INSTITUT FÜR WIENER KLANGSTIL

TAP User Manual

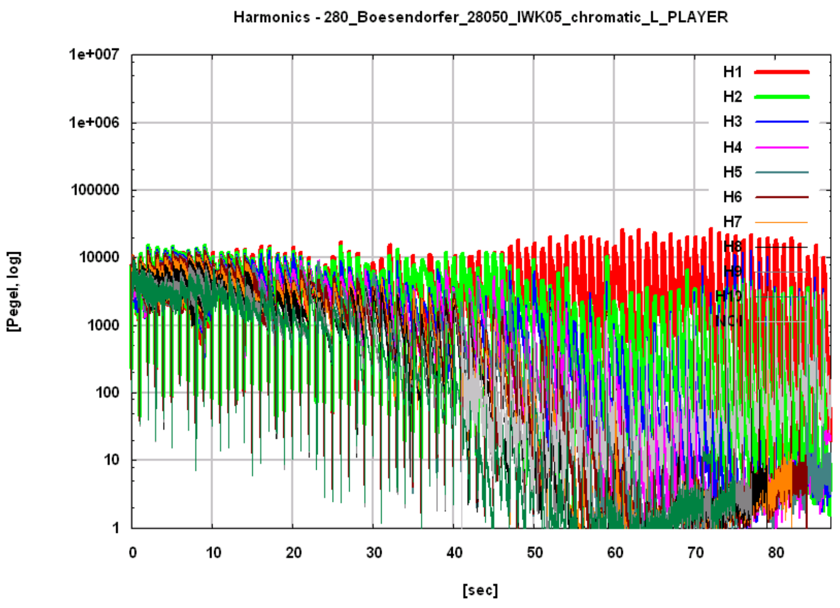
Stand Januar 2011

Wilfried Kausel

Institut für Wiener Klangstil (Musikalische Akustik),
Universität für Musik und darstellende Kunst Wien
Anton von Webernplatz 1 / MO2
A - 1030 Wien

email: ima@mdw.ac.at

http://www.bias.at/



Inhalt

Einleitung

Die Transfer Analysis Package (TAP) ist eine Sammlung von kleinen Programmen, sozusagen ein Baukasten mit Programm-Modulen, die per Kommandozeile oder Scriptfile zusammengefügt werden, um komplexe Aufgaben der Klanganalyse oder Signalverarbeitung vollautomatisch durchzuführen. Z.B. erlaubt das Paket die Entwicklung von neuen Klangdarstellungen, die skriptgesteuert auch eine große Anzahl von zu untersuchenden Aufnahmen analysieren können. Für die graphische Ausgabe auf Bildschirm oder Drucker in TAP ist das mächtige freie Graphikprogramm „GNU-Plot" integriert.

Das Signal Analyse Programmpaket TAP wurde von Wilfried Kausel und Herbert Nachtnebel entwickelt und ursprünglich an der Technischen Universität Wien für die Auswertung der übertragungstechnischen Eigenschaften von Telefonie-Schaltkreisen eingesetzt. Am Institut für Wiener Klangstil wurde es dann in wesentlichen Funktionen erweitert und für die Analyse und Visualisierung von Klängen verwendet.

Das Programmpaket TAP stellt eine Vielzahl von elementaren signalverarbeitenden Einzelprogrammen zur Verfügung, die über die Windows-NT Kommandozeile aufgerufen werden und deren Ein- und Ausgabedaten über den Kommandozeilenoperator | („pipe") miteinander verknüpft werden können. Dabei kann das Verhalten jedes Einzelprogramms durch sogenannte Befehlszeilenoptionen genau gesteuert werden.

Zur Illustration des Prinzips möge das folgende Beispiel dienen:

```
C:\>stim -freq 440 -sample 44100 -len 1000 -noise 0.1 | window -kaiser 10 | dft | db -ref 9 | plot -dx 44.1
```

Das Einzelprogramm stim (stim.exe) ist der Stimulusgenerator. Im Beispiel wird ein Sinussignal mit einer Frequenz von 440Hz bei einer Abtastfrequenz von 44100 Abtastwerten pro Sekunde erzeugt, dem ein Rauschpegel von 0.1V_{rms} überlagert wurde. Als Länge des zu erzeugenden Datenstromes wurden 1000 Werte angegeben.

Die Ausgabe des Stimulusgenerators wird dann mit dem pipe-Operator (|) zur window-Funktion (window.exe) weitergeleitet, die ein Kaiserfenster mit Beta=10 erzeugt und den Datenstrom mit den Werten dieser Fensterfunktion multipliziert. Auf diese Weise wird das Signal im auszuwertenden Zeitraum weich ein und ausgeblendet, was den störenden Effekt reduziert, den hart abgeschnittene Signalperioden bei der Fourier-Transformation bewirken. Eine Voraussetzung für die Anwendbarkeit der Fourier-Transformation ist ja die strenge Periodizität des zu untersuchenden Signals.

Das Programm dft (dft.exe) führt die Fourier-Transformation durch. Ohne Angabe von weiteren Befehlszeilen-Parametern wird nur der Betrag des Spektrums weitergegeben.

Die db-Funktion (db.exe) interpretiert seine Eingangsdaten entsprechend der linearen Skala und gibt die entsprechenden logarithmischen Werte in [dB] aus. Im Beispiel wird der aktuelle Wert der neunten Spektrallinie, das ist der Beitrag des 440Hz Sinustones als Bezugspegel (0dB) gewählt.

Die plot-Funktion (plot.exe) erzeugt dann eine graphische Darstellung des Ergebnis-Spektrums in einem neuen Fenster ohne selbst weitere Ausgabedaten zu erzeugen. Im Plotfenster stehen dann Menü und Mausfunktionen zur Verfügung mit deren Hilfe eine Vielzahl von Darstellungsparametern, wie z.B. Plotbereich, Maßstab, Skala, Beschriftungen, Liniendarstellung u.s.w. verändert werden können. Auch das Ausdrucken sowie das Exportieren der graphischen Daten in andere Programme ist hier vorgesehen.

Im Anhang findet sich die vollständige Beschreibung aller zur Verfügung stehender Einzelprogramme samt einer Beschreibung der unterstützten Befehlszeilenparameter. Die Funktionen lassen sich dabei in mehrere Gruppen unterteilen:

1. Signalquellen (wie z.B. der Stimulusgenerator oder das häufig verwendete Programm zum Einlesen von Wave-Files)
2. Signalsenken (wie z.B. die plot-Funktion oder verschiedene Analyse-Programme)
3. Signalverarbeitende Funktionen für Zeit und/oder Frequenzbereich
4. Arithmetische Operationen für ein und mehrere Datenströme
5. Steuerungsfunktionen zum Programmieren von Programmschleifen und Automatisieren von komplexeren Analyseaufgaben

Die mit TAP am Institut für Wiener Klangstil durchgeführten Klanganalysen setzen auf parametrisierbaren Analyse-Skripts auf, die für diesen Zweck entwickelt wurden. Die Benutzung dieser Skripts erfordert nicht mehr die Kenntnis und das Verständnis der oben erwähnten Einzelprogramme, sondern setzt nur mehr sehr rudimentären Umgang mit dem NT-Kommandozeilen-Interpreter voraus.

Im Idealfall muss ein vorbereitetes Skript nur mehr im Windows-Explorer ausgeführt werden, um eine so komplexe Aufgabe, wie das Auswerten von Wave-Files mit Aufnahmen von chromatischen Tonleitern, gespielt auf 31 verschiedenen Klavieren, vollautomatisch durchführen zu können. Diese Prozedur umfasst z.B. das Segmentieren, d.h. das Auffinden von Anfang und Ende aller angeschlagenen Töne, die Grundtonerkennung, die statische und dynamische harmonische Analyse, die graphische Darstellung und die Erzeugung der vielfältigen Printouts für die Dokumentation.

Derzeit stehen zwei unterschiedliche parametrisierbare Analysemodule zur Verfügung:

- Harmonische Analyse von Tonleitern *[Harmonic Analysis on Scales (harmonics.ana)]*
- Analyse der Spektraldynamik *[Spectral Dynamic Analysis (specdens.ana)]*

Die Module werden in den Batch-Skripts mit frei definierbaren Parametern aufgerufen (options) und liefern verschiedene spezifizierbare Ausgabedaten (targets). Options und Targets der beiden Makromodule finden sich im Anhang C, eine Beschreibung des Programmes „do“ zum Ausführen von TAP Makro-Modulen im Anhang A.

Die Kommandozeile für die Erzeugung aller Plots zu einer chromatischen Tonleiter lautet z.B.:

```
C:\>do -f harmonics.ana all WAV=myWaveFile min=500 f0=40
```

Die Segmentierungs-Schwelle (min=500) ergibt sich aus dem Rauschpegel der Aufnahme und kann einer vorhergehenden RMS Analyse entnommen werden. Die Startfrequenz (f0=40) ist die erwartete Frequenz des ersten angeschlagenen Tones und soll die Grundtonerkennung erleichtern.

Die drei Schnittstellen-Ebenen, das sind die Ebenen der Einzelprogramme, der Makro-Module und der projektspezifischen Skripts, machen TAP zu einem äußerst flexiblen Werkzeug, das umso mächtiger wird, je besser der Anwender sich mit den vielen Einstellmöglichkeiten vertraut macht. Ein nicht zu unterschätzender Vorteil des Konzepts der vielen Einzelprogramme, die durch Pipes (|) verbunden werden, ist die Multi-Prozessor-Fähigkeit. Das Betriebssystem Windows parallelisiert solche Aufgaben, indem jedes Einzelprogramm einem sogenannten Thread zugeordnet wird. Diese Threads werden auf die verfügbaren Prozessoren aufgeteilt und kommunizieren miteinander über die im Skript definierten Pipes.

