

Analyzing Incomplete Time Series Data with
CLEAN

Zachary Kessin

13 May 2003

Abstract

In astronomical observation it is common to have incomplete data. From the 24 hour cycle of the Earth's rotation and other sampling effects it is frequent to have data that are sampled less than half of the time. Blindly running a Fourier transform on data with such a duty cycle will produce huge side lobes which are very difficult to tell from the true signal. In *Roberts et al (1987)* a method was described using CLEAN algorithm, which was adopted from aperture synthesis, to correct for this effect. By deconvolution of the spectrum from the beam an accurate picture of the original spectrum can be arrived.

In this thesis I will attempt to show how to apply these methods when in addition to sampling effects the input domain is also influenced by noise. I will show some results on how to separate real features of the spectrum from noise when there is an understanding of what the noise should be.

<i>CONTENTS</i>	3
-----------------	---

Contents

1 Defining the Problem	4
2 CLEAN on data with no noise	4
2.1 How CLEAN works	7
2.2 Using CLEAN on a simple spectrum	12
2.3 Using CLEAN with partially sampled data	12
2.4 Using CLEAN on a more complex spectrum	12
3 CLEAN on data with Noise	16
3.1 How to run clean on data with Noise	17
3.2 Properties of Noise	20
3.3 Relationship of Number of CLEANS to SNR in the CLEAN spectrum	23
4 Conclusions	24
A A Short Introduction to the Scheme Language	27
A.1 Scheme Resources	27
A.1.1 Web pages on Scheme	27
A.1.2 Books on Scheme	27
B Numeric Experiments	28
B.1 Mean High Peak	31
B.2 Noise Level Test	37
C Data Formats	41
D Scheme Code	43
D.0.1 Scheme Utility Code	44
D.0.2 Scheme works code	53
D.0.3 Scheme Test Code	58
E Presentation code	60
F Fortran Code	64
G Gnuplot Code	83

1 Defining the Problem

In observational astronomy it is often the case that the data that is in hand for observations of an object are less than perfect. In addition to a finite sampling rate which can cause ringing in a spectrum and other problems, there is the Earth's 24 hour day/night cycle, and depending on the season and positions of an object in the sky it is possible that in some cases it will only be observable 6 hours a day or less. Add to this the uncertainty of weather and observatory scheduling can create quite a headache for an observer.

It may seem to make sense to take the data as acquired and blindly run a Fourier transform over it. This is not a good idea. The effects of missing data on the spectrum can be quite large. The CLEAN algorithm had been shown to be quite adept at dealing with this problem.

2 CLEAN on data with no noise

When there is no noise in the input spectrum, the results of CLEAN are reasonably well understood. CLEAN will remove all significant artifacts with a gain of 0.1 and a sufficient number of cleans; 500 to 1000 cleans in most cases will suffice to separate the actual signal from side lobes and other artifacts produced as a result of data sampling.

Fourier's theorem tells us that for any function $f(t)$ there is an equivalent

function $F(\nu)$ such that

$$F(\nu) \equiv FT[f(t)] \equiv \int_{-\infty}^{+\infty} dt f(t) e^{-2\pi i \nu t} \quad (1)$$

and assuming that $f(t)$ is real then the negative frequencies will be the complex conjugates of the positive ones, so $F(-\nu) = F^*(\nu)$.

However a real sampled signal is not defined for all time. The signal is sampled at a finite number of real data points which have some minimum separation and some maximum resolution. The sampling function $s(t)$ can be best defined as weighted sum of Dirac delta functions, $\delta(x)$. A good definition is:

$$s(t) = C \frac{\sum_{r=1}^N w_r \delta(t - t_r)}{\sum_{r=1}^N w_r} \quad (2)$$

The constant C and w_r are defined to be one if all the sample points have equal weights. Thus the sampled signal is:

$$f_s(t) \equiv f(t)s(t) = \frac{1}{N} \sum_{r=1}^N f(t) \delta(t - t_r) \quad (3)$$

Given Fourier's theorem it follows that the Fourier transform of a sampled signal $D(\nu) \equiv FT[f_s]$ is the convolution of the spectrum with the Fourier transform of the sampling function, $W(\nu) \equiv FT[s]$:

$$D(\nu) = F(\nu) \otimes W(\nu) \equiv \int_{-\infty}^{+\infty} d\nu' F(\nu') W(\nu - \nu'). \quad (4)$$

We will refer to the functions $D(\nu)$ and $W(\nu)$ as the Dirty Spectrum and

the Spectral Window functions¹ respectively. They can be computed simply by the formula, which reduce to the summation in the case of a discrete Fourier transform:

$$D(\nu) = \int_{-\infty}^{+\infty} dt f_s(t) e^{-2\pi i \nu t} = \frac{1}{N} \sum_{r=1}^N f_r e^{-2\pi i \nu t_r} \quad (5)$$

$$W(\nu) = \int_{-\infty}^{+\infty} dt s(t) e^{-2\pi i \nu t} = \frac{1}{N} \sum_{r=1}^N e^{-2\pi i \nu t_r} \quad (6)$$

When data are sampled by a computer it is of course not sampled in a continuous way but by taking a reading once per period Δt . This results in a function that can best be described as a set of delta functions approximating the shape of the curve. This sampling function will produce a ringing in the Fourier spectrum. Changing the sample rate and other parameters of the Fourier transform will affect the specifics of this ringing.

Normally the Fourier transform is computed such that the frequency separation is one over the separation in time of the data points.

$$\Delta\nu = \frac{1}{\Delta t}$$

If the data are uniformly sampled over a period of time, then for N data samples we can use the recursive Fast Fourier transform which will compute the Fourier series in time proportional to $O(N \log N)$, where N is the number of data points in the original data. However in this problem the data can not be assumed to be uniformly sampled. Missing data points and a partial

¹Sometimes also called the “beam” in astronomy.

duty cycle create a time series that is not uniform in its spacing. Therefore the Fast Fourier Transform can not be used here, and we must use the slower normal discrete Fourier transform. This takes time proportional to the square of the number of data points $O(N^2)$.

Under normal circumstances one will compute the Fourier spectrum at a series of frequencies ν_r such that

$$\{\nu_r\} \equiv \left\{ \frac{r}{\Delta f} \right\}, r = 1, 2, \dots, N$$

However one can put in any real value for ν and in some cases it makes sense to adjust the spacing of the computed frequencies. We often wish to compute the frequency spacings such $\Delta\nu$ is less than the default value. In most of the experiments that we performed we sampled 4 points per beam “PPB”, Which is to say that

$$\Delta\nu = \frac{1}{4\Delta t}$$

plots shown in this thesis should be assumed to have been computed at 4 ppb unless otherwise noted.

2.1 How CLEAN works

To apply CLEAN start with a single harmonic component with amplitude A frequency $\hat{\nu}$ and phase constant ϕ ,

$$f(t) = A \cos(2\pi\hat{\nu}t + \phi) \tag{7}$$

The spectrum of f is

$$F(\nu) = a\delta(\nu - \hat{\nu}) + a^*\delta(\nu + \hat{\nu}) \quad (8)$$

where

$$a = \frac{A}{2}e^{+i\phi} \quad (9)$$

The spectrum is complex and contains a pair of components at $\pm\hat{\nu}$ with amplitudes of $A/2$. Which is why a wave with amplitude A will transform to a peak with amplitude $A/2$ the phases of the two components are $\pm\phi$, respectively. The sampled data gives a dirty spectrum of

$$D(\nu) = aW(\nu - \hat{\nu}) + a^*W(\nu - \hat{\nu}) \quad (10)$$

Each real peak casts a set of side lobes radiating outward from that feature. And these features can reflect across the 0 frequency. As the negative frequencies are defined (as the complex conjugate of the positive frequencies) Then each feature will produce a real peak at $\pm\hat{\nu}$ and a set of side lobes that radiate out from those peaks. Each real peak will produce a set of side lobes that radiate out from that peak, with the signature of the dirty beam. In the case of a 50% duty cycle the side lobes for each peak will be that peak convolved with the Fourier spectrum of a square wave, centered at the spectral feature $\hat{\nu}$. Since the features are both positive and negative, the side lobes of both halves will be present in the positive spectrum.

However, we may take the contamination into account by noting that

in the case of a spectrum containing a single harmonic element, the dirty spectrum will be

$$D(\hat{\nu}) = aW(0) + a^*W(2\hat{\nu})$$

at the frequency of that component. Using the equations above we can find the amplitude of that component in terms of its frequency,

$$a = \frac{D(\hat{\nu}) - D^*(\hat{\nu})W(2\hat{\nu})}{1 - |W(2\hat{\nu})|^2} \quad (11)$$

Because of the contamination of the positive and negative frequency peaks by the sidelobes of the other peaks it is often not possible to determine the exact frequency of even a single spectral component from the dirty spectrum. However, the peaks $\tilde{\nu}$ of D usually lie within $\delta\nu$ of the corresponding spectral components. Given the frequency of clean component estimated from D , its complex amplitude is found from a function defined similarly to the previous function

$$\alpha = \frac{D(\nu) - D^*(\nu)W(2\nu)}{1 - |W(2\nu)|^2} \quad (12)$$

If CLEAN is used properly it will make small errors in doing this, but it will correct them in further iterations. There are a few cases where sidelobes may combine to form a peak higher than any real peak that could cause CLEAN to fail. Or clean may fail if there is too much noise in the signal.

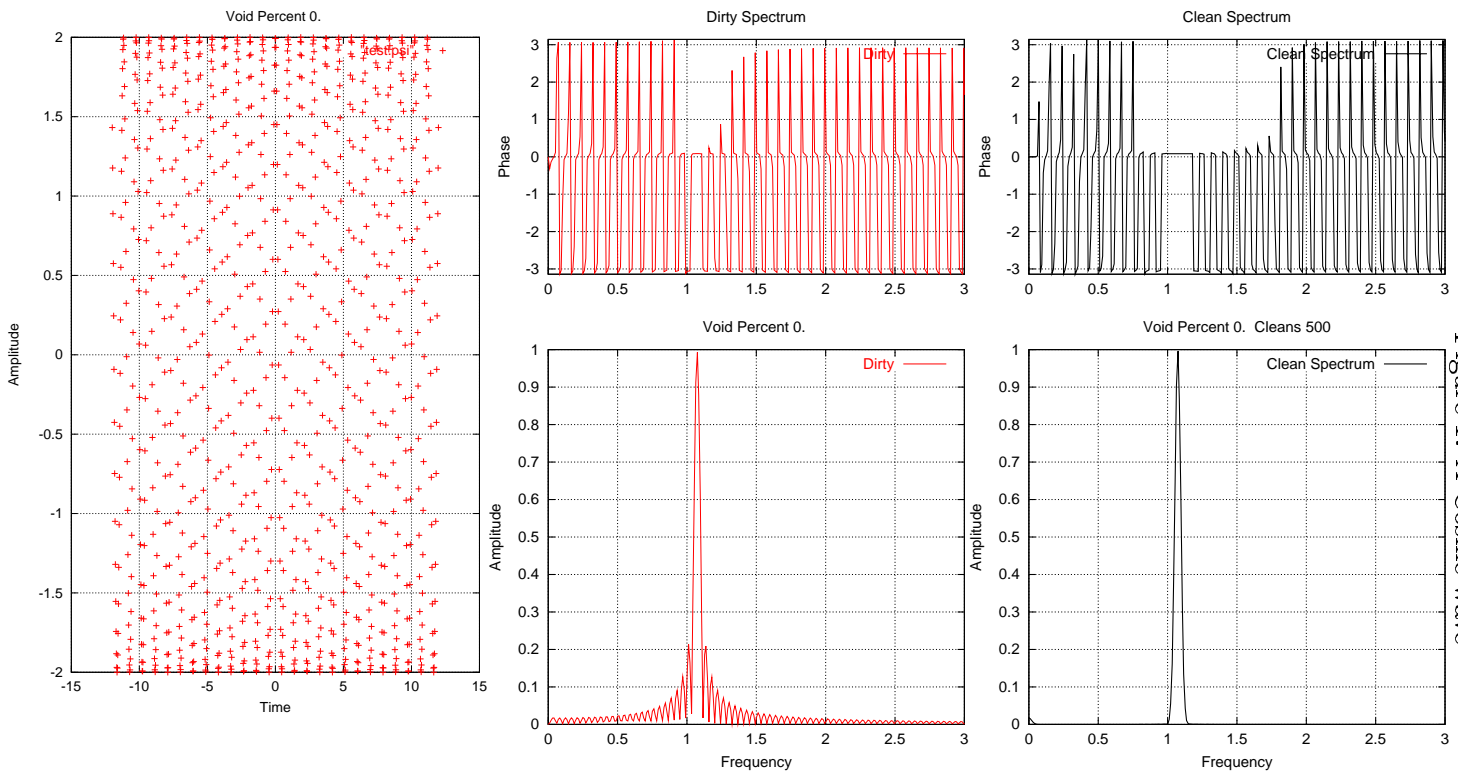


Figure 1: A Cosine wave

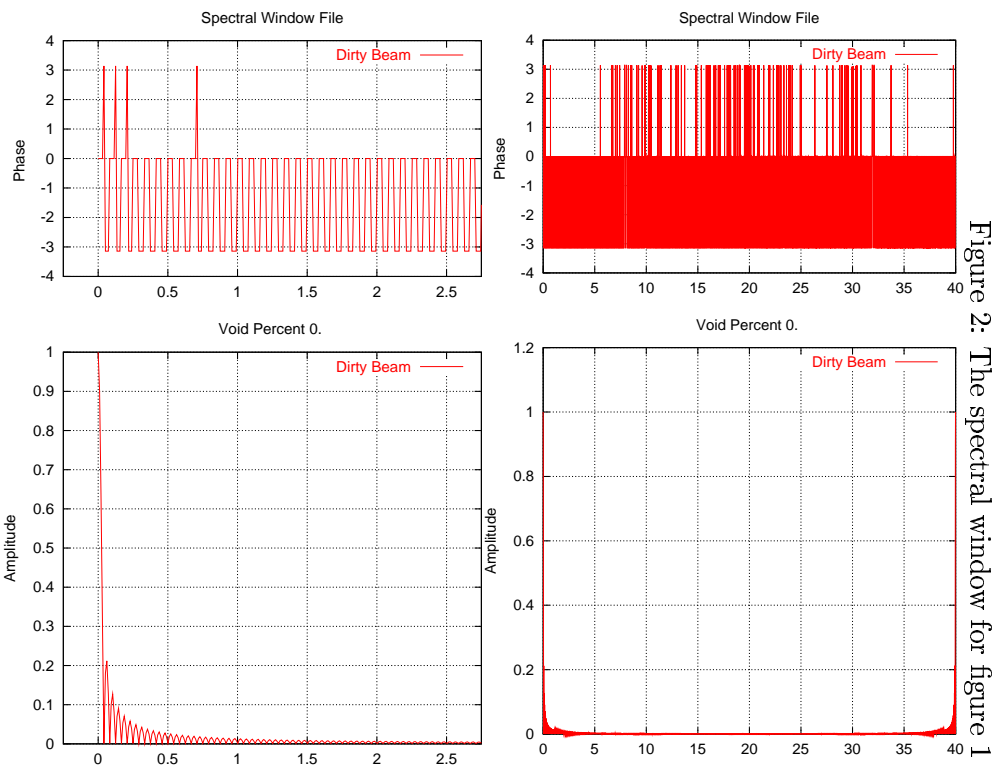


Figure 2: The spectral window for figure 1

2.2 Using CLEAN on a simple spectrum

In the simple case of a simple cosine wave with a reasonable sampling rate ($\Delta\nu < \nu_{nyquist}$) it is reasonably simple to visually pick out the real features in a dirty spectrum. See figure 1 on page 10. However even in this case it is easy to see the effects of the sample function in the dirty spectrum. However in the clean spectrum effectively all of the sampling effects have been removed.

