

# Frequency Stability Analysis Using R

W.J. Riley  
Hamilton Technical Services  
Beaufort, SC 29907 USA  
[bill@wriley.com](mailto:bill@wriley.com)  
Rev. C June 4, 2020

## ABSTRACT

This document describes techniques for frequency stability analysis using as their basis the R program for statistical computing and graphics. It describes how R can be used for quantifying the stability of a frequency source in the time and frequency domains, providing information about practical methods for conducting such an analysis, including an R package of functions for that purpose.

## 1 INTRODUCTION

R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing<sup>1</sup>. It runs on a large number of UNIX/Linux, Windows and MacOS platforms. C/C++ and Fortran programs can be linked for speed and efficiency. Much useful information about the R language and its programming environment will be found in Reference [3]<sup>2</sup>.

Why would someone want to use R for frequency stability analysis rather than, say, a special-purpose tool like the comprehensive and freely-

available Stable32<sup>3</sup> program? Perhaps to allow greater flexibility and support experimentation with the underlying algorithms, or to generate plainer graphical results more suited for publication and presentations. The R computing environment is better suited to performing specific, perhaps customized, functions rather than supporting a large integrated application. The R console resembles the UNIX command line, with small, tersely-named, single-purpose functions. Larger functions can easily be built by combining them on-screen or in a script. The programming environment resembles a C or Python interpreter, with easy variable handling and memory management, which encourages calculator-like experimentation and testing. [RStudio](#) is available as a GUI R programming environment.

R has become somewhat of a standard for general-purpose and academic statistical analysis, is free, and is well-supported, including many additional packages. But, except for basic Allan variance functions (see Section 3.1 below), those do not directly support frequency stability analysis. This paper will (hopefully) improve that. Related R spectral analysis functionality is available in Reference [2] and its supporting R code<sup>4</sup>.

---

<sup>1</sup> R Development Core Team (2010). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL <http://www.R-project.org>.

<sup>2</sup> This document refers to the original 2007 version which is available as a free download (see [3]).

---

<sup>3</sup> The Stable32 program and its documentation is freely available from the International Electrical and Electronic Engineers (IEEE) Ultrasonics, Ferroelectrics, and Frequency Control (UFFC) Society at: <https://ieeef.org/frequency-control/frequency-control-software/stable32/>.

<sup>4</sup> See: <http://faculty.washington.edu/dbp/sauts.html>

## 1.1 Getting Started with R

This document is not intended as a tutorial on R. We will simply say that one should start by downloading and installing the R software package from [www.R-project.org](http://www.R-project.org), and consult the material there along with other on-line sources and books (such as Reference [3] to the extent needed. If you already have R installed, it is recommended that you re-install the latest version. Then, if new to R, one can try commands to start to become familiar with it.

We will not describe a complete R package of functions for frequency stability analysis (although such a collection would be desirable), but rather simply show a number of individual functions for certain key operations. The R environment favors the use of many small single-purpose functions, and you can build up a collection of those to suit you specific needs.

We show R commands in red as in the default R console.

## 1.2 R Examples

The examples of R code herein use R 4.0.0 under Windows and are intended to illustrate its use for frequency stability analysis.

Make sure that you have full read/write permission to the R library folder on your computer. Packages can then be installed and loaded using the Packages/Install package(s)... and Load package... menu items. The following packages should be installed and loaded for the exercises herein:

- ‘RobPer’ (for TK95())
- ‘sazedR’ (for downsample())
- ‘allanvar’ (for Allan variance, etc.)
- ‘avar’ (for Allan variance)
- ‘zoo’ (for rollapply())
- ‘fsa’ (see Appendix 6)

## 1.3 R Frequency Stability Analysis

This document is intended to introduce a frequency stability analyst to the use of R for that purpose. It leads a new R analyst through a

number of examples emphasizing analysis techniques rather than program operation. A key aspect of the tutorial is the ability to generate power law noise as test data to use for exploring the various analysis methods.

This document contains basic information about some aspects of frequency stability analysis, with references to further details (e.g., the math), principally the Reference [5] *Handbook of Frequency Stability Analysis*. Examples are included for some topics to stimulate further study. It is recommended that, after reading about a topic in this document, the reader consult the referenced section of the *Handbook*, the references cited herein, and then their referenced documents as you get deeper into these subjects.

Most analysts who use the techniques of frequency stability analysis have frequency sources they wish to characterize and measuring systems for doing so. But all users of precision frequency sources need to understand those techniques, and hands-on experience with them, using actual or simulated data, is the best way to become familiar with them.

## 2 TIME SERIES ANALYSIS

Frequency stability analysis is an example of time series analysis which applies statistical measures to describe the properties of a time-ordered set of data, in this case usually either phase data in seconds or dimensionless fractional frequency deviations. The analyses can be performed in either the time or frequency domain. Chapter 22 of *The R Book* is concerned with Time Series Analysis.

The field of time series analysis is very broad and well-established, and an internet search will produce a vast amount of material. In the case of frequency source (“oscillator”, “frequency standard” or “clock”) characterization one is typically concerned with describing a finite sample of clock data for average frequency, slow trends (“drift” and “aging”) and shorter-term noise-like fluctuations. The latter are often described in the time domain by variances and

in the frequency domain by spectral densities. More specifically, there are specialized statistical variances (e.g., the Allan variance, AVAR) and spectral density types (e.g., the SSB phase noise,  $\mathcal{L}(f)$ , dBc/Hz) that have been developed for frequency stability analysis. Those measures can be implemented by extensions to the basic R package.

Frequency stability analysis generally applies to equally-spaced discrete phase or frequency measurements (a *time series*) taken at a particular *measurement interval* denoted by the lower-case Greek letter *tau* ( $\tau$ ). Other words used for this quantity are *sampling interval*, *measurement time*, *sampling time* or *averaging time*. The measurement and sampling terms are usually associated with the measurement process itself, while the averaging time applies to the analysis. The basic measurement interval is often denoted as  $\tau_0$  while the analysis averaging time is simply called  $\tau$ . As noted above, phase data have units of seconds, while frequency data are dimensionless<sup>5</sup> fractional frequency.

Fractional frequency data can be converted to a longer sampling time by arithmetic averaging. That averaging can be performed by a function similar to `downsample()` in the ‘sazedR’ package<sup>6</sup>:

```
favg<-function(data,af=2)
+ {
+ return(rollapply(data,width=af,
+ by=af,FUN=mean))
+ }
```

For example:

```
>y<-1:20
>y
[1] 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20
> w<-favg(y, 4)
> w
2.5 6.5 10.5 14.5 18.5
```

The function returns the averaged frequency points.

<sup>5</sup> Units of Hz/Hz are sometimes associated with fractional frequency values, but that seems rather awkward.

<sup>6</sup>See: <https://www.google.com/search?q=package%20sazedR>

Similarly, phase data can be converted to a longer sampling time by downsampling (omitting intermediate points). That downsampling can be accomplished by another `downsample()` function<sup>7</sup>:

```
> pavg<-function(x,af)
+ {
+ seed<-c(TRUE,rep(FALSE,af-1))
+ cont<-
+ rep(seed,ceiling(length(x)/af))
+ [1:length(x)]
+ return(x[which(cont)])
+ }
> x
[1] 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20
> pavg(x,3)
[1] 1 4 7 10 13 16 19
```

## 2.1 Clock Data

The data for analyzing the stability of a frequency source is a time series comprising a set of equally-spaced phase (time) or fractional frequency values at some sampling time  $\tau$ . These data are often accompanied by timetags<sup>8</sup>. Phase data is generally preferred, and can be converted to fractional frequency data by their 1<sup>st</sup> differences divided by  $\tau$ . Conversely, fractional frequency data can be converted to phase data (with an arbitrary constant, generally zero) by numerical integration. Those conversions can be performed quite simply in R code, e.g., `freq=diff(phase)/tau` and `phase=diffinv(freq)*tau`. For example:

```
> diff(c(2,3,5,18,4,6,4))
[1] 1 2 13 -14 2 -2
> diffinv(c(1,2,13,-14,2,-2))
[1] 0 1 3 16 2 4 2
```

where the latter values differ from the original ones by the constant 2.

<sup>7</sup>See: <http://evertqin.blogspot.com/2011/03/simple-downsample-function-for-vectors.html>

<sup>8</sup> Modified Julian Data (MJD) timetags are commonly used for time and frequency data, usually including a fractional part having a 1-second resolution.

## 2.2 Power Law Noise

The phase and frequency fluctuations of a frequency source can often be well-described by one or more power law noise processes having a spectral characteristic of  $S(f)=h_{\alpha}f^{\alpha}$  where  $\alpha$  is the (usually integer) power law exponent ranging from -2 to +2 for noise processes from Random Walk FM through White PM (see Figure 1).

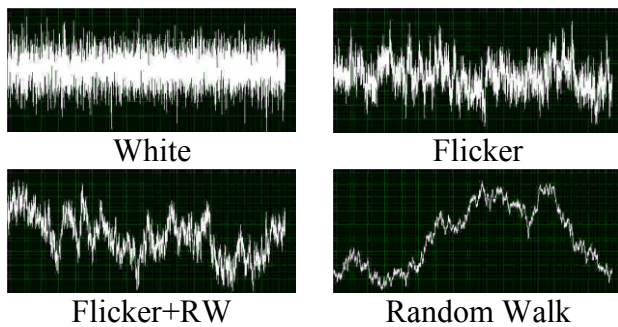


Figure 1. Common Types of Power Law Noise

The various noise types can apply to either phase or frequency data. Note that power law noise doesn't necessarily have to have an integer exponent – mixtures of noise types are possible.

Because the white, flicker, and random walk noise types can apply to either phase or frequency data, these three noise types, along with phase-frequency conversions, will cover all five common noises. Note that those conversions change the exponent by 2, and that W FM noise is the same as RW PM (both  $\alpha=0$ ).

## 2.3 Power Law Noise Simulation

It is frequently useful to simulate a set of power law noise as an analysis sample or to model a frequency source. There are several ways to accomplish this, and, in R, one is provided by the `TK95(N, alpha)` function of the CRAN ‘RobPer’ package<sup>9</sup> based on the power law noise generation method of Reference [6]. Note that the alpha argument has the opposite sign as the symbol  $\alpha$  is commonly used for frequency stability analysis(see Table 1):

<sup>9</sup> See: <https://repo.bppt.go.id/cran/web/packages/RobPer/RobPer.pdf>.

Noise Type	Color	$\alpha$		TK95 alpha
		Phase	Freq	
White	White	2	0	0
Flicker	Pink	1	-1	1
Random Walk	Brown	0	-2	2

For example, 2000 points of simulated flicker noise can be generated and plotted (see Figure 2) with the commands<sup>10</sup>:

```
> #Generate power law noise with expo
+ nent alpha=1.0
> y<-TK95(N=2000, alpha=1.0)
> t<-seq(along=y)
> #Show time series:
> plot(t,y,type="l",main="Power Law
+ Noise",xlab="t",ylab="y")
```

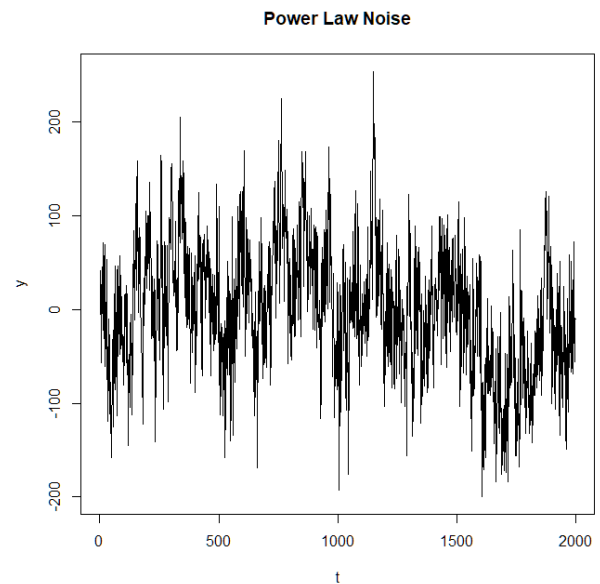


Figure 2. Simulated Flicker Noise

The noise exponent need not be an integer, and the “colored” noise can be considered to be either phase or frequency data. The resulting noise is only close to having a zero mean and its variance is not known, so these attributes have to be adjusted and scaled as desired (for example,  $\mu=-0.866$  and  $\sigma_y(1)=33.07$  for the above per the Stable32 Statistics screen of Figure 3).

<sup>10</sup> Watch out for “ (NG) versus " (OK) quotes in R code.

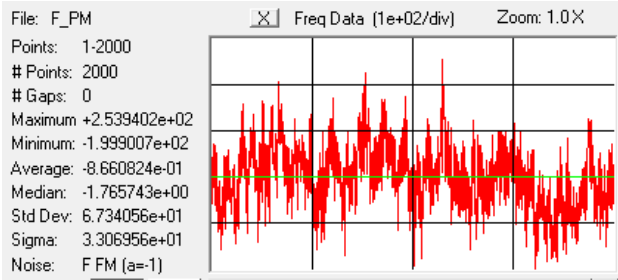


Figure 3. Statistics for Simulated F FM Noise

For the following examples, it is useful to generate and save a set of 5 power law noise data files each containing 4096 points of the noise types and file names shown in Table 2:

Table 2. Power Law Noise Data Files			
Data Type	Noise Type	$\alpha$	File Name
Phase	White PM	+2	W_PM.dat
	Flicker PM	+1	F_PM.dat
Frequency	White FM	0	W_FM.dat
	Flicker FM	-1	F_FM.dat
	RW FM	-2	RW_FM.dat

The generated flicker PM data can be saved to disk with the following command:

```
> write(y, "C:\\Data\\F_PM.dat", 1)
```

A complete function to generate a certain number of points of noise having a certain power law exponent, zero mean and a certain Allan variance (see Appendix 1) would call TK95(), calculate its ADEV, scale it accordingly and then remove its average value.

## 2.4 Data Plots

Data plots are an important analysis tool. The default R data plot format using simply `plot(x)` produces a very reasonable result and is quite adequate for general purposes, producing an x-y scatter plot with points denoted by circles as shown in Figure 4.

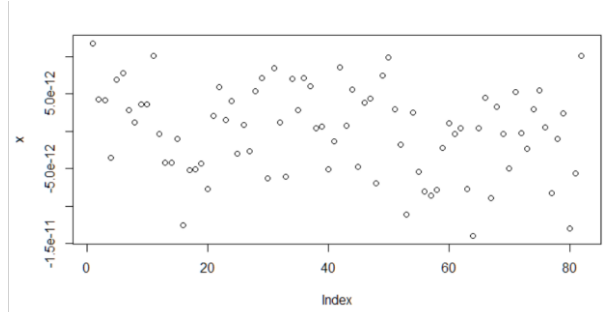


Figure 4. Scatter Style Plot

But better formatting is also quite easy. One improvement (see Figure 5) is to use lines without points for phase data, `plot(x, type="l")`, and steps for frequency data (see Figure 6), `plot(y, type="s")`, to indicate that they represent an average over the tau interval, and to distinguish the two (at least when there are relatively few points):

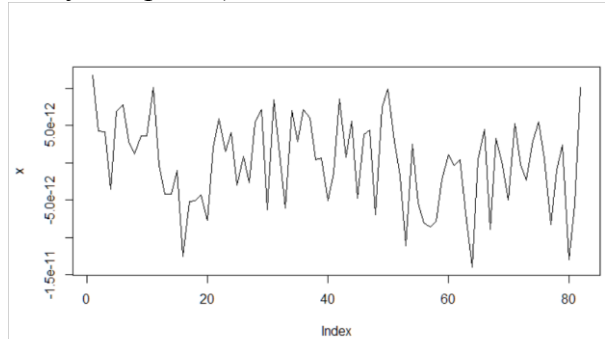


Figure 5. Line Style Plot

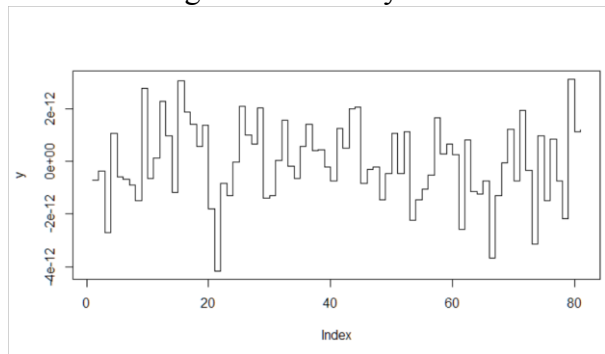


Figure 6. Step Style Plot

Many fancier plotting options are available, especially for presentation graphics.

### 3 TIME DOMAIN FREQUENCY STABILITY MEASURES

The main measures of domain frequency stability in the time domain are a number of specialized variances designed to handle divergent noise types (the standard variance doesn't converge for  $\alpha < 0$ ), distinguish between white and flicker PM noise, ignore linear frequency drift, provide higher confidence and support larger averaging factors. These statistics are described in Reference [5].