Anwendungsbeispiel

In der Folge werden einige Beispiele für verschiedene skriptgesteuerte TAP-Analysen gegeben. Die verwendeten Skripts basieren auf dem Makro-Modul zur harmonische Analyse von Tonleitern und erlauben die Ausgabe der Ergebnisgraphiken wahlweise auf dem Bildschirm, dem Drucker oder direkt in ein „.pdf“ oder Postscript Dokument.

Die X-Achse beider Analysegrafiken entspricht dem zeitlichen Verlauf einer auf einem Klavier gespielten chromatischen Tonleiter über die gesamte Klaviatur.

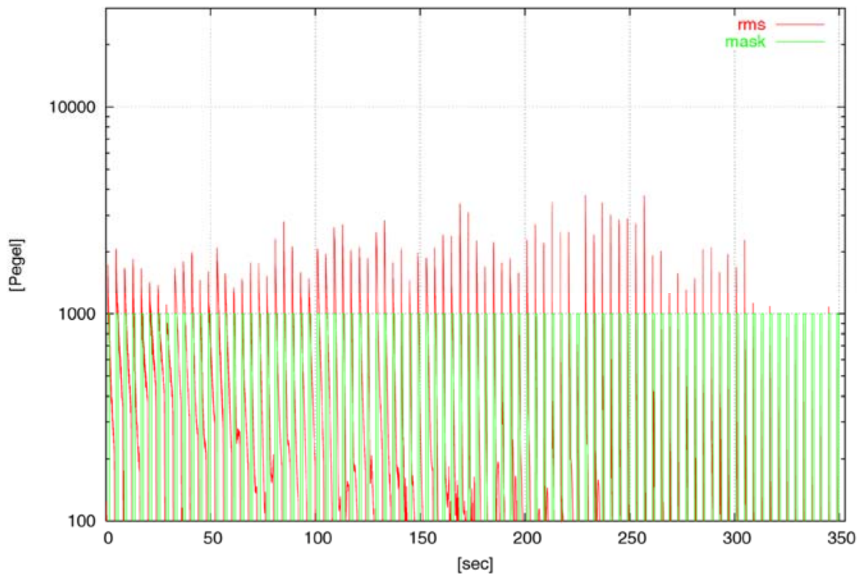


Abb. 1: TAP Analyse 1 (RMS/ Mask): Im ersten Schritt wird der RMS der aufgenommenen Töne berechnet (rote Kurve). Diese dient zur Segmentierung, bei der jeweils eine Sekunde pro Tastenanschlag in einem neuen Soundfile zusammengeschnitten werden. Die grünen Balken zeigen die Maske an, welche zum Schneiden verwendet wird.

Die in TAP verfügbare Segmentierung basiert entweder auf der Anschlagerkennung mit Hilfe einer Pegelschwelle oder auch auf einem fixen Zeitraster, das aber eher nur bei maschinell oder streng mit Metronom eingespielten Aufnahmen Verwendung finden kann. Für den eigentlichen Schnitt können sowohl fixe (Anschnitt) als auch variable (reale) Tonlängen herangezogen werden. Die statische Spektralanalyse erfordert jedoch fixe Tonlängen.

Die weiteren Auswertungen basieren auf dem segmentierten Soundfile, das für jede Taste eine Sekunde Klangsignal enthält, das mit dem Anschlag beginnt. Die Zeitangabe in Sekunden entspricht daher auch der Tonnummer.

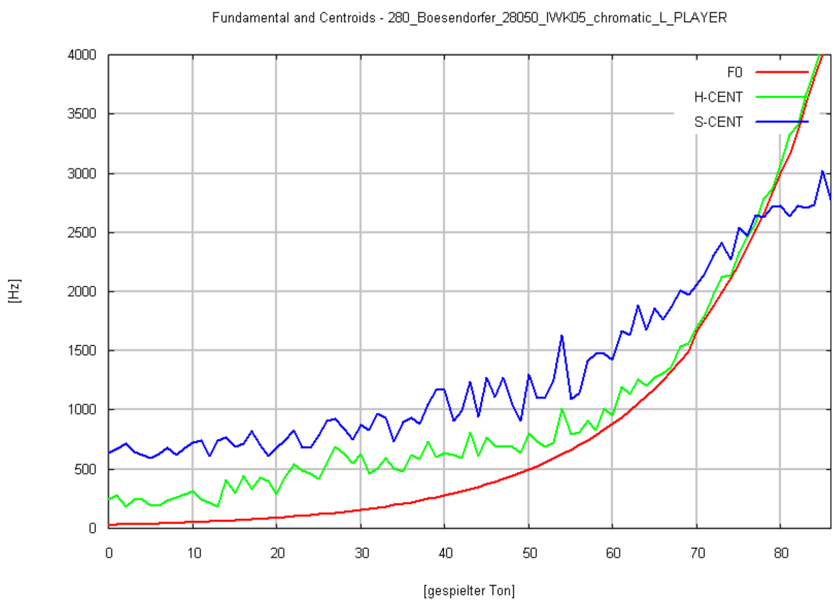


Abb. 2: TAP Analyse 2 (F0 + Centroids): Die Darstellung zeigt das Ergebnis der Grundtonextraktion (F0) als roten Kurvenverlauf, basierend auf dem segmentierten Soundfile, in welchem jeder Ton eine Sekunde andauert. Die Zeit-Achse entspricht daher auch der Tastennummer

Die grüne Kurve ist das „Hamonic Centroid", der pegelgewichtete Mittelwert der ersten 10 Teiltonfrequenzen aller gespielten Töne.

Die blaue Kurve stellt das „Spectral Centroid" dar, die pegelgewichtete Mittelfrequenz des Gesamtspektrums aller gespielten Töne.

Die obige Abbildung zeigt klangliche Schwankungen benachbarter Töne (z.B. Registerbrüche) als unregelmäßigen Verlauf der spektralen Mittelwertkurven (Centroids).

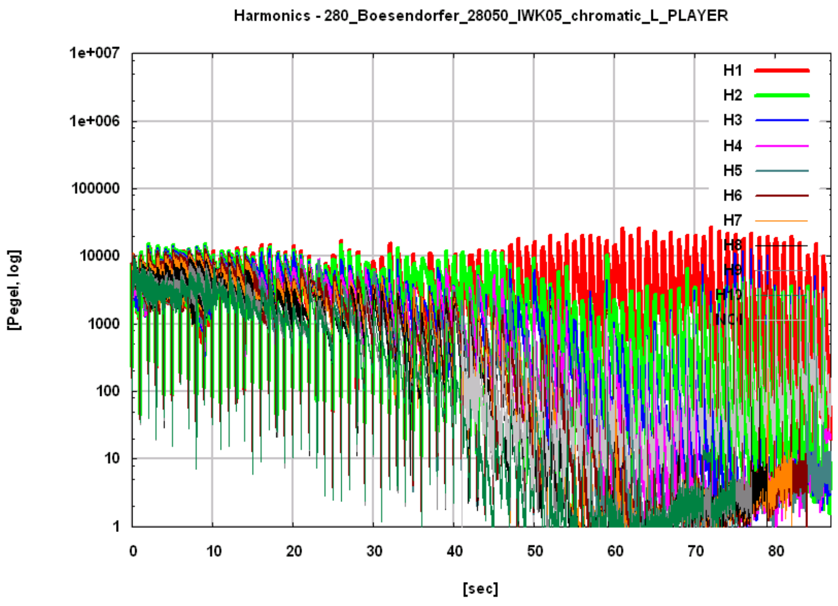


Abb. 3: TAP Analyse 3a (Ten Harmonics + NOI): Die X-Achse entspricht sowohl der Zeit in Sekunden als auch der Tastennummer. Die Y-Achse zeigt den Pegel in einem logarithmischen 20dB Raster.

In verschiedenen Farben (siehe Legende) wird der Intensitätsverlauf der ersten 10 Harmonischen dargestellt.

Zusätzlich zu den 10 Teiltönen (H1...H10) werden die Anschlaggeräusche als elfte Kurve (NOI) dargestellt.

Während die obige Abbildung einen Überblick über den zeitlichen Teiltonverlauf im gesamten Tonumfang gibt, zeigen Ausschnitte einzelner Bereiche viel mehr Details. In den Auswertungsskripts wurden daher von jedem Klavier automatisch gleich drei weitere Detailplots für das tiefe, das mittlere und das hohe Register ausgegeben. Die folgende Abbildung zeigt einen noch kleineren Ausschnitt von 6 Tönen.

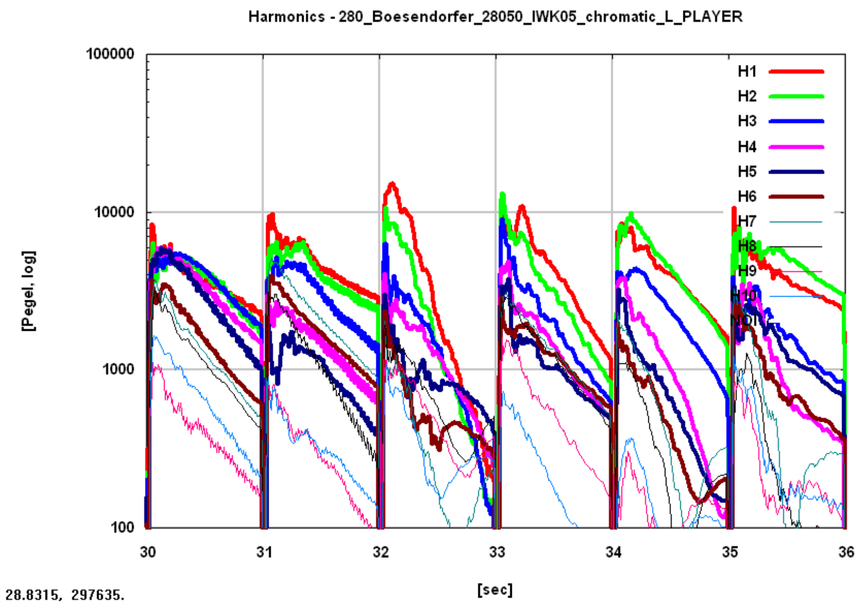


Abb. 4: TAP Analyse 3b (Ten Harmonics + NOI ZOOM): Diese Abbildung zeigt einen kleinen Ausschnitt in entsprechender zeitlicher Vergrößerung. Der dargestellte Ausschnitt umfasst sechs Töne des mittleren Registers.