2.3 Using CLEAN with partially sampled data

However, if in addition to the sampling frequency we also impose a duty cycle, and drop 50% or even 75% of the data, things get significantly more complex. In this case the side lobes can grow to 90% or more of the height of the highest peak. The dirty spectrum shows a great deal of artifact from the missing data.

Figure 3 shows a simple cosine wave from which was dropped 75% of the datapoints. In the Dirty spectrum side lobes rise to over 90% of the main central peak. After 500 CLEANS all visible artifacts of the sampling function have been removed. The clean spectrum $c(\nu)$ is at this point a very good representation of $F(\nu)$, the spectrum of the original signal $f(t)$.

2.4 Using CLEAN on a more complex spectrum

Figure 3: Cosine wave with a duty Cycle

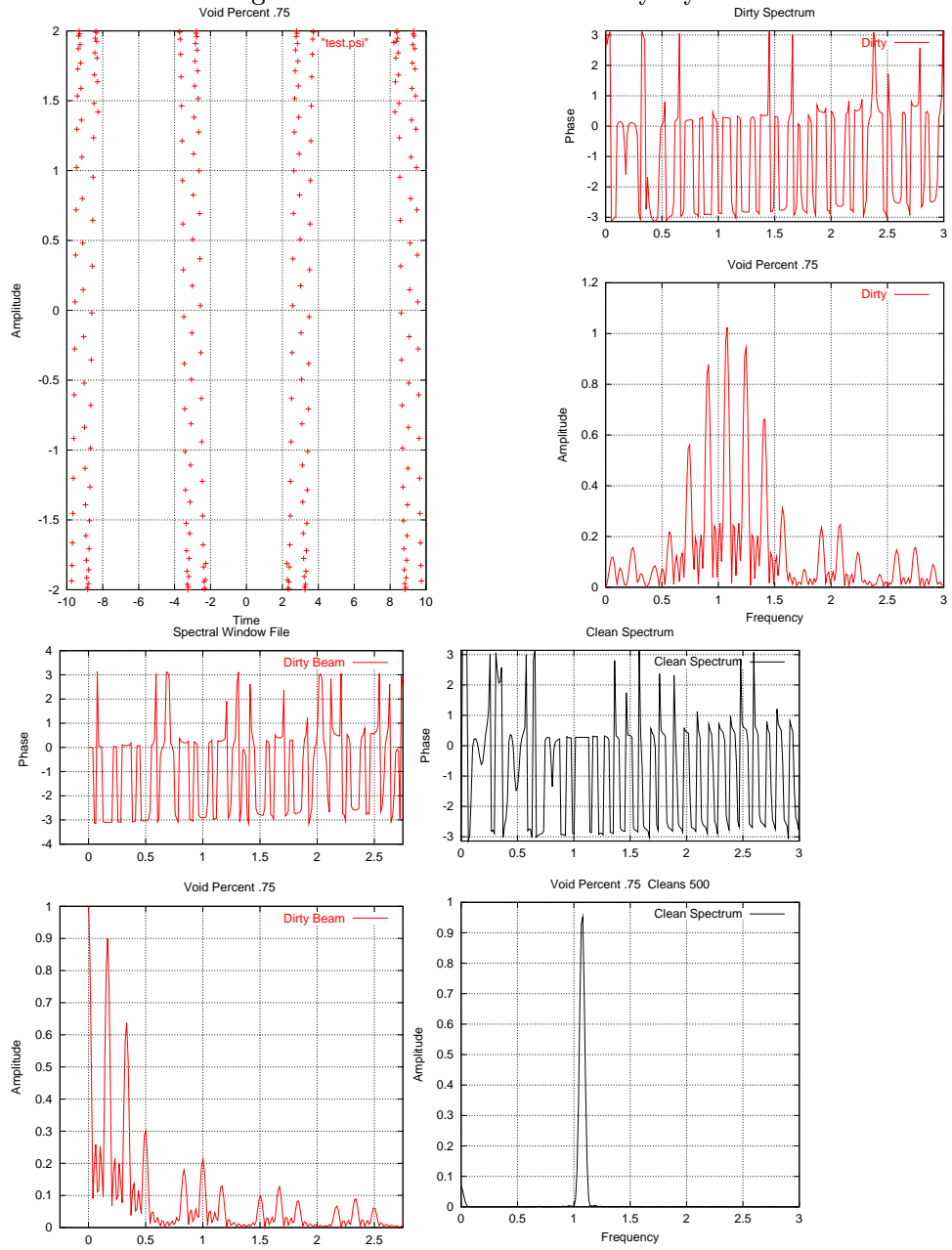


Figure 4: A complex spectrum

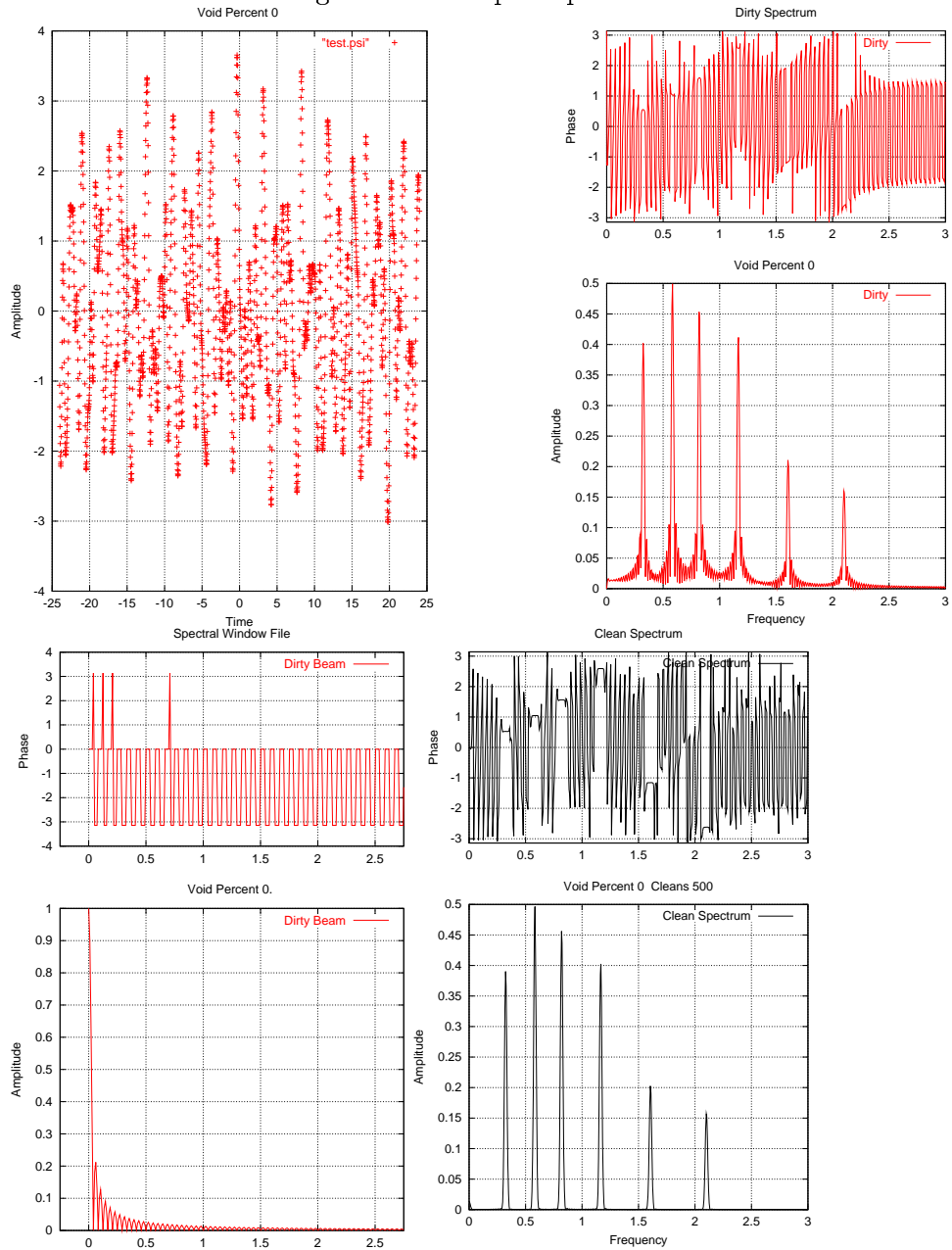
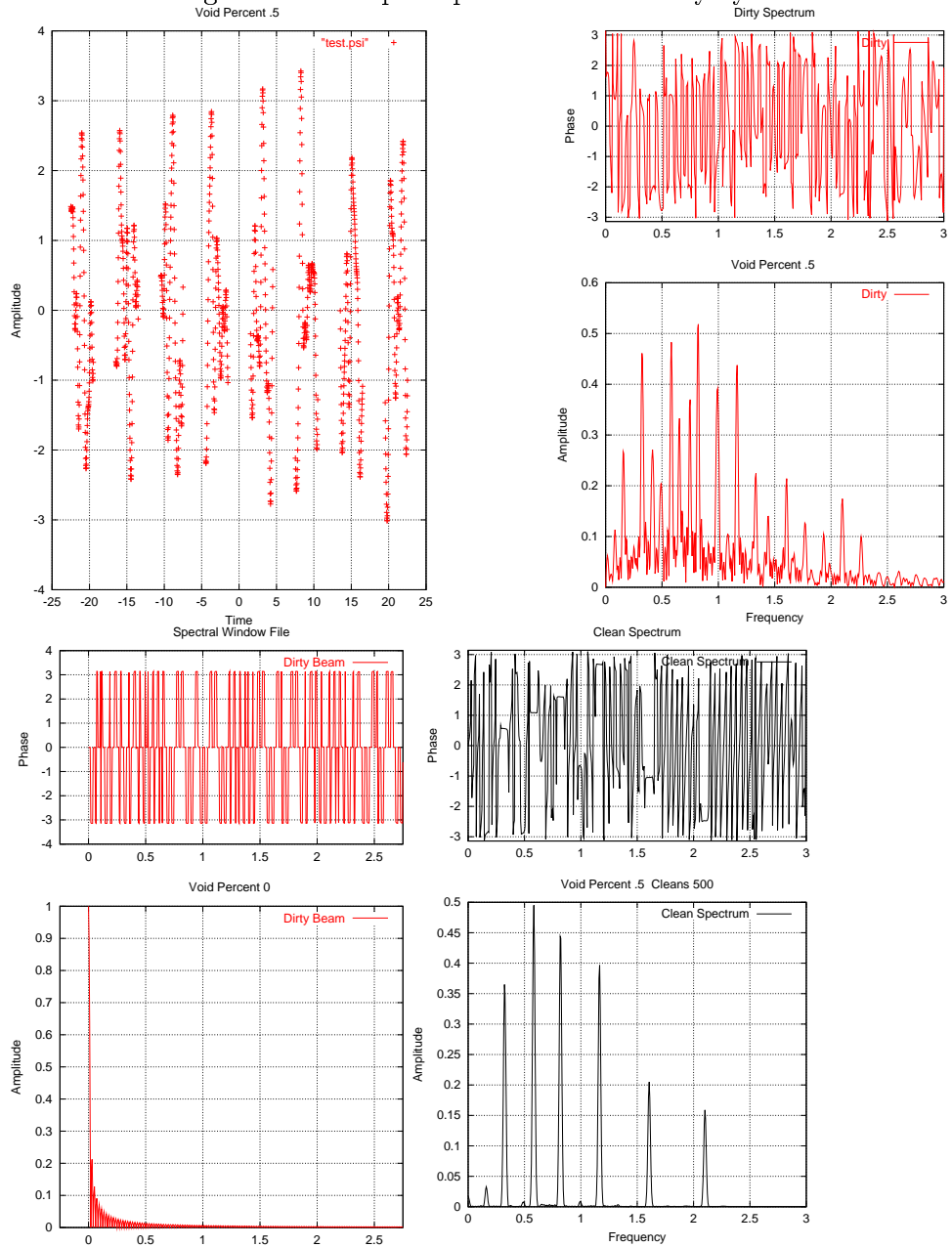


Figure 5: A complex spectrum with a duty cycle



Given a spectrum with a single real peak it is rather simple to see what is real and what is sampling effect. However when there are multiple real peaks in a spectrum things get much more complex. Each of the real features in the spectrum create a pattern of side lobes that radiate outward from that feature, these side lobes then overlap and interfere with each other and the real peaks. Each pattern of side lobes is the convolution of the real peak with the beam. So in the case of a single peak the side lobes are easy to distinguish. However Given a spectrum with several real features and a 50% duty cycle it is impossible to select the real peaks from the side lobes by eye. Side lobes rise to over 75% of the height of the highest real peak and are higher than some of the lesser real peaks. CLEAN however can remove the artifacts with great effect.

3 CLEAN on data with Noise

The artificial data samples so far have been made with no noise. I created a pure signal and imposed a 50% duty cycle to simulate the rising and setting of a source or a day/night cycle. When noise is added to the signal it complicates the picture. Given a spectrum with noise, it is much harder to sort out real peaks from artifacts. When there is no noise there is only the effect of the original signal $f(t)$ and the sampling function $s(t)$; however, when noise is added to the signal then there is of course added to that the “hum” of the noise. Given a plot of just noise with no signal then the

spectrum of the noise should be spread across the range of frequencies in an approximately uniform distribution. Given a noise with a strong signal it is reasonably easy to see the true signal as the noise tends to spread uniformly across the spectrum. (See figure 7 on page 19 which has noise and figure 8 which has noise and a 50% duty cycle).

