-\beta_i}{\alpha_i}\right)$. And finally everything that was needed in equation (1) ($g\left(x\right)=\sum_{i=1}^{m}\rho_i\cos\left(\omega_i x+\varphi_i\right)$) has been found.

5. Conclusion

This tutorial has fully detailed the basic trigonometric fitting methods that were briefly described in [1] and [2]. Those methods have a serious drawback. Despite what seems a correct and reasonable symbolic analysis, the practical numerical implementation can suffer from several numerical precision issues.

One of the common numerical issues is related to the method used in both sections 3.d and 4.a to solve an over-determined, over-constrained system of linear equations, for which we have more equations than unknowns to solve. In order to solve this issue both sections 3.d and 4.a have used the same Least-Squares (LS) method using what are called the normal equations which are well known to be ill-defined and not ideal for numerical implementations. The reader of this tutorial interested to address this issue is encouraged to follow-up with the topic of Singular Value Decomposition (SVD) described in [5] and [9].

Another common numerical issue is related to the use of limited machine precision floating point implementations. Trying to use the LS on very large sets of data even with SVD methods can very easily exhibit the limits of the implementation accuracy and give results that are very unsatisfying. A reasonable workaround that can be used in some cases is to reduce the sampling frequency of the data set used, at the expense of the accuracy of the approximation and solution.

The last common numerical issue to be mentioned in this conclusion is related to the Prony method itself. Firstly the method itself is not guaranteed to be optimal since minimizing the error on polynomial coefficients does not insure to find optimal frequencies, and secondly the method has known numerical issues when the signal analyzed is composed of frequencies very close to each other, or when the signal has a non negligible noise component.

Finally trigonometric curve fitting is not a fully solved problem. There exist several other trigonometric curve fitting methods such as Pisarenko's method, the Matrix Pencil method, MUSIC, TAM, ESPRIT, and KT methods. The reader interested to pursue further investigations related to the area of trigonometric curve fitting is encouraged to read [10] and [11] for more details.

The positive side of this incomplete solution is that there is always an interest to discover new trigonometric curve fitting algorithms, and the reader of this tutorial can take part in this research effort.

6. References

1. J.M Hurst, The Profit Magic of Stock Transaction Timing, Traders Press (2000)
2. C.E. Cleeton, The Art of Independent Investing, Prentice-Hall (1976)
3. G. Riche de Prony, Essai éxperimental et analytique, Journal de l'École Polytechnique (1795). Also available at the address http://www.polytech.unice.fr/~leroux/PRONY.pdf
4. R. Barrett et al., Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM (1994)
5. W.H. Press et al., Numerical Recipes in C++: The Art of Scientific Computing, Cambridge University Press (2002)
6. J.P. Pante, Computer Algorithms in Power System Analysis, World Wide Web site at the address http://www.geocities.com/SiliconValley/Lab/4223/fault/ach03.html, (1999)
7. Durand-Kerner method, World Wide Web site at the address http://en.wikipedia.org/wiki/Durand-Kerner_method
8. L. Volpi, Practical Methods for Roots Finding, World Wide Web site at the address http://digilander.libero.it/foxes/poly/Polynomial_zeros.pdf, (2006)
9. G.H. Golub et al., Matrix Computations, Johns Hopkins University Press, (1996)
10. R. Badeau, High Resolution Methods for Estimating and Tracking Modulated Sinusoids, World Wide Web site at the address http://pastel.paristech.org/1234/01/Thesis.pdf, (2005)
11. A. Van der Veen, Subspace Based Signal Analysis using Singular Value Decomposition, Proceedings IEEE, vol. 81, (1993)

7. Appendix. Non optimized C++ code.

```
#include <vector>
#include <complex>

using namespace std;
typedef complex<double> COMPLEX;
const double kPI = 3.14159265358979323846;
```

// Returns in vector all roots of input polynomial using Durand-Kerner / Weirstrass method

```
vector<COMPLEX> PolyRoots( vector<double> &poly, const double eps, const long maxIter, const double
setupRadius )
{
	size_t degree = poly.size() - 1;
	vector<COMPLEX> roots( degree );
	for ( size_t i = 0; i < degree; i++ )
	{
		roots[ i ] = polar( setupRadius, 2.0 * i * kPI / degree );
	}

	double error = eps;
	for ( long iter = maxIter; 0 <= iter, eps <= error; iter-- )
	{
		error = 0.0;
		for ( size_t i = 0; i < degree; i++ )
		{
			COMPLEX divisor( poly[ degree ] );
			COMPLEX delta( poly[ degree ] );
			for ( size_t j = 0; j < degree; j++ )
			{
				if ( i != j )
				{
					divisor *= roots[ i ] - roots[ j ];
				}
				delta = delta * roots[ i ] + poly[ degree - 1 - j ];
			}
			delta /= divisor;
			roots[ i ] -= delta;
			error = max( error, abs( delta ) );
		}
	}

	return roots;
}
```