#### 3.1 Allan Variance

The Allan variance, AVAR, (and its square root, the Allan deviation, ADEV) is the most common time domain measure of frequency stability. Its calculation is supported in R by the Allan Variance Analysis 'allanvar' package<sup>11</sup> that is freely-available under the GPL-2 license<sup>12</sup> in the CRAN<sup>13</sup> repository. This package contains several functions for calculating and plotting ADEV, with an emphasis on describing sensors and gyros. A sample ADEV calculation and plot<sup>14</sup> is shown in Figure 7.

```
library(allanvar)
#Load data
data(gyroz)
#Allan variance computation using avar
avgyroz <- avar(gyroz@.Data, frequency(gyroz))
plotav(avgyroz)
```

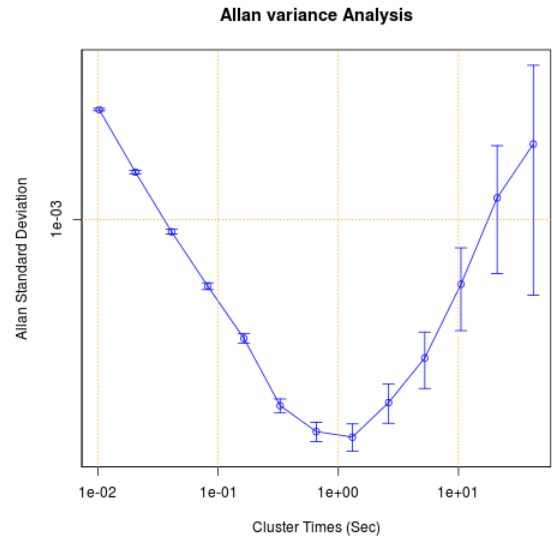


Figure 7. allanvar Calculation and Plot

The "avar" package<sup>15</sup> also provides several Allan variance-related functions, as shown in Figure 8. Its emphasis is also on sensors and gyros. In particular, the avar() function implements the overlapping AVAR for frequency data, and the avari() function does so for phase data. Those functions perform a full AVAR run at octave-spaced points, and show the results in a table along with the associated error bars. Note that one must take the square root of the displayed av values to obtain the ADEV. The 'allanvar' demo is shown in Appendix 1.

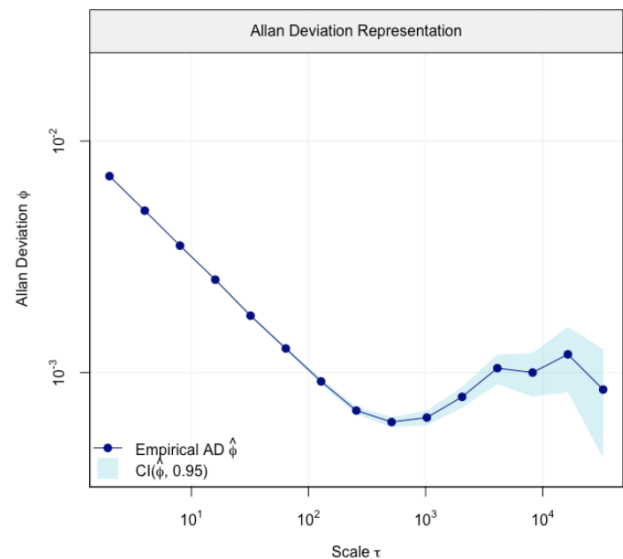


Figure 8. avar Calculation and Plot

<sup>11</sup> See: <http://www2.uaem.mx/r-mirror/web/packages/allanvar/allanvar.pdf> .

<sup>12</sup> See [Wikipedia: GNU General Public License](https://en.wikipedia.org/wiki/GNU_General_Public_License).

<sup>13</sup> Comprehensive R Archive Network.

<sup>14</sup> See: <https://rdrr.io/cran/allanvar/man/plotav.html>.

<sup>15</sup> See: <https://cran.r-project.org/web/packages/avar/avar.pdf>.

This plot includes a very nice display of the ADEV confidence limits.

In considering the Allan (and related) variances one needs to keep in mind the following:

1. Several types of variance are used for frequency stability analysis, as follows:
  - Allan Variance, AVAR
  - Modified Allan Variance, MVAR
  - Time Variance, TVAR
  - Hadamard Variance, HVAR
  - Total Variance, TVAR
  - Theo1
2. One should distinguish between a variance (e.g., AVAR) and its square root, a deviation (e.g., ADEV). The former is often used in a general sense, while the latter is almost always the form actually used.
3. One should distinguish between the expected value of statistics like AVAR and the computational means used to estimate them. For example, one can estimate AVAR with either non-overlapping or overlapping samples, where the latter is generally preferred because of its higher confidence.
4. The confidence level of a statistic (its error bars, the range of an estimate around its nominal value at a certain confidence factor) depends on the statistic, its estimation method, the number of samples used for the estimate, and the properties of the data (noise) used in the estimation.
5. The confidence level of an estimate of a variance is generally based on the number of equivalent  $X^2$  degrees of freedom that apply, a quantity that can be determined either analytically or by Monte-Carlo simulation.
6. Clock stability data can be in the form of either phase (time) variations,  $x(t)$  in units of seconds or dimensionless fractional frequency data,  $y(t) = \Delta f/f_0 = (f - f_0) / f_0$ , where  $f_0$  is the nominal frequency.
7. Since frequency is the rate of change of phase, frequency data can be obtained from phase data by taking 1<sup>st</sup> differences, while phase data can be obtained from frequency data by numerical integration.
8. The fluctuations of a frequency source are often modeled as an integer power law process in the frequency domain  $S(f)=h_\alpha f^\alpha$

where  $\alpha$  is the power law exponent ranging from -2 to +2 for noise processes from Random Walk FM through White PM (see below). Examples of the most common noise types was shown in Figure 1.

9. The power law noise exponent determines the slope of a log ADEV versus log tau plot for three common variance types as shown in Table 3.
10. Always follow R.W. Hamming's admonition that "the purpose of computing is insight, not numbers" [4].

Noise Type	$\alpha$	Stability Plot Noise Slope		
		ADEV	MDEV	TDEV
W PM	+2	-1	-3/2	-1/2
F PM	+1	-1	-1	0
W FM	0	-1/2	-1/2	+1/2
F FM	-1	0	0	+1
RW FM	-2	+1/2	+1/2	+3/2

### 3.2 Other Variances

This author knows of no R packages that support other variance types such as the Modified and Hadamard variances. Nor are the two Allan variance R packages ideal for frequency stability analysis. It therefore is desirable to develop an R package to provide a more complete suite of time domain frequency stability analysis tools, preferably written in C for speed and efficiency (see Reference [7]). We take a few steps toward that goal herein (see Appendices 2 and 7).

### 3.3 Autocorrelation

R makes obtaining the autocorrelation sequence of a time series very easy: just type `acf(z)` to show it for time series  $z$ . You can immediately see the difference between uncorrelated white noise and a sample of more divergent flicker or random walk noise as indicated by their lag 1 autocorrelation scatter plots with `lag.plot(z)` as shown in Figure 9.

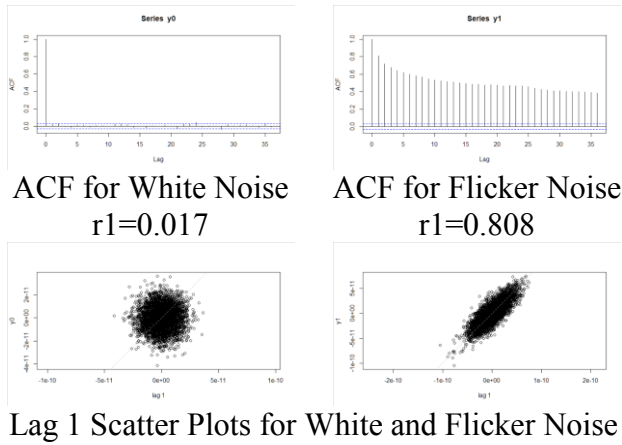


Figure 9. White and Flicker Noise  
ACF and Lag 1 Scatter Plots

The lag 1 value can be used to identify the power lay noise type (see the `nid()` function in Appendix 2), or to show data quantization.

### 3.4 Histograms

It is occasionally helpful to examine phase or frequency data in a histogram. Most such data is dominated by random noise having the familiar bell-shaped Gaussian distribution as shown in Figure 10. However a histogram can show when the data are less normal, perhaps bimodal or highly quantized. It is easy to produce a histogram in R, simply execute `hist(z)`, where `z` is the name of a vector of phase or frequency data.

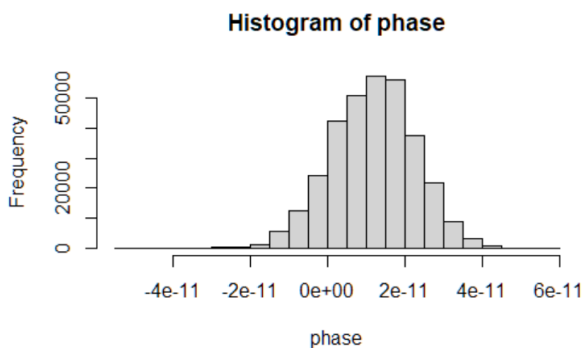


Figure 10. Histogram of White Gaussian Phase Noise

## 4 FREQUENCY DOMAIN FREQUENCY STABILITY MEASURES

The main measures of frequency stability in the frequency domain are a number of specialized power spectral densities (PSD),  $S_x(f)$ ,  $S_\phi(f)$ , and  $\mathcal{L}(f)$  for phase data and  $S_y(f)$  for frequency data.

### 4.1 Raw Periodogram

Obtaining a raw periodogram in R is as simple as typing `spectrum(z)` where `z` is the name of a time series data vector. For example, Figure 11 shows one for a set of phase data in seconds<sup>16</sup>:

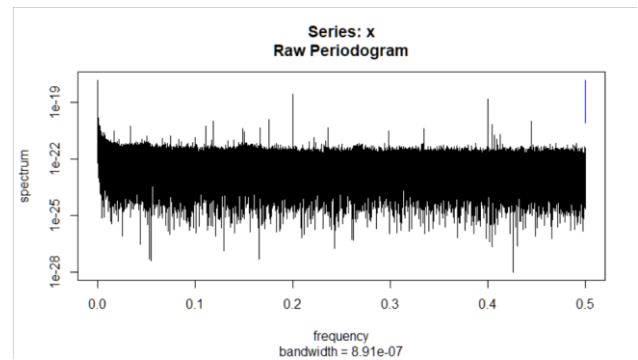


Figure 11. Raw Periodogram

These data represent the nominally white PM noise floor of a phase measuring system, with an ADEV of about  $1.4e-11$  at their 1-second sampling interval. The phase noise does indeed look quite white (the estimated  $\alpha$  is  $+1.7$ ) and the ADEV does closely follow a  $\tau^{-1}$  characteristic. The scale factor of the spectral intensity appears to take into account of the noise bandwidth because it nearly agrees with that of the Figure 12 Stable32  $S_x(f)$  plot in  $\text{sec}^2/\text{Hz}$ . The Fourier frequency scales are both 0.5 Hz full scale<sup>17</sup>.

<sup>16</sup> Note that the default `spectrum()` function uses a 10% cosine taper, so it is not entirely “raw”. One can eliminate it with `taper=0` in the call; there is hardly any difference in the resulting spectrum.

<sup>17</sup> The # of FFT points is probably 524,288, the next power of 2 above the # of data points, so, with the 1-second sampling time, the Fourier bin size is about 1.9  $\mu\text{Hz}$ , which, for a rectangular window, should also be the noise bandwidth. The plot annotation is different.



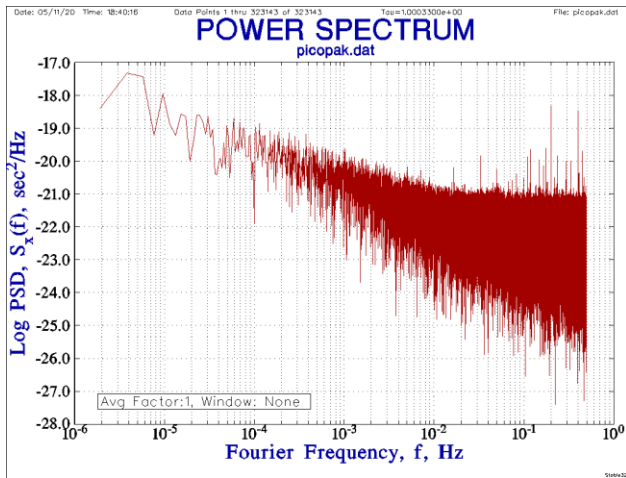


Figure 12. Stable32  $S_x(f)$  Plot

To further process the spectrum, we need to extract and scale the Fourier frequency and PSD results as numeric vectors:

```
> s<-spectrum(phase)
> freq<-s$freq
> psd<-2*s$spec
```

The Fourier frequency is in cycles per sampling period, which, for the 1-second sampling, makes it in Hz. Scaling will be needed for other sampling rates. To scale the spectrum so that its total area is equal to the time series variance, the PSD values need to be multiplied by  $2^{18}$ . We can then re-plot the spectrum as desired. For example:

```
> plot(log10(freq), log10(psd),
type="l")
```

This plot (Figure 13) closely resembles the unwindowed Stable32  $S_x(f)$  PSD plot. However it should be noted that the `spectrum()` function, by default, prewhitens the time series data by removing any mean and linear trend before computing the spectrum (those are not a factor in this case).

<sup>18</sup> The mean value for the  $2^{\text{nd}}$  half of the PSD points is about  $2.5 \times 10^{-22}$ . The approximate area of the flat white noise spectrum,  $2.5 \times 10^{-22} * 0.5 = 1.25 \times 10^{-22}$  is about equal to the variance of the phase noise,  $(1.4 \times 10^{-11})^2 = 2.0 \times 10^{-22}$ . If a domain conversion is conducted for W PM with  $ADEV=1.4 \times 10^{-11}$ , the resulting  $S_x(f=1 \text{ Hz})=1.3 \times 10^{-22}$  for a system  $BW=0.5 \text{ Hz}$ .

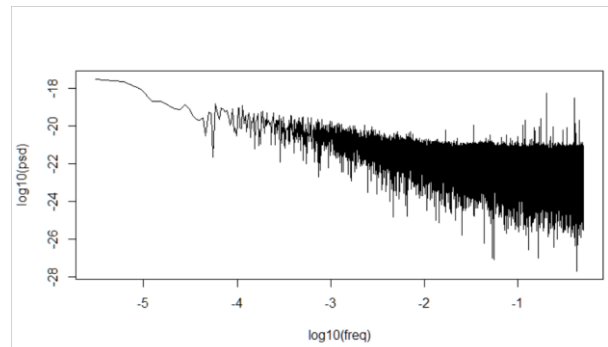


Figure 13. PSD Plot with Log Scales

One can eliminate the large amplitude low frequency leakage-induced components as shown in Figure 14.

```
plot(log10(freq[-(1:100)]),
log10(psd[-(1:100)]), type="l")
```

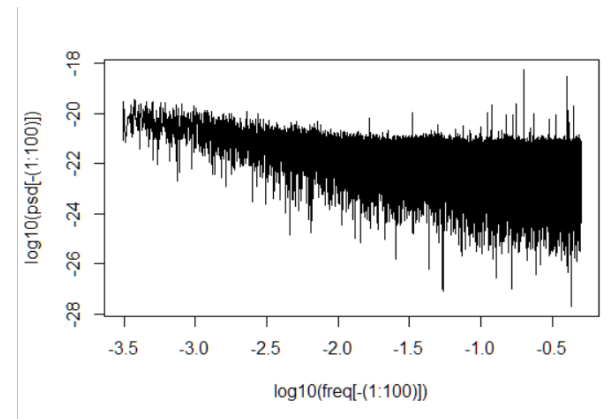


Figure 14. PSD Plot with Low Fourier Frequency Components Removed

The  $S_x(f)$  PSD data can be converted into more commonly-used  $\mathcal{L}(f)$  values using the relation  $\mathcal{L}(f) = 10 \log_{10}[2 \cdot \pi^2 \cdot \nu_0^2 \cdot S_x(f)]$  in dBc/Hz where  $\nu_0$  is the RF carrier frequency, as shown in Figure 15.

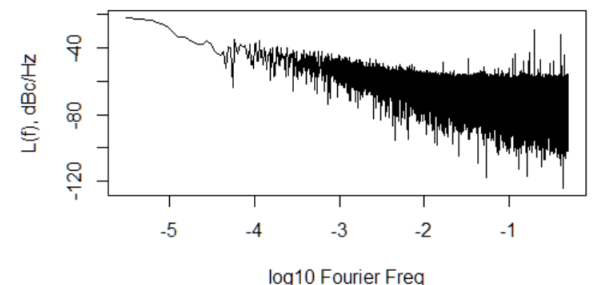


Figure 15.  $\mathcal{L}(f)$  Plot

## Smoothed Periodogram

The spectrum can be smoothed with the `span` argument of the `spectrum()` function. Repeating the analysis with `span=20` results in the PSD plot of Figure 16.