3.1 How to run clean on data with Noise

Given data as I have described, with noise in as well as with a duty cycle the question is how best to employ CLEAN. The user must decide how many points per beam to compute in the Fourier spectrum and then what gain and how many times to run CLEAN. If CLEAN is not allowed enough iterations it will leave many of the artifacts in a spectrum. Running CLEAN too many times will apparently not harm the output, but will take a very long time to run. A test run of with 5,000,000 iterations of CLEAN, which is probably more than would ever be needed, shows a result that is roughly identical to a run with 5,000 CLEANs. However not surprisingly the test with 5,000,000 CLEAN iterations took a lot longer to run. Therefore a balance should be sought between to few runs leaving artifacts and too many which will simply use to much computer time. In our tests a number between 500 and 5,000 CLEANs seemed to fit the bill.

Figure 6: Clean at different levels

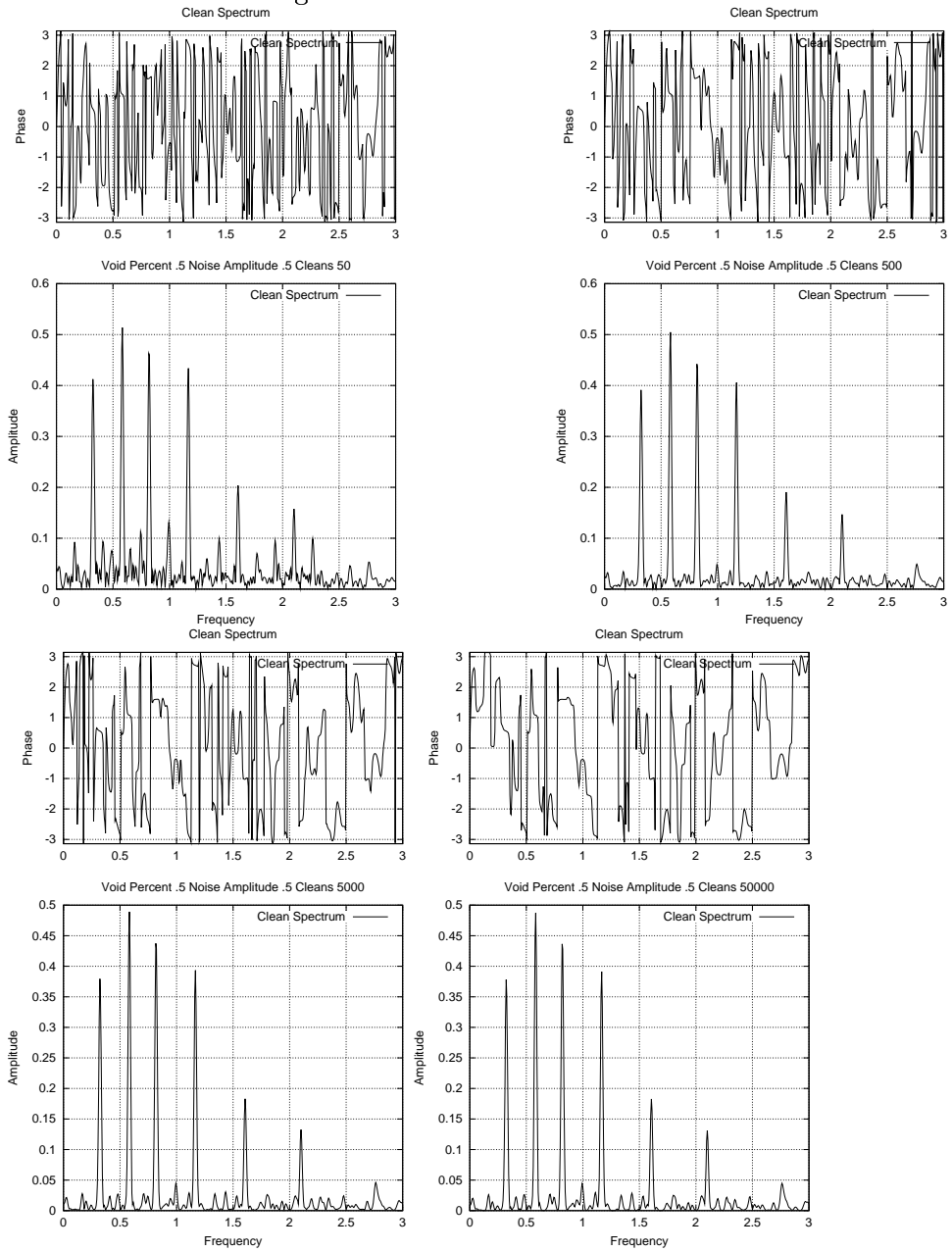
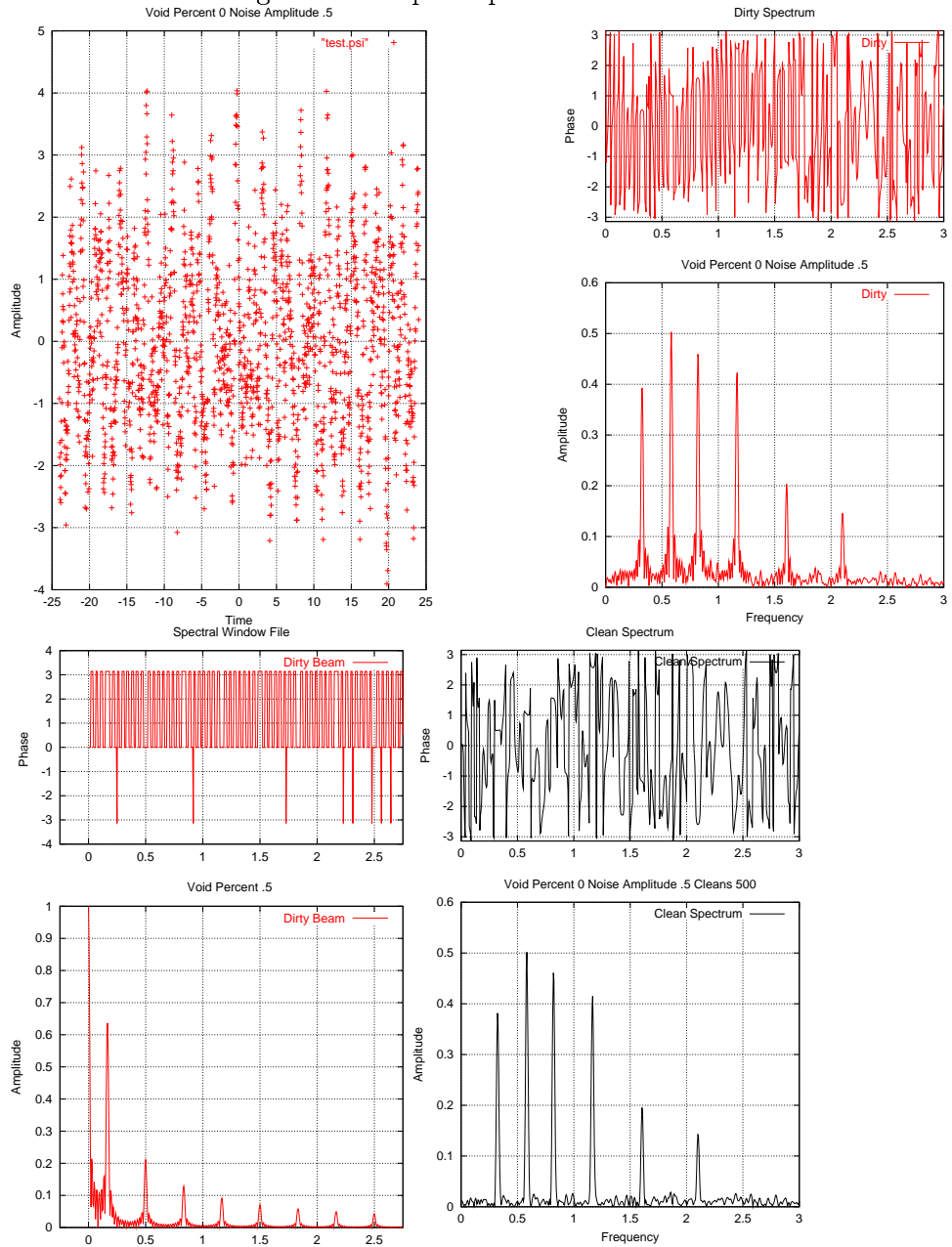


Figure 7: Complex Spectrum with noise



3.2 Properties of Noise

We generated a spectrum as above and added a Gaussian noise to it with a variable noise amplitude. By holding the signal amplitude constant and changing the noise amplitude we were able to test CLEAN with a number of different parameters. We plotted noise amplitude vs. number of CLEANs. We removed the real signal from these test and plotted the mean value of the highest peak left in the spectrum.

The noise used here was generated with a random number generator with a Gaussian distribution. Specifically it was generated using a scheme port of the code for a random number generator in “Numerical Recipes in C”. The random seeds were taken from the Linux */dev/random* file which should provide a strong source of random numbers.

As all of the descriptions here are on strict numeric experiments units are somewhat arbitrary. The Noise amplitude is defined such that if it is one, then, the noise when viewed in time space, will be a Gaussian with a standard deviation of 1. In phase space it will result in a hum across the entire spectrum.

Figure 8: Complex Spectrum with noise and a 50% duty Cycle

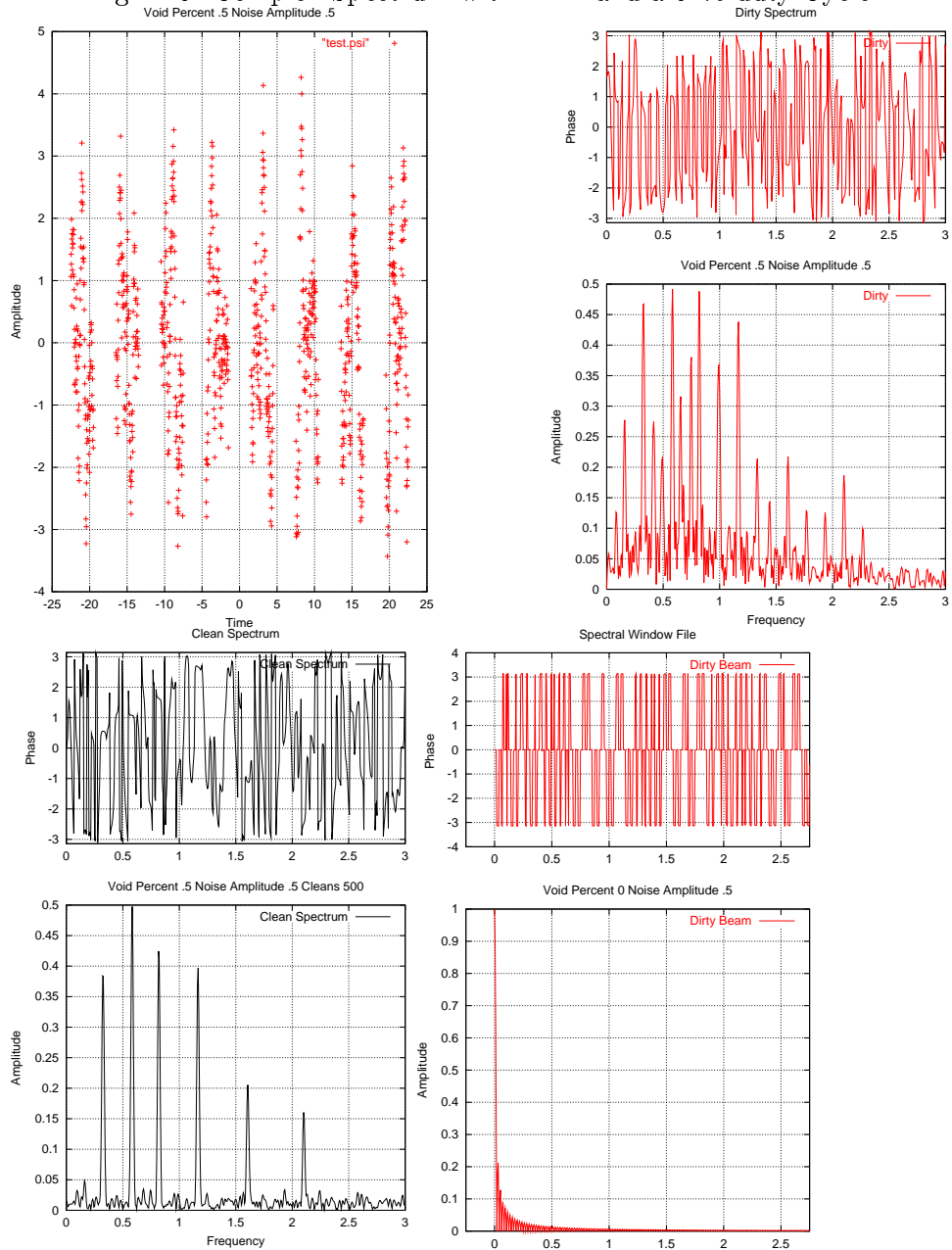
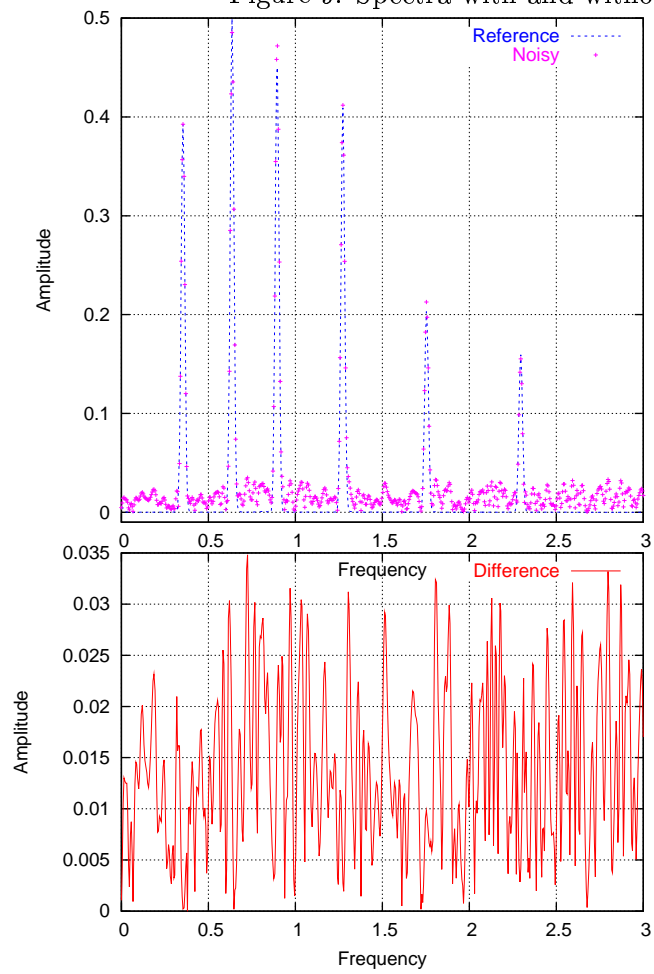


Figure 9: Spectra with and without noise



3.3 Relationship of Number of CLEANs to SNR in the CLEAN spectrum

Given a signal with noise the relevant question must be, after running a Fourier transform, what in the left-over spectrum is a real signal and what is a product of the sampling function $s(t)$ and what is a product of the noise. As I have shown if there is no noise in the signal then CLEAN will remove effectively all of the results of the sampling function. However if there is noise in the spectrum things get more complex. The noise used in these tests was a Gaussian signal as provided by “Numerical Recipes” and therefore contributes noise across the spectrum of the signal. This noise overlies both real features and artifacts and makes it difficult to sort out smaller real peaks from the noise.