In Ausschnitten wie diesem wird die zeitliche Entwicklung der Teiltöne weitaus deutlicher. Bei weiterer Vergrößerung wird auch der Einschwingvorgang sichtbar.

Auch die nächsten Darstellungen beruhen auf dem segmentierten Soundfile, bei dem die erste Sekunde aller Tastenanschläge zusammengeschnitten wurden. Im Gegensatz zu den bisherigen Analysen sind bei den nächsten Auswertungen nur ein Mittelwert pro Sekunde d.h. pro Ton dargestellt und nicht ein Verlauf über der Zeit.

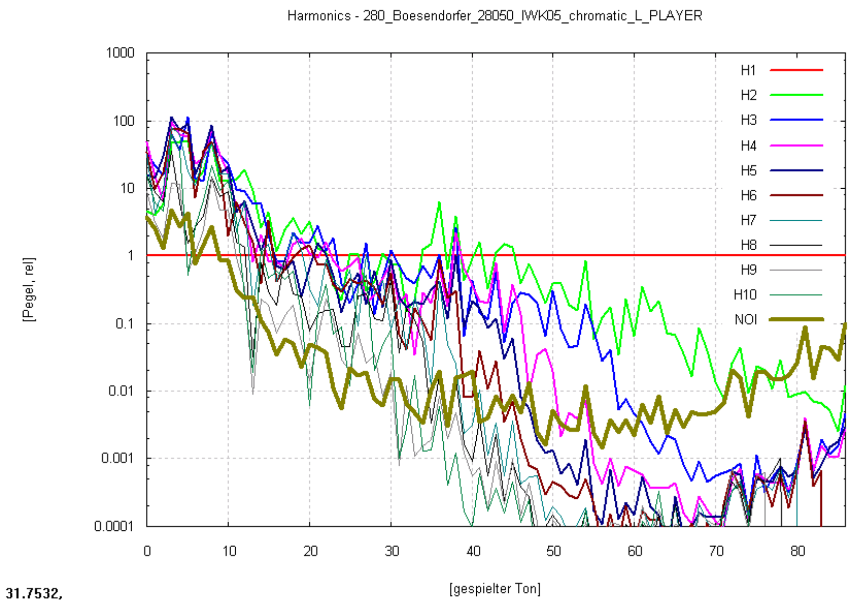


Abb. 5: TAP Analyse 4 (Mean Relative Harmonics): Von jedem angeschlagenen Ton wurde ein Mittelwertspektrum berechnet (erste Sekunde des Anschlags) und die Intensität der ersten zehn Teiltöne sowie des Rausch(Rest-)anteils relativ zur Stärke des Grundtones (1. Harmonische) aufgezeichnet.

Während der Grundton im hohen Register den Klang dominiert, ist seine Intensität im tiefen Register um bis zu 40 dB geringer, als die höherer Harmonischer.

Anhang A: TAP Reference Manual, Basic Level

Table of Contents

Index of TAP functions

Signal Sources

Stimulus Generator (stim)

Generates a binary stream representing a signal to be used by other TAP operators. It combines a sine-wave source, a DC-source, a rectangular signal source, a single Dirac pulse source, various noise sources, a ramp generator and different PCM quantisers.

The sine-wave generator recognizes the level specification (option -lev in dBm0) which is the voltage level relative to the zero reference level. The overload level (option -ovlev in dB) is the maximum level which can be created. It corresponds to an output magnitude specified by option -vpeak. The defaults for -ovlev and -vpeak are normally 0 dB and 1, that means a sine wave between -1 and +1 is generated when no level options are given.

A-law, µ-law and linear (certain number of bits) quantisation can be enabled. The quantisation range is the overload level. If A-law quantisation (-Alaw) is requested then the defaults are adjusted to 3.14 dB and 4096, in µ-law (-Ulaw) mode they are 3.17 dB and 8192. In linear PCM mode (-linear #bits) -vpeak should be set to the available number range and #bits to the selected resolution.

The full scale tuning pitch with standard audio resolution would be specified by:

```
C:\>stim -freq 440 -linear 16 -vpeak 32767 -sample 44100 ...
```

The frequency can either be specified by the number of signal periods in the buffer (-periods) or directly (-freq). In this case the frequency input is adjusted in a way to have an integer (-adjust 1) or odd integer (-adjust 2) number of periods (at least one) in the buffer. If no frequency adjustment is allowed then -adjust 0 must explicitly be specified. If the frequency is set to zero then the sine-wave source is turned off.

Levels of the other signal sources are specified in Volts or units depending on whether an analogue or a PCM source is to be simulated. The other signal sources are a DC source (option -dc), a ramp generator with programmable step size (option -step), a square wave source with given amplitude and frequency (option -square, no frequency adjustment!), a normal distributed white noise source with given RMS value (option -noise), a random noise source with specified peak value (option -rnoise), a pseudo random digital noise source (option -prnoise) implemented as a recirculating shift register and a sine sweep source with a given sweep rate (option -sweep).

The option -pulse affects only the first sample of the buffer. It will be set to the chosen full-scale amplitude.

The pseudo random digital noise source can be used as generator for maximum length sequence (MLS) signals. To generate MLS streams the following seeds can be used:

bits	*period*	*seeds*
8	255	29, 43, 45, 77, 95, 99, 101, 105, 113, 135, 141, 169, 195, 207, 231, 245
9	511	17, 27, 33, 45, 51, 89, 95, 105, 111, 119, 125, 135, 149, 163, 165, 175, 183, 189, 207, 209, 219, 245, 249, 275, 277, 287, 291, 305, 315, 335, 347, 353, 363, 365, 371, 383, 389, 399, 437, 441, 455, 459, 461, 469, 473, 483, 489, 507
10	1023	9, 27, 39, 45, 101, 111, 129, 139, 197, 215, 231, 243, 255, 269, 281, 291, 305, 317, 323, 343, 363, 389, 399, 407, 417, 455, 485, 503, 507, 531, 533, 549, 567, 579, 591, 603, 633, 639, 649, 693, 705, 723, 735, 765, 791, 797, 801, 825, 839, 845, 853, 857, 867, 893, 909, 915, 945, 987, 1011, 1017
11	2047	5, 23, 43, 45, 71, 99, 101, 113, 123, 141, 149, 159, 169, 177, 207, 209, 225, 231, 235, 245, 269, 275, 293, 297, 315, 317, 325, 329, 337, 347, 371, 373, 383, 387, 399, 427, 429, 441, 455, 473, 485, 503, 513, 519, 531, 533, 553, 585, 609, 621, 633, 639, 645, 657, 669, 679, 683, 691, 693, 725, 735, 745, 751, 753, 763, 771, 777, 785, 819, 831, 833, 843, 857, 863, 869, 879, 893, 903, 907, 915, 917, 943, 951, 957, 969, 987, 989, 999, 1005, 1035, 1037, 1049, 1055, 1111, 1121, 1131, 1139, 1157, 1161, 1175, 1179, 1181, 1203, 1215, 1223, 1229, 1235, 1237, 1251, 1257, 1271, 1283, 1295, 1309, 1319, 1325, 1345, 1351, 1365, 1369, 1379, 1391, 1393, 1427, 1439, 1449, 1467, 1469, 1481, 1495, 1499, 1505, 1511, 1525, 1541, 1565, 1569, 1575, 1579, 1587, 1593, 1607, 1611, 1621, 1631, 1649, 1659, 1661, 1665, 1683, 1695, 1699, 1723, 1743, 1757, 1779, 1785, 1803, 1817, 1841, 1847, 1885, 1899, 1901, 1909, 1923, 1937, 1943, 1947, 1959, 1965, 1973, 1997, 2003, 2021, 2025
12	4095	83, 105, 123, 125, 153, 209, 235, 263, 287, 291, 315, 335, 343, 353, 363, 389, 435, 473, 479, 525, 567, 573, 615, 627, 639, 697, 705, 715, 783, 797, 801, 825, 831, 845, 881, 921, 931, 937, 1031, 1073, 1079, 1103, 1117, 1127, 1141, 1191, 1197, 1235, 1295, 1309, 1357, 1427, 1477, 1495, 1501, 1515, 1545, 1607, 1621, 1625, 1701, 1725, 1813, 1817, 1859, 1861, 1909, 1929, 1965, 1971, 1983, 1985, 2135, 2141, 2193, 2199, 2233, 2287, 2331, 2357, 2369, 2405, 2427, 2443, 2481, 2493, 2505, 2511, 2535, 2587, 2603, 2611, 2665, 2699, 2769, 2785, 2805, 2827, 2835, 2847, 2903, 2961, 2983, 3007, 3009, 3027, 3077, 3089, 3095, 3111, 3149, 3207, 3231, 3237, 3259, 3269, 3273, 3279, 3315, 3335, 3363, 3395, 3409, 3419, 3445, 3461, 3465, 3605, 3609, 3631, 3653, 3665, 3687, 3699, 3727, 3811, 3857, 3867, 3879, 3953, 3993, 4027, 4029, 4041
13	8191	27, 39, 53, 83, 101, 111, 139, 141, 159, 165, 175, 187, 189, 195, 201, 225, 243, 269, 277, 297, 303, 315, 323, 359, 363, 377, 393, 407, 413, 447, 449, 455, 461, 479, 483, 497, 507, 537, 549, 567, 573, 579, 603, 605, 633, 639, 649, 663, 667, 691, 703, 717, 751, 759, 763, 773, 807, 811, 839, 853, 857, 879, 881, 893, 903, 909, 917, 931, 937, 945, 951, 955, 993, 1005, 1017, 1035, 1043, 1055, 1061, 1065, 1085, 1105, 1111, 1121, 1133, 1151, 1155, 1179, 1181, 1205, 1215, 1217, 1223, 1227, 1251, 1289, ...
14	16383	43, 57, 83, 95, 123, 169, 175, 187, 189, 207, 235, 243, 269, 275, 315, 323, 411, 413, 423, 429, 437, 469, 473, 497, 525, 599, 609, 639, 645, 669, 711, 715, 717, 739, 745, 751, 777, 801, 831, 893, 903, 917, 943, 969, 1003, 1005, 1035, 1091, 1139, 1235, 1237, 1247, 1251, 1275, 1323, 1337, 1369, 1391, 1433, 1439, 1445, 1463, 1477, 1495, 1511, 1523, 1535, 1551, 1565, 1575, 1589, 1607, 1625, 1635, 1649, 1659, 1701, 1733, 1743, 1755, 1841, 1871, 1919, 1959, ...
15	32767	3, 17, 23, 45, 53, 95, 119, 129, 135, 147, 165, 195, 207, 221, 231, 245, 257, 277, 293, 343, 349, 353, 365, 389, 417, 423, 441, 459, 461, 479, 509, 531, 571, 581, 639, 649, 655, 667, 715, 729, 791, 795, 811, 819, 839, 845, 863, 867, 873, 881, 907, 921, 943, 957, 965, 977, 1049, 1059, 1071, 1073, 1079, 1127, 1133, 1145, 1155, 1175, 1185, 1205, 1247, 1271, 1277, 1309, 1313, 1319, 1331, 1351, 1355, 1375, 1393, 1403, 1409, 1421, 1443, 1457, 1477, 1481, 1499, 1517, 1523, 1545, 1553, 1565, ...
16	65535	45, 57, 63, 83, 189, 215, 303, 317, 335, 349, 407, 417, 429, 447, 455, 533, 537, 549, 559, 605, 621, 645, 657, 673, 741, 797, 843, 873, 881, 903, 909, 927, 931, 989, 1017, 1065, 1111, 1127, 1155, 1161, 1169, 1215, 1217, 1331, 1351, 1385, 1415, 1475, 1501, 1515, 1601, 1611, 1619, 1675, 1731, 1899, 1901, 1913, 1923, 2033, 2061, 2145, 2239, 2261, 2271, 2275, 2289, 2299, 2361, 2429, 2443, 2469, 2479, 2499, 2501, 2535, 2547, 2685, 2689, 2747, 2757, 2817, 2835, 2837, 2897, 2909, 3021, …
17	131071	9, 15, 33, 45, 51, 63, 65, 85, 105, 123, 141, 153, 163, 175, 187, 197, 245, 267, 269, 281, 293, 317, 343, 353, 359, 365, 383, 387, 449, 455, 459, 473, 497, 525, 547, 553, 561, 567, 581, 619, 633, 639, 643, 657, 693, 711, 735, 739, 745, 765, 777, 785, 791, 795, 807, 819, 821, 851, 887, 945, 951, 963, 977, 987, 1003, 1031, 1045, 1049, 1055, 1071, ...
18	262143	39, 63, 77, 123, 129, 219, 231, 237, 263, 335, 401, 483, 489, …