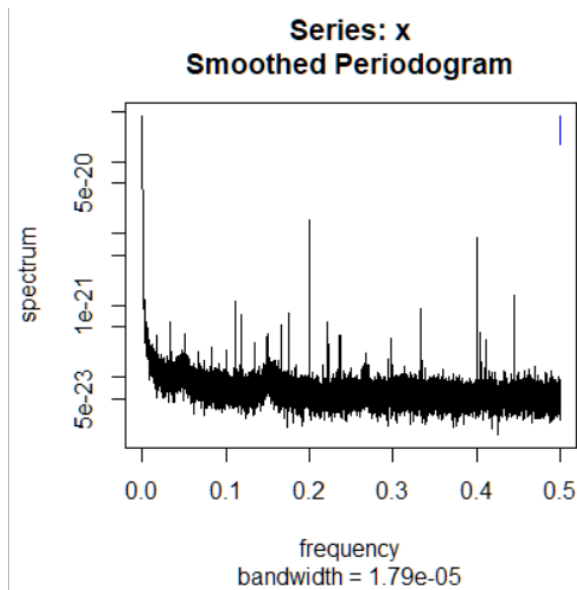


Figure 16. Smoothed Periodogram

And, after reprocessing to use log plot scales, it is shown in Figure 17.

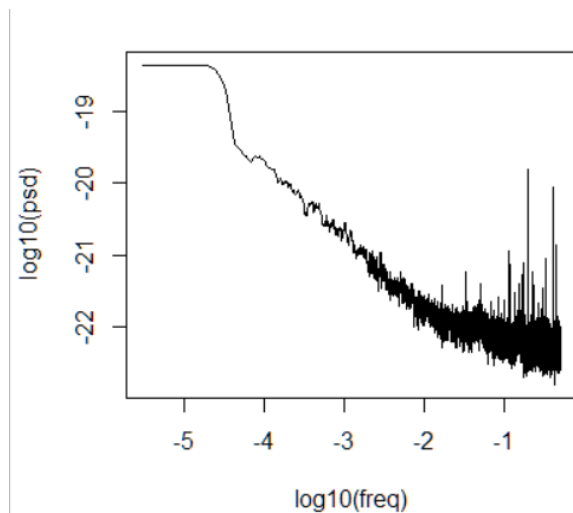


Figure 17. Smoothed Periodogram with Log Scales

## 4.2 Power Spectral Density Analysis

Reference [2] contains many examples of power spectral analysis using R<sup>19</sup>, including several involving atomic clocks. It describes many ways to make a spectral analysis have less variance (more consistency) and less bias.

## 5 OTHER TOPICS

We conclude with two miscellaneous topics. Timetags are often associated with phase or frequency data, and they are an important way to document it. Documentation is vital for any significant frequency stability analysis, as it also is for custom R functions and the R programming environment in general.

### 5.1 Timetags

MJD timetags<sup>20</sup> are often used with clock data, generally in the 1<sup>st</sup> column of a row of data. There are lots of ways to read a 2-column data file with timetags. For example: `d<-read.table("C:\\Data\\clock.dat")` will read it into the data.frame table `d`, and `mjd<-d[1]` and `phase<-d[2]` will separate the timetags and data into two vectors. One line of the table can be printed with `d[n,]` where `n` is the line #:

```
> d[996,]:
           v2                v2
996 58150.8244382 -9.39369201665e-12
```

And the table data can be plotted with `plot(d)` as shown in Figure 18.

<sup>19</sup> R code is available for download, and can be pasted into the R console command line. It can be hard to make a snippet of code run by itself, but just looking at it can be very informative.

<sup>20</sup> The Modified Julian Date (MJD) is based on the astronomical Julian Date, the # of days since noon on January 1, 4713 BC. The MJD is the Julian Date - 2,400,000.5; it starts at zero at midnight on November 17, 1858.

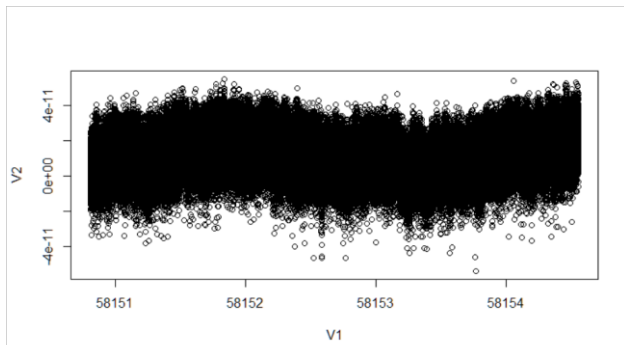


Figure 18. Data Plot with MJD Time Scale

The `mjd` and phase values can be put into separate vectors with `mjd<-d[,1]` and `phase<-d[,2]`.

The MJD timetags could be manipulated to show the x-scale as hours, etc.

One can obtain the current MJD from the computer clock with:

```
(as.numeric(Sys.time())/86400)+
40587.
```

## 5.2 Documentation

The R program is well-documented, including on-line manuals and tutorials, and several books<sup>21</sup>, and the additional packages in the CRAN repository are generally well-documented also. The R help system (? followed by a package or function name) is quite effective. The R computing environment lends itself to creating custom functions, either in an ad hoc fashion or as a organized collection. They should usually be brief, perform a single operation, and have a short intuitive name<sup>22</sup>. Functions can be saved as a file and loaded with the `source()` command. It is important to document your functions so that one can recall their purpose and arguments, perhaps following the format used by the CRAN repository.

<sup>21</sup> For example, [The R Book](#).

<sup>22</sup> See: <https://nicercode.github.io/guides/functions/>

## 6 AN R PACKAGE FOR FREQUENCY STABILITY ANALYSIS

An example of an R package for frequency stability analysis is given in Appendix 6, as described in Appendices 2 through 4. Appendix 5 contains some R code for regression analysis and modeling of phase and frequency data.

## 7 CONCLUSIONS

The R programming environment can be a useful tool for frequency stability analysis. While less suitable than a specialized application such as Stable32 for most analysis work, R is particularly effective for academic study and evaluation of new techniques and algorithms with immediate feedback provided by its interpreted language.

## ACKNOWLEDGMENT

The author acknowledges excellent work of the R Core Team and the CRAN repository in supporting the R language and its computing environment, as well as other contributors of R code.

He also recognizes the important work of many contributors to the field of frequency stability analysis.

## ACRONYMS AND ABBREVIATIONS

ACF	Autocorrelation Function
ADEV	Allan Deviation
AVAR	Allan Variance
CRAN	Comprehensive R Archive Network
FFT	Fast Fourier Transform
F FM	Flicker Frequency Modulation
F PM	Flicker Phase Modulation
FW FM	Flicker Walk Frequency Modulation
FW PM	Flicker Walk Phase Modulation
HDEV	Hadamard Deviation
HVAR	Hadamard Variance
IEEE	Institute of Electrical and Electronic Engineers
MAD	Median Absolute Deviation
MJD	Modified Julian Date
NIST	National Institute of Standards and Technology
PM	Phase Modulation
PSD	Power Spectral Density
R	The R Programming Language
RF	Radio Frequency
RW FM	Random Walk Frequency Modulation
RW PM	Random Walk Phase Modulation
RR FM	Random Run Frequency Modulation
RR PM	Random Run Phase Modulation
SSB	Single Sideband
STS	Short Term Stability
TDEV	Time Deviation
TVAR	Time Variance
T&F	Time and Frequency
UFFC	Ultrasonics Ferroelectrics and Frequency Control
W FM	White Frequency Modulation
W PM	White Phase Modulation

28 on p. 26 the tau exponent should be 2, not 3. The Overlapping Samples section # should be 5.2.3. And most of the page #s in the index are wrong.

6. J. Timmer and M. König, “On Generating Power Law Noise”, *Astronomy and Astrophysics*, Vol. 2.3, pp. 1-4, March 1995.
7. R Core Team, [Writing R Extensions](#), 2018.

## REFERENCES

1. [The R Project for Statistical Computing](#).
2. D.B. Percival and A.T. Walden, [Spectral Analysis for Univariate Time Series](#), ISBN 978-1-107-02814-2, Cambridge University Press, 2020.
3. M.J. Crawley, [The R Book](#), ISBN 978-0-470-51024-7, John Wiley & Sons, Ltd., 2007. Note that this link provides a free download of this entire book (the PDF format is  $\approx 25$  MB).
4. R.W. Hamming, [Numerical Methods for Scientists and Engineers](#), ISBN 007025887-2, McGraw-Hill Book Company, 1973.
5. W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#). Please note that there are several typos in SP1065: Eq. 9 on p.15 is missing brackets around the inner summation – use Eq. 11 instead. Eq. 18 on p. 20 has a spurious m in the denominator of its leading term. In Eq.

## Appendix 1

### Package 'allanvar' Demo: demo(allanvar)

```
> demo(allanvar)

      demo(allanvar)
      ---- ~~~~~~

Type <Return> to start :

> #Loading values
> data(gyroz)

> #Allan variance computation using avar
> avgyroz <- avar(gyroz@.Data[1:1000], frequency(gyroz))
[1] "Calculating..."

> plotav(avgyroz)
Waiting to confirm page change...

> abline(1.0+log(avgyroz$time[1]), -1/2, col="green", lwd=4, lty=10)
> abline(1.0+log(avgyroz$time[1]), 1/2, col="green", lwd=4, lty=10)
> legend(0.11, 1e-03, c("Random Walk"))
> legend(2, 1e-03, c("Rate Random Walk"))

> #Allan variance computation using avarn
> avngyroz <- avarn(gyroz@.Data[1:1000], frequency(gyroz))
[1] "Calculating..."

> plotav(avngyroz)
Waiting to confirm page change...

> abline(1.0+log(avngyroz$time[1]), -1/2, col="green", lwd=4, lty=10)
> abline(1.0+log(avngyroz$time[1]), 1/2, col="green", lwd=4, lty=10)
> legend(0.11, 1e-03, c("Random Walk"))
> legend(2, 1e-03, c("Rate Random Walk"))

> ##Allan variance computation using avari
> ##Simple integration of the angular velocity
> igyroz <- cumsum(gyroz@.Data[1:1000] * (1/frequency(gyroz)))

> igyroz <- ts(igyroz, start=c(igyroz[1]), delta=(1/frequency(gyroz)))

> avigyroz <- avari(igyroz@.Data, frequency(igyroz))
[1] "Calculating..."

> plotav(avigyroz)
Waiting to confirm page change...

> abline(1.0+log(avigyroz$time[1]), -1/2, col="green", lwd=4, lty=10)
> abline(1.0+log(avigyroz$time[1]), 1/2, col="green", lwd=4, lty=10)
> legend(0.11, 1e-03, c("Random Walk"))
```

```

> legend(2, 1e-03, c("Rate Random Walk"))

> #Plotting all
> plot (avgyroz$time,sqrt(avgyroz$av),log= "xy", xaxt="n" , yaxt="n", type="l",
col="blue", xlab="", ylab="")
Waiting to confirm page change...

> lines (avngyroz$time,sqrt(avngyroz$av), col="green", lwd=1)

> lines (avigyroz$time,sqrt(avigyroz$av), col="red")

> axis(1, c(0.001, 0.01, 0.1, 0, 1, 10, 100, 1000, 10000, 100000))

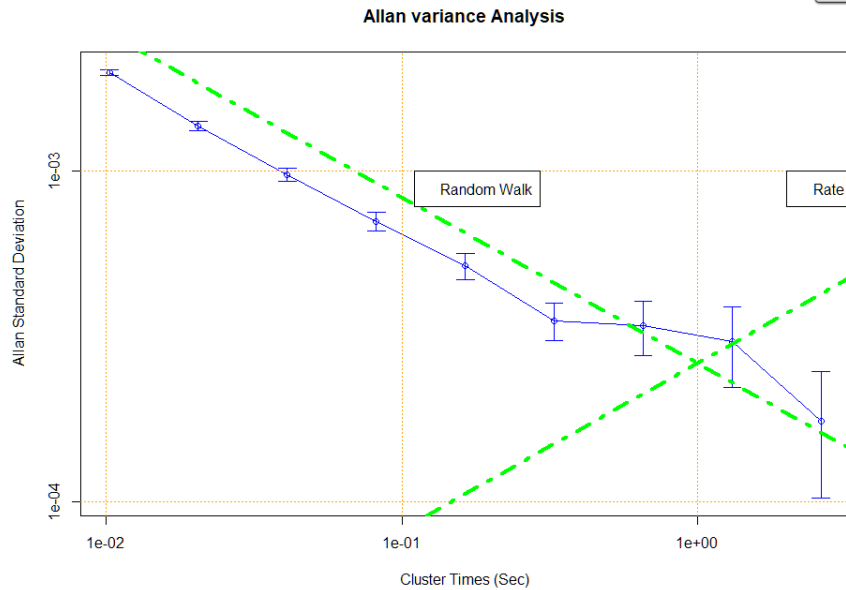
> axis(2, c(0.00001, 0.0001, 0.001, 0.01, 0.1, 0, 1, 10, 100, 1000, 10000))

> grid(equiloggs=TRUE, lwd=1, col="orange")

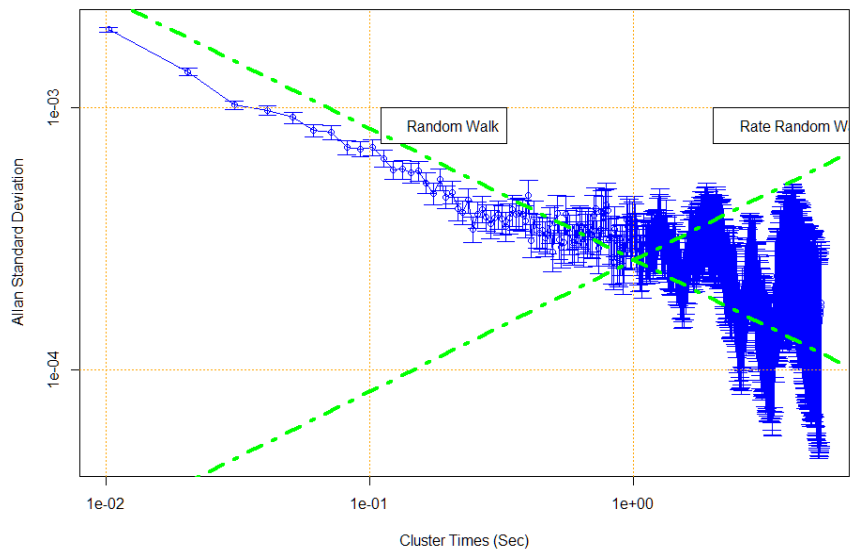
> title(main = "Allan variance Analysis Comparison", xlab = "Cluster Times (Sec)",
ylab = "Allan Standard Deviation (rad/s)")

> legend(1, 1e-03, c("GyroZ (avar)", "GyroZ (avarn)", "GyroZ (avari)"), fill =
c("blue", "green", "red"))
Warning messages:
1: In plot.xy(xy.coords(x, y), type = type, ...) :
  "log" is not a graphical parameter
2: In plot.xy(xy.coords(x, y), type = type, ...) :
  "log" is not a graphical parameter
3: In plot.xy(xy.coords(x, y), type = type, ...) :
  "log" is not a graphical parameter
>

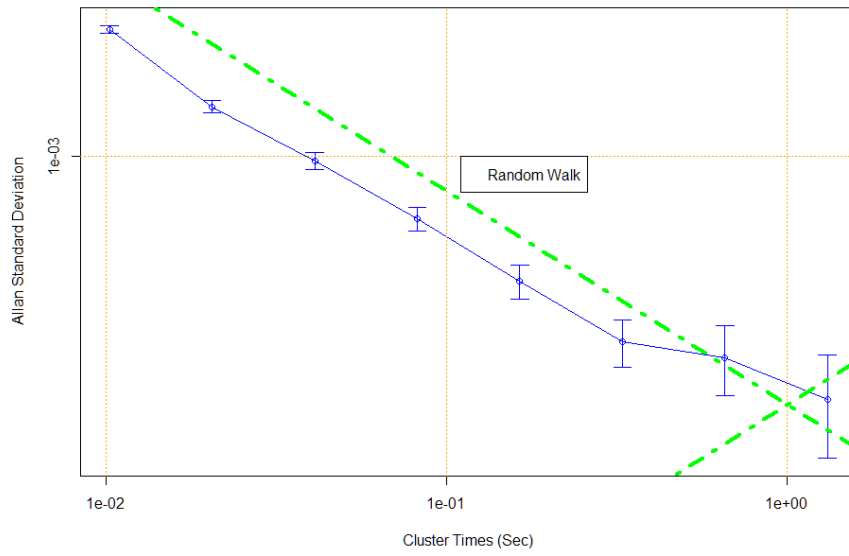
```



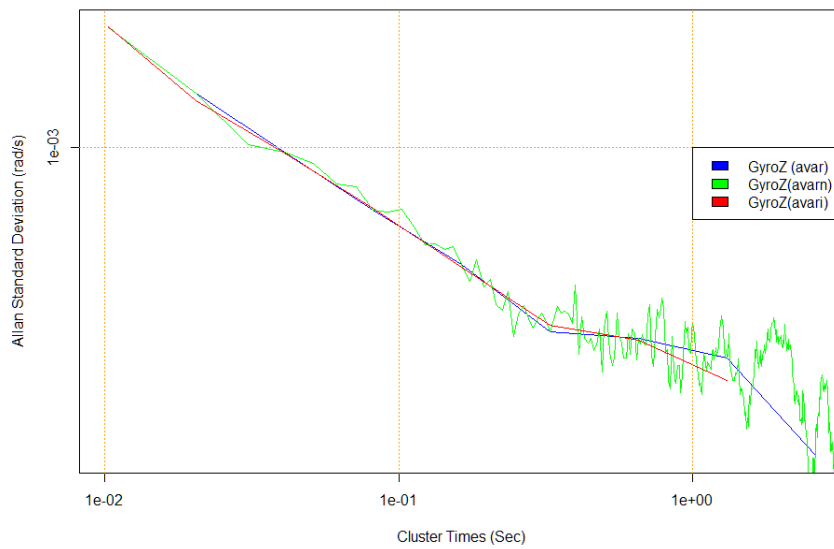
Allan variance Analysis



Allan variance Analysis



Allan variance Analysis Comparison



## Appendix 2

### R Code for Basic Frequency Stability Analysis Functions

The R code for these functions can be copied to the file fsa.R from Appendix 6.