To analyze resulting noise in a clean spectrum it is first necessary to remove the real signal. To do this I created a reference spectrum with the same signal and the same duty cycle that I would use for the noisy spectra. I applied a large number of CLEANs (10,000) to the reference spectrum. I then created a noisy spectrum that had the same base signal and duty cycle as the reference signal. The remaining noise in this case was defined as the absolute value of the difference of the absolute value of the two spectra. (The spectra are complex and phase is ignored here.)

$$s_{NoiseOnly}(\nu) = ||s_{Reference}(\nu)| - |s_{WithNoise}(\nu)|| \quad (13)$$

This approach also finds the components of the noise that may happen

to have a frequency that overlaps a real feature in the spectrum. In the clean spectrum without noise at first appearance all points not on one of the peaks seem to be 0. However on closer look the reference spectrum does have some very small residual side lobes. These are not visible on the plot unless the y-axis is blown up or plotted on a log scale. The reference plot shows a part of the plot with an exaggerated X and Y axis (See figure 9 on page 22)

I then took the mean value highest peak left over and plotted it against the amplitude of the noise that was input into the initial function. (See figure 10 on page 26). The center of this plot is roughly linear. The value of the highest remaining peak goes up linearly with the noise level. And down with the number of CLEANs. More CLEANs seem to lower the noise level up to a point at which they seem just to shuffle the noise around.

It appears that running CLEAN 1000 times seems to have a definite improvement over running CLEAN 500 times. More CLEANs seem to lower the mean high peak a little bit but not a significant amount. The difference between the 500 line and the 5000 line on figure 10 is quite small, at a noise amplitude of 0.5 the difference is between about 0.04 and 0.045.

If an estimate of the noise in the original signal can be made then it should be possible to determine which of the peaks in a spectrum are real and which are a result of the noise.

4 Conclusions

CLEAN seems to deal reasonably well with noise in the input spectrum, assuming at least a few basic criteria are met. First the noise level is

reasonably low. This can be best quantified as saying that the mean value of the noise in phase space is smaller than the value of the real data.

A A Short Introduction to the Scheme Language

The Scheme programming language is a derivative of *Lisp* that has been designed for simplicity. It was originally created by Guy Lewis Steele and Gerald Jay Sussman at MIT in the 1970's. The version of scheme that has been used in this project is Gambit Scheme by Marc Feeley from the University of Montreal.

Scheme features a very simple syntax, built in complex numbers, a solid debugger, and a knowledgeable user base. It also considers functions as first order data, which is to say that a function can be passed in any variable. Scheme also allows for very fast prototyping of ideas.

For a quick introduction to scheme see Teach Yourself Scheme in Fixnum days, or the book the *The Little Schemer*. For the official scheme standard see the *The Revised(5) report on the Algorithmic language Scheme (R5RS)*.

A.1 Scheme Resources

A.1.1 Web pages on Scheme

- Schemers.org a general scheme web page start here : <http://www.schemers.org>
- Gambit Scheme System:
<http://www.iro.umontreal.ca/~gambit/>
- Teach Yourself Scheme in Fixnum days:
<http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html>
- The Scheme Programming language: <http://www.scheme.com/tspl2d/index.html>
- The Revised(5) report on the Algorithmic language Scheme (R5RS):
The official standard of the scheme language
http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html

A.1.2 Books on Scheme

- Structure and Interpretation of Computer programs, (SICP) by Harold Ableson and Gerald Jay Sussman, can also be found online at <http://mitpress.mit.edu/sicp/full-text/book/book.html>
- Friedman, Daniel, The Little Schemer.
- Friedman, Daniel, The Seasoned Schemer

B Numeric Experiments

These files make up the numeric experiments that were used in this thesis.

gain_test.scm

```
;; This is the basic function that crates a wave and runs a Fourier transform and CLEAN
;; on the data. the tr and tc functions down bellow wrap this nicely and feed in
;; Reasonable default values for some of the paramaters.
;;

(define (test-fc ppb percent void_period noise-amp gain clean_list psi t1)
  (define delta 0.025)
  (define (dx old)
    (+ old delta ))

  (let*
    (
      (fileroot "test")
      (plot-title (string-append "set title \"Void Percent "
        (number->string percent)
        (if (> noise-amp 0)
          (string-append " Noise Amplitude "
            (number->string noise-amp))
            " ")
        "\""))
      (noise-func (lambda () (* noise-amp (grandom 1))))
      (time-data (print-tabular-data
        (generate-mean-wave (- t1) t1 dx
          (make-void void_period 0 percent)
          psi noise-func)
        (string-append fileroot ".psi"))))
      (for-each (lambda (cln)
        (let* ((cr
          (fourth_and_clean time-data 4 gain cln))
          (dsf (car cr))
          (swf (cadr cr))
          (rsf (caddr cr)) ;; Find highest value here, this is the noise
          (ccf (caddr cr))
          (csf (caddr (cdr cr))))
          ; (rcon (recon_wrap ccf dsf 0 (- t1) t1 0 0 )))
          (print-amp-phase-data dsf (string-append fileroot ".pdsf"))
```

```

                (print-amp-phase-data rsf (string-append fileroot ".prsf"))
        (print-amp-phase-data ccf (string-append fileroot ".pccf"))
        (print-amp-phase-data csf (string-append fileroot ".pcsf"))
        (gnuplot "wave.gp" (list plot-title (if printp (gnuplot_print_ps))))
; ; (gnuplot "fc.gp" (list plot-title (if printp (gnuplot_print_ps))))
        (gnuplot "dirty.gp" (list plot-title (if printp (gnuplot_print_ps))))
        (gnuplot "clean.gp" (list (string-append "set title \"Void Percent \"
                (number->string percent)
                (if (> noise-amp 0)
        (string-append " Noise Amplitude \"
(number->string noise-amp))
" ")
        " Cleans \"
        (number->string cln) "\\\"")
        (if printp (gnuplot_print_ps))))
;   (gnuplot "swf.gp" (list plot-title (if printp (gnuplot_print_ps))))
    ))
clean_list)))

```

```
;; These are wrapper functions that are quite useful in experiments)

```

```

(define (tc na vper vp gain cleans psi units)
  (test-fc 4 vp 5.3
    na
    gain cleans psi
    (* 3 units units)))

```

```

;; This is a wrapper that gets used a lot,
;; It presets most of the things that can be preset like the length of the duty cycle
;; and the clean gain.
;;

```

```

(define (tr noise-amp data-length vpercent cleans psi)
  (test-fc 4 vpercent 6 noise-amp .1 cleans psi data-length))

```

B.1 Mean High Peak

This file is the outer wrapper for the mean high peek experiment. In the first parameter is the number of times to run the test and the second is the number of cleans to run. “stats_run.pl” will put itself in the background and then run the “stats_test.scm” script with a *nice -3* as so not to take up the entire CPU of the computer its on.

In truth the experiment was often started by hand so that each process could be run on a separate computer. The software is designed to run in parallel on one or more computers.

run_test.sh

```
#!/bin/bash
##
export clean_runs=50

echo "*"
./stats_run.pl \${clean_runs} 10

echo "*"
./stats_run.pl \${clean_runs} 50

echo "*"
./stats_run.pl \${clean_runs} 100

echo "*"
./stats_run.pl \${clean_runs} 500

echo "*"
./stats_run.pl \${clean_runs} 1000

echo "*"
./stats_run.pl \${clean_runs} 5000

echo "*"
./stats_run.pl \${clean_runs} 10000

echo "*"
./stats_run.pl \${clean_runs} 50000
```

```
gnuplot clean_test_100.gp
rm dfile*
```

`stats_run.pl` This script runs the mean high peak experiment.

```
#!/usr/bin/perl -w
## This section causes the output
## of the carp, warn, die etc to go to
## the stats_run_log file, which can be
## monitored by the user.
BEGIN
{
    use CGI::Carp qw(carpout);
    open (LOG,">>stats_run_log") or
        die "Can't open log \${!}";
    carpout(LOG);
}
use strict;
my \$runs          = shift;

my \$cleans        = shift;

my @noise_level = (0,0.1,0.2,0.3,0.4,0.5,
    0.6,0.7,0.8,0.9,1.0,
    1.1,1.2,1.3,1.4,1.5);

open(OUTFILE,"> clean_test_" . \$runs . "_" . \$cleans . ".dat")
    || die "Can't open output file";
## this puts the process into the background and
## detaches it from the terminal
fork && exit;
fork && exit;
select (OUTFILE);
my \$nl;
foreach \$nl (@noise_level)
{
    my \$cmd = "nice -3 ./stats_test.scm \$runs \$nl \$cleans";
    carp "\$ENV{'HOSTNAME'} \$cmd \n";
```

```

    my \$result = '\$cmd';
    print \$result;
}

```

`stats_test.scm` This file is called by “stats_run.pl” to run the mean high peak experiment.

```

#! /usr/local/bin/gsi -f

(load "load-all")

;; Set to false to turn off plotting of charts during a run
(define plotp #f)
;; plot a chart when the # of the run is divisible by this number
;;
(define plot_interval 3)

;; Don't change these
(set! amp 1)
(set! period 2)

;; *****
(define (psi t)
  (* amp (+ (* 1.00 (cos (+ 1.0 (* 2 period t))))
            (* 0.79 (cos (+ 0.5 (* 1.1131 period t))))
            (* 0.93 (cos (+ 1.5 (* 2.8131 period t))))
            (* 0.82 (cos (+ 2.5 (* 4 period t))))
            (* 0.41 (cos (+ 5.0 (* 5.512311 period t))))
            (* 0.32 (cos (+ 3.5 (* 7.213 period t)))))))

;; *****
(define logfile (open-output-file (string-append fileroot "_log")))

(define (stats_test runs ppb percent void_period noise-amp gain cln psi tl delta )
  ;; *****
  (define (no-noise)
    0)

  ;; *****

```

```

(define (dx old)
  (+ old delta))

;; *****
(define (find-highest-peak data)
  (define highest 0)
  (for-each (lambda (y)
    (if (> y highest)
        (set! highest y)))
    (map (lambda (x) (cadr x)) data))
  (display (string-append "(" (number->string highest) ")") logfile)
  (newline logfile)
  (flush-output logfile)
  highest)

;; *****
;; Noise function
(define (noise-func)
  (* noise-amp (grandom 1)))

;; *****
(define (stats-iter ref-data n)
  (if (<= n runs)
      (begin
        (display (string-append "#"
          (number->string n)
          " out of "
          (number->string runs)))
        (newline)
        (cons (let* ((nspectrum
(fourt_and_clean (generate-mean-wave (- t1) t1 dx
  (make-void void_period 0 percent)
  psi noise-func)
  4 gain c1n))
          (dsf (car nspectrum))

          (rsf (cadr nspectrum))

          (noisy-data

```

```

(caddr
  (cdr nspectrum)))

      (diff-data (map
        (lambda (ref noisy)
          (list (car ref)
                (abs (- (magnitude (cadr ref))
                        (magnitude (cadr noisy))))))
          ref-data
          noisy-data)))
      (plot ref-data noisy-data diff-data cln noise-amp n)
      (find-highest-peak diff-data))

(stats-iter ref-data (+ n 1)))
'())

;; *****
(let* ((cspectrum
(fourt_and_clean (generate-mean-wave (- t1) t1 dx
  (make-void void_period 0 percent)
  psi no-noise)
  4 gain 10000))
  (dsf (car cspectrum))
  (rsf (cadr cspectrum))

  (clean-data
(caddr
  (cdr cspectrum))))

  (stats-iter clean-data 1)))

;; *****
(define (find-mean data)
  (define (mean-iter d)
    (if (null? d)
0

```

```

(+ (car d)
  (mean-iter (cdr d))))
(/ (mean-iter data) (length data))

;; *****
(define (plot_data data)
  (let*((file (open-output-file "highpoint.dat")))
    (for-each (lambda (x)
      (display x file)
      (newline file))
      data)))
;; *****
(define (plot ref-data noisy-data diff-data cln noise-amp n)
  (if (and plotp
    (integer? (/ n plot_interval)))
    (begin
      (set! fileroot "noise-test")
      (print-amp-phase-data ref-data      (string-append fileroot ".pref"))
      (print-amp-phase-data noisy-data    (string-append fileroot ".pnoisy"))
      (print-amp-phase-data diff-data     (string-append fileroot ".pdif"))
      (gnuplot "stats_check.gp" (list (string-append "set title \"Num Cleans \"
        (number->string cln)
        " Noise Amp "
        (number->string noise-amp)
        " run "
        (number->string n) "\\;")
          (gnuplot_print_ps))))))
;; *****
(define (st runs noise-amp cleans)
  (stats_test runs 4 .5 6 noise-amp .1 cleans psi 24 .05))
;; *****
(define runs (string->number (cadr (argv))))
(define noise-amp (string->number (caddr (argv))))
(define cleans (string->number (cadddr (argv))))

(display "#Runs   Noise   Cleans   Mean Highest Peak   Sigma")
(newline)