The procedure of generating the above seed values using TAP is described as one of the examples given in the Appendix B.

Using the -merge option it is possible to add the generated stimulus signal to an already existing input data stream which must match in length, of course. This way it is possible to apply quantisation to an input signal. This option also allows composition of input stimuli containing more than one sine (multi-tone), ramp (piece wise linear) or square wave sources or to add a dither signal to an input wave form.

The option -extend creates an extra prologue to the signal without effecting the calculation of signal frequencies or signal periods. The actual length of the generated output stream is given by the -len option plus the -extend option. The idea is to allow a simulated system to settle. Before processing the system's response the clip command can be used to remove that extra prologue again leaving -len samples to reflect the steady state system response.

The -rep option simulates a sample and hold term by repeating each sample several times. This does not actually increase the buffer length specified by -len. It rather divides the visible sampling frequency specified by the -sample option. It does not effect the frequency or the number of periods of a sine-wave signal but it does effect the frequency of a square-wave signal and the slope of a ramp.

The sine-sweep generator is turned on using the -sweep option. It will use the sine-wave generator's settings as its initial frequency and it will sweep according to its specification given in decades per second.

Number of signal periods (nsp)	Round(freq / sample * len)
Adjusted signal frequency (fs)	nsp * sample / len
Time step (T)	rep / sample
Generated sine-wave	peak * 10^((lev-3.17) / 20) * sin(2 Pi fs n T)

Table 1: Stimulus Generator Formulas

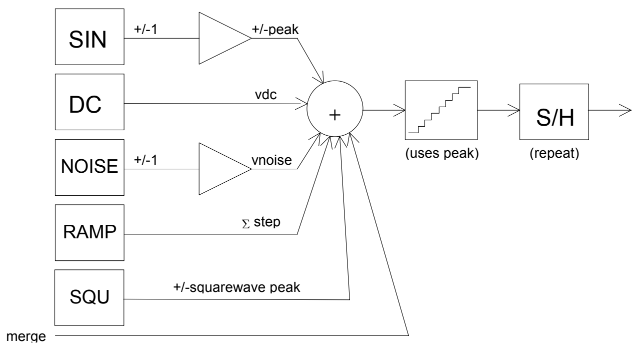


Figure 1: Stimulus Generator

Usage: stim [options] [< infile[1]] > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-freq double	[Hz]	signal frequency (approximate)	(1023.44 Hz)
-adjust int		adjust frequency for {0=any | 1=int | 2=odd} number of periods	(2)
-periods double		number of signal periods in buffer	<off>
-phase double	[rad]	initial phase for sine wave	(0 rad)
-sweep double	[Decades / sec]	frequency sweep rate	(2.3518 Decades / sec)
-lev double	[dBm0]	signal level relative to vpeak	(0 dBm0)
-ovlev double	[dBm0]	overflow level corresponding to vpeak	(0 dBm0)
-vpeak double	[V|units]	peak value (corresponding to 3.14/3.17 dB)	(1 V|units)
-sample double	[Hz]	sampling rate	(8000 Hz)
-dc double	[V]	dc offset	(0 V)
-noise double	[V]	white noise RMS (normal distributed)	(0 V)
-rnoise double	[V]	random noise peak	(0 V)
-prnoise bitlen seed zero one *int bitlen* *unsigned seed* *double zero* *double one*	 *[V]* *[V]*	pseudo random digital noise *shift register length* *seed (defines feedback polynom)* *amplitude assigned to logical 0* *amplitude assigned to logical 1*	<off> *(12)* *(83)* *(1.0 V)* *(-1.0 V)*
-step double	[V]	ramp generator step	(0 V)
-Alaw		enable A-law quantiser G.711	<off>
-Ulaw		enable U-law quantiser G.711	<off>
-lin int		linear PCM representation	(0)
-square vpeak freq *double vpeak* *double freq*		sqare wave *peak value* *frequency*	<off>
-pulse		create Dirac pulse	<off>
-rep int	[times]	repeat each sample	(1 times)
-len int	[samples]	buffer length	(1024 samples)
-extend int	[samples]	extend buffer length with settling time	(0 samples)
-merge		merge (add) with data from stdin	<off>
-query		output actual frequency and periods	<off>
-info		print actual settings to statistic	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

[1] If merge option set

Floating point (ASCII file) to binary stream conversion (flo2bin)

Converts an input stream containing a table of numerical values into a TAP compatible binary stream. Reads ASCII integer or floating point numbers or packed binary int32 arrays.

Especially when signals are generated by external programs (e.g. a VHDL simulator) it is necessary to convert an ASCII file containing a table of numerical values into the binary output stream required by the other signal processing functions contained in this package.

The ASCII input file usually contains one number per text line. It should start with an integer number representing the number of floating-point values to follow (if this is not the case automatic size detection can be enabled by using the -nocnt option). This starting number must be followed by the indicated number of floating-point values separated by white space characters. The file must not contain anything except valid IEEE floating-point formats. However, a known number of header lines containing text in any format will be skipped when specified using the -header option.

Alternatively a packed binary array of 32-bit machine precision integers can be read using the -int32 option. The number of values is determined from the file size and must not be given in that case.

Usage: flo2bin [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nocnt		no number of values in input stream	<off>
-int32		input stream is array of int32 (no count upfront)	<off>
-header		number of header lines to skip in input stream	(0)
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

PCM (ASCII file) to binary stream conversion (pcm2bin)

Converts an input stream containing an ASCII table of PCM values into a TAP compatible binary stream.

Especially when signals are generated by external programs e.g. a VHDL simulator it is necessary to convert an ASCII file containing a table of PCM values in binary, octal, decimal or hexadecimal format into a binary output stream required by the other signal processing functions contained in this package.

The ASCII input file usually contains one number per text line. It should start with an integer number representing the number of PCM values to follow (if this is not the case automatic size detection can be enabled by using the -nocnt option). This starting number must be followed by the indicated number of PCM values. The PCM numbers must conform to the format and number base specified (hexadecimal, decimal, octal, binary). The default number format is hexadecimal with no sign extension (expecting sign & magnitude).

If a fixed number of input bits (less than 32) is specified then the sign bit expected at position 2^{bits} is extended in order to convert the number to 32 bit binary integer representation. Specifying -bits 0 selects a sign magnitude format with variable number of digits.