Basic Frequency Stability Analysis Functions in R package fas.R		
Function	Description	Remarks
pavg	Average Phase Data	Downsample by certain averaging factor
favg	Average Frequency Data	Average by certain averaging factor
ptof	Phase to Frequency Conversion	First differences
ftop	Frequency to Phase Conversion	Numerical integration
noise	Generate Power Law Noise	TK95 method for certain ADEV
nid	Power Law Noise Identification	Uses Lag 1 Autocorrelation
bs	Show Basic Statistics	List basic statistics and show data plot
co	Count Outliers	Using MAD limit
padev	Calculate Allan Deviation for Phase Data	At basic sampling interval
fadev	Calculate Allan Deviation for Frequency Data	At basic sampling interval
poadev	Calculate Overlapping ADEV for Phase Data	At selected averaging factor
foadev	Calculate Overlapping ADEV for Freq Data	At selected averaging factor
adevrun	Calc overlapping ADEV for an Octave Run	At range of octave-s[aced taus
pmdev	Calculate Mod Allan Deviation for Phase Data	At basic sampling interval – TDEV also
phdev	Calculate Hadamard Deviation for Phase Data	At basic sampling interval
fhdev	Calculate Hadamard Deviation for Freq Data	At basic sampling interval
theo1	Calculate the Thêo1 Statistic for Phase Data	At selected averaging factor
psd	Calculate and Plot a Power Spectral Density	PSD may be smoothed, optional log scales

Additional packages required:

allanvar for avar() and avari()

RobPer for TK95()

zoo for rollapply()

A 1000-point data set used as a test suite for frequency stability analysis statistics can be downloaded from: [https://www.wriley.com/tst\\_suit.dat](https://www.wriley.com/tst_suit.dat), and the results for a collection of such statistics will be found at: <http://www.wriley.com/paper1ht.htm>, see References [20] and [21] therein. After downloading the test suite data, it can be read into R with: `> ts<-scan("C:\\Data\\test_suite.dat")` where the file name is edited appropriately for where it was stored.

If you prefer, you can generate the test suite yourself with the following R code:

```
# Generate the 1000-point test suite
# Initializations
ts<-1:1000
ts[1]=1234567890
# Generate data
for(i in 2:1000)
{
  ts[i]=(16807*ts[i-1])%%2147483647
}
```

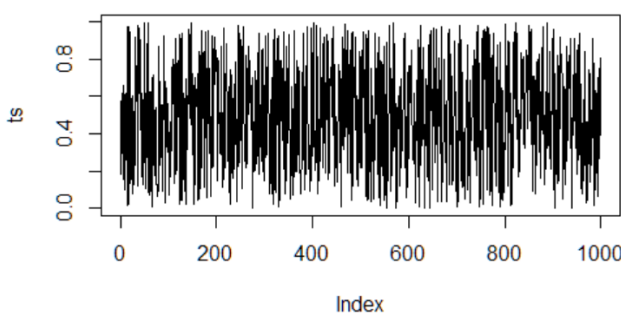
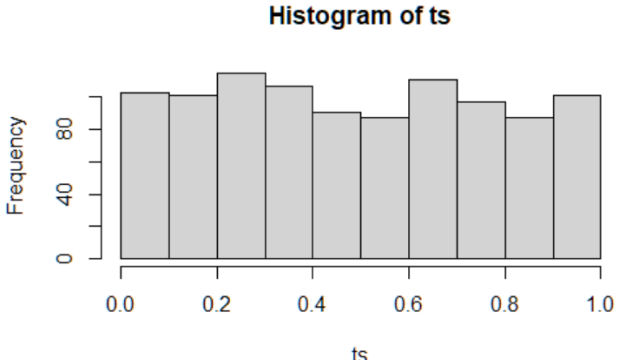


```

# Scale data
for(i in 1:1000)
{
  ts[i]=ts[i]/2147483647
}
# Optionally show and plot the generated data
# Increase # R display digits to max
# options(digits=22)
# ts
# plot(ts, type="l")
# Restore normal # digits
# options(digits=7)
# Convert freq data to phase data
tsp<- diffinv(ts)

```

The test data can be confirmed as follows:

<pre> &gt; length(ts) [1] 1000 &gt; ts[1] [1] 0.5748905 &gt; max(ts) [1] 0.9957453 &gt; min(ts) [1] 0.00137176 &gt; mean(ts) [1] 0.4897745 &gt; median(ts) [1] 0.4798849 </pre>	
<p>A histogram can be produced with <code>hist(ts)</code>. Notice that these test data are uniformly (not Gaussian) distributed. That does not affect their usefulness for testing frequency stability analysis methods.</p>	

Examples of tests with these test data on some of the R functions are as follows:

<pre> &gt; fadev(ts) [1] 0.2922319 &gt; fhdev(ts) [1] 0.2943883 &gt; ts10&lt;-favg(ts,10) &gt; length(ts10) [1] 100 &gt; fadev(ts10) [1] 0.09965736 </pre>	<pre> &gt; tsp&lt;-ftop(ts,1) &gt; padev(tsp) [1] 0.2922319 &gt; phdev(tsp,1) [1] 0.2943883 </pre>	<pre> &gt; tsp10&lt;-pavg(tsp,10) &gt; length(tsp10) [1] 101 &gt; padev(tsp10,1) [1] 0.9965736 </pre>
--	--	---

**Description**

Function to average phase data to a larger averaging factor.

**Usage**

```
pavg(x, af)
```

**Arguments**

x	The vector of phase data to be averaged.
af	The averaging factor to be applied (default=2)

**Return Value**

The averaged phase data.

**Example**

Average a set of phase data x by an averaging factor of 10:  

```
pavg(x, 10)
```

**Reference**

R. Everett, "A simple downsample function for vectors in R", March 2011.  
<http://evertqin.blogspot.com/2011/03/simple-downsample-function-for-vectors.html>

**Code:**

```
# Function to average phase data
pavg<-function(x,af)
{
  seed<-c(TRUE,rep(FALSE,af-1))
  cont<-rep(seed,ceiling(length(x)/af))[1:length(x)]
  return(x[which(cont)])
}
```

**Description**

Function to average frequency data to a larger averaging factor.

**Usage**

```
favg(x, af)
```

**Arguments**

y            The vector of fractional frequency data to be averaged.  
af            The averaging factor to be applied (default=2).

**Return Value**

The averaged frequency data.

**Example**

Average a set of frequency data y by an averaging factor of 10:  
favg(y, 10)

**Reference**

M. Toller, T. Santos & R. Kern, “Parameter-Free Domain-Agnostic Season Length Detection in Time, R package ‘sazedR, September 2019.

<https://www.google.com/search?q=package%20sazedR>

**Code**

```
# Function to average frequency data
favg<-function(data,af=2)
{
  (return(rollapply(data,width=af,by=af,FUN=mean))
}
```

**Description**

Function to convert phase data to fractional frequency data.

**Usage**

ptof(x, tau)

**Arguments**

x            The vector of phase data to be converted.  
tau          The data sampling interval, tau (default=1).

**Return Value**

The converted fractional frequency data.

**Example**

Convert a set of phase data in seconds to dimensionless fractional frequency data for data having a sampling interval of 10 seconds:

```
ptof(x, 10)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function for phase to frequency conversion  
ptof<-function(x,af)  
{  
  return(diff(x)/tau)  
}
```

**Description**

Function to convert fractional frequency data to phase data

**Usage**

ftop(y, tau)

**Arguments**

y            The vector of fractional frequency data to be converted.  
tau          The data sampling interval, tau (default=1).

**Return Value**

The converted phase data.

**Example**

Convert a set of dimensionless fractional frequency data to phase data for data having a sampling interval of 10 seconds:

```
ftop(y, 10)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function for frequency to phase conversion  
ftop<-function(y,tau)  
{  
  return(diffinv(y)*tau)  
}
```

**Description**

Function to generate a certain number of points of noise having a certain power law exponent, zero mean and a certain Allan deviation. The data type can be either phase or frequency with a certain sampling time for the former.

**Imports**

Package RobPer required.

**Usage**

```
noise(num, exp, sigma, type, tau)
```

**Arguments**

num            The number of points to generate.  
 exp            The power law exponent (not necessarily an integer)  
               White                    0  
               Flicker                 1  
               Random Walk            2

Note that these values are not the same as those normally associated with  $\alpha$ , the power law noise exponent used in the field of frequency stability analysis.

Power Law Noise Data Files			
Data Type	Noise Type	$\alpha$	exp
Phase=0	White PM	+2	0
	Flicker PM	+1	1
Frequency=1	White FM	0	0
	Flicker FM	-1	1
	RW FM	-2	2

sigma         The desired Allan deviation at the basic sampling interval of the data.  
 type          The type of data to be generated, 0=phase, 1=frequency (default=0).  
 tau            The sampling time for the phase data (N/A for frequency data) (default=1)

**Return Value**

The resulting power law noise data.

**Example**

```
Generate 4096 points of flicker FM noise with AVAR=1e-11:
y <- noise(4096, 1, 1e-11, 1) # No tau required
Write these data to a file as frequency data:
write(y, "C:\\Data\\F_FM.frd", 1)
Read the data file into a vector:
z <- scan("C:\\Data\\F_FM.frd")
```

**Reference**

J. Timmer and M. K ning, "On Generating Power Law Noise", *Astronomy and Astrophysics*, Vol. 2.3, pp. 1-4, March 1995.

## Notes

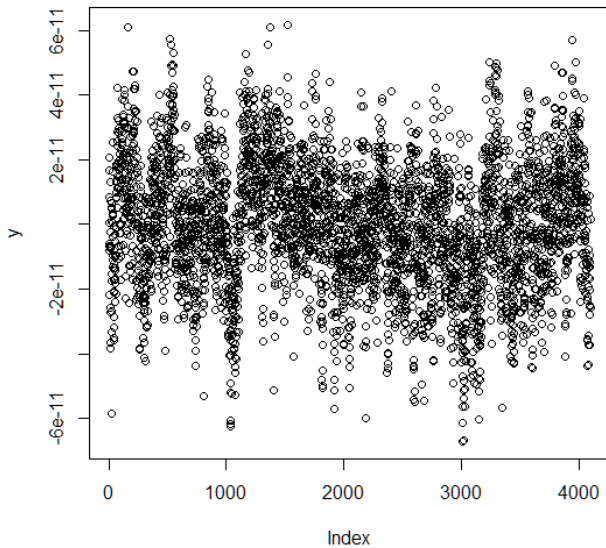
An offset can be added to the generated data with  $z=z+offset$ , and it can be scaled with  $z=z*scale$ . A linear slope can be added with: `for(i in 1:length(y)) y[i]=y[i]+i*slope` where `slope` is the desired slope per sampling interval. Generic filename extensions for phase or frequency data are typically `.dat` or `.txt`. Stable32 optionally uses the extensions `.phd` and `.frd` for phase and frequency data respectively.

## Code

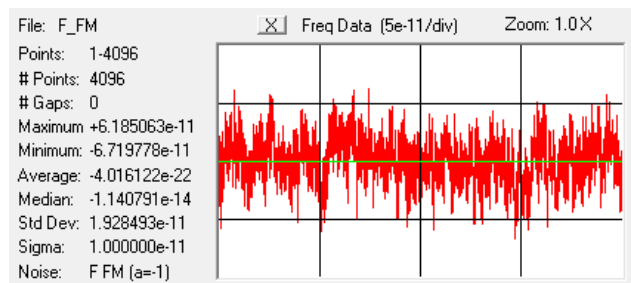
```
# Function to generate power law noise
noise<-function(num,exp,sigma,type,tau=1)
{
  z<-TK95(num,exp)
  if(type==0) d<-padev(z,tau)
  else d<-fadev(z)
  z=(z/d)*sigma
  m=mean(z)
  z=z-m
  return(z)
}
```

## Test Case

```
> y <- noise(4096, 1, 1e-11, 1)
> write(y,"C:\\Data\\F_FM.frd",1)
> plot(y)
```

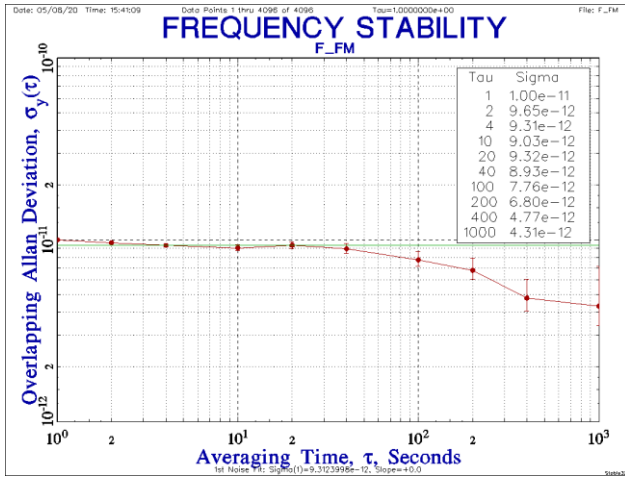


R Data Plot

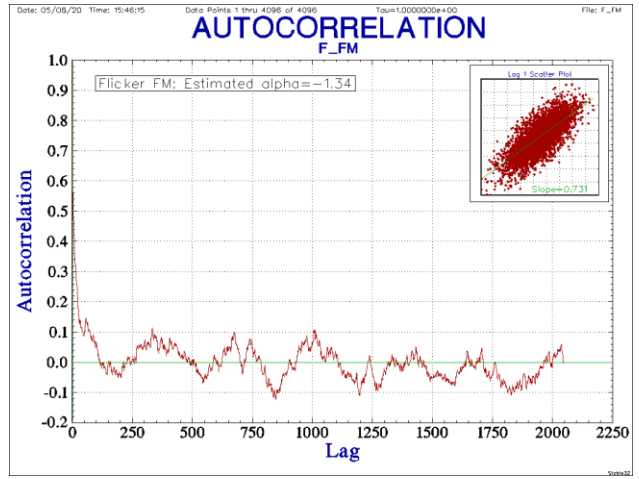


Stable32 Stats

Note the correct noise type and sigma at the basic sampling interval, with negligible offset. However the Stable32 stability plot shows significant departure from a flicker noise characteristic at longer averaging times, and the Stable32 autocorrelation plot indicates that the power law exponent,  $\alpha$ , is -1.34 rather than the requested -1.



Stable32 Stability Plot



Stable32 Autocorrelation Plot



**Description**

Function to ID the dominant type of power law noise in phase or frequency data using the lag 1 autocorrelation. It requires a minimum of about 30 data points.

**Usage**

```
nid(z)
```

**Arguments**

`z` The vector of phase or fractional frequency data to be examined.

**Return Value**

The estimated power law noise exponent,  $\alpha$ , at the basic sampling interval.

**Example**

ID the dominant noise type of a set of phase data:

```
> nid(x0)
```

```
[1] 2.01985990379
```

The nominal power law noise type is W PM

**Reference**

W.J. Riley and C.A. Greenhall, "[Power Law Noise Identification Using the Lag 1 Autocorrelation](#)," *Proceedings of the 18<sup>th</sup> European Frequency and Time Forum*, April 2004.

**Note**

The  $\alpha$  value returned by this function refers only to the data itself and not whether it represents phase or frequency information. Thus the returned value is the correct  $\alpha$  for phase data, but 2 must be subtracted from it for frequency data.

**Code**

```
# Find noise type using the lag 1 ACF method
nid<-function(z)
{
  nD=0 # Difference order
  # Save original data
  zz<-z
  # Calc lag 1 autocorrelation r1
  r1=acf(z,1, "cor",F)
  r1=r1$acf[2]
  # Find d = r1/(1+r1)
  d=r1/(1+r1)
  # If d<0.25, must apply increment operator
  if(d>0.25)
  {
    while(d>=0.25)
    {
      # Take 1st differences
      Z<-diff(z)
      nD=nD+1
      # Calc lag 1 autocorrelation r1
```

```
        r1=acf(z,1, "cor",F)
        r1=r1$acf[2]
        # Find d = r1/(1+r1)
        d=r1/(1+r1)
    }
}
# Calc alpha
alpha=-2*d -2*nD +2
# Restore original data
z<-zz
return (alpha)
}
```

**Description**

Function to show the basic statistics and plot for phase or frequency data.