```

```

;; *****
(define (tab) (display #\tab))

;; *****
(define data (st runs noise-amp cleans))

;(plot_data data)

(define dmean (find-mean data))
(define sigma (std_deviation data))

;(display "Runs           = ")
(display runs)
(tab)

;(display "Noise-amp      = ")
(display noise-amp)
(tab)

;(display "Cleans         = ")
(display cleans)
(tab)

;(display "Mean Highest Peak = ")
(display dmean)
(tab)

;(display "Std Deviation   = ")
(display sigma)
(newline)

```

B.2 Noise Level Test

noise_level_test.scm

```

#! /usr/local/bin/gsi -f
(load "load-all")

```

```

(define (no-wave t) 0)
(set! amp 1)
(set! period 2)
(set! delta 0.025)
(define (plot-spectrum s1 s2 lst)
  (print-amp-phase-data s1 "test1.pf")
  (print-amp-phase-data s2 "test2.pf")
  (gnuplot "spectrum.gp" lst))

(define (find-mean data)
  (define (mean-iter d)
    (if (null? d)
        0
        (+ (magnitude (cadr (car d)))
            (mean-iter (cdr d)))))
  (/ (mean-iter data) (length data)))
;; *****
(define (psi t)
  (* amp (+ (* 1.00 (cos (+ 1.0 (* 2 period t))))
            (* 0.79 (cos (+ 0.5 (* 1.1131 period t))))
            (* 0.93 (cos (+ 1.5 (* 2.8131 period t))))
            (* 0.82 (cos (+ 2.5 (* 4 period t))))
            (* 0.41 (cos (+ 5.0 (* 5.512311 period t))))
            (* 0.32 (cos (+ 3.5 (* 7.213 period t)))))))

(define (no-signal t)
  0)
(define (tab) (display #\tab))

(define (static_noise_test noise-amp psi t1 cln ppb tests)
  (define (dx old)
    (+ old delta))
  (define (no-void t) #t)
  (define (noise-func)
    (* noise-amp (grandom 1)))
  (let* ((signal (generate-mean-wave (- t1) t1 dx
                                     no-void psi noise-func))
         (cr (fourth_and_clean signal ppb 0.1 cln))
         (dsf (car cr)))

```

```

        (csf (caddr (cdr cr))))
;(plot-spectrum dsf csf (list (string-append "set title = "
;(number->string noise-amp))))

;; Run tests
(list->vector (map (lambda (funct)
    (funct noise-amp ppb cln dsf csf))
    tests))))

(define (show-noise-amp  noise-amp ppb cleans dsf csf)
  noise-amp)

(define (show-ppb      noise-amp ppb cleans dsf csf)
  ppb)

(define (show-cleans   noise-amp ppb cleans dsf csf)
  cleans)

(define (show-dirty-mean noise-amp ppb cleans dsf csf)
  (find-mean dsf))

(define (show-clean-mean noise-amp ppb cleans dsf csf)
  (find-mean csf))

(define (show-dirty-rms  noise-amp ppb cleans dsf csf)
  (rms dsf))

(define (show-clean-rms  noise-amp ppb cleans dsf csf)
  (rms csf))

(define (snt noise-amp ppb cleans)
  (static_noise_test noise-amp no-signal 12 cleans ppb
    (list show-dirty-mean
          show-clean-mean

```

```

                                show-dirty-rms
                                show-clean-rms)))

(define (many-snt count noise-amp ppb cleans)
  (define (iter j)
    (cons (snt noise-amp ppb cleans)
          (if (< j count)
              (iter (+ j 1))
              '()))))
  (define data (iter 0))
  (display count)
  (tab)
  (display noise-amp)
  (tab)
  (display ppb)
  (newline)
  (pp (map (lambda (lst)
            (cons (mean lst)
                  (cons (std_deviation lst)
                        '()))))
        (map (lambda (element)
              (map (lambda (vec)
                    (vector-ref vec element))
                  data))
            '(0 1 2 3)))))

(many-snt 25 .25 1 500)
;(for-each snt '(0.25 0.50 .75 1.00))
```

C Data Formats

The files all follow a basic tab delimited format. The input time series file should look like this:

Time	Value
------	-------

And the phase space files should all look like this:

Frequency	Real	Imaginary
-----------	------	-----------

These files can be read into and written out of scheme with the functions in the *print_data.scm* module. There are a number of functions to print out data and they all work the same way and vary only in how they format the data. The basic form is (`print_<format>_data data file`) where format can be one of tabular, amp-phase, scheme, real. The second parameter should be either the name of a file or a file handle.

Form	Use
tabular	Print out data in three columns, time/frequency, real, imaginary. For time data the imaginary column should normally be 0
scheme	Print out the data in a format that can be quickly used by scheme. This can be read back into scheme with <i>loadfile</i> which may be useful
amp-phase	print out the data as in tabular format, but in polar coordinates. Useful for plotting.
real	As Tabular but ignore any imaginary components.

To read data into scheme use the (`read-data-file <file>`) command. This takes the name of a file and returns a scheme data structure that contains the information.

Internal to scheme all time and spectral data is stored in a standard format. Scheme formats all data as a list of lists, each element in the list consisting of a pairs that consist of a time or frequency and value. The value can be a complex number in the frequency domain. In the case of the result of an inverse Fourier transform back into time space it is probable that some of the time values will have very small imaginary components due to numeric errors.

Example data set:

```
((.02 .028797241643053566+2.1133096081676417e-4i)
(.041 .020507782577238846+3.9219710253428515e-4i)
(.062 .011697462453592888+4.985291608917845e-4i)
(.083 .005333216900598863+4.3327667247881793e-4i)
(.104 .0018219287662072045-5.405790480047062e-4i)
(.125 2.0427465077917326e-4-.02215485280302419i))
```

D Scheme Code

D.0.1 Scheme Utility Code

gambc.scm Gambit will load this file, if present, automatically when it starts. It simply loads the next file.

```
(load "load-all")
```

load-all.scm

```
;; Load in all the functions in some kind of order.
;;
;; lists all the files that contain code that does computation
;;
(define files '(
                ;; Basic Definitions
                constants
math
                ;; Utility Functions
input-parse

                ;; Seed the random number generator and set it up
                ;;
                random

                ;; Direct port of the random number code in
                ;; Numerical recipes
gaussian-random

                ;; A wrapper for gnuplot
                gnuplot

makewave
drop_points
util
print_data
void_function
stats
                filter
clean_wrapper

mean_time_generator))
```

```
;; This function loads all the files listed above
(define (load-all)
  (for-each (lambda (file)
             (load (symbol->string file)))
           files))
;; This function compiles everything, it is not used.
(define (compile-all)
  (for-each (lambda (file)
             (compile-file (symbol->string file))) files ))

;; Load everything
(load-all)
```

constants.scm Constants and other such useful things This file just defines a bunch of mathematical constants And such. no actual computation in this file.

```
(define e (exp 1))
(define i (sqrt -1))
(define pi (* 4 (atan 1)))
(define twopi (* 2 pi))
(define minusTwoPi (* -2 pi))
;; If this is set to true "#t" gnuplot will be caused to
;; output to a postscript file in some places.
;;
(define printp #f)
```

math.scm Functions that take care of common math stuff

```
;; Take the complex conjugate of a number
(define (cong z)
  (+ (real-part z) (* -1 i (imag-part z))))

;; X squared
(define (sqr z)
  (* z z))
```

random.scm This calls linux's /dev/urandom device to produce a solid random seed.

```
;;
(define (make-rand)
  (let ((randfile (open-input-file "/dev/urandom")))
    (define (int-rand) (+ (char->integer (read-char randfile))
                        (* 256 (char->integer (read-char randfile)))
                        (* 65536 (char->integer (read-char randfile)))
                        (* 16777216 (char->integer (read-char randfile)))))
      ;; return a float between 0 and 1
      (define (float-rand)
        (/ (int-rand) 4294967296.0))
      (list int-rand float-rand)))
```

gnuplot.scm A wrapper for *gnuplot* this function takes two parameters, a file to plot, and a list containing any additional parameters to pass to that file. Those parameters are placed in the file “param.gp”, so the plot file should include that file.

If a postscript version of the plot is desired add the output of the (`gnuplot_print_ps`) function. It will tell *gnuplot* to output the image to a file called “dfile nnn .ps”.

```
;; A wrapper for gnuplot
;; Params takes list and puts each element of that list into
;; a file that can be loaded in gnuplot.
;;

(define (gnuplot plotfile params)
  (define file (open-output-file "params.gp"))
  (for-each (lambda (param)
              (display param file)
              (newline file))
            params)
  (flush-output file)
  ; (display (string-append "gnuplot -persist " plotfile))
  ; (newline)
  (##shell-command (string-append "gnuplot -persist " plotfile)))

;; Call this function to have gnuplot output a postscript file
;;
(define (gnuplot_print_ps)
  (string-append "set terminal postscript portrait color;"
                "set output \"\" (create-file-name) ".ps\";"))
```

stats.scm This file has basic code for statistical applications such as finding the mean, variance and RMS of a data set.

```
(define (mean data)
  (define (mean-iter data)
    (if (null? data)
        0
        (+ (car data)
            (mean-iter (cdr data)))))
  (if (null? data)
      0
      (/ (mean-iter data)
          (length data))))

(define (var data)
  (define datamean
    (mean data))
  (define (var-iter data)
    (if (null? data)
        0
        (+ (sqr (- (car data) datamean))
            (var-iter (cdr data)))))
  (if (null? data)
      0
      (/ (var-iter data) (- (length data) 1))))

(define (std_deviation data)
  (sqrt (var data)))

(define (rms data)
  (define (rms-iter data)
    (if (null? data)
        0
        (+ (sqr (magnitude (cadr (car data))))
            (rms-iter (cdr data)))))
  (if (null? data)
      0
      (sqrt (/ (rms-iter data) (length data)))))
```

print_{data} This file handles input and output of data, See appendix D for full details.

```
(load "input-parse.scm")
(load "util.scm")
(load "constants.scm")

;; All the print functions in this file take a "file" as a paramater.
;; You can pass in a string which will be taken as a file name, or a
;; file handle which will be used to outpu the data. If you pass in
;; anything else (Say an empty list) the data will be sent to the
;; current output port.

;; This prints out a data set in the form
;; time    real    imaginary
(define (print-tabular-data data file)
  (cond
    ;;If it is a string do open the file
    ((string? file)
     (set! file (open-output-file file)))
    ;;If it is already a port do nothing
    ((output-port? file)
     '())
    ;; If its anything else use the current output port
    (else (set! file (current-output-port))))

  (for-each (lambda (x)
              (if x
                  (begin
                    (display (/ (floor (* (car x) 1000)) 1000) file)
                    (display #\tab file)
                    (display (real-part (cadr x)) file)
                    (display #\tab file)
                    (display (imag-part (cadr x)) file)
                    (display #\newline file))))
              (if (vector? data)
                  (vector->list data)
                  data)))
```

```

(flush-output file)
data)

;; Print out the data in the form of a scheme vector
;;
;; This function outputs the data in the form of a scheme s-expression
;; So if it is known that you will be re-reading the data into scheme
;; this is a good way to output it as the data will be in a format
;; scheme already knows what to do with.

(define (print-scheme-data data file)
  (cond
    ;;If it is a string do open the file
    ((string? file)
     (set! file (open-output-file file)))
    ;;If it is already a port do nothing
    ((output-port? file)
     '())
    ;; If its anything else use the current output port
    (else (set! file (current-output-port))))
  (display "'#(" file)
  (display #\newline file)
  ;; Display each record of the dataset
  (for-each (lambda (x)
    (begin
      (display "'(" file)
      (display (/ (floor (* (car x) 1000)) 1000) file)
      (display #\tab file)
      (display (real-part (cadr x)) file)
      (display #\tab file)
      (display (imag-part (cadr x)) file)
      (display ")" file)
      (display #\newline file))))
    (if (vector? data)
      (vector->list data)
      data))

  (display ")" file)
  (flush-output file)

```

```
data)

;; This just takes a complex number Z and returns
;; a list of the magnitude and angle of that number.

(define (amp-phase z)
  (list (magnitude z)
        (angle z)))

;; This is similar to the first function in that it takes The data and
;; prints it out in a tab delimited form for GNUplot or the like but
;; It prints it out in polar coordinates.

(define (print-amp-phase-data data file)
  (cond
    ;;If it is a string do open the file
    ((string? file)
     (set! file (open-output-file file)))
    ;;If it is already a port do nothing
    ((output-port? file)
     '())
    ;; If its anything else use the current output port
    (else (set! file (current-output-port))))

  (for-each (lambda (x)
              (if x
                  (begin
                     (display (/ (floor (* (car x) 1000)) 1000) file)
                     (display #\tab file)
                     (display (magnitude (cadr x)) file)
                     (display #\tab file)
                     (display (angle (cadr x)) file)
                     (display #\newline file))))
              (if (vector? data)
                  (vector->list data)
                  data))
            (flush-output file)
            data)
```

```

;; This prints out a data set using only the
;; Real half of a complex number.

(define (print-real-data data file)
  (cond
    ;;If it is a string do open the file
    ((string? file)
     (set! file (open-output-file file)))
    ;;If it is already a port do nothing
    ((output-port? file)
     '())
    ;; If its anything else use the current output port
    (else (set! file (current-output-port))))

  (for-each (lambda (x)
              (if x
                  (begin
                     (display (/ (floor (* (car x) 1000)) 1000) file)
                     (display #\tab file)
                     (display (real-part (cadr x)) file)
                     (display #\newline file))))
              (if (vector? data)
                  (vector->list data)
                  data))
            (flush-output file)
            data)

;; This reads in a data file and returns a list of lists. Each sub
;; list represents one line of the file, and the total thing is all
;; the data in the file. It can be turned into a vector with
;; list->vector

(define (read-data-file file)

  ;; Check for EOF
  (define (split string)
    (if (eof-object? string)
        '()
        (string-split string)))

  (string-split file))

```

```
(define (rf-iter)
  (set! line (map string->number (split (read-line file))))
  (if (not (null? line))
      (if (number? (car line))
          (cons (list (car line)
                      ;; the last 2 items make a complex number a+bi
                      (+ (cadr line)
                        (* (caddr line) i)))
                (rf-iter))
          (rf-iter)))
      '()))

(cond
  ;;If it is a string do open the file
  ((string? file)
   (set! file (open-input-file file)))
  ;;If it is already a port do nothing
  ((input-port? file)
   '())
  ;; If its anything else use the current output port
  (else (set! file (current-input-port))))
(read-line file)
(rf-iter))
```

D.0.2 Scheme works code

This is the code that forms the works of the scheme code to work with clean;

mean_time_generator.scm This function generates a data set such that the mean time in it is at 0. This preserves the phase in the fourier transform.