Usage: pcm2bin [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nocnt		no number of values in input stream	<off>
-bits int		number of bits of PCM value	(32)
-lin int		linear PCM representation	(0)
-Alaw		use A-law conversion	<off>
-Ulaw		use U-law conversion	<off>
-radix int		input radix	(16)
-hex		hexadecimal input	<off>
-dec		decimal input	<off>
-oct		octal input	<off>
-bin		binary input	<off>
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Windows Sound File (.WAV file) to Binary Stream Conversion (wav2bin)

Converts a windows .WAV sound file (Microsoft RIFF format) into a TAP compatible binary stream.

Read windows sound file (.wav) and generate a binary output stream required by the other signal processing functions contained in this package. If the sound file is a stereo file then left and right channel data words are interleaved. Data words are numbers in the range of -32768 to +32767 if the wave file resolution is 16 bit. 8, 16 and 24 bit audio files are supported.

The sound file name must be specified with the -fname option. If the -merge option is set then the content of the sound file is appended to the input stream.

To separate f.e. the right channel of a stereo sound file the stream can be processed like in the following example:

```
C:\>wav2bin -fname mySound.wav | table -sig 2 -extr 2 | ...
```

`Usage: wav2bin [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-fname char*		name of existing wave file to read from	<off>
-merge		append wave file data to input stream	<off>
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Windows Video File (.AVI file) to Binary Stream Conversion (avi2bin)

Converts a windows .AVI video file (Microsoft RIFF format) into a TAP compatible binary stream.

Read compressed or uncompressed video file (.avi) and generate a binary output stream required by the other signal processing functions contained in this package.

Specify a certain subset of video frames using the options -start, -stop and -step. Select a clipping window using the -clip option. If detailed format information is required the -term option (do not propagate data stream) may be given in conjunction with the -stat option (write status information to external text file).

The video file name must be specified with the -fname option. If the -merge option is set then the pixel content of the video file is appended to the input stream.

To do image processing a video file can be read, modified and written back like in the following example:

```
C:\>avi2bin -fname myVideo.avi | map  ... | bin2avi -fn out.avi ...
```

Usage: avi2bin [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-fname char*		name of existing wave file to read from	<off>
-start int		first frame to output	(first)
-stop int		last frame to output	(last)
-step int		frame step for output	(1)
-clip *left int* *right int* *bottom int* *top int}*		clipping rectangle *(negative means from opposite edge)* *(0 for full width)* *(0 for full height)*	<0 0 0 0>
-merge		append wave file data to input stream	<off>
-vfw		use video_for_windows avi parser	<off>
-term		do not generate output stream (file check only)	<off>
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

DAC (ASCII file) to binary stream conversion (dac2bin)

Simulates a switched capacitor DAC driven by clock signals specified in an ASCII table. Reads clock states from textual input stream and creates a TAP compatible binary output stream containing the simulated DAC output signal. Can be used to analyse the effect of data-weighted-averaging if corresponding clock signals are applied.

A nine level switched-capacitor digital-to-analogue converter with integrated low-pass filter can be simulated. The nine level circuit allows dynamic averaging even of absolute gain. Its circuit diagram with a complete functional description is shown in Appendix B as Figure 19.

By controlling the DAC elements in a certain manner (specifying an appropriate SC-clock sequence in the input stream), data weighted averaging of all capacitor values can be achieved. Input data is interpreted as a set of bits corresponding to the states of DAC element control lines in the order PHA[8], PHB[8], PHC[8], reserved, reserved, reserved, PH1. It is assumed that PH1 (the preparation phase) is the inverse of PH2 (the working phase). The output values are created by summing up the contributions of all DAC elements which are enabled by their corresponding control lines during PH2. No output is generated during PH1. Two input lines are therefore required to produce one output sample.

The ASCII input file usually contains one circuit state per text line. It should start with an integer number representing the number of data lines to follow (if this is not the case automatic size detection can be enabled by using the -nocnt option). This starting number must be followed by the indicated number of data lines. The data format must conform to the format and number base specified by options (hexadecimal, binary). The default number format is seven-digit hexadecimal with right adjusted status data.

Capacitor mismatch can be specified and the continuous time feedback capacitor can be varied to achieve different filter cut-off frequencies. The clock signals driving the CMOS switches of the simulated circuit are usually generated by a logic or behavioural simulator like VERILOG or VHDL.

Usage: dac2bin [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nocnt		no number of values in input stream	<off>
-radix int		input radix	(16)
-hex		hexadecimal input	<off>
-dec		decimal input	<off>
-oct		octal input	<off>
-bin		binary input	<off>
-vref double	[V]	reference voltage (+- full range)	(1.4125 V)
-fb double	[pF]	feedback capacitor (tau * fsample * Cunit)	(13 pF)
-cap		capacitor values (Cunit +- dC)	<off>
C1 double	*[pF]*	*.value of feedback capacitor 1*	*(1.00870 pF)*
C2 double	*[pF]*	*.value of feedback capacitor 2*	*(1.00873 pF)*
C3 double	*[pF]*	*.value of feedback capacitor 3*	*(1.01947 pF)*
C4 double	*[pF]*	*.value of feedback capacitor 4*	*(1.00983 pF)*
C5 double	*[pF]*	*.value of feedback capacitor 5*	*(1.00354 pF)*
C6 double	*[pF]*	*.value of feedback capacitor 6*	*(0.99873 pF)*
C7 double	*[pF]*	*.value of feedback capacitor 7*	*(0.991411 pF)*
C8 double	*[pF]*	*.value of feedback capacitor 8*	*(1.00877 pF)*
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Signal Sinks

Plot (plot)

Creates a PX-Graph or GNU-Plot window to display input stream data. For the window a new interactive child process is created. The graphical window offers a zooming function and allows creation of postscript files for a printer. Options control the shape of the plot, the number of signals, plot and axes labels and other features. Plot -all combines all open plots in a single plot window.

If the option -signals is set to a value > 1 then multi column input is assumed. If no y columns are selected using the option -ycol then all columns are displayed in one frame. If an x-axis column is specified by the option -xcol then this column is used to label the x-axes instead of being displayed as a separate trace. Column numbers are starting with the number one.

If no column for the x-axis is given then an x-axis start value can be specified using the option -x0 and an x-axis increment can be entered using the option -dx.

A plot title can be specified using the -title option. Axis labels are entered using the -xunit and -yunit strings. If enough signal names are provided using the -names option then signals are named rather than numbered. The options -ltype and -lwidth allow to specify lists of line type codes and line width numbers.

The options -lnx and -lny select logarithmic axis scaling providing data values are positive and non-zero. The option -grid turns grid display on. The -opts string is directly passed to GNUPlot.

The combination of the -mask option (gt[2], gtn, td[3], tdn, wn[4], tn[5], sd[6], ob[7], gf[8], gfx, ov[9], gd[10]), the -dir option (tx or rx) and the -law option (a or u) selects several specification masks used in the ITU telecommunication standards G711, G712 and others.

[2] Gain Tracking

[3] Total Distortion

[4] Weighted Noise

[5] Total Noise

[6] Single Frequency Distortion

[7] Out of Band Noise

[8] Gain over Frequency

[9] Overflow Level

[10] Group Delay Distortion

To create a GNUPlot console window linked to a plot window hit the <space> bar. Hit the <h> key to display a list of all such key stroke commands in that console window. The console window contains additional online help functions. The [Replot] button will execute all modifications which might have been made. Zooming can be accomplished by clicking two corners of a zoom window using the right mouse button. The <p> key will bring you back to the previous view.

The window menu, the icon at the left top corner of a plot window, contains important options like print and copy to clipboard.

Usage: plot [options] < infile

Options:

-isignal ipipe		read input signal from	(stdin)
-signals int		number of columns in stream	(1)
-nowait		do not enter interactive mode	<off>
-surface		plot surface spanned by signals	<off>
-xcol int		select column for x axis	<off>
-ycol int *{ int }*		select column for y axis *more than one columns may be specified*	<off>
-normcol int		scale columns, let RMS of this column be unity	<off>
-pluscol int		add this column to ycols	<off>
-minuscol int		subtract this column from ycols	<off>
-timescol int		multiply this column to ycols	<off>
-divcol int		divide ycols by this column	<off>
-yoffs double		subtract this value from ycols	<off>
-xref double		y value of this sample is offset	<off>
-x0 double		x axis begin	(0)
-dx double		y axis increment	(1)
-title char*		plot title	<off>
-name char* *{ char* }*		signal name *more than one name may be specified*	<off>
-ltype int *{ int }*		line type *more than one type may be specified*	<off>
-lwidth int *{ int }*		line width *more than one width may be specified*	<off>
-dots		use dots for plotting	<off>
-xunit char*		x axis label	(x)
-yunit char*		y axis label	(y)
-lnx		logarithmic x axis	<off>
-lny		logarithmic y axis	<off>
-grid		draw grid lines	<off>
-mask char*		name of predefined mask	<off>
-dir char*		transmit or recieve mask (tx or rx)	<off>
-law char*		A-law or U-law mask (a or u)	<off>
-cmask double		draw horizontal line at y position	<off>
-opts char*		additional GNUPLOT options or commands (use ; and ' as delimiters)	<off>
-all		join all active plots	<off>
-swap		exchange adjacent bytes of values	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Binary stream to PCM (ASCII file) conversion (bin2pcm)

Converts a TAP compatible binary input stream into a textual output stream containing an ASCII table of PCM values.

While most functions are getting binary input streams and creating binary output streams it is sometimes necessary to convert such a binary result into a readable ASCII representation. The ASCII file created contains one number per text line. It normally starts with an integer number representing the number of PCM values to follow (if this is not turned off by using the -nocnt option). This starting line is followed by the indicated number of data lines. The binary input values are converted to PCM values either according to G.711 A-law or U-law or truncated to a certain number of fixed point fractional bits.