**Usage**

```
bs(z, type, tau)
```

**Arguments**

z	The vector of phase or fractional frequency data to be examined.
type	The type of data to be generated, 0=phase, 1=frequency (default=0)
tau	The sampling time for the phase data (N/A for frequency data) (default=1)

**Return Value**

The basic statistics are printed and the data are plotted.

**Example**

See Appendix 4

**Reference**

None – See similar Stable32 Stats function

**Note**

The noise type ID covers the range between  $\alpha = -2$  (RW FM) to  $+2$  (W PM)

**Code**

```
# Function to show basic statistics of phase or frequency data
bs <- function(z,type,tau)
{
  print("Basic Statistics:", quote=FALSE)
  txt=paste("File =", deparse(substitute(z)))
  print(txt, quote=FALSE)if(type==0)
  {
    print("Type = Phase", quote=FALSE)
  }
  else
  {
    print("Type = Frequency", quote=FALSE)
  }
  txt=paste("Tau =", tau)
  print(txt, quote=FALSE)
  txt=paste("# Points =", length(z))
  print(txt, quote=FALSE)
  txt=paste("Max =", max(z))
  print(txt, quote=FALSE)
  txt=paste("Min =", min(z))
  print(txt, quote=FALSE)
  txt=paste("Span=", max(z)-min(z))
  print(txt, quote=FALSE)
  txt=paste("Mean =", mean(z))
  print(txt, quote=FALSE)
}
```

```

txt=paste("Median =", median(z))
print(txt, quote=FALSE)
txt=paste("MAD =", mad(z))
print(txt, quote=FALSE)
txt=paste("Std Dev =", sqrt(var(z)))
print(txt, quote=FALSE)
if(type==0) # Phase data
{
  txt=paste("Sigma =", padev(z,tau))
  print(txt, quote=FALSE)
  alpha=nid(z)
}
else # Freq data
{
  txt=paste("Sigma =", fadev(z))
  print(txt, quote=FALSE)
  alpha=nid(z)-2
}
txt=paste("Alpha =", alpha )
print(txt, quote=FALSE)
if(alpha>1.5)
{
  txt=paste("Noise = W PM")
  print(txt, quote=FALSE)
}
else if(alpha>0.5)
{
  txt=paste("Noise = F PM")
  print(txt, quote=FALSE)
}
else if(alpha>-0.5)
{
  txt=paste("Noise = W FM")
  print(txt, quote=FALSE)
}
else if(alpha>-1.5)
{
  txt=paste("Noise = F FM")
  print(txt, quote=FALSE)
}
else
{
  txt=paste("Noise = RW FM")
  print(txt, quote=FALSE)
}
plot(z)
}

```

## Drift

The least-squares linear drift of a set of fractional frequency data can be determined with the function call: `lm(y~t)` where `y` is the frequency data and `t<=1:length(y)`. It can be removed by: `for(i in 1:length(y)) y[i]=y[i]-i*slope`, where `slope` is the calculated slope.

Similarly, the least-squares quadratic drift fit for a set of phase data can be determined by: `t2<-t^2` and `lm(x~t+t2)` where `t<=1:length(x)`.

**Description**

Function to count the number of outliers in phase or frequency data.

**Usage**

co(z, limit)

**Arguments**

z                The vector of phase or fractional frequency data to be examined.  
limit            The MAD factor limit for OK data (default=5)

**Return Value**

The number of outliers that exceed the  $\pm\text{MAD}*\text{limit}$ .

**Reference**

Gernot M.R. Winkler, "[Introduction to Robust Statistics and Data Filtering](#)," Tutorial at 1993 IEEE Frequency Control Symposium, June 1993.

**Code**

```
# Function to count outliers in phase or frequency data
co <- function(z, limit=5)
{
  # Find MAD
  m=mad(z)
  # Count outliers
  n=sum(z<(-m*limit))+sum(z>(m*limit))
  return (n)
}
```

**Description**

Function to calculate the estimated ADEV of a set of phase data at its basic sampling interval.

**Usage**

```
padev(x, tau)
```

**Arguments**

x            The vector of phase data to be analyzed.  
tau          The data sampling interval, seconds (default=1).

**Return Value**

The estimated Allan deviation for the phase data at its basic sampling interval.

**Example**

Find the estimated ADEV for a set of phase data:

```
padev(x,1)
```

**Reference**

Function `avari()` in R package ‘allanvar’. This function adapts that code to calculate the ADEV at a single averaging factor. See: <https://rdrr.io/cran/allanvar/src/R/avari.R>.

**Code**

```
# Function to calculate the ADEV for phase data
padev <- function (x, tau=1)
{
  N=length(x)
  s=0
  for (i in 1:(N-2))
  {
    s = s + (x[i+2]-(2*x[i+1])+x[i])^2
  }
  av = s/(2*(tau^2)*(N-2))
  return (sqrt(av))
}
```

**Test Case**

Classic NBS Monograph 140 Annex 8.E frequency values from: B.E. Blair (Editor), Time and Frequency: Theory and Fundamentals, *NBS Monograph 140*, [Annex 8.E](#), p. 181, May 1974.

```
> nbs<-c(892,809,823,798,671,644,883,903,677)
> nbs
[1] 892 809 823 798 671 644 883 903 677
> nbsi<-diffinv(nbs)
> nbsi
[1] 0 892 1701 2524 3322 3993 4637 5520 6423 7100
> padev(nbsi,1)
[1] 91.22945
```

**Description**

Function to calculate the estimated ADEV of a set of fractional frequency data at its basic sampling interval.

**Imports**

Package RobPer required.

**Arguments**

y                    The vector of fractional frequency data to be analyzed.

**Return Value**

The estimated Allan deviation for the frequency data at its basic sampling interval.

**Example**

Find the estimated ADEV for a set of frequency data:  
fadev(y)

**Reference**

Function avar() in R package ‘allanvar’. This function adapts that code to calculate the ADEV at a single averaging factor. See: <https://rdrr.io/cran/allanvar/src/R/avar.R>.

**Code**

```
# Function to calculate the ADEV for frequency data
fadev <- function(y)
{
  N=length(y)
  s=0
  for (i in 1:(N-1))
  {
    s = s + (y[i+1]-y[i])^2
  }
  av=s/(2*(N-1))
  return (sqrt(av))
}
```

**Test Case**

Classic NBS Monograph 140 Annex 8.E frequency values from: B.E. Blair (Editor), Time and Frequency: Theory and Fundamentals, *NBS Monograph 140*, [Annex 8.E](#), p. 181, May 1974.

```
> nbs
[1] 892 809 823 798 671 644 883 903 677
> fadev(nbs)
[1] 91.22945
```



**Description**

Function to calculate the estimated overlapping ADEV of a set of phase data.

**Usage**

```
poadev(x, tau, af)
```

**Arguments**

x            The vector of phase data to be analyzed.  
tau          The data sampling interval of the phase data, seconds (default=1).  
af           The averaging factor for the ADEV estimate (default=1).

**Return Value**

The estimated Allan deviation for the phase data at a certain averaging factor.

**Example**

Find the estimated overlapping ADEV for a set of phase data with tau=1 at AF=10

```
poadev(x,1,10)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function to calculate the overlapping Allan deviation from phase data
poadev <- function(x, tau=1, m=1)
{
  N=length(x)
  s=0
  for(i in 1:(N-2*m))
  {
    s = s + (x[i+2*m]-2*x[i+m]+x[i])^2
  }
  s = s/(2*m^2*(N-2*m)*tau^2)
  return (sqrt(s))
}
```

**Test Case**

1000-Point Test Suite phase data. See: W.J. Riley, "A Test Suite for the Calculation of Time Domain Frequency Stability", *Proceedings of the 1995 IEEE International Frequency Control Symposium*, pp. 360-366, June 1995. It may be downloaded as frequency data from:

[https://www.wriley.com/tst\\_suit.dat](https://www.wriley.com/tst_suit.dat)

```
> poadev(tsp,1,1)
[1] 0.2922319
> poadev(tsp,1,10)
[1] 0.09159953
> poadev(tsp,1,100)
[1] 0.03241343
```

**Description**

Function to calculate the estimated overlapping ADEV of a set of fractional frequency data.

**Requires**

Functions ftop() and poadev() in this package.

**Usage**

```
foadev(y, tau, af)
```

**Arguments**

y	The vector of fractional frequency data to be analyzed.
tau	The sampling interval of the frequency data, (default=1).
af	The averaging factor for the ADEV estimate (default=1).

**Return Value**

The estimated Allan deviation for the frequency data at a certain averaging factor.

**Example**

Find the estimated ADEV for a set of frequency data with tau=1 at AF=10.

```
fadev(y)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Note**

It is faster and more efficient to convert frequency data to phase data before calculating the overlapping Allan deviation for frequency data, thereby avoiding nested summations.

**Code**

```
# Function to calculate the overlapping Allan deviation from freq data
foadev <- function(y, tau=1, af=1)
{
  x=ftop(y,tau)
  ad=poadev(x,tau,af)
  return (ad)
}
```

**Test Case**

1000-Point Test Suite frequency data. See: W.J. Riley, "A Test Suite for the Calculation of Time Domain Frequency Stability", *Proceedings of the 1995 IEEE International Frequency Control Symposium*, pp. 360-366, June 1995. . Downloaded from: [https://www.wiley.com/tst\\_suit.dat](https://www.wiley.com/tst_suit.dat)

```
> foadev(ts,1,1)
[1] 0.2922319
> foadev(ts,1,10)
[1] 0.09159953
> foadev(ts,1,100)
[1] 0.03241343
```

**Description**

Function to calculate the estimated overlapping ADEV of a set of phase or frequency data over a range of octave averaging factors

**Usage**

```
adevrun(z, type, tau)
```

**Arguments**

`z` The vector of phase or frequency data to be analyzed.  
`type` The data type, 0=phase, 1=frequency (default=0)  
`tau` The data sampling interval of the data, seconds (default=1).

**Return Value**

An 'allanvar'-compatible data frame containing tau, AVAR and error bar values for the estimated Allan deviations for the data over a range of octave averaging factors.

**Example**

Find the estimated overlapping ADEVs for a set of phase data with tau=1 over a range of octave averaging factors:

```
adevrun(x,0,1)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function to calculate the overlapping Allan deviation
# from phase or data over a range of octave averaging factors
# The function writes a table of af and ADEV values
# and returns a data frame compatible with the 'allanvar' package
# containing time, av (AVAR not ADEV) and error vectors
# which can be plotted with plotav()
adevrun <- function(z, type=0, tau=1)
{
  # If frequency data, convert it to phase data
  if(type==1)
  {
    x<-ftop(z)
  }
  else
  {
    x<-z
  }
  # Initializations
  N=length(x)
  af=1
  n=1 # point #
  # Loop thru AFs up to limit, calculating ADEV
  # The maximum AF is floor(N/4)
```

```

# The # stability points is:
# ceiling(log10(floor(N/4))/log10(2))
p=ceiling(log10(floor(N/4))/log10(2))
# Create results table per 'allanvar' package
# Note that that table has AVAR not ADEV
time<-1:p
av<-1:p
error<-1:p
while(af<=floor(N/4))
{
  ad=poadev(x,tau,af)
  print(paste0("AF= ",af," ADEV=",ad))
  av[n]=ad^2
  time[n]=tau*af
  # Equation for error AV estimation per allanvar
  # See Papoulis (1991) for further information
  error[n]=1/(sqrt(2*(N/(af-1))))
  af=af*2
  n=n+1
}
return (data.frame(time,av,error))
}

```

### Test Case

1000-Point Test Suite frequency data. See: W.J. Riley, "A Test Suite for the Calculation of Time Domain Frequency Stability", *Proceedings of the 1995 IEEE International Frequency Control Symposium*, pp. 360-366, June 1995. It may be downloaded from:

[https://www.wriley.com/tst\\_suit.dat](https://www.wriley.com/tst_suit.dat).

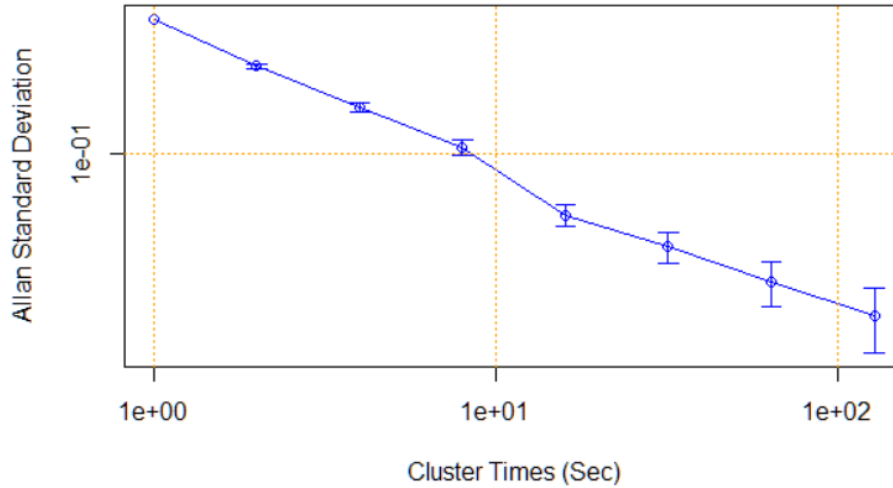
```

> r<-adevrun(ts,1,1)
[1] "AF= 1 ADEV=0.29223187810676"
[1] "AF= 2 ADEV=0.201016042170939"
[1] "AF= 4 ADEV=0.144791307218438"
[1] "AF= 8 ADEV=0.1057038500787"
[1] "AF= 16 ADEV=0.0619147784187454"
[1] "AF= 32 ADEV=0.0480821426212821"
[1] "AF= 64 ADEV=0.036237212985705"
[1] "AF= 128 ADEV=0.0276738558206943"

> plotav(r)

```

### Allan variance Analysis

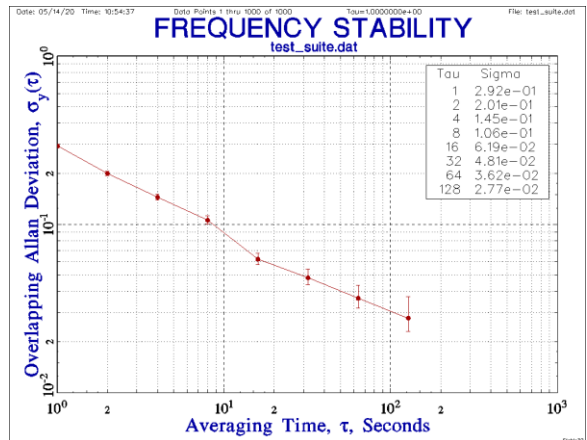


Equivalent Stable32 stability table and plot results are as follows:

```

STATISTICS FOR FILE: test_suite.dat
Frequency Data Points 1 thru 1000 of 1000
  Maximum = 9.957453e-01
  Minimum = 1.371760e-03
  Average = 4.897745e-01
Sigma Type: Overlapping Allan
Confidence Factor = 0.683
Deadtime T/Tau = 1.000000
    
```

AF	Tau	#	Alpha	Min Sigma	Sigma	Max Sigma
1	1.0000e+00	999	0	2.8515e-01	2.9223e-01	2.9987e-01
2	2.0000e+00	997	0	1.9520e-01	2.0102e-01	2.0738e-01
4	4.0000e+00	993	0	1.3931e-01	1.4479e-01	1.5098e-01
8	8.0000e+00	985	0	1.0038e-01	1.0570e-01	1.1198e-01
16	1.6000e+01	969	0	5.7696e-02	6.1915e-02	6.7217e-02
32	3.2000e+01	937	0	4.3654e-02	4.8082e-02	5.4202e-02
64	6.4000e+01	873	0	3.1755e-02	3.6237e-02	4.3377e-02
128	1.2800e+02	745	0	2.3045e-02	2.7674e-02	3.7027e-02



**Description**

Function to calculate the estimated MDEV of a set of phase data.

**Usage**

```
pmdev(x, tau, af)
```

**Arguments**

x            The vector of phase data to be analyzed.  
tau          The data sampling interval of the phase data, seconds (default=1).  
af          The averaging factor for the MDEV estimate (default=1).

**Return Value**

The estimated Modified Allan deviation for the phase data at a certain averaging factor.

**Example**

Find the estimated MDEV for a set of phase data with data sampling interval tau=1 and AF=10  
pmdev(x,1,10)

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function to calculate Modified Allan deviation
# MVAR for phase data
# Argument tau is basic data sampling interval
# Each analysis tau is tau*m
# where argument m is averaging factor 1 to N/3
pmdev<-function(x, tau=1, m=1)
{
  N=length(x)
  mvar=0
  # Outer loop
  for(j in 1:(N-3*m+1))
  {
    s=0
    # Inner loop
    for(i in j:(j+m-1))
    {
      s=s+(x[i+(2*m)]-2*x[i+m]+x[i])
    }
    mvar=mvar+s^2
  }
  # Scaling
  mvar=mvar/(2*m^2*m^2*tau^2*(N-3*m+1))
  return (sqrt(mvar))
}
```

## Test Case

1000-Point Test Suite phase data. See: W.J. Riley, “A Test Suite for the Calculation of TimeDomain Frequency Stability”, *Proceedings of the 1995 IEEE International Frequency Control Symposium*, pp. 360-366, June 1995. It may be downloaded as frequency data from: [https://www.wriley.com/tst\\_suit.dat](https://www.wriley.com/tst_suit.dat).

```
> pmdev(tsp)
[1] 0.2922319
> pmdev(tsp,1,10)
[1] 0.06172376
> pmdev(tsp,1,100)
[1] 0.02170921
```

## Time Deviation

One can easily get the time deviation, TDEV, from MDEV by multiplying by  $\sqrt{\tau^2/3}$ :

```
> pmdev(tsp)*sqrt(1*1/3)
[1] 0.1687202
```

**Description**

Function to calculate the estimated HDEV of a set of phase data at its basic sampling interval.