It takes 6 paramaters, the start and end time, a function to get the next time, a function to apply voiding, the function to generate the wave and noise function.

The data returned will be set to have a mean time value of 0, so that the Fourier code will preserve the phase on it. The Fourier code defines phase as the cos of the mean time of the sample.

To call this function, you need six parameters, the first two are the start and end time, generally set to $\pm T$, the next four parameters are scheme functions.

The dx function takes a given time t and returns the next time that should be used. Normally this is given as $t + \delta t$ but could be setup to return any time $t' > t$.

The next function is a voiding function which is any function that takes a time and returns $\#t$ (true) or $\#f$ (false). In general to apply a regular duty cycle use the *(make-void...)* function which will return a function that does the right thing.

To use *make-void* call it with 3 parameters, the length of the duty cycle, the phase which can often be left at 0 and the percent of points to drop. It will return a scheme λ function that can be directly passed to *generate-mean-wave*.

The next parameter is the wave function itself. This should be a function that takes a time and returns a real value.

The last function is the noise function. This will be called for every point in the spectrum and added to the value of the base spectrum.

```
(generate-mean-wave (- t1) t1 dx
  (make-void void_period 0 percent)
  psi noise-func)
```

It is called from *gain_test.scm* and other places, so an example can be found there.

```
(define (generate-mean-wave tmin tmax dx void psi noise-func)
```

```

(define (make-time)
  (define (make-time-iter ctime)
    (if (<= ctime tmax)
        (if (void ctime)
            (cons ctime
                  (make-time-iter (dx ctime)))
            (make-time-iter (dx ctime)))
        '()))
  (make-time-iter tmin))

(define (FindTMean time-series)
  (define (TMean-Iter mean data)
    (if (not (null? data))
        (TMean-Iter (+ (car data) mean)
                    (cdr data))
        mean))
  (/ (TMean-Iter 0 time-series ) (length time-series)))
; (display "generate-mean-wave")
; (newline)
(let* ((times (make-time))
      (mean-time (FindTMean times)))

  (map (lambda (time)
; (display ".")
      (let* ((nt (- time mean-time))
            (list nt
                  (+ (psi nt)
                    (noise-func))))
        times)))

```

drop_points.scm This function takes a data set (list or vector) and drops numpoints points at random.

```

; ;#! /usr/local/bin/gsi -f
; ;
; ;

```

```
(define (drop-points data-set numpoints)
  (if (list? data-set)
      (vector->list (drop-points (list->vector data-set) numpoints))
      (if (eq? numpoints 0)
          data-set
          (begin
             (vector-set! data-set
                           (modulo (generator) (vector-length data-set)))
             #f)
          (drop-points data-set (- numpoints 1))))))
```

void_function.scm This is a factory function, which is to say that it returns another function that will void out the wave generator. The void function returned will return true where the function is defined and false where it is voided out.

The phase is defined such that if it is 0 then it will return true from $t=0$ to $T = \text{percent}/\text{period}$

```
(define (make-void period phase percent)
  (set! period (/ twopi period))
  ; (set! phase (+ phase pi))
  (lambda (t)
    (< (* percent twopi)
        (+ pi (angle (make-polar 1 (+ phase (* period t))))))))
```

clean_wrapper.scm This file contains the wrappers that interface the scheme code to the fortran code that actually does the fourier transform and CLEAN.

The function that should be used is `four_t_and_clean` which takes 4 parameters, the time series data, the number of points per beam (default 4) to run the fourier transform at, the gain for clean, and the number of cleans to run.

It then returns a list of lists of the output of the fourier transform and clean processes. The list is in this order,

1. Dirty Spectrum File

2. Spectral Window File (Beam)
3. Residual Spectrum file
4. Clean Component File
5. Clean Spectrum File

While this function is running it will create a bunch of files of the form "dfile nnn .???" these will be deleted when it finishes. There will also be a lockfile called "dfile nnn _lock" which has to be deleted manually.

```
(define (create-file-name)
  (let ((basename (string-append "" (symbol->string (gensym 'dfile))))
        (if (file-exists? (string-append basename "_lck"))
            (create-file-name)
            (begin
              (display (string-append "touch " basename "_lck"))
              (newline)
              (##shell-command (string-append "touch " basename "_lck")
                                basename))))))

(define fileroot (create-file-name))
(define (fourt_and_clean time-series ppb gain cleans)
  (let* ((fourt_cmd (string-append "fourt "
                                   fileroot ".psi "
                                   "MEAN MEAN "
                                   (number->string ppb) " "
                                   (number->string 0) " "
                                   fileroot ".dsf "
                                   fileroot ".swf "))
        (clean_cmd (string-append "clean "
                                   fileroot ".dsf "
                                   fileroot ".swf "
                                   (number->string gain) " "
                                   (number->string cleans) " "
                                   fileroot ".rsf "
                                   fileroot ".ccf "
                                   fileroot ".csf" ))
        (rm_cmd (string-append "rm " fileroot ".*")))
    (fourt_cmd)
    (clean_cmd)
    (rm_cmd)))
```

```
(print-tabular-data time-series (string-append fileroot ".psi"))

(##shell-command fourt_cmd)

(##shell-command clean_cmd)
(set! return_vals (list (read-data-file (string-append fileroot ".dsf"))
  (read-data-file (string-append fileroot ".swf"))
  (read-data-file (string-append fileroot ".rsf"))
  (read-data-file (string-append fileroot ".ccf"))
  (read-data-file (string-append fileroot ".csf"))))

(##shell-command rm_cmd)

return_vals
))
```

D.0.3 Scheme Test Code

Files in this section exist to test other scheme code.

void_test.scm

```
(load "load-all.scm")

(define (make-void period phase percent)
  (set! period (/ twopi period))
  ; (set! phase (+ phase pi))
  (lambda (t)
    (< (* percent twopi)
      (+ pi (angle (make-polar 1 (+ phase (* period t))))))))

(define (psi t) (cos t))

(define delta 0.05)

(define (dx old)
  (+ old delta ))

(define (nonoise) 0)

(define test-void (make-void 8 2 .25))
(trace make-void)
(set! voiddata
  (generate-wave -20 20 dx test-void psi nonoise ) )

(begin (print-tabular-data voiddata "voidwave.dat") '())
(untrace make-void)
```

clean_wrap_test.scm This tests the “clean_wrapper.scm” functions which are the code that actually calls the fortran code.

```
(define (test-clean ppb void_percent void_period gain ncleans psi)
```

```

(define (nonoise) 0)
(load "clean_wrapper")

(let* ((timedata (generate-wave -12 12 dx
  (make-void void_period 0 void_percent)
  psi nonoise))
  (spectrum (fourier timedata "MEAN" "MEAN" ppb 0))
  (fileroot "test")
  (dsf (car spectrum))
  (swf (cadr spectrum))
  (clean_result (clean-wrap dsf swf gain ncleans '()))
  (rsf (car clean_result))
  (ccf (cadr clean_result))
  (csf (caddr clean_result)))
  ;; plot stuff here but for now just see if it runs...
  (step)
  (print-tabular-data timedata "wave.psi")
  (print-amp-phase-data dsf (string-append fileroot ".pdsf"))
  (print-amp-phase-data csf (string-append fileroot ".pcsf"))
  ;   (##shell-command "gnuplot -persist wave.gp")
  (##shell-command "gnuplot -persist angleplot.gp")
  'done))

(define (tc gain cleans)
  (test-clean 2 .25
    6 gain cleans
    (lambda (t) (* 2 (cos (* 6.1 twopi t))))))
(tc 0.2 50)

```

E Presentation code

These files contain the code that made the images for the presentations and this thesis.

make_pres.scm

```
;; Make this images for the PHYS301b presentation
;;
(define printp #f)

(load "gain_test")

(define period 1)
(define (psi t) (* amp (cos (* 2.1 period t))))

(tr 0 20 .75 '(500) psi)
(set! period 2)
(define amp 1)

(define (psi t)
  (* amp (+ (* 1.00 (cos (+ 1.0 (* 2 period t))))
            (* 0.79 (cos (+ 0.5 (* 1.1131 period t))))
            (* 0.93 (cos (+ 1.5 (* 2.8131 period t))))
            (* 0.82 (cos (+ 2.5 (* 4 period t))))
            (* 0.41 (cos (+ 5.0 (* 5.512311 period t))))
            (* 0.32 (cos (+ 3.5 (* 7.213 period t))))))))

(define period 2)
(define printp #f)

(define (run-many n)
  (define (run-iter j)
    (if (< j n)
        (begin
          (tr .5 24 .5 '(1000) psi)
          (run-iter (+ j 1))))
        (run-iter 0)))
```

```
(ts .5 24 .5 '(100))
```

```
(define (make_pres) (load "make_pres"))
```

make_thesis_plots.scm This script makes most of the plots in this thesis.

```
(load "gain_test")
(define printp #t)
(define period 3.214)
(define amp 2)
(define num_cleans 500)
(define (psi_simple t)
  (* amp (cos (* 2.1 period t))))

(tr 0 12 0.0 '(num_cleans) psi_simple)
(tr 0 12 .75 '(num_cleans) psi_simple)

(set! period 1.831)
(set! amp 1)

(define (psi t)
  (* amp (+ (* 1.00 (cos (+ 1.0 (* 2 period t))))
            (* 0.79 (cos (+ 0.5 (* 1.1131 period t))))
            (* 0.93 (cos (+ 1.5 (* 2.8131 period t))))
            (* 0.82 (cos (+ 2.5 (* 4 period t))))
            (* 0.41 (cos (+ 5.0 (* 5.512311 period t))))
            (* 0.32 (cos (+ 3.5 (* 7.213 period t)))))))

;; No void, no noise
(tr 0.0 24 0 '(num_cleans) psi)
;; Void no noise
(tr 0.0 24 .5 '(num_cleans) psi)
;; No void, but with noise
(tr 0.5 24 0 '(num_cleans) psi)
```

```
;;; Void with noise
(tr 0.5 24 .5 '(num_cleans) psi)
;; Different clean values
(tr 0.5 24 .5 '(50
500
5000
50000
5000000) psi)
(set! amp 0)
;; Noise only
(tr 1 24 0 '(num_cleans) psi_simple)
```

F Fortran Code

Makefile This Makefile builds the fortran code and installs it.

```
FFLAGS = -ff77 -fvxt
FC      = g77
FILES = clean_s fourt recon
INSTDIR = /usr/local/bin
all: $(FILES)

clean_s : clean_s.f inout
$(FC) -c $(FFLAGS) clean_s.f
$(FC) $(FFLAGS) clean_s.o inout.o -o clean_s

fourt : fourt.f inout
$(FC) -c $(FFLAGS) fourt.f
$(FC) $(FFLAGS) fourt.o inout.o -o fourt

inout : inout.f
$(FC) -c $(FFLAGS) inout.f

#plots : plots.f inout
# $(FC) -c $(FFLAGS) plots.f
# $(FC) $(FFLAGS) plots.o inout.o -o plots

recon : recon.f inout
$(FC) -c $(FFLAGS) recon.f
$(FC) $(FFLAGS) recon.o inout.o -o recon

clean:
rm -f *.o
rm *~
rm clean_s fourt plots recon
rm core

install: all
mv recon $(INSTDIR)
mv clean_s $(INSTDIR)/clean
mv fourt $(INSTDIR)
```

four.f This code has been modified from Joe Lehar's original code. It has been ported to run on Linux and compile with the gnu g77 Fortran compiler. This should also make it easier to port to most other modern unix type operating systems.

In addition it has been changed to allow for larger data sets than the original code allowed, and longer file names.

Finally it has been modified to take its parameters on the unix command line, it should be called like this

```
clean <input.dat> TSUB XSUB ppb max_freq <output.dsf> <output.swf>
```

The first parameter is the name of the input data file. The next four parameters control how the Fourier transform works. *TSUB* is the mean time and *XSUB* is the mean amplitude, in normal cases these should simply be set to "MEAN". *ppb* is the number of points per beam to compute, if set to 0 it will compute 4 ppb, which is a good default. The last two parameters are the dirty spectrum and spectral window function respectively. These files should not exist when the function is called.

To build all the Fortran programs use the included *Makefile*.

The *four* program defines the phase as the cos at the mean time of the sampled data. So a cos sampled from $-x : +x$ should have phase 0.

```
C*--- FOURS.f
```

```
PROGRAM FOURS
```

```
C-----
C Author:      J. Lehar                               Date: 13-JUL-87
C Written for VAX FORTRAN 77   (May require some changes for other machines)
C NOTE: This code was written for clarity, not for efficiency.
C       Some optimization may be desired for certain applications.
C-----
C Produces the dirty spectrum and the spectral window for a time series
C contained in XFILE. The time average and the data mean are automatically
C removed (see paper). The frequency sampling is determined by the user.
C       / freq. resolution = 1/T   ; T is the overall time interval
C defaults: < Max. frequency   = 1/2dt ; dt is the smallest time spacing
C           \ Points-per-beam = 4   ; Freq. spacing, dF = 0.25*(1/T)
C The dirty spectrum is written to DFILE, and the spectral window to WFILE.
C NOTE: the filenames are limited to 14 characters. This is so they can
C       be propagated to subsequent programs through the spec. file headers
C-----
```

```

C CALLS:      RDDATA      to read the time series data \_ in INOUT.FOR
C             WRSPEC      to write spectra /
C             FOUR1       performs the discrete FT, time-->freq
C-----
      PARAMETER (MAXN= 5000)          ! maximum # data samples
      PARAMETER (MAXM= 10000)        ! maximum index for FT array
C Declare arrays and some variables
      REAL      XSUB,ppb,df,FMAX
      COMPLEX   D(0:MAXM),W(0:2*MAXM),FOUR1
      REAL      T(MAXN),X(MAXN),ONES(MAXN),F(0:2*MAXM)

C Changed file names to be 64 charecters to deal with modern OS's (IE Unix)
C 20 Oct 2002 --ZDK
      CHARACTER XFILE*64,DFILE*64,WFILE*64,HEADER*80
      CHARACTER ARG*20
      DATA ONES/MAXN*1.0/          ! fill the ONES array with ones
C Changed code to take parameters at the command line
C --ZDK

C Get execution parameters and filenames
      CALL GETARG(1,XFILE)

C Time subtraction (phase centre):
      CALL GETARG(2,ARG)
      READ (ARG,*, ERR=10) TSUB
10      TSUB= -1E20                  ! flag: use Tmean

C Data subtraction (DC level):
      CALL GETARG(3,ARG)
      READ (ARG,*,ERR=12) XSUB
12      XSUB= -1E20                  ! flag: use Tmean

C Frequency increment:
      CALL GETARG(4,ARG)
      READ (ARG,*) dF

C Max. freq. (Dirty spec.):
      CALL GETARG(5,ARG)
      READ (ARG,*) FMAX
C OUTPUT FILES