The default number format for A-law and U-law is two-digit hexadecimal, for linear PCM eight-digit hexadecimal. Other number systems like decimal or octal can be selected, too. If the number of bits is set to zero then sign-magnitude representation with a variable number of digits is selected otherwise sign-extension and zero-padding for the indicated number of bits is chosen.

```
Usage: bin2pcm [options] < infile > outfile
```

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nocnt		suppress generation of values count	<off>
-bits int		number of bits of PCM value	(32)
-lin int		linear PCM representation	(0)
-Alaw		use A-law conversion	<off>
-Ulaw		use U-law conversion	<off>
-radix int		output radix to use	(16)
-hex		generate hexadecimal output	<off>
-dec		generate decimal output	<off>
-oct		generate octal output	<off>
-bin		generate binary output numbers	<off>
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Binary to floating point (ASCII file) conversion (bin2flo)

Converts a TAP compatible binary input stream into a textual output stream containing an ASCII table of floating point values.

While most functions are getting binary input streams and creating binary output streams it is sometimes necessary to convert such a binary result into a readable ASCII representation. The ASCII file created contains as many numbers per text line as specified by the -columns option. It normally starts with a separate line containing an integer number representing the number of floating-point values to follow (if this is not turned off by using the -nocnt option). This starting line is followed by the indicated number of data lines. The default number format can be overwritten by specifying a format string following C-language printf syntax.

ASCII files will be used to interface the VHDL simulator and to export the final numerical result of a sequence of stream processing operators.

`Usage: bin2flo [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nocnt		suppress generation of values count	<off>
-format char*		format string of double values	(%20.12lg)
-columns int		number of columns written to stream	(1)
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Binary Stream to Windows Sound File (.WAV file) Conversion (bin2wav)

Converts a TAP compatible binary input stream into a windows .WAV sound file (Microsoft RIFF Format).

Take a TAP compatible binary input stream and create a windows sound file in RIFF format (standard wave file which usually has the extension .wav). Only PCM format is supported (-format 1)

The sound file format can be specified using the -channels, -samples and -bits options or by using the -like option to specify the file name of an unrelated wave file with a matching format.

The format to be used to create an audio CD is:

```
    -channels 2 -samples 44100 -bits 16
```

If a stereo file is to be generated then input samples of left and right channel must be interleaved. The filename of the new wave file must be given using the -fname option.

`Usage: bin2wav [options] < infile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-fname char*		name of new wave file to create	<off>
-like char*		name of another existing wave file to copy header descriptions from	<off>
-stat char*		write program statistics to file	<off>
-format int		wave file format	(1)
-channels int		number of channels	(1)
-samples int		number of samples per second	(8000)
-bits int		number of bits per sample	(8)
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Binary Stream to Windows Multimedia File (.AVI file) Conversion (bin2avi)

Converts a TAP compatible binary input stream into a windows .AVI multimedia file (Microsoft RIFF Format).

Take a TAP compatible binary input stream and create a windows multimedia file in RIFF format (standard movie file which usually has the extension .avi). Compressed and uncompressed frame format is supported (-compress FOURCC)

The frame format of the output video may be controlled independently of the dimension of the input stream by using the -width, -height and -interpolation options.

The command line to be used to create an uncompressed video stream with 400x400 RGB pixes from a 20x20 input data stream using green shades for positive displacement amplitudes and red shades for negative displacement is:

```
-fname test.avi -wid 20 -height 20 -interpol 20 -colpair -fps 15 -auto
```

The playback rate is set to 15 fps and automatic scaling of input level is used. The filename of the generated file must be given using the -fname option.

The samples of the input stream after scaling are either interpreted as color-map indices (-colmap option) or as signal amplitudes to be mapped onto a range of intensities of a color pair (-colpair option) or gray scale (-colbw option) or directly as RGB values.

The stream has to contain any number of complete frames, each consisting of a specified number of lines (-height option) with a specified number of pixels (-width option).

Input pixels are over-sampled by a user specified interpolation factor (-interpolation option) to create target resolutions higher than input resolutions. Available interpolation methods are pixel repetition, linear interpolation and overlapping 2D-Hamming window weighting.

Animation title (-title), sampling time, frame count or an external frame identifier (-time) as well as a color bar legend (-legend) can be included. A background image stored in a Windows bitmap file (*.bmp) may also be specified (-background).

A straight line may be defined (-xsection) specifying any point and any inclination angle which will define a cross-section through the two dimensional data plane. A line plot of the signal magnitude profile across that line is then plotted and updated in all animation frames.

```
Usage: bin2avi [options] < infile
```

Options:

-isignal ipipe		read input signal from	(stdin)
-fname char*		name of new avi file to create	<off>
-width int		width of input stream in pixels	(32)
-height int		height of input stream in pixels	(24)
-interpolation		interpolate (increase avi resolution)	<off>
factor int		*oversampling factor*	*(1)*
order int		*interpolation order:*	*(2)*
		0..sample repetition	
		1..linear interpolation	
		2..Hamming filter	
yfactor int		*oversampling factor in y-direction*	*(0)*
		(if different from x)	
-circular		circular symmetry	<off>
orad int		*radius of outer edge*	*(0)*
irad int		*radius of inner edge*	*(0)*
list int		*list of radii (one radius per input row,*	
{ int }		*smallest first)*	
-cplx		input stream is complex (mag,arg)	<off>
resolution int		*number of frames per period*	*(36)*
periods int		*number of periods to animate*	*(2)*
-ac		eliminate common magnitude per frames	<off>
-clip		clip amplitude range to avoid color periodicity	<off>
-colpair		display neg/pos input using color pair	<off>
		(0=Blue, 1=Green, 2=Red)	
pcol int		*positive color*	*(1)*
ncol int		*negative color*	*(2)*
-colbw		display black and white only	<off>
-colmap		specify custom color map (RGB triples)	<off>
		colors are assigned from most	
		negative to most positive values	
*fname char**		*name of binary data file*	*("rgb.bin")*
upsampling int		*number of interpolated color triples*	*(254)*
lines int		*number of contour lines (pos..white, neg..black)*	*(0)*
lwidth int	*[%]*	*width of contour lines*	*(20)*
-compression		name of codec to be used (FOURCC)	<off>
char*			*("TSCC")*
-fps int		frames per second to be generated	(5)
-scale double		initial scale	(1.0)
-range		signal amplitude range (if -autoscale or ext	<off>
		then only distance is preserved!)	
llim double		*lower amplitude limit*	*(-1.0)*
ulim double		*upper amplitude limit*	*(1.0)*
*ext char**		*read upper limits from external binary file*	
-autoscale		rescale dynamically	<off>
tconst double	*[s]*	*recovering time constant*	*(10)*
		(0 for immediate recover)	
-log		logarithmic compression	<off>
base double		*base of logarithm*	*(2.0)*
range double		*logarithmic range (exponent)*	*(12.0)*
-background		background image	<off>
*filename char**		*name of bitmap file (*.bmp)*	*("bg.bmp")*
saturation double		*saturation factor (0=BW)*	*(0.5)*
brightness double		*adjust brightness (1=white)*	*(0.5)*
-time		annotate time or frame number	<off>
*format char**		*format string (e.g. '%2.3fms' for time in ms)*	*("%6.0f")*
fps double		*original frame rate (1 for frame count)*	*(1)*
x int	*[pixel]*	*x position (0=center | neg=from right)*	*(5)*
y int	*[pixel]*	*y position (0=middle | neg=from bottom)*	*(5)*
points int	*[points]*	*height of font in points*	*(8)*
*font char**		*name of font*	*("Arial")*
col int	*[RGB]*	*text color*	*(0xffffff)*
bgcol int	*[RGB]*	*text background color*	*(0x000000)*
*ext char**		*read from external binary file*	

-title *format char** *x int* *y int* *points int* *font char** *col int* *bgcol int*	 *[pixel]* *[pixel]* *[points]* *[RGB]* *[RGB]*	annotate animation title *title string* *x position (0=center | neg=from right)* *y position (0=middle | neg=from bottom)* *height of font in points* *name of font* *text color* *text background color*	<off> *("")* *(5)* *(8)* *("Arial")* *(0xffffff)* *(0x000000)*
-legend *width int* *height int* *hpos int* *vpos int* *points int* *font char** *col int* *bgcol int*	 *[pixel]* *[pixel]* *[pixel]* *[pixel]* *[points]* *[RGB]* *[RGB]*	include legend *x legend bar width (-1 full width)* *legend bar height (-1 full height)* *distance to left edge (neg=to right)* *distance to top edge (neg=to bottom)* *height of font in points* *name of font* *text color* *text background color*	<off> *(10)* *(-1)* *(-5)* *(0)* *(8)* *("Arial")* *(0xffffff)* *(0x000000)*
- xsection *scale int* *fi int* *x int* *y int* *col int* *wid int* *typ int* *xcol int* *typ int*	 *[%]* *[deg]* *[pixel]* *[pixel]* *[RGB]* *[pixel]* *[0..4]* *[RGB]* *[0..4]*	plot cross-sectional displacment *relative scale* *cross-section angle (90=vertical)* *x position (0=center)* *y position (0=center)* *line color* *line width* *line type (solid,dash,dot,dashdot,dashdotdot)* *axis color* *axis line type*	<off> *(50)* *(0)* *(0)* *(0)* *(0xffff80)* *(1)* *(0)* *(0x808080)* *(2)*
-BMPout *name char** *frame int*		create a series of bitmap files *name of file (group)* *frame (-1 for all)*	<off> *(0)*
-binout		create binary output pixel stream	<off>
-noavi		suppress avi file generation	<off>
-debug		write debug output to statistics file	<off>
-stat char*		write program statistics to file	<off>
-swap		exchange adjacent bytes of values	<off>
-? | -help		generates this help screen	<act>

Time Domain Processing

Envelope Sort (envsort)

The envelope sort operation sorts a buffer containing a certain number of signal periods in a way to move all samples into a single period increasing the effective sampling frequency. This operation is very similar to what a sampling oscilloscope does.