**Usage**

```
phdev(x, tau)
```

**Arguments**

x            The vector of phase data to be analyzed.  
tau          The data sampling interval, seconds (default=1).

**Return Value**

The estimated Hadamard deviation for the phase data at its basic sampling interval.

**Example**

Find the estimated HDEV for a set of phase data:

```
phdev(x,1)
```

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function to calculate the Hadamard deviation from phase data
phdev <- function (x, tau=1)
{
  N=length(x)
  s=0
  for (i in 1:(N-3))
  {
    s = s +(x[i+3] -3*x[i+2] +3*x[i+1] -x[i])^2
  }
  hv = s/(6*(tau^2)*(N-3))
  return (sqrt(hv))
}
```

**Test Case**

Classic NBS Monograph 140 Annex 8.E frequency values from: B.E. Blair (Editor), Time and Frequency: Theory and Fundamentals, *NBS Monograph 140*, [Annex 8.E](#), p. 181, May 1974.

```
> nbs<-c(892,809,823,798,671,644,883,903,677)
> nbs
[1] 892 809 823 798 671 644 883 903 677
> nbsi<-diffinv(nbs)
> nbsi
[1] 0 892 1701 2524 3322 3993 4637 5520 6423 7100
> phdev(nbsi,1)
[1] 70.80607
```



**Description**

Function to calculate the estimated HDEV of a set of fractional frequency data at its basic sampling interval.

**Imports**

Package RobPer required.

**Arguments**

y                    The vector of fractional frequency data to be analyzed.

**Return Value**

The estimated Hadamard deviation for the frequency data at its basic sampling interval.

**Example**

Find the estimated HDEV for a set of frequency data:  
fhdev(y)

**Reference**

W.J. Riley, [Handbook of Frequency Stability Analysis](#). You can buy a printed copy of this July 2008 book at [Handbook](#), and download it as [NIST Special Publication 1065](#).

**Code**

```
# Function to calculate the Hadamard deviation from freq data
fhdev <- function(y)
{
  N=length(y)
  s=0
  for (i in 1:(N-2))
  {
    s = s + (y[i+2] -2*y[i+1] +y[i])^2
  }
  hv=s/(6*(N-2))
  return (sqrt(hv))
}
```

**Test Case**

Classic NBS Monograph 140 Annex 8.E frequency values from: B.E. Blair (Editor), Time and Frequency: Theory and Fundamentals, *NBS Monograph 140*, [Annex 8.E](#), p. 181, May 1974.

```
> nbs
[1] 892 809 823 798 671 644 883 903 677
> fhdev(nbs)
[1] 70.80607
```

**Description**

Function to calculate the Théo1 statistic for a set of phase data.

**Arguments**

x                    The vector of phase data to be analyzed.  
 tau                  The sampling interval of the phase data, seconds (default=1).  
 af                    The averaging factor for the analysis (must be even, default=2)

**Return Value**

The estimated value of the Théo1 statistic for the phase data at a certain averaging factor.

**Example**

Find Théo1 for a set of phase data having a tau=1 second at an averaging factor of 2:  
 fhdev(y)

**Reference**

D.A. Howe and T.K. Pepler, “Very Long-Term Frequency Stability: Estimation Using a Special-Purpose Statistic”, *Proceedings of the 2003 IEEE International Frequency Control Symposium*, May 2003.

**Code**

```
# Find Theo1 per Howe and Pepler (2003)
# x = phase data vector (1 to N)
# tau = data sampling interval
# m = averaging factor (2 to N-1)
# m must be even
# Analysis tau = m*tau
# Stride = 0.75*m*tau
theo1<-function(x, tau=1, m=2)
{
  # Initializations
  N=length(x)
  t1=0

  # Outer sum
  for( i in 1:(N-m))
  {
    sum=0
    # Inner sum
    for( d in 0:((m/2)-1))
    {
      s=(1/((m/2)-d))*((x[i]-x[i-d+(m/2)]+x[i+m]-x[i+d+(m/2)])^2)
      sum=sum+s
    }
    t1=t1+sum
  }

  # Scaling factor
  t1=t1/(0.75*(N-m)*(m*tau)^2)
}
```

```
# Return Theo1 deviation
return (sqrt(t1))
}
```

## Test Case

Test data of Appendix I of Reference:

```
> t
```

```
[1] 1.00 2.50 0.65 -3.71 -3.30 1.08 0.50 2.20 4.68 3.29
```

Test results:

```
> theo1(t,1,4)
```

```
[1] 1.509405
```

```
> theo1(t,1,6)
```

```
[1] 1.412349
```

```
> theo1(t,1,8)
```

```
[1] 1.148758
```

**Description**

Function to calculate and plot a power spectral density (PSD) for phase or frequency data.

**Arguments**

`z` The time series to be analyzed  
`span` # of smoothing spans to use (default=10)  
`logx` Flag to use log x scale (default=TRUE)  
`logy` Flag to use log y scale (default=TRUE)  
`title` Plot title (default="PSD Plot")

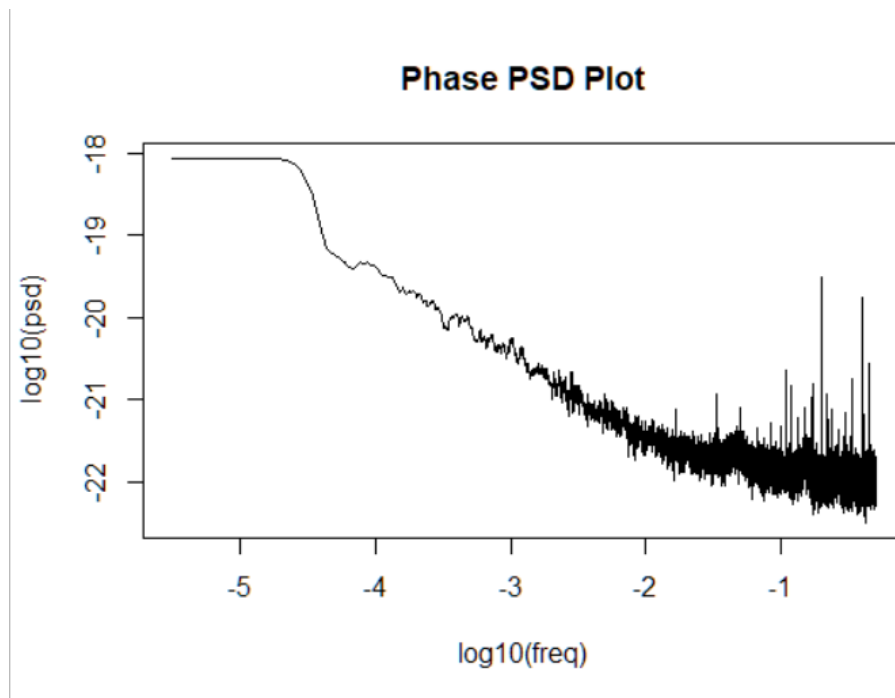
**Return Value**

The requested PSD plot.

**Example**

Plot the PSD for a set of phase data:

```
psd(phase, span=10, logx=TRUE, logy=TRUE, title="Phase PSD Plot")
```

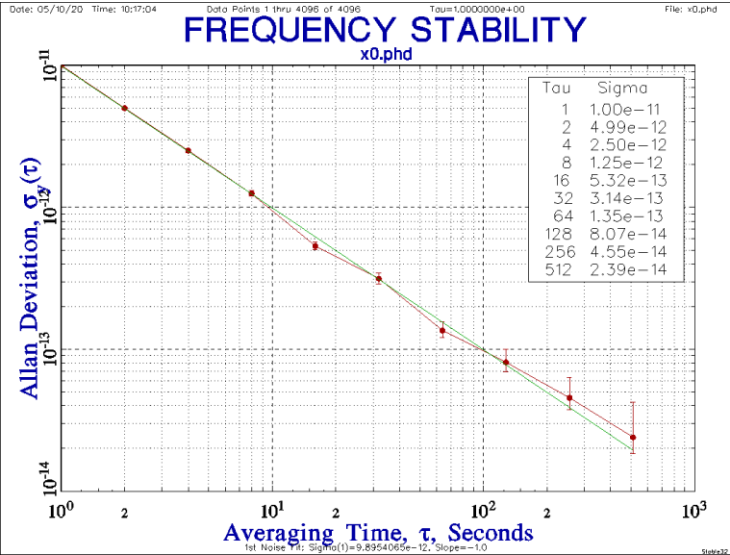
**Code**

```
# Function to calculate and plot a power spectral density
# for phase or frequency data
psd <- function(z, span=10, logx=TRUE, logy=TRUE, title="PSD Plot")
{
  s<-spectrum(z, span)
  freq<-s$freq
  psd<-2*s$spec
```

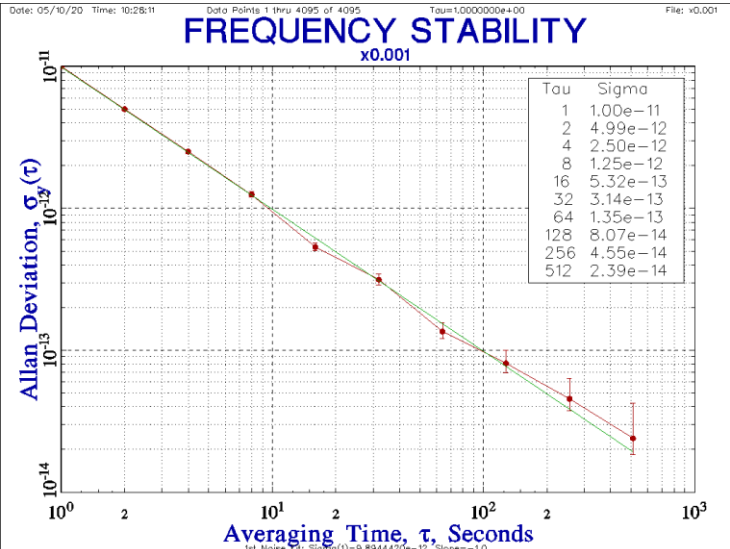
```
if( logx==FALSE & logy==FALSE)
{
    plot(freq,psd,type="l",main=title)
}
else if(logx==FALSE & logy==TRUE)
{
    plot(freq,log10(psd),type="l",main=title)
}
else if(logx==TRUE & logy==FALSE)
{
    plot(log10(freq),psd,type="l",main=title)
}
else(logx==TRUE & logy==TRUE)
{
    plot(log10(freq),log10(psd),type="l",main=title)
}
}
```

### Appendix 3 Notes for padev() and fadev()

These functions calculate the Allan deviation at a single unity averaging factor. At that basic sampling interval there is no difference between non-overlapping and overlapping samples, nor between the Allan and Modified Allan deviations. The functions pavg() and favg() can be used to average the phase or frequency data to a longer tau before applying padev() or fadev().

<pre style="font-family: monospace; font-size: 0.9em;"> &gt; # Save x0 data file &gt; x&lt;-x0 &gt; length(x) [1] 4096 &gt; # Calculate ADEV over a range &gt; # of octave averaging factors &gt; padev(x) [1] 1e-11 &gt; x&lt;-pavg(x) &gt; length(x) [1] 2048 &gt; padev(x,2) [1] 4.99267805585e-12 &gt; x&lt;-pavg(x) &gt; length(x) [1] 1024 &gt; padev(x,4) [1] 2.50433298546e-12 &gt; x&lt;-pavg(x) &gt; length(x) [1] 512 &gt; padev(x,8) [1] 1.24888717539e-12 </pre>	 <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; font-size: 0.8em;"> <thead> <tr> <th>Tau</th> <th>Sigma</th> </tr> </thead> <tbody> <tr><td>1</td><td>1.00e-11</td></tr> <tr><td>2</td><td>4.99e-12</td></tr> <tr><td>4</td><td>2.50e-12</td></tr> <tr><td>8</td><td>1.25e-12</td></tr> <tr><td>16</td><td>5.32e-13</td></tr> <tr><td>32</td><td>3.14e-13</td></tr> <tr><td>64</td><td>1.35e-13</td></tr> <tr><td>128</td><td>8.07e-14</td></tr> <tr><td>256</td><td>4.55e-14</td></tr> <tr><td>512</td><td>2.39e-14</td></tr> </tbody> </table>	Tau	Sigma	1	1.00e-11	2	4.99e-12	4	2.50e-12	8	1.25e-12	16	5.32e-13	32	3.14e-13	64	1.35e-13	128	8.07e-14	256	4.55e-14	512	2.39e-14
Tau	Sigma																						
1	1.00e-11																						
2	4.99e-12																						
4	2.50e-12																						
8	1.25e-12																						
16	5.32e-13																						
32	3.14e-13																						
64	1.35e-13																						
128	8.07e-14																						
256	4.55e-14																						
512	2.39e-14																						

Notice that the tau is entered for the averaged padev() calculations.

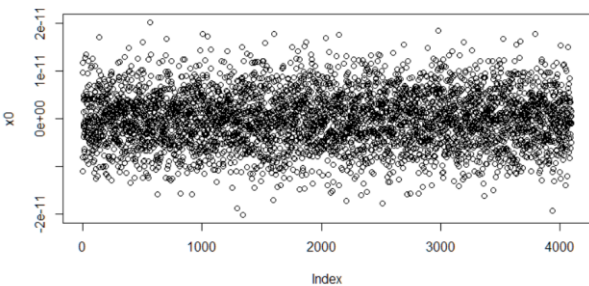
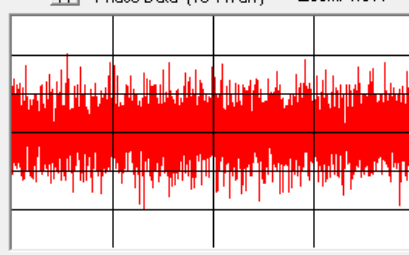
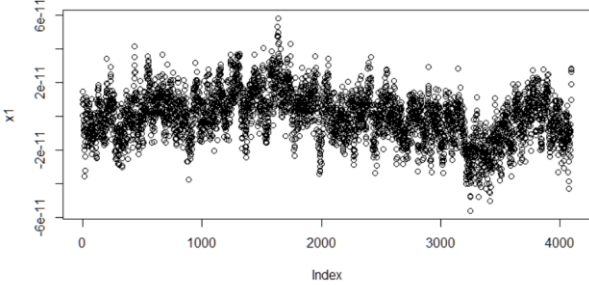
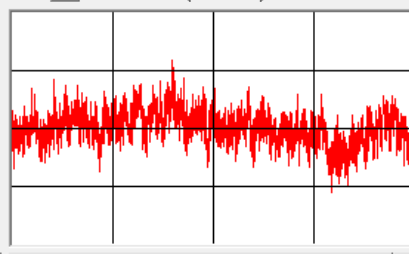
<pre style="font-family: monospace; font-size: 0.9em;"> # Repeat for frequency data &gt; y&lt;-ptof(x0) &gt; length(y) [1] 4095 &gt; fadev(y) [1] 1e-11 &gt; y&lt;-favg(y) &gt; length(y) [1] 2047 &gt; fadev(y) [1] 4.99267805585e-12 &gt; y&lt;-favg(y) &gt; fadev(y) [1] 2.50433298546e-12 &gt; y&lt;-favg(y) &gt; length(y) [1] 511 &gt; fadev(y) [1] 1.24888717539e-12 </pre>	 <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; font-size: 0.8em;"> <thead> <tr> <th>Tau</th> <th>Sigma</th> </tr> </thead> <tbody> <tr><td>1</td><td>1.00e-11</td></tr> <tr><td>2</td><td>4.99e-12</td></tr> <tr><td>4</td><td>2.50e-12</td></tr> <tr><td>8</td><td>1.25e-12</td></tr> <tr><td>16</td><td>5.32e-13</td></tr> <tr><td>32</td><td>3.14e-13</td></tr> <tr><td>64</td><td>1.35e-13</td></tr> <tr><td>128</td><td>8.07e-14</td></tr> <tr><td>256</td><td>4.55e-14</td></tr> <tr><td>512</td><td>2.39e-14</td></tr> </tbody> </table>	Tau	Sigma	1	1.00e-11	2	4.99e-12	4	2.50e-12	8	1.25e-12	16	5.32e-13	32	3.14e-13	64	1.35e-13	128	8.07e-14	256	4.55e-14	512	2.39e-14
Tau	Sigma																						
1	1.00e-11																						
2	4.99e-12																						
4	2.50e-12																						
8	1.25e-12																						
16	5.32e-13																						
32	3.14e-13																						
64	1.35e-13																						
128	8.07e-14																						
256	4.55e-14																						
512	2.39e-14																						

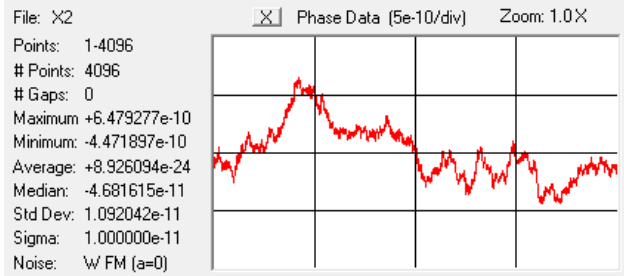
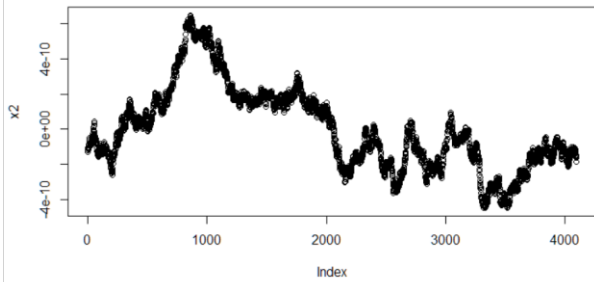
But, for higher-confidence, it is better to use the entire data set and overlapping samples as implemented in avari() of the 'allanvar' package and in poadev() and foadev().

The padev() and fadev() functions are used by the noise() generation function to set the desired Allan deviation for their respective data type. They are also used by the bs() function to show that quantity.

## Appendix 4

### Examples of Noise Generation, Data Plots, Noise Identification and Basic Statistics with the functions noise(), plot(), nid(), and bs()

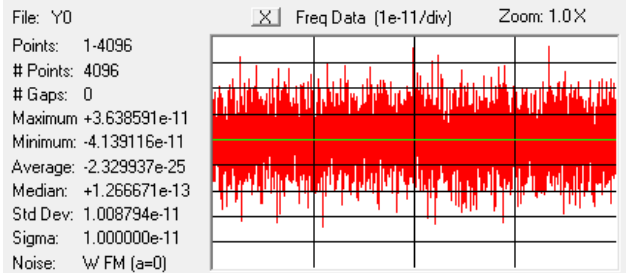
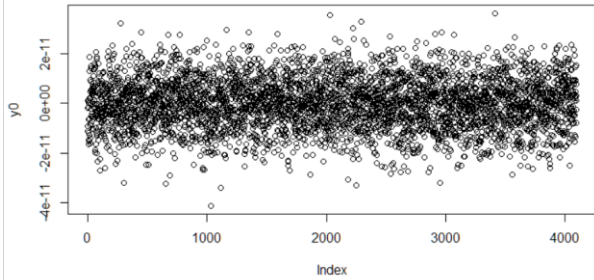
	<div style="border: 1px solid gray; padding: 5px;"> <p>File: X0 <span style="float: right;">Phase Data (1e-11/div) Zoom: 1.0X</span></p> <p>Points: 1-4096</p> <p># Points: 4096</p> <p># Gaps: 0</p> <p>Maximum +2.029786e-11</p> <p>Minimum: -2.013143e-11</p> <p>Average: -2.725939e-25</p> <p>Median: +4.359510e-14</p> <p>Std Dev: 8.132550e-12</p> <p>Sigma: 1.000000e-11</p> <p>Noise: W PM (a=2)</p>  </div>
<p style="text-align: center;">W PM</p> <pre style="font-family: monospace;"> x0&lt;-noise(4096,0,1e-11,0,1) plot(x0) bs(x0) write(x0,"C:\\Data\\x0.phd",1) </pre>	<pre style="font-family: monospace;"> [1] Basic Statistics: [1] File = x0 [1] Type = Phase [1] Tau = 1 [1] # Points = 4096 [1] Max = 2.02978610909385e-11 [1] Min = -2.01314251491493e-11 [1] Span= 4.04292862400878e-11 [1] Mean = -8.26356594567044e-29 [1] Median = 4.70758166928985e-14 [1] MAD = 5.73682842157458e-12 [1] Std Dev = 5.72330213491854e-12 [1] Sigma = 9.99999999999999e-12 [1] nid = 2.01985990379316 [1] Alpha = 2.01985990379316 [1] Noise = W PM </pre>
	<div style="border: 1px solid gray; padding: 5px;"> <p>File: X1 <span style="float: right;">Phase Data (5e-11/div) Zoom: 1.0X</span></p> <p>Points: 1-4096</p> <p># Points: 4096</p> <p># Gaps: 0</p> <p>Maximum +5.848072e-11</p> <p>Minimum: -5.623309e-11</p> <p>Average: +4.626409e-25</p> <p>Median: +6.781125e-13</p> <p>Std Dev: 8.883410e-12</p> <p>Sigma: 1.000000e-11</p> <p>Noise: F PM (a=1)</p>  </div>
<p style="text-align: center;">F PM</p> <pre style="font-family: monospace;"> x1&lt;-noise(4096,1,1e-11,0,1) F PM plot(x1) bs(x1) write(x1,"C:\\Data\\x1.phd",1) </pre>	<pre style="font-family: monospace;"> [1] Basic Statistics: [1] File = x1 [1] Type = Phase [1] Tau = 1 [1] # Points = 4096 [1] Max = 5.84807189016316e-11 [1] Min = -5.62330868955754e-11 [1] Span= 1.14713805797207e-10 [1] Mean = -3.51577740975912e-28 [1] Median = 6.85181196368336e-13 [1] MAD = 1.38708603479839e-11 [1] Std Dev = 1.43574889402762e-11 [1] Sigma = 1e-11 [1] nid = 0.729755489755081 [1] Alpha = 0.729755489755081 [1] Noise = F PM </pre>



RW PM