```

```

C Dirty spectrum file
  CALL GETARG(6,DFILE)
C Spectral window file
  CALL GETARG(7,WFILE)

C      WRITE (*,1000) '** Source File=',XFILE,' Tsub=',TSUB,' Xsub=',XSUB
C      WRITE (*,1000) '** df= ',dF, ' FMax=',FMAX
C      WRITE (*,1000) '** DFILE= ',DFILE, ' WFILE=',WFILE

C Read the times T(1:N) and the data X(1:N) from XFILE
  N= MAXN                ! defines max. N for RDDATA
  ERR= RDDATA(N,T,X,XFILE,HEADER)
C      PRINT *,HEADER
  IF(ERR.NE.0.) PRINT *,'Array truncated at N=',N
C Find the mean T,X ; min & max time separation
C ---mean T,X
  TMEAN= 0.
  XMEAN= 0.
  DO I=1,N
    TMEAN= TMEAN+ T(I)
    XMEAN= XMEAN+ X(I)
  ENDDO
  TMEAN= TMEAN/N
  XMEAN= XMEAN/N
C      WRITE (*,1000) '*** TMEAN = ',TMEAN
C ---min & max time separation
  SMIN= 1.E20
  DO I=2,N
    SEP= T(I) - T(I-1)
    IF(SEP.LT.SMIN) SMIN= SEP
  ENDDO
  SMAX= T(N)-T(1)
C Set default parameter values & confirm
  IF(TSUB.EQ.-1E20) TSUB = TMEAN          ! time sub (phase centre)
  IF(XSUB.EQ.-1E20) XSUB = XMEAN          ! data subtraction (DC)

  IF(PPB.EQ.0.) PPB = 4                    ! frequency increment
  df = 1.0 / ( PPB * SMAX)
  IF(FMAX.EQ.0.) FMAX= 1./(2.*SMIN) ! max. frequency
C      PRINT *,'Tsub,Xsub = ',TSUB,XSUB

```

```

C      PRINT *, 'df, Fmax = ', df, FMAX
C subtract the time, data means from T, X
      DO I=1, N
          T(I) = T(I) - TSUB
          X(I) = X(I) - XSUB
      ENDDO
C set up the frequency array F(0:M)
      M = INT(FMAX/df)           ! maximum freq. element (for D)
      DO J=0, 2*M
          F(J) = df * J
      ENDDO
C generate the dirty spectrum D(0:M)
C ---calculate the dirty spectrum
C      PRINT *, 'Computing the dirty spectrum...'
      DO J=0, M
          D(J) = FOUR1(F(J), N, T, X)
      ENDDO
C ---create the header & write to file
C      WRITE(HEADER, 1000) DFILE, XFILE, N, TSUB, ', Xsub=', XSUB
      CALL WRSPEC(M, F, D, DFILE, HEADER)
C      PRINT *, HEADER
C generate the spectral window W(0:2M)
C ---calculate the spec. window
C      PRINT *, 'Computing the spectral window...'
      DO J=0, 2*M
          W(J) = FOUR1(F(J), N, T, ONES)
      ENDDO
C ---create header & write to file
      WRITE(HEADER, 1000) WFILE, XFILE, N, TSUB, ', Tavg=', TMEAN
      CALL WRSPEC(2*M, F, W, WFILE, HEADER)
C      PRINT *, HEADER
C exit the program
      CALL EXIT
1000  FORMAT(A14, ', df=', A14, ', N=', I5, ', Tsub=', 1P, E14.7, A6, 1P, E14.7)
      END

```

COMPLEX FUNCTION FOUR1(F, N, T, X)

C Author: J. Lehar Date: 13-JUL-87

```

C-----
C Returns the Fourier transform of the time series specified by T(1:N)
C and X(1:N), evaluated at the frequency F.
C The form of the Fourier transform is taken from Bracewell (1965)
C
C
C          1 .--. N          -i*2pi*F*T(j)
C   FOUR1(F) = --- >      X(j) e
C              N  '--'j=1
C
C The DFT is normalized to have the data mean at FREQ=0.
C-----
      COMPLEX    SUM
      REAL      X(N),T(N)
C initialize some variables
      TWOPI= 8 * atan(1.0)
      SUM= (0.,0.)          ! accumulation variable
C Evaluate FT at F...
      DO J=1,N
         PHASE= -TWOPI*F*T(J)
         SUM= SUM + X(J)*CMLPX(COS(PHASE),SIN(PHASE))
      ENDDO
C return DFT normalized to FOUR1(0)=Xmean
      FOUR1= SUM/N
      RETURN
      END

```

clean_s.f This code has been modified from Joe Lehar's original code. It has been ported to run on Linux and compile with the gnu g77 Fortran compiler. This should also make it easier to port to most other modern unix type operating systems.

In addition it has been changed to allow for larger data sets than the original code allowed. Filename lengths have been expanded to 64 characters.

Finally it has been modified to take its parameters on the unix command line, it should be called like this

```
clean <file.dsf> <file.swf> gain num_cleans <file.rsf> <file.ccf> <file.csf>
```

The first two parameters are the dirty spectrum file, and the spectral window file, which are produced by the *fourt* command. The next two are the gain and number of iterations of CLEAN to run. The last 3 are the output files. These files should not exist when the program is called or it will not operate properly.

To build all the Fortran programs use the included *Makefile*.

```
C*--- CLEAN.FOR
```

```
PROGRAM CLEAN
```

```
C-----
C Author: J. Lehar Date: 13-JUL-87
C Written for VAX FORTRAN 77 (May require some changes for other machines)
C NOTE: This code was written for clarity, not for efficiency.
C Some optimization may be desired for certain applications.
C-----
C Deconvolves the spectral window in WFILE from the dirty spectrum in DFILE,
C by using an iterative, one-dimensional, complex, Hoegbom CLEAN algorithm.
C The resulting clean spectrum is in SFILE, the residual and CLEAN component
C spectra are found in RFILE & CFILE, respectively.
C The user determines the gain and the number of cleans, the latter
C either explicitly or by specifying a CLEAN level. (when the maximum in
C the residual spectrum is less than the CLEAN level, cleaning stops.)
C Since all spectra from real data are Hermitian, we define only the non-
C negative frequencies. The negative frequency elements are recovered by
C the use of the function CVAL, which returns the complex conjugate for
C negative frequencies, and zero for frequencies outside the defined range.
C NOTE: the filenames are limited to 14 characters. This is so they can
C be propagated to subsequent programs through the spec. file headers
C-----
```

```

C CALLS:      RDSPEC          to read spectra  \_ in INOUT.FOR
C             WRSPEC          to write spectra /
C             CLEAN1         performs 1 iteration of CLEAN
C             FITBEAM        to fit a beam to the spectral window
C             RESTORE        to create the clean spectrum

```

```

C-----
      PARAMETER (MAXM = 10000)
C      MAXM = 10000          ! maximum index for spectral arrays
C  Declare arrays and some variables
      COMPLEX  W(2 * MAXM),B(MAXM),R(MAXM),C(MAXM),S(MAXM)
      REAL     F(2*MAXM)
      CHARACTER WFILE*64,DFILE*64,RFILE*64,CFILE*64,SFILE*64,HEADER*80
      CHARACTER ARG*20
C  get execution parameters & filenames

C Dirty Spectrum File
      CALL GETARG (1,DFILE)

C Spectral Window File
      CALL GETARG (2, WFILE)

C GAIN
      CALL GETARG(3,ARG)
      READ (ARG,*) GAIN

C      PRINT '( '$[ >0 = #CLEANS, <0 = CLEAN level ] Cleans: '$)'
      CALL GETARG(4,ARG)
      READ (ARG,*) CLEANS

C Residual spectrum file
      CALL GETARG (5, RFILE)

C CLEAN components file:
      CALL GETARG (6, CFILE)
C Clean spectrum file:
      CALL GETARG (7, SFILE)

C  ---determine depth of CLEAN
      IF(CLEANS.GE.0.)THEN
          NCLEAN= INT(CLEANS)  ! clean to the specified # of cleans

```

```

        RCLEAN= 0.          ! no level specified.
    ELSE
        NCLEAN= 32000.     ! more than anyone is likely to want
        RCLEAN= ABS(CLEANS) ! specifies the desired level
    ENDIF
C read the input spectra, get time info for beam, determine freq. range.
C ---dirty spectrum, read into the residual R(0:MS)
    MS= MAXM                ! max. allowed MS, for RDSPEC
    ERR= RDSPEC(MS,F,R,DFILE,HEADER)
C PRINT *,HEADER
C IF(ERR.NE.0.) PRINT *,'Truncated dirty spectrum at M=',M
C ---spectral window W(0:MW)
    MW= 2*MAXM              ! max. allowed MW, for RDSPEC
    ERR= RDSPEC(MW,F,W,WFILE,HEADER)
C PRINT *,HEADER
    READ(HEADER,'(46X,E14.7,6X,E14.7)') TSUB,TMEAN ! from WFILE
C IF(ERR.NE.0.) PRINT *,'Truncated spec.window at M=',M
C ---determine frequency range
    dF= F(2)-F(1)           ! frequency increment
    IF(2*MS.GT.MW) MS= MW/2 ! ensure that MS < MW/2
C CLEAN the residual spectrum, storing the components in C(0:MS)
C PRINT *,'CLEANing up to ',NCLEAN,' iterations, or down to ',RCLEAN
    DO K=1,NCLEAN           ! on until NCLEAN
        CALL CLEAN1(MW,W,MS,R,C,GAIN,RMAX) ! execute a single CLEAN
        IF(RMAX.LT.RCLEAN) GOTO 10        ! skip out if down to RCLEAN
    ENDDO
10 NCLEAN= K-1             ! actual number of CLEANs
C PRINT *,'NCLEAN=',NCLEAN,' RMAX=',RMAX
C Generate the beam B(0:MB) and the clean spectrum S(0:MS)
    MB= MS                 ! max. possible MB, for FITBEAM
    CALL FITBEAM(MB,B,MW,W,dF,TSUB,TMEAN) ! create the beam
    CALL RESTORE(MS,S,C,R,MB,B) ! restore S (convolve CxB, add R)
C Write the output spectra
C ---residual spectrum
C WRITE(HEADER,1000) RFILE,DFILE,WFILE,GAIN,CLEANS
    CALL WRSPEC(MS,F,R,RFILE,HEADER)
C PRINT *,HEADER
C ---CLEAN components
C WRITE(HEADER,1000) CFILE,DFILE,WFILE,GAIN,CLEANS
    CALL WRSPEC(MS,F,C,CFILE,HEADER)

```

```

C      PRINT *,HEADER
C      ---clean spectrum
C      WRITE(HEADER,1000) SFILE,DFILE,WFILE,GAIN,CLEANS
C      CALL WRSPEC(MS,F,S,SFILE,HEADER)
C      PRINT *,HEADER
C      - format statement for spectrum file headers...
1000  FORMAT(A14,',From:',A14,',and:',A14,',';G=', F7.4 , 'C=',1P,E11.4)
C exit the program
      CALL EXIT
      END

```

SUBROUTINE CLEAN1(MW,WIN,MS,RES,CMP,GAIN,RMAX)

```

C-----
C Author:   J. Lehar                               Date: 13-JUL-87
C-----
C Performs one iteration of the RLD complex, 1-D, CLEAN algorithm.
C o The component CCPOS responsible for the max. in the residual spectrum
C   RES at L, is estimated; taking into account the interference with
C   its conjugate CCNEG at -L.  ( through RES(L)= CCPOS +CCNEG*WIN(2L) )
C o The spectral window WIN is matched to these components, multiplied by
C   a GAIN factor and subtracted from the residual spectrum.
C o GAIN*CCPOS is added to the CMP spectrum.
C o returns RMAX, the maximum in the residual (modulus)
C-----
C CALLS:    CVAL                determines spectral values beyond (0:M)
C-----
      COMPLEX WIN(0:MW),RES(0:MS),CMP(0:MS),WIN2L,CCPOS,CCNEG,CVAL
      DATA ERROR/0.0001/                ! allowed error in WNORM
C find the location L of max. in the residual spectrum RES(0:M)
      RMAX= CABS(RES(0))
      L= 0
      DO J=1,MS
        IF(CABS(RES(J)).GT.RMAX)THEN
          RMAX= CABS(RES(J))
          L= J
        ENDIF
      ENDDO
C estimate the component at L which yields RES(L) through WIN(L)
      WIN2L= WIN(2*L)                ! (L,-L) interference

```

```

WNORM= 1.0-CABS(WIN2L)**2
IF(WNORM.LT.ERROR) CCPOS= 0.5*RES(L) ! prevent singularities
IF(WNORM.GE.ERROR) CCPOS= (RES(L)-WIN2L*CVAL(RES,-L,MS))/WNORM
C Remove effects of both +L and -L components from RES, add CCPOS to CMP
CCPOS= GAIN*CCPOS ! remove GAIN*CCPOS
CCNEG= CONJG(CCPOS) ! conjugate at -L
DO J=0,MS
RES(J)= RES(J) -CCPOS*CVAL(WIN,J-L,MW) -CCNEG*WIN(J+L)
ENDDO ! (from L ) (from -L )
CMP(L)= CMP(L) +CCPOS ! add it to CMP
IF(L.EQ.0) CMP(L)= CMP(L) +CCNEG ! also NEG if L=0
C return to the calling program
RETURN
END