// Returns the inverse of input matrix using full pivot Gauss Jordan method

```
vector<double> InverseMatrix( vector<double> M, const long n )
{
vector<bool> freeIndices( n, true );

	vector<double> res( n * n, 0.0 );
	for ( long i = 0; i < n; i++ )
	{
		res[ i * n + i ] = 1.0;
	}

	for ( long p = 0; p < n; p++ )
	{
		double pivot = 0.0;
		long mi, mj;
		for ( long i = 0; i < n; i++ )
		{
			for ( long j = 0; j < n; j++ )
			{
				if ( ( fabs( pivot ) < fabs( M[ i * n + j ] ) )
				 && freeIndices[ i ] && freeIndices[ j ] )
				{
					mi = i;
					mj = j;
```

```
                        pivot = M[ i * n + j ];
                    }
                }
            }

            freeIndices[ mj ] = false;

            for ( long j = 0; j < n; j++ )
            {
                swap( M[ mi * n + j ], M[ mj * n + j ] );
                swap( res[ mi * n + j ], res[ mj * n + j ] );
                M[ mj * n + j ] /= pivot;
                res[ mj * n + j ] /= pivot;
            }

            for ( long i = 0; i < n; i++ )
            {
                if ( i != mj )
                {
                    double scale = M[ i * n + mj ];
                    for ( long j = 0; j < n; j++ )
                    {
                        M[ i * n + j ] -= scale * M[ mj * n + j ];
                        res[ i * n + j ] -= scale * res[ mj * n + j ];
                    }
                }
            }
        }

        return res;
}
```

// Small utility functions for Prony algorithm

```
double Sgk( const vector<double> &values, const long m, const long x, const long i )
{
        if ( i != m )
        {
            return values[ x + i ] + values[ x + 2 * m - i ];
        }
        else
        {
            return values[ x + m ];
        }
}

double CosSin( const std::vector<double> &frequencies, const long m, const long x, const long i )
{
        if ( i < m )
        {
            return sin( frequencies[ i ] * x );
        }
        else
        {
            return cos( frequencies[ i - m ] * x );
        }
}

vector<double> MultiplyMatrixVector( const vector<double> &M, const long n, const vector<double> &V )
{
        vector<double> res( n, 0.0 );
        for ( long i = 0; i < n; i++ )
```

```
	{
		for ( long j = 0; j < n; j++ )
		{
			res[ i ] += M[ i * n + j ] * V[ j ];
		}
	}
	return res;
}
```