```
x2<-noise(4096,2,1e-11,0,1) RW PM
plot(x2)
bs(x2)
write(x2,"c:\\Data\\x2.phd",1)
```

```
[1] Basic Statistics:
[1] File = x2
[1] Type = Phase
[1] Tau = 1
[1] # Points = 4096
[1] Max = 6.47927717483002e-10
[1] Min = -4.47189694402041e-10
[1] Span= 1.09511741188504e-09
[1] Mean = -9.51670086404567e-27
[1] Median = -4.67813969398479e-11
[1] MAD = 2.52791718953365e-10
[1] Std Dev = 2.37408182839263e-10
[1] Sigma = 1e-11
[1] nid = -0.276641140000412
[1] Alpha = -0.276641140000412
[1] Noise = W FM
```

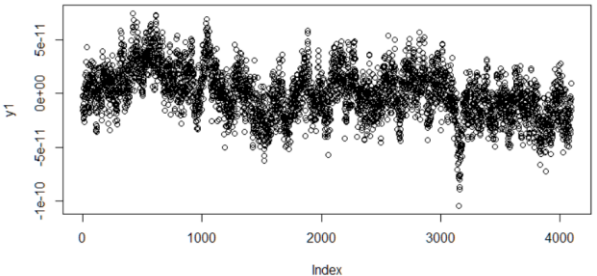
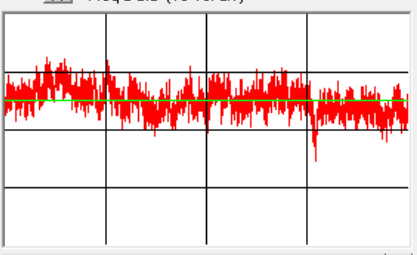
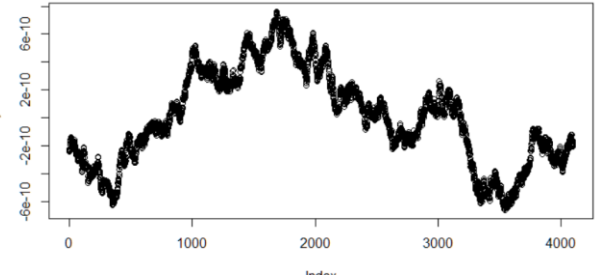
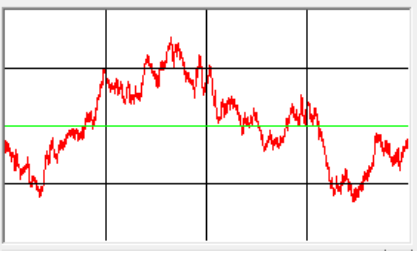


W FM

```
y0<-noise(4096,0,1e-11,1,1)
plot(y0)
bs(y0,1,1)
write(y0,"C:\\Data\\y0.frd",1)
```

```
[1] Basic Statistics:
[1] File = y0
[1] Type = Frequency
[1] Tau = 1
[1] # Points = 4096
[1] Max = 3.63859071683338e-11
[1] Min = -4.13911634846489e-11
[1] Span= 7.77770706529828e-11
[1] Mean = -1.96120527724863e-28
[1] Median = 1.29435902553715e-13
[1] MAD = 9.93182203459801e-12
[1] Std Dev = 1.00879385318981e-11
[1] Sigma = 1e-11
[1] nid = 1.9659025637474
[1] Alpha = -0.0340974362525981
[1] Noise = W FM
```



	 <pre> File: Y1 Points: 1-4096 # Points: 4096 # Gaps: 0 Maximum: +7.493713e-11 Minimum: -1.049010e-10 Average: +4.535698e-25 Median: -1.058002e-13 Std Dev: 2.282321e-11 Sigma: 1.000000e-11 Noise: F FM (a=-1) </pre>
<p style="text-align: center;">F FM</p> <pre> y1&lt;-noise(4096,1,1e-11,1,1) plot(y1) bs(y1,1,1) write(y1,"c:\\Data\\y1.frd",1) </pre>	<pre> [1] Basic Statistics: [1] File = y1 [1] Type = Frequency [1] Tau = 1 [1] # Points = 4096 [1] Max = 7.49371347343171e-11 [1] Min = -1.04900962344573e-10 [1] Span= 1.79838097078891e-10 [1] Mean = -3.07898420209187e-28 [1] Median = -1.04051185404546e-13 [1] MAD = 2.25373829312755e-11 [1] Std Dev = 2.28232081099642e-11 [1] Sigma = 1e-11 [1] nid = 0.718287670833863 [1] Alpha = -1.28171232916614 [1] Noise = F FM </pre>
	 <pre> File: Y2 Points: 1-4096 # Points: 4096 # Gaps: 0 Maximum: +7.632633e-10 Minimum: -6.631584e-10 Average: +4.533110e-26 Median: -2.152941e-12 Std Dev: 3.386186e-10 Sigma: 1.000000e-11 Noise: RW FM (a=2) </pre>
<p style="text-align: center;">RW FM</p> <pre> y2 &lt;-noise(4096,2,1e-11,1,1) plot(y2) bs(y2,1,1) write(y2,"c:\\Data\\y2.frd",1) </pre>	<pre> [1] Basic Statistics: [1] File = y2 [1] Type = Frequency [1] Tau = 1 [1] # Points = 4096 [1] Max = 7.63263284355859e-10 [1] Min = -6.63158350087911e-10 [1] Span= 1.42642163444377e-09 [1] Mean = 6.91003326225927e-27 [1] Median = -1.93595507701568e-12 [1] MAD = 3.84900008281505e-10 [1] Std Dev = 3.38618639221366e-10 [1] Sigma = 1e-11 [1] nid = -0.290301177620439 [1] Alpha = -2.29030117762044 [1] Noise = RW FM </pre>

This table shows six examples of power law phase and frequency noise from W PM ( $a=2$ ) to RW FM ( $a=-2$ ), where RW PM and W FM have the same  $a=0$ . The data plots are for their respective phase or frequency data types, and Stable32 Stats results are shown to their right. The Basic Statistics listings (with extra nid items) are in the panel below that, and show the same results. The bottom left panel of each set shows the R commands used to generate the noise, plot and analyze it, and save it to a file.

**Appendix 5**  
**Regression Analysis for Phase and Frequency Data**

Frequency Offset			
Data	Method	R Code	Remarks
Phase	Linear fit to slope $x(t)=a+bt, y(t)=b$	<code>t&lt;=1:length(x)</code> <code>lm(x~t)</code>	Common
	Average of 1 <sup>st</sup> differences $y(t)=[x(t+\tau)-x(t)]/\tau$	<code>f=mean(diff(x))</code>	
	Endpoints $slope=(x[n]-x[1])/(n-1)$	<code>n=length(x)</code> <code>f=(x[n]-x[1])/(n-1)</code>	Match endpoints
Freq	Arithmetic average (mean)	<code>f=mean(y)</code>	Most common
Frequency Drift			
Phase	Quadratic fit $x(t)=a+bt+ct^2$ , where $y(t)=x'(t)=b+2ct$ , $slope=y'(t)=2c$	<code>t&lt;=1:length(x)</code> <code>t2&lt;-t^2</code> <code>lm(x~t+t2)</code>	Most common
	Average of 2 <sup>nd</sup> differences $y(t)=[x(t+\tau)-x(t)]/\tau$ , $slope=[y(t+\tau)-y(t)]/\tau =$ $[x(t+2\tau)-2x(t+\tau)+x(t)]/\tau$	<code>d=mean(diff(x,1,2))</code>	May have numerical precision problems
	3-point fit $slope=4[x(n)-2x(n/2)+x(1)]/(n\tau)^2$	<code>n=length(x)</code> <code>d=4*(x[n]-2*x[floor(n/2)]+x[1])/</code> <code>(n*tau)^2</code>	
	Greenhall fit 4-point cumulative sum estimator using start, 10%, 90% & end points	<code>w=cumsum(x)</code> <code>n=length(w)</code> <code>d=-4*w[1]+5*w[floor(n/10)]-</code> <code>5*w[floor(9*n/10)]+4*w[n]</code> <code>d=d*50/(3*n*n)</code>	
Freq	Linear fit to slope $y(t)=a+bt, y'(t)=b$	<code>t&lt;=1:length(y)</code> <code>lm(y~t)</code>	See example below
	Bisection fit $slope=2 [ y(2nd\ half) - y(1st\ half) ] / (n\cdot\tau)$ , where n=# points	<code>n=length(y)</code> <code>h1&lt;-y[1]:y[floor(n/2)]</code> <code>h2&lt;-y[floor(n/2)+1]:y[n]</code> <code>m1=mean(h1)</code> <code>m2=mean(h2)</code> <code>d=2*(m2-m1)/(n*tau)</code>	Uses averages of first and last halves of data
Nonlinear Models for Aging Stabilization			
Freq	Log $y(t)=a\cdot\ln(bt+1)+c$ , $slope=y'(t)=ab/(bt+1)$	<code>t&lt;=1:length(y)</code> <code>&gt; a=initial estimate</code> <code>&gt; b=initial estimate</code> <code>&gt; c=initial estimate</code> <code>&gt; nls(rafts ~</code> <code>a*(log(b*t+1))+c,</code> <code>start=list(a=a,b=b,c=c))</code>	MIL-O-55310B

	<b>Diffusion</b> $y(t) = a + b(t+c)^{1/2}$ , $\text{slope} = y'(t) = \frac{1}{2} \cdot b(t+c)^{-1/2}$ .	<pre>t&lt;-1:length(y) &gt; a=initial estimate &gt; b=initial estimate &gt; c=initial estimate &gt; nls(rafs ~ a+b*((t+c)^0.5), start=list(a=a,b=b,c=c))</pre>	
<b>Prewhitening Methods</b>			
Phase	Remove Slope	Calc slope (see above), then: <pre>for(i in 1:length(x)) x[i]=x[i]-i*slope</pre>	
Freq	Remove Drift	Calc drift (see above), then: <pre>for(i in 1:length(y)) y[i]=y[i]-i*slope</pre>	
Both	Remove AR(1) fit $z(t) = z(t+1) - r(1) \cdot z(t)$	<pre>w=acf(z) r1=w[1] for(i in 1:length(z)) z[i]=z[i+1]-r1*z[i]</pre>	z is x or y. r(1)=lag 1 autocorrelation coefficient.

<p>To plot frequency data with a regression line:</p> <pre>&gt; plot(d,type="s",ylab="Freq") &gt; t&lt;-1:length(d) &gt; fit&lt;-lm(d~t) &gt; abline(fit,col="red")</pre>	
---	--

## Appendix 6

### R code for Frequency Stability Analysis Package

#### fsa.R

```
# R Functions for Basic Frequency Stability Analysis
# W.J. Riley
# Hamilton Technical Services, Beaufort, SC 29907 USA
# License: MIT
# Version 1.0
# May 18, 2020

# Packages required
library(allanvar) # For avar() and avari()
library(RobPer) # For TK95()
library(zoo) # For rollapply()

# Note that the code for the fsa.R functions does not include argument validation,
# nor do they handle data with gaps.

# Function to average phase data
pavg<-function(x,af=2)
{
  seed<-c(TRUE,rep(FALSE,af-1))
  cont<-rep(seed,ceiling(length(x)/af))[1:length(x)]
  return(x[which(cont)])
}