```

SUBROUTINE FITBEAM(MB,BEAM,MW,WIND,dF,TSUB,TMEAN)

```

C-----
C Author: J. Lehar Date: 13-JUL-87
C-----
C Fits a clean beam, BEAM, to the primary peak of the spectral window, WIND.
C The modulus is a Gaussian, matched to have the same half-width,half-max.
C through linear interpolation between frequency samples.
C The phase is a linear function of frequency, determined by the frequency
C increment dF and the time offset from the mean time (TSUB-TMEAN).
C-----

COMPLEX BEAM(0:MB),WIND(0:MW)
C Find the half-width,half-max of CABS(WIND)
HALFMAX= 0.5*CABS(WIND(0)) ! Max. of WIND at J=0
HALFWID= 0. !<---(not yet found)
DO J=1,MB ! step along WIND
IF(CABS(WIND(J)).LT.HALFMAX)THEN !\ When .LT.HALFMAX,
WCURR= CABS(WIND(J)) ! \ use lin. inter-
WLAST= CABS(WIND(J-1)) ! > polation to find
SLOPE= 1./(WCURR-WLAST) ! / the half-width
HALFWID= FLOAT(J-1) + SLOPE*(HALFMAX-WLAST)!/
GOTO 10 ! then skip out
ENDIF
ENDDO ! keep on until up to MB
10 IF(HALFWID.LE.0) PRINT *,'*Could not find HALF-WIDTH,HALF-MAX*'
C match a Gaussian of width SIGMA to the above, determine phase increment

```

```

      TWOPI= 8.*ATAN2(1.,1.)           ! calculate 2*PI to precision
      SIGMA= HALFWID/SQRT(2.*ALOG(2.)) !\ parameters for Gaussian
      CONST= 1./(2.*SIGMA*SIGMA)      !/
      dPHASE= TWOPI*dF*(TSUB-TMEAN)   ! dPHASE from time info.
C   Fill the BEAM array
      MB= INT(5.*SIGMA) +1            ! only fill out to 5 sigmas
      DO J= 0,MB
          X= FLOAT(J)
          GAUSS= EXP(-CONST*X*X)       ! Gaussian envelope
          PHASE= dPHASE*X              ! phase linear in F
          BEAM(J)= GAUSS*CMPLX(COS(PHASE),SIN(PHASE))
      ENDDO
C   return to the calling program
      RETURN
      END

```

```

      SUBROUTINE RESTORE(MS,SP,CC,RS,MB,BM)

```

```

C-----
C   Author:   J. Lehar                      Date: 13-JUL-87
C-----
C   Creates the clean spectrum SP(0:MS), by convolving the CLEAN
C   components CC(0:MS) with the clean beam BM(0:MB), and finally
C   adding the residual spectrum RS(0:MS) to the result.
C   This produces a spectrum estimate which more properly represents
C   the frequency resolution (through the beam width) and which includes
C   the fluctuations in the residual spectrum. The CLEAN spectrum will
C   also combine components within one beam-width of each other to build
C   up a signal which might appear as several smaller ones in CC. These
C   should be combined, because frequency separations of less than one
C   beam width are not reliable.
C-----
C   CALLS:    CVAL                          determines spectral values beyond (0:M)
C-----
      COMPLEX SP(0:MS),CC(0:MS),RS(0:MS),BM(0:MB),CVAL
C   Convolve CC with BM, then add RS, to form SP
      DO J=0,MS
          SP(J)= (0.,0.)      ! reset SP(J)
          DO K=-MB,MB
              SP(J)= SP(J)+CVAL(BM,K,MB)*CVAL(CC,J-K,MS) ! convolution
          
```

```

        ENDDO
        SP(J)= SP(J)+RS(J)
    ENDDO
C  return to the calling program
    RETURN
END

```

COMPLEX FUNCTION CVAL(SPEC,L,M)

```

C-----
C  Author:    J. Lehar                               Date: 13-JUL-87
C-----
C  Returns the "value" of the spectrum array SPEC(0:M) at the "location" L.
C    if L >= 0    CVAL returns SPEC(L)
C    if L > 0    CVAL returns the complex conjugate of SPEC(L)
C    if |L| > M  CVAL will return (0.,0.)
C-----

```

COMPLEX SPEC(0:M)

```

C  set value to conjugate of absolute location if necessary
    LOC= ABS(L)                ! where necessary information is found
    IF(LOC.GT.M)THEN           !\
        CVAL= (0.,0.)         ! \  if LOC exceeds range
        RETURN                ! /  return (0.,0.)
    ENDIF                      !/
    IF(L.GE.0) CVAL= SPEC(LOC)  ! return SPEC(L)
    IF(L.LT.0) CVAL= CONJG(SPEC(LOC)) ! return SPEC(-L)
    RETURN
END

```

inout.f This code has been modified from Joe Lehar's original code. It has been ported to run on Linux and compile with the gnu g77 fortran compiler. This should also make it

C*--- INOUT.FOR

```

      FUNCTION RDDATA(N,TIME,DATA,DFILE,HEADER)
C-----
C  Author:  J. Lehar                               Date: 13-JUL-87
C  Written for VAX FORTRAN 77   (May require some changes for other machines)
C  Source file:  INOUT.FOR
C-----
C  Reads in the times, data values for a time series from DFILE
C  In addition, a descriptive header is read in.  N (input) specifies the
C  maximum number of samples allowed and returns (output) the number of
C  samples found.  OPEN uses UNIT=1
C  ARGUMENTS:
C      N          (input) #samples allowed, (output) #samples found
C      TIME       (output) array for time samples
C      DATA      (output) array for data samples
C      DFILE      (input) data file name
C      HEADER     (output) descriptive file header
C  RDDATA error values:
C      0.0       everything went well
C      1.0       #samples exceeds N, array is truncated at N
C-----
      REAL TIME(N),DATA(N)
      CHARACTER DFILE*(*),HEADER*80
      ERR= 0.
C  open the file & read in the header
      OPEN(UNIT=1,FILE=DFILE,STATUS='OLD')
      READ(1,'(A80)') HEADER
C  read in the time series, each record has (Time,Data)
      I=1
C  step through the file until END-OF-FILE
10  READ(1,*,END=11) TIME(I),DATA(I)
      I= I+1
      IF(I.GT.N)THEN
          ERR= 1.
          GOTO 11

```

```

                ENDIF
                GOTO 10
11      CONTINUE
C   close the file & return #samples found & error code
        CLOSE(UNIT=1)
        N= I-1
        RDDATA= ERR
        RETURN
        END

```

```

                SUBROUTINE WRDATA(N,TIME,DATA,DFILE,HEADER)

```

```

C-----
C   Author:   J. Lehar                               Date: 13-JUL-87
C   Written for VAX FORTRAN 77   (May require some changes for other machines)
C   Source file:  INOUT.FOR
C-----
C   Writes the times, data values for a time series from DFILE
C   In addition, a descriptive header is written. OPEN uses UNIT=1
C   ARGUMENTS:
C       N           (input) #samples
C       TIME        (input) array for time samples
C       DATA       (input) array for data samples
C       DFILE       (input) data file name
C       HEADER      (input) descriptive file header
C-----
        REAL TIME(N),DATA(N)
        CHARACTER DFILE*(*),HEADER*80
C   open the file & write the header
        OPEN(UNIT=1,FILE=DFILE,STATUS='NEW')
C   REMOVE THE HEADER IT SCREWS UP GNUPLOT --ZDK
        WRITE(1,'(A80)') HEADER
C   write the time series
        DO I=1,N
            WRITE(1,'(2(1X,1P,G14.7))') TIME(I),DATA(I)
        ENDDO
C   close the file & return
        CLOSE(UNIT=1)
        RETURN
        END

```

```

      FUNCTION RDSPEC(M,FREQ,SPEC,SFILE,HEADER)
C-----
C Author: J. Lehar                               Date: 13-JUL-87
C Written for VAX FORTRAN 77 (May require some changes for other machines)
C Source file: INOUT.FOR
C-----
C Reads in the frequency and spectral values for a spectrum to SFILE.
C In addition, a descriptive header is read in. M (input) specifies the
C maximum frequency element allowed and returns (output) the maximum
C frequency element found. OPEN uses UNIT=1
C ARGUMENTS:
C     M      (input) max. freq element allowed, (output) max. elt. found
C     FREQ   (output) array for frequency samples
C     SPEC   (output) array for spectrum samples
C     SFILE  (input) spectrum file name
C     HEADER (output) descriptive file header
C RDSPEC error values:
C     0.0    everything went well
C     1.0    Max freq element exceeds M, array is truncated at M
C-----
      COMPLEX SPEC(0:M),CBUF
      REAL FREQ(0:M),RBUF(2)
      CHARACTER SFILE*(*),HEADER*80
      EQUIVALENCE (RBUF,CBUF)
      ERROR= 0.
C open the file & read in the header
      OPEN(UNIT=1,FILE=SFILE,STATUS='OLD')
      READ(1,'(A80)') HEADER
C read in the spectrum, each record has (Freq,Real.comp,Imag.comp)
      I=0
C step through the file until END-OF-FILE
10 READ(1,*,END=11) FREQ(I),RBUF(1),RBUF(2)
      SPEC(I)= CBUF
      I= I+1
      IF(I.GT.M)THEN
          ERR= 1.
          GOTO 11
      ENDIF

```

```

          GOTO 10
11      CONTINUE
C close the file, return the max. freq element found & error code
      CLOSE(UNIT=1)
      M= I-1
      RDSPEC= ERR
      RETURN
      END

          SUBROUTINE WRSPEC(M,FREQ,SPEC,SFILE,HEADER)
C-----
C Author: J. Lehar                               Date: 13-JUL-87
C Written for VAX FORTRAN 77 (May require some changes for other machines)
C Source file: INOUT.FOR
C-----
C Writes the frequency and spectral values for a spectrum to SFILE
C In addition, a descriptive header is written. OPEN uses UNIT=1
C ARGUMENTS:
C     M      (input) max. frequency element
C     FREQ   (input) array for frequency samples
C     SPEC   (input) array for spectrum samples
C     SFILE  (input) spectrum file name
C     HEADER (input) descriptive file header
C-----
      COMPLEX SPEC(0:M),CBUF
      REAL FREQ(0:M),RBUF(2)
      CHARACTER SFILE*(*),HEADER*80
      EQUIVALENCE (RBUF,CBUF)
C open the file & read in the header
      OPEN(UNIT=1,FILE=SFILE,STATUS='NEW')
      WRITE(1,'(A80)') HEADER
C write out the spectrum, the spec. file lists FREQ,REAL(SPEC),IMAG(SPEC)
      DO I=0,M
          CBUF= SPEC(I)
          WRITE(1,'(3(1X,1P,G14.7))') FREQ(I),RBUF(1),RBUF(2)
      ENDDO
C close the file & return
      CLOSE(UNIT=1)
      RETURN

```

END

G Gnuplot Code

Gnuplot is a general plotting program that uses a script file to generate plots. Gnuplot can be used to plot to the screen or to a postscript file. To find out more about gnuplot see the gnuplot webpage <http://www.gnuplot.info> or the gnuplot FAQ page <http://www.uccie/gnuplot/gnuplot-faq.html>

Gnuplot scripts end in “.gp” and can for the most part be run directly at the command line with gnuplot like this `gnuplot -persist clean.gp` the “-persist” is needed if gnuplot is plotting to the screen to tell it to leave the plot visible after the program ends. Most of the gnuplot files here can be also called from within scheme with the gnuplot command.

wave.gp This file prints out the wave in time space.

```
set size 1,1;
set origin 0.0,0.0;

set xrange[:]
set ylabel "Amplitude";
set xlabel "Time"
set grid
set data style points
load "params.gp"
plot "test.psi"
```

dirty.gp The “dirty.gp” and “clean.gp” files are the same except for legends and filenames.

```
# Print out the dirty spectrum plots
#
# Amplitude
  load "params.gp"
  set multiplot;

  set lmargin 10
  set rmargin 4
  set size 1,0.6;
  set origin 0.0,0.0;
  set grid
  set yrange [*:*];
  set xrange [0:3]
  set data style lines
```

```
set ylabel "Amplitude";

set xlabel "Frequency";
plot "test.pdsf" using 1:2 title "Dirty"
```

```
#Phase
set title "Dirty Spectrum";
set size 1,0.4;
set origin 0.0,0.6;
set lmargin 10
set rmargin 4
set grid
set data style lines
set ylabel "Phase"

set yrange [-3.1415926:3.1415926];
set xlabel "";
plot "test.pdsf" using 1:3 title "Dirty"

set nomultiplot;
```

clean.gp

```
## This file plots out the clean plots and is called from
## a number of scheme files.
##
# Amplitude
set terminal X11
load "params.gp"
set multiplot;

set lmargin 10
set rmargin 4
set size 1,0.6;
set origin 0.0,0.0;
set grid
```

```
set yrange [*:*];
set xrange [0:3]
set data style lines
set xlabel "Frequency"
set ylabel "Amplitude";
plot "test.pcsf" using 1:2 title "Clean Spectrum" lt -1
```

#Phase

```
set title "Clean Spectrum"
set size 1,0.4;
set origin 0.0,0.6;
set lmargin 10
set rmargin 4
set grid
set data style lines
set ylabel "Phase"
set xlabel ""

set yrange [-3.1415926:3.1415926];

plot "test.pcsf" using 1:3 title "Clean Spectrum" lt -1
set nomultiplot;
```

params.gp This file is generated by the gnuplot function in scheme> it should not be modified.

stats_check.gp

```
set terminal X11
load "params.gp"
set multiplot;
set lmargin 10
set size 1,0.4
set origin 0.0,0.0
set rmargin 4
set grid
set nologscale
set yrange [-0.01:*]
set xrange [0:4]
```

```
set data style points
set xlabel "Frequency"
set ylabel "Amplitude";

plot          "noise-test.pdif"    using 1:2 title "Difference" lt 1

set size 1,0.6
set origin 0.0,0.4
set yrange [-0.01:*]
set title ""
plot "noise-test.pref"    using 1:2 title "Reference" lt 3,\
    "noise-test.pnoisy" using 1:2 title "Noisy" lt 4

set nomultiplot

clean_test_100.gp

set terminal x11

set grid

set title "100 runs"
set xlabel "Noise Amplitude"
set ylabel "Mean value of Highest Noise peak"

set data style linespoints

plot "clean_test_100_10.dat"    using 2:4 title "10 Cleans" ,\
    "clean_test_100_50.dat"    using 2:4 title "50 Cleans" lt 7,\
    "clean_test_100_100.dat"   using 2:4 title "100 Cleans",\
    "clean_test_100_500.dat"   using 2:4 title "500 Cleans",\
    "clean_test_100_1000.dat"  using 2:4 title "1000 Cleans" lt 9,\
    "clean_test_100_5000.dat"  using 2:4 title "5000 Cleans",\
    "clean_test_100_10000.dat" using 2:4 title "10000 Cleans"
```

References

- [1] Roberts, D.H., Lehar, j and Dreher, J.W. (1987). *Astrophysical Journal*, 93(4), 968.
- [2] Kalikow, Louis (1991). *Investigations of the Reliability of Analyzing Time Series with CLEAN*. Senior Thesis, Brandeis University.