// Returns up to m frequencies, amplitudes and phases from input data vector

```
void PronyFit( const vector<double> &values, const long &m, vector<double> &frequenciesRad,
vector<double> &amplitudes, vector<double> &phases )
{
	long p = 2 * m;
	long n = static_cast<long>( values.size() );

	// CREATE VECTOR AND MATRIX TO GET POLYNOMIAL COEFFICIENTS

	vector<double> A( m * m, 0.0 );
	vector<double> B( m, 0.0 );
	for ( long i = 0; i < m; i++ )
	{
		for ( long x = 0; x < n - p; x++ )
		{
			for ( long j = 0; j < m; j++ )
			{
				A[ i * m + j ] += Sgk( values, m, x, i + 1 ) * Sgk( values, m, x, j + 1 );
			}
			B[ i ] -= Sgk( values, m, x, i + 1 ) * Sgk( values, m, x, 0 );
		}
	}

	vector<double> inverseA = InverseMatrix( A, m );
	vector<double> X = MultiplyMatrixVector( inverseA, m, B );

	// CREATE LINEAR COMBINATION OF CHEBYSHEV POLYNOMIALS

	vector<double> Tnm2( 1, 1.0 ); // 1
	double coeffsTnm1[] = { 0.0, 1.0 }; // X
	vector<double> Tnm1( coeffsTnm1, coeffsTnm1 + sizeof( coeffsTnm1 ) / sizeof( double ) );
	double coeffsP[] = { X[ m - 1 ], 2.0 * X[ m - 2 ] };
	vector<double> P( coeffsP, coeffsP + sizeof( coeffsP ) / sizeof( double ) );

	for ( long i = m - 3; -1 <= i; i-- )
	{
		vector<double> Tn( Tnm1.size() + 1, 0.0 );
		for ( size_t j = 0; j < Tnm1.size(); j++ )
		{
			Tn[ j + 1 ] += 2.0 * Tnm1[ j ];
		}
		for ( size_t j = 0; j < Tnm2.size(); j++ )
		{
			Tn[ j ] -= Tnm2[ j ];
		}

		Tnm2 = Tnm1;
		Tnm1 = Tn;

		double k = 2.0;
		if ( 0 <= i )
		{
```

```
                        k *= X[ i ];
                }

                for ( size_t j = 0; j < Tn.size(); j++ )
                {
                        Tn[ j ] *= k;
                }
                for ( size_t j = 0; j < P.size(); j++ )
                {
                        Tn[ j ] += P[ j ];
                }

                P = Tn;
        }

        // SOLVE FOR COSINE OF FREQUENCIES

        const double epsilon = 1.e-6;
        vector<COMPLEX> roots = PolyRoots( P, epsilon, 100, 2.0 );

        // CONVERT TO FREQUENCIES IF ROOTS ARE VALID

        for ( long i = 0; i < m; i++ )
        {
                if ( ( fabs( roots[ i ].imag() ) < epsilon ) && ( fabs( roots[ i ].real() ) < 1.0 + epsilon ) )
                {
                        frequenciesRad.push_back( acos( max( min( roots[ i ].real(), 1.0 ), -1.0 ) ) );
                }
        }

        // CREATE VECTOR AND MATRIX TO GET AMPLITUDES AND PHASES

        long nbFreq = static_cast<long>( frequenciesRad.size() );
        long m2 = 2 * nbFreq;

        vector<double> A2( m2 * m2, 0.0 );
        vector<double> B2( m2, 0.0 );
        for ( long i = 0; i < m2; i++ )
        {
                for ( long x = 0; x < n; x++ )
                {
                        for ( long j = 0; j < m2; j++ )
                        {
                                A2[ i * m2 + j ] += CosSin( frequenciesRad, nbFreq, x, i )
                                * CosSin( frequenciesRad, nbFreq, x, j );
                        }
                        B2[ i ] += CosSin( frequenciesRad, nbFreq, x, i ) * values[ x ];
                }
        }

        vector<double> inverseA2 = InverseMatrix( A2, m2 );
        vector<double> X2 = MultiplyMatrixVector( inverseA2, m2, B2 );

        // CONVERT TO FINAL AMPLITUDE AND PHASES

        for ( long i = 0; i < nbFreq; i++ )
        {
                amplitudes.push_back( sqrt( X2[ i ] * X2[ i ] + X2[ i + nbFreq ] * X2[ i + nbFreq ] ) );
                phases.push_back( -atan2( X2[ i ], X2[ i + nbFreq ] ) );
        }
}
```

// Example program using PronyFit

```
int main( int argc, char* argv[] )
{
	vector<double> originalData( 32, 0.0 );

	vector<double> originalFrequencies;
	originalFrequencies.push_back( 0.3 * 2.0 * kPI / originalData.size() );
	originalFrequencies.push_back( 1.7 * 2.0 * kPI / originalData.size() );
	originalFrequencies.push_back( 3.5 * 2.0 * kPI / originalData.size() );
	double originalAmplitudes[] = { 3.0, 5.0, 0.7 };
	double originalPhases[] = { 0.0, 1.0, 2.0 };

	for ( size_t i = 0; i < originalData.size(); i++ )
	{
		for ( size_t j = 0; j < originalFrequencies.size(); j++ )
		{
			originalData[ i ] += originalAmplitudes[ j ] * cos( originalFrequencies[ j ] * i
			+ originalPhases[ j ] );
		}
	}

	vector<double> pronyFrequencies;
	vector<double> pronyAmplitudes;
	vector<double> pronyPhases;
	PronyFit( originalData, 3, pronyFrequencies, pronyAmplitudes, pronyPhases );

	vector<double> pronyData( originalData.size(), 0.0 );
	for ( size_t i = 0; i < pronyData.size(); i++ )
	{
		for ( size_t j = 0; j < pronyFrequencies.size(); j++ )
		{
			pronyData[ i ] += pronyAmplitudes[ j ] * cos( pronyFrequencies[ j ] * i
			 + pronyPhases[ j ] );
		}

		double error = fabs( pronyData[ i ] - originalData[ i ] );
		printf( "Original: % .6f / Prony: % .6f / Error: % .6f\n", originalData[ i ], pronyData[ i ], error );
	}

	return 0;
}
```