# Function to average frequency data
favg<-function(data,af=2)
{
  return(rollapply(data,width=af,by=af,FUN=mean))
}

# Function for phase to frequency conversion
ptof<-function(x,tau=1)
{
  return(diff(x)/tau)
}

# Function for frequency to phase conversion
ftop<-function(y,tau=1)
{
  return(diffinv(y)*tau)
}

# Function to generate power law noise
noise<-function(num,alpha,sigma,type=0,tau=1)
{
  z<-TK95(num,alpha)
  if(type==0) d<-padev(z,tau)
  else d<-fadev(z)
  z=(z/d)*sigma
  m=mean(z)
  z=z-m
  return(z)
}

# Function to find the noise type using the lag 1 ACF method
nid<-function(z)
{
  nD=0 # Difference order
  # Save original data
  zz<-z
  # Calc lag 1 autocorrelation r1
  r1=acf(z,1, "cor",F)
```

```

r1=r1$acf[2]
# Find d = r1/(1+r1)
d=r1/(1+r1)
# If d<0.25, must apply increment operator
if(d>0.25)
{
  while(d>=0.25)
  {
    # Take 1st differences
    z<-diff(z)
    nD=nD+1
    # Calc lag 1 autocorrelation r1
    r1=acf(z,1, "cor",F)
    r1=r1$acf[2]
    # Find d = r1/(1+r1)
    d=r1/(1+r1)
  }
}
# Calc alpha
alpha=-2*d -2*nD +2
# Restore original data
z<-zz
return (alpha)
}

# Function to show basic statistics for phase or frequency data
bs <- function(z,type=0,tau=1)
{
  print("Basic Statistics:", quote=FALSE)
  txt=paste("File =", deparse(substitute(z)))
  print(txt, quote=FALSE)
  if(type==0)
  {
    print("Type = Phase", quote=FALSE)
  }
  else
  {
    print("Type = Frequency", quote=FALSE)
  }
  txt=paste("Tau =", tau)
  print(txt, quote=FALSE)
  txt=paste("# Points =", length(z))
  print(txt, quote=FALSE)
  txt=paste("Max =", max(z))
  print(txt, quote=FALSE)
  txt=paste("Min =", min(z))
  print(txt, quote=FALSE)
  txt=paste("Span=", max(z)-min(z))
  print(txt, quote=FALSE)
  txt=paste("Mean =", mean(z))
  print(txt, quote=FALSE)
  txt=paste("Median =", median(z))
  print(txt, quote=FALSE)
  txt=paste("MAD =", mad(z))
  print(txt, quote=FALSE)
  txt=paste("Std Dev =", sqrt(var(z)))
  print(txt, quote=FALSE)
  if(type==0) # Phase data
  {
    txt=paste("Sigma =", padev(z,tau))
    print(txt, quote=FALSE)
    # txt=paste("nid =", nid(z))
    # print(txt, quote=FALSE)
    alpha=nid(z)
  }
  else # Freq data
  {

```

```

    txt=paste("Sigma =", fadev(z))
    print(txt, quote=FALSE)
    # txt=paste("nid =", nid(z))
    # print(txt, quote=FALSE)
    alpha=nid(z)-2
}
txt=paste("Alpha =", alpha )
print(txt, quote=FALSE)
if(alpha>1.5)
{
    txt=paste("Noise = W PM")
    print(txt, quote=FALSE)
}
else if(alpha>0.5)
{
    txt=paste("Noise = F PM")
    print(txt, quote=FALSE)
}
else if(alpha>-0.5)
{
    txt=paste("Noise = W FM")
    print(txt, quote=FALSE)
}
else if(alpha>-1.5)
{
    txt=paste("Noise = F FM")
    print(txt, quote=FALSE)
}
else
{
    txt=paste("Noise = RW FM")
    print(txt, quote=FALSE)
}
plot(z)
}

# Function to count outliers in phase or frequency data
co <- function(z, limit=5)
{
    # Find MAD
    m=mad(z)
    # Count outliers
    n=sum(z<(-m*limit))+sum(z>(m*limit))
    return (n)
}

# Function to calculate the ADEV for phase data
padev <- function (x, tau=1)
{
    N=length(x)
    s=0
    for (i in 1:(N-2))
    {
        s = s + (x[i+2]-(2*x[i+1])+x[i])^2
    }
    av = s/(2*(tau^2)*(N-2))
    return (sqrt(av))
}

# Function to calculate the ADEV for frequency data
fadev <- function(y)
{
    N=length(y)
    s=0
    for (i in 1:(N-1))
    {
        s = s + (y[i+1]-y[i])^2
    }
}

```

```

}
av=s/(2*(N-1))
return (sqrt(av))
}

# Function to calculate the overlapping Allan deviation from phase data
poadev <- function(x, tau=1, m=1)
{
  N=length(x)
  s=0
  for(i in 1:(N-2*m))
  {
    s = s + (x[i+2*m]-2*x[i+m]+x[i])^2
  }
  s = s/(2*m^2*(N-2*m)*tau^2)
  return (sqrt(s))
}

# Function to calculate the overlapping Allan deviation from frequency data
foadev <- function(y, tau=1, af=1)
{
  x=ftop(y,tau)
  ad=poadev(x,tau,af)
  return (ad)
}

# Function to calculate the overlapping Allan deviation from phase or data
# over a range of octave averaging factors
adevrun <- function(z, type=0, tau=1)
{
  # If frequency data, convert it to phase data
  if(type==1)
  {
    x<-ftop(z)
  }
  else
  {
    x<-z
  }
  # Initializations
  N=length(x)
  af=1
  # Loop thru AFs up to limit, calculating ADEV
  # The maximum AF is floor(N/4)
  while(af<=floor(N/4))
  {
    ad=poadev(x,tau,af)
    print(paste0("AF= ",af," ADEV=",ad))
    af=af*2
  }
}

# Function to calculate Modified Allan deviation
# MVAR for phase data
# Argument tau is basic data sampling interval
# Each analysis tau is tau*m
# where argument m is averaging factor 1 to N/3
pmdev<-function(x,tau=1,m=1)
{
  N=length(x)
  mvar=0
  # Outer loop
  for(j in 1:(N-3*m+1))
  {
    s=0
    # Inner loop
    for(i in j:(j+m-1))

```

```

    {
      s=s+(x[i+(2*m)]-2*x[i+m]+x[i])
    }
    mvar=mvar+s^2
  }
  # Scaling
  mvar=mvar/(2*m^2*m^2*tau^2*(N-3*m+1))
  return (sqrt(mvar))
}

# Function to calculate the Hadamard deviation for phase data
phdev <- function (x, tau=1)
{
  N=length(x)
  s=0
  for (i in 1:(N-3))
  {
    s = s +(x[i+3] -3*x[i+2] +3*x[i+1] -x[i])^2
  }
  hv = s/(6*(tau^2)*(N-3))
  return (sqrt(hv))
}

# Function to calculate the Hadamard deviation for frequency data
fhdev <- function(y)
{
  N=length(y)
  s=0
  for (i in 1:(N-2))
  {
    s = s + (y[i+2] -2*y[i+1] +y[i])^2
  }
  hv=s/(6*(N-2))
  return (sqrt(hv))
}

# Find Theo1 per Howe and Pepler (2003)
# x = phase data vector (1 to N)
# tau = data sampling interval
# m = averaging factor (2 to N-1)
# m must be even
# Analysis tau = m*tau
# Stride = 0.75*m*tau
theo1<-function(x, tau=1, m=2)
{
  # Initializations
  N=length(x)
  t1=0

  # Outer sum
  for( i in 1:(N-m))
  {
    sum=0
    # Inner sum
    for( d in 0:((m/2)-1))
    {
      s=(1/((m/2)-d))*((x[i]-x[i-d+(m/2)]+x[i+m]-x[i+d+(m/2)])^2)
      sum=sum+s
    }
    t1=t1+sum
  }

  # Scaling factor
  t1=t1/(0.75*(N-m)*(m*tau)^2)

  # Return Theo1 deviation
  return (sqrt(t1))
}

```



```

}

# Function to calculate and plot a power spectral density
psd <- function(z, span=10, logx=TRUE, logy=TRUE, title="PSD Plot")
{
  s<-spectrum(z,span)
  freq<-s$freq
  psd<-2*s$spec

  if( logx==FALSE & logy==FALSE)
  {
    plot(freq,psd,type="l",main=title)
  }
  else if(logx==FALSE & logy==TRUE)
  {
    plot(freq,log10(psd),type="l",main=title)
  }
  else if(logx==TRUE & logy==FALSE)
  {
    plot(log10(freq),psd,type="l",main=title)
  }
  else(logx==TRUE & logy==TRUE)
  {
    plot(log10(freq),log10(psd),type="l",main=title)
  }
}

```

## Appendix 7

### Adding Frequency Stability Analysis Functionality to R with C++, Rcpp, and Rtools

#### Using C/C++ with R

C or C++ functions may be called from R to provide significantly faster execution. This can be an important advantage for core frequency stability analysis functions (e.g., variances) that involve nested loops performed on large data arrays. This appendix briefly describes how that can be done for a Windows R installation.

#### Rcpp and Rtools

The easiest way to use C/C++ code in R is with the Rcpp and Rtools tool chain. Rcpp supports calling C++ from R, while Rtools compiles C++ code under R. The C++ code for the small functions usually involved closely resembles plain C with the significant advantage of easier memory management, and Rcpp is much easier to use than the older C interface. The current version 4.0.0 of R requires rtools40 which can be installed on a 64-bit Windows system with rtools40-x86\_64.exe. As usual, one should install the latest version of R, RStudio and Rtools (see: <https://cran.r-project.org/bin/windows/Rtools/> and follow the instructions therein).

Then, the Rcpp/Rtools environment can be verified with the following on the R console command line (see “Getting started with C++” in “High performance functions with Rcpp” at [adv-r.had.co.nz](http://adv-r.had.co.nz)):

```
> cppFunction('int add(int x, int y, int z)
+ {
+   int sum = x + y + z;
+   return sum;
+ }')
> add(1,2,3)
[1] 6
```

The approach shown above using `cppFunction()` is fine for a small piece of C++ code, but it is more common for a larger project to call C++ code from a separate `.cpp` source file that begins with:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
```

and is brought into R using `sourceCpp("filename.cpp")`.

For example, this C++ code in `size.cpp` calculates the size of a data vector:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double size(NumericVector x)
{
  return x.size();
}
```

which can be compiled and run in R with:

```

> library(Rcpp)
>
> sourceCpp("C:\\R\\size.cpp")
>
> d<-c(1:100)
> length(d)
[1] 100
> size(d)
[1] 100

```

The R, RStudio, Rcpp and Rtools installation is working nicely and one can now quite easily write, compile, and run C++ functions in R to perform frequency stability analysis. In particular, these functions can leverage existing C code to obtain better performance along with the convenience of R.

### Timing Function Execution

The time required for a function to execute in R can be determined by successive calls to the `Sys.time()` function:

```

t1<-Sys.time()
some_function_to_be_timed()
t2<-Sys.time()
t2-t1

```

where the code needs to be executed as a block. For example, we can generate 10,000 points of W FM noise phase data using the `noise()` function from the `fsa.R` package with:

```

> d<-noise(10000,0,1,0,1)

```

and time the execution of the `theo1()` function from the `fsa.R` package with:

```

> t1<-Sys.time()
> theo1(d,1,5000)
[1] 0.0007709487
> t2<-Sys.time()
> t2-t1
Time difference of 2.884525 secs

```

We can compare that execution time with the same `Théo1` function (named `theo` instead of `theo1`) implemented in C++ code as follows:

```

#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]

double theo(NumericVector x, double tau, int m)
{
  // Note: x is 0-based
  // Local variables
  int i; // Outer index
  int d; // Inner index
  int N; // # phase data points
  double t1; // Theo1
  double sum; // Inner sum
  double s; // Partial sum

  // Initializations
  N=x.size();
  t1=0;

  // Outer sum
  for(i=1; i<=(N-m); i++)
  {
    sum=0;
    // Inner sum
    for(d=0; d<=((m/2)-1); d++)
    {
      s=(x[i-1]-x[i-d+(m/2)-1]+x[i+m-1]-x[i+d+(m/2)-1]);
      sum+=(s*s/((m/2)-d));
    }
    t1+=sum;
  }

  // Scaling factor
  t1/=(0.75*(N-m)*m*m*tau*tau);

  // Return Theo1 deviation
  return sqrt(t1);
}

```

The theo() function is compiled with:

```
> sourceCpp("C:\\R\\theo.cpp")
```

and executed, with timing, with:

```

> t1<-Sys.time()
> theo(d,1,5000)
[1] 0.0007709487
> t2<-Sys.time()
> t2-t1
Time difference of 0.01994205 secs

```

The R version of Théo1 takes a significant 2.88 seconds to execute while the C++ version takes only 0.02 second, clearly demonstrating the big advantage of using C++ to implement lengthy functions in R.

As a final example, here is a C++ function that can be used with R to calculate the Total deviation from phase data. Note that any needed bias correction must be applied separately.

```

/*****
/*
/*          totdev()
/*
/*  Function to calculate TOTDEV using doubly reflected phase data
/*  C++ code for use with the R statistical computing environment
/*
/*  Parameters:   NumericVector x = phase data (double)
/*                double tau     = data sampling interval
/*                int af         = analysis averaging factor
/*
/*  Return:      double          = TOTDEV
/*                or -1 if bad argument error
/*                or -2 if memory alloc error
/*                or -3 if no result error
/*                or -4 if negative variance error
/*
/*  Install:     sourceCpp("path to tc.cpp")
/*
/*  Call:        totdev(x, tau, af)
/*
/*  Notes:       1. Adapted for C++ from FrequenC.DLL TotvarCalc()
/*                for use with R.
/*                2. Function signature changed.
/*                3. Windows/WIN32 code/style removed.
/*                4. Phase data vector w/o start, end or # points.
/*                5. No gap handling.
/*                6. No progress indicator.
/*                7. Calculation is done entirely with a new array.
/*                that is deleted after the function closes.
/*                8. The phase data need NOT be endmatched before
/*                calling this function.
/*
/*  Reference:   D.A. Howe and C.A. Greenhall, "Total Variance:
/*                A Progress Report on a New Frequency Stability
/*                Characterization", Proc. 29th PTTI Meeting,
/*                December 1997.
/*
/*  Revision record:
/*      06/03/20  Adapted from TotvarCalc() of FrequenC.DLL
/*
/*  (c) Copyright 1997-2020 Hamilton Technical Services License: MIT
/*
*****/

```

```

// Headers
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]

// TOTDEV Calculation
double totdev(NumericVector x, double tau, int af)
{

```

```

// Local variables
int i;           // Index
int j;           // Auxiliary index
int n;           // # points (in full vector x)
int c=0;         // # analysis points

double *p=NULL; // TOTVAR phase array pointer
double s=0.0;   // TOTVAR summation
double e;       // Phase data value at edge of reflection
double totdev;  // Total deviation

// Initializations
n=x.size(); // # phase data points = N

// Check arguments
// AF must be >= 1, tau must be > 0, Max AF is N-1
if( (af<1) || (tau<=0.0) || (n<af+1))
{
    return -1.0; // Bad argument error
}

// Allocate a new "virtual" phase data array to size 3N-4, where N
// is the # of phase data points. This virtual array is the result
// of extension by reflection about both endpoints. N-2 reflected
// points are added at both ends of the original phase data.
p=new(std::nothrow) double [3*n-4];
if(p==NULL)
{
    return(-2.0); // Error - memory allocation failed
}

// Note that in the referenced paper the index of the virtual phase data
// array goes from 3-N (a negative number) to 2N-2 (a positive number),
// with the original data having indices from 1 to N. Our indices start at
// 0 and go to 3N-5. The lower reflected data has indices from 0 to N-3.
// The original data in the middle of the virtual array has indices from
// N-2 to 2N-3. The upper reflected data has indices from 2N-2 to 3N-5.

// Copy original phase data array x[] to (headerless) working array p[]
// i is the index into the virtual array p[]. j is the variable part
// of the index into the original phase data array x[].
j=0;
for(i=n-2; i<2*n-2; i++)
{
    p[i]=x[j];
    j++;
}

// Fill the lower reflected phase data
// These values are twice the first phase data point minus the particular
// data point value to be reflected.
j=0;
e=2*p[n-2];

```

```

for(i=0; i<n-2; i++)
{
    p[i]=e-p[2*n-4-j];
    j++;
}

// Fill the upper reflected phase data
// These values are twice the last phase data point minus the particular
// data point value to be reflected.
j=0;
e=2*p[2*n-3];

for(i=2*n-2; i<3*n-4; i++)
{
    p[i]=e-p[2*n-4-j];
    j++;
}

// Calc TOTVAR - See Eq (3) of Reference
for(i=n-1; i<2*n-3; i++)
{
    // Sum 2nd differences squared
    s+=(p[i-af]-2*p[i]+p[i+af])*(p[i-af]-2*p[i]+p[i+af]);
    c++;
}

// Scale result - See Eq (3) of Reference
if(c)
{
    s/=(tau*tau*af*af*2*c);
}
else
{
    return -3.0; // No results error
}

// Find TOTDEV
if(s>0.0)
{
    totdev=sqrt(s);
}
else
{
    return 4.0; // Negative variance error
}

// Free memory
delete [] p;

return(totdev); // Return Total deviation
}

```

```

/*****/

```