The effect of the envsort command can be illustrated most easily by comparing the results of the following two command lines:

```
stim | plot
stim | envsort | plot
```

Using just the default settings of the stimulus generator, a 1024 sample long binary stream is generated representing a sine wave signal with exactly 131 periods contained in the interval. The envelope sort command resorts those 1024 samples in a way to represent a sine wave signal with one period sampled at a virtual sampling rate which is 131 times higher than the original sampling rate.

The required up-sampling rate is tried to be recognized automatically but it can be specified using the -periods option. In order to work properly an integer number of signal periods must be contained in the original buffer. If the number of original periods is prime then there is no redundancy in the signal and the resorted result will look most smoothly.

The -orgphase option will keep the original phase, while the default setting tries to resort the signal in a way to present a zero phase output signal.

`Usage: envsort [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-periods unsigned		number of periods	<off>
-orgphase		preserve original phase	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Sigma Delta Modulator (sigma)

The sigma delta modulator simulates a first or second order, single bit, switched capacitor sigma delta loop. Symmetric (+1/-1) output as well as logic (1/0) output is provided. Cyclical reset of output and feedback can be selected. Integrator gains as well as operating ranges are programmable.

The sigma function takes an input stream representing an analogue input signal and simulates a first or second order sigma delta modulator with a single bit output. The operating range of the first order version is up to ±vref while the second order version requires an additional margin of about 3 dB. When the default configuration of the circuit is not altered using the options -order, -g1, -g2, -lim1, -lim2, -vref then the working input range is ±1V and a DC output offset of 0.5 V is generated.

The single bit output stream when filtered by a good low pass filter represents with very high precision the analogue input signal. The -sym option switches from the [1,0] set of output states to the [1,-1] set thus eliminating the output DC-offset.

The following command lines will illustrate the operation of sigma delta modulation. The sinc term is the decimation filter, the func term compensates the 3dB loss caused by the second order sigma delta circuit.

```
stim -per 1 | sigma -sym | plot
stim -per 1 | sigma -sym | sinc -n 16 | func -att 3 | plot
```

By zeroing every nth feedback value long strings of the same logic level can be avoided when big input magnitudes are present. This helps to overcome overrun conditions of simple comb decimation filters.

`Usage: sigma [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-order int		mudulator order (1 or 2)	(2)
-sym		symetric output	<off>
-g1 double		gain of first integrator	(0.25)
-g2 double		gain of second integrator	(0.5)
-lim1 double	[V]	operating range of first integrator	(1.5 V)
-lim2 double	[V]	operating range of second integrator	(1.5 V)
-zero int		zero every nth feedback value	<off>
-vref double	[V]	feedback reference voltage	(1.41254 V)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Decimation, Interpolation, DC-component (sinc)

A Sinc decimator applies a digital filtering operation (comb or Sinc filter) to the input stream and then down-samples the signal to a lower sampling rate. The Sinc interpolator up-samples the signal obtained by the input stream and then applies a Sinc interpolation filter at the higher sampling rate. The first order sinc interpolator (n<0) is actually a sample value repeater.

The filtering operation which is generally referred to as Sinc or comb filter creates zeroes at the lower sampling rate and its multiples. Orders up to three are implemented. A first order Sinc decimator is equivalent to a block-wise running average (accumulate and dump operation). A first order Sinc interpolator basically is a circuit which repeats each input sample n times. A second order Sinc interpolator is a linear interpolation circuit.

The spectral effect of a Sinc filter on the pass-band can be compensated by a subsequent window operation using the -invsinc option.

Using the -a option or setting the decimation rate to the buffer length (which is what the -a option actually does) outputs the mean value of all samples which is the dc-component of the signal if first order is selected.

$$f(z,D)=\frac{1-z^{-D}}{1-z^{-1}} \qquad h(x,D)=\left.\frac{1-\left(\cos(x)+i\sin(x)\right)^{-D}}{1-\cos(x)+i\sin(x)}\right|_{x=2\pi\frac{f}{fs}}$$

`Usage: sinc [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-order int		decimation or interpolation order	(3)
-n int		decimation rate (negative for interpolation)	(2)
-phase int		starting index for zero order decimation	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

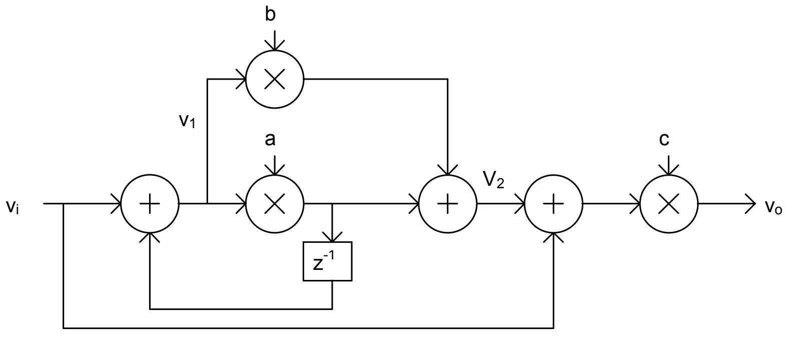
Linear Section (Infinite Impulse Response Filter) (integr)

Simulates a first order section taking a limited number range and different overflow handling into account.

This function simulates a lossy integrator. The limited number range [-min.. -max] can be enforced by saturation. Other realistic numerical behaviour like wrap around or zeroing can be simulated at all three arithmetic circuit nodes.

As a first order section a topology has been chosen which guarantees no loss of numerical precision in the pass-band and sufficient accuracy in the stop-band. This was achieved with an architecture containing a direct data-path between input and output. The z-domain equivalent circuit representing a general 1st order section and its circuit equations are shown in Figure 2. The options -a, -b and -scale of the integr function correspond to the variables *a*, *b* and *c* in the figure.

Choosing *b* = -1 a high-pass filter can be derived. Its transmission function $H_{HP}(z) = \frac{c\,a\,(z-1)}{z-a}$ can be calculated using the relations given in Figure 2. At frequencies which are higher than the cut-off frequency of the high-pass filter (normal pass-band operation) the signal $v_2(z) = v_1(z) \times (a-1) = \frac{z\,(a-1)}{z-a}$ becomes very small and does no longer significantly contribute to the output. The adder adding v_i and v_2 mainly propagates the filter input v_i without loosing any numerical precision.



$$v_o = c\,(v_i + v_2)$$
$$v_1 = v_i + \frac{v_1 a}{z}$$
$$v_2 = v_1(a+b)$$

$$H(z) = \frac{a\,c - (c + a\,c + b\,c)\,z}{a - z}$$

$$H(z) = \frac{v_o}{v_i} = \frac{A_1 + A_0\,z}{B_1 + z}$$

$$a = -B_1 \qquad b = B_1 + \frac{B_1\,A_0}{A_1} - 1 \qquad c = \frac{A_1}{B_1}$$

Figure 2: General First Order IIR Section

With default coefficients the circuit behaves as a first order low-pass with a 3 dB cut-off at 39% of the sampling frequency. The plot below was created by the command line:

```
stim -freq 0 -prn 12 83 -len 4095 -sampl 44100  | integr | dft | db | clip -keep -pro 2048 |
 plot -dx 10.77 -title "Lossy Integrator Gain" -gr -xu [Hz] -yu [dB] -opt "set yr [-3.5:0.5]"
```

It shows how the perfectly flat spectrum of an MLS signal is filtered by the first order low-pass section. From the symmetric spectrum just the first half is passed to the plot procedure.

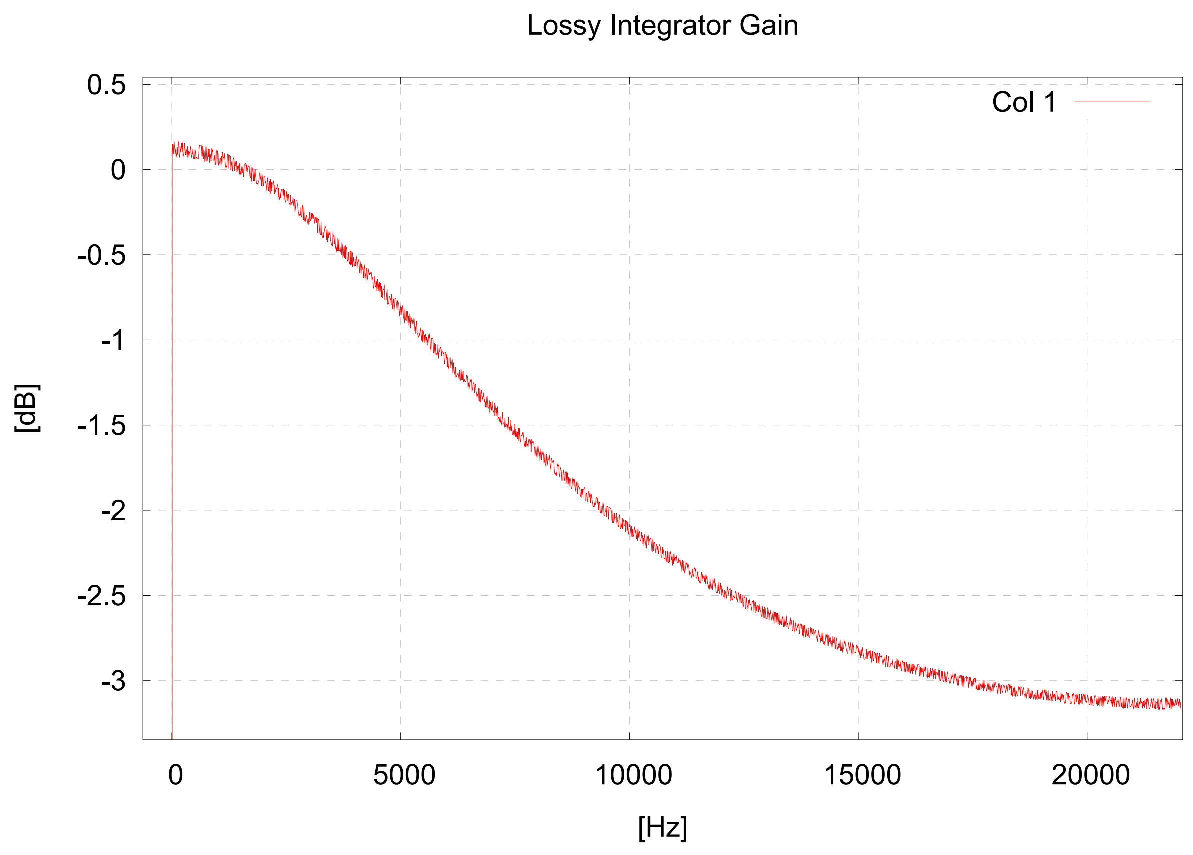


Figure 3: MLS filtered by integr

Usage: integr [options] < infile > outfile

Options:

-isignal ipipe		Read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-scale double		scale factor	(0.399447)
-a double		coeffecicient a	(0.347656)
-b double		coeffecicient b	(0.655273)
-min double		lower limit	(-1)
-max double		upper limit	(1)
-wrap		wrap around output if outside limits	<off>
-sat		saturate output if outside limits	<off>
-zero		bound output to zero if outside limits	<off>
-wrap1		wrap around node 1 if outside limits	<off>
-sat1		saturate node 1 if outside limits	<off>
-zero1		bound node 1 to zero if outside limits	<off>
-wrap2		wrap around node 2 if outside limits	<off>
-sat2		saturate node 2 if outside limits	<off>
-zero2		bound node 2 to zero if outside limits	<off>
-block		indicate saturation by forcing y=x (debug)	<off>
-allnodes		write all internal node to stream	<off>
-debug		generate internal debug information	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Biquadratic Section (Infinite Impulse Response Filter) (biquad)

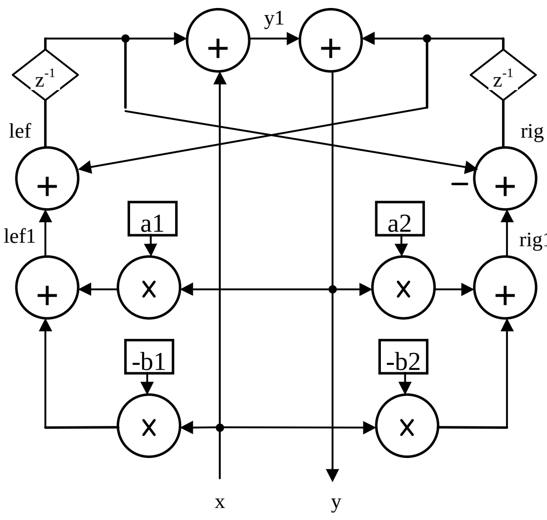
Simulates a second order wave digital filter section taking a limited number range and different overflow handling into account.

The biquad operator performs a digital filtering operation on the input stream. The four filter coefficients a1, a2, b1, b2 and the scale factor are input by options. Scaling is applied to the input signal before it enters the actual filter block which otherwise would not have the gain as a free parameter.

The wave digital biquad which is simulated by this operation has been described by Fettweis[11]. Its z-domain representation together with its network equations are presented in Figure 4.

The solution in the z-domain is a biquadratic term which has one parameter less than the most general biquadratic section. It can be seen that any general transmission function can be realized if the DC-gain is defined externally. The relationship of the four factors a_1, a_2, b_1, b_2 and the general biquad coefficients are described in Table 2.

For frequency domain analysis of z-domain transmission functions the substitution $z \to e^{j2\pi\frac{f}{f_s}}$ can be used to plot magnitude and phase or group delay over frequency.



$$lef_1 = a_1\, y - b_1\, x$$

$$rig_1 = a_2\, y - b_2\, x$$

$$lef = lef_1 + \frac{rig}{z}$$

$$rig = rig_1 - \frac{lef}{z}$$

$$y_1 = x + \frac{lef}{z}$$

$$y = y_1 + \frac{rig}{z}$$

$$y(z) = x(z)\frac{1 + b_1 - b_2 - b_1 z - b_2 z + z^2}{1 + a_1 - a_2 - a_1 z - a_2 z + z^2}$$

Figure 4: Wave Digital Biquad (z-Domain Representation)

[11] **Fettweis;** *Wave Digital Filters: Theory and Practice*; Proc. IEEE, vol.74, pp. 270-327, Feb. '86

$$H(z) = \frac{A_2 + A_1 z + A_0 z^2}{B_2 + B_1 z + z^2}$$

$$a_1 = \frac{B_2 - B_1 - 1}{2} \qquad a_2 = \frac{1 - B_2 - B_1}{2} \qquad b_1 = \frac{A_0 - A_2 - A_1}{2 A_0} \qquad b_1 = \frac{A_2 - A_1 - A_0}{2 A_0}$$

Table 2: Wave Digital Biquad (Transmission Function and Coefficients)

There are no restrictions on the coefficients A_i and B_i that means they can be optimised freely. As long as the transformation rules are applied correctly in order to derive the actual coefficients a_i and b_i this second order section enjoys all the stability properties of wave digital filters, including forced-response stability providing proper scaling and number representation.

The biquad architecture is very efficient in terms of overflow handling because it calculates the output signal from many partial sums which may overflow in two's complement arithmetic as long as the final output of the filter stage does not overflow. The input signal and the output signal of the stage are multiplied with constant coefficients less than one so there will not be any multiplier overflow as long as input signal and output signal do not overflow. If the signal is scaled properly then there will not be any overflow in the output signal and saturation circuits are not required.

In order to analyse the effect of different implementations simulation of overflow conditions is provided by the biquad function. Anyhow, fixed point arithmetic has not been implemented so internal quantization distortion cannot be simulated.

Because of two partial sums *lef* and *rig* which have to be stored in the memory representing the two z^{-1} products the overflow rule for partial sums cannot be applied completely. Three typical overflow handling methods (saturate, wrap-around and set-to-zero) can be applied to the memory terms *lef* and *rig* using the options -satmem, -wrapmem and -zeromen.

Output overflow can be simulated using the options -sat, -wrap and -zero. If only the propagated signal should be effected but not the internal signal fed back to the multiplications by the coefficients a1 and a2 then the options -satout, -wrapout and -zeroout have to be used.

The default filter coefficients yield a low-pass with unity gain and a cut-off frequency close to one quarter of the sampling frequency. The plot below was created by the command line:

```
stim -freq 0 -prn 12 83 -len 4095 -sampl 44100  | biquad | dft | db | clip -keep -pro 2048 |
 plot -dx 10.77 -title "Biquad Gain" -gr -xu [Hz] -yu [dB] -opt "set yr [-30.5:0.5]"
```

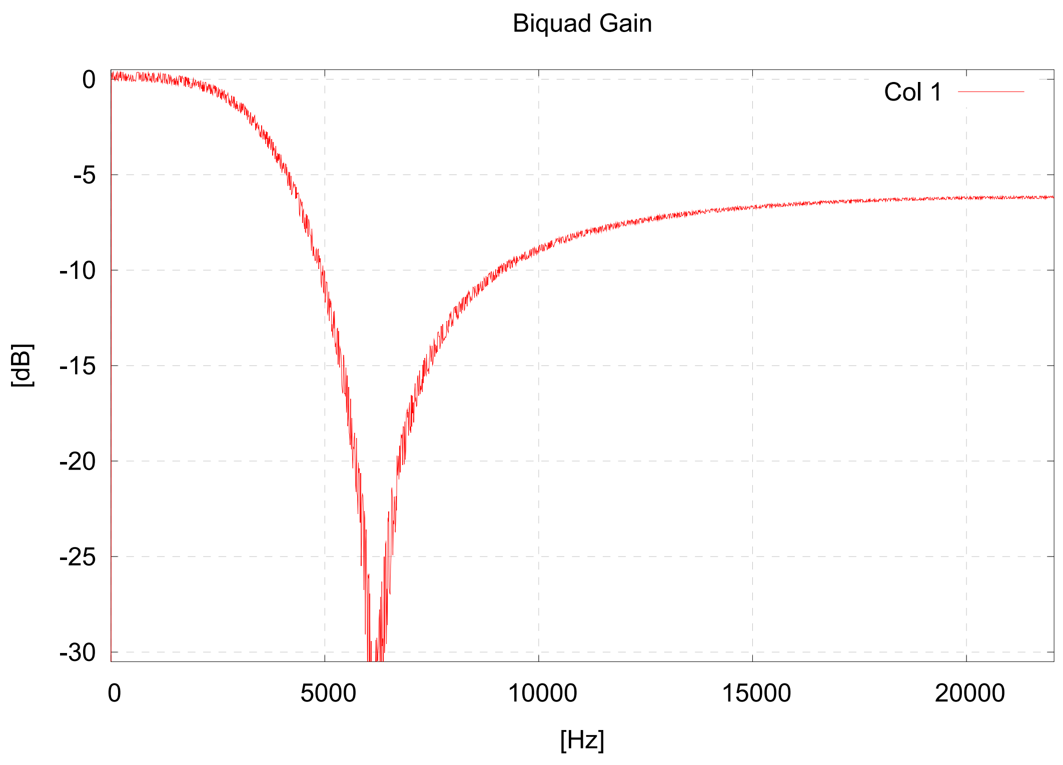


Figure 5: MLS filtered by biquad

```
Usage: biquad [options] < infile > outfile
```

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-scale double		scale factor for x	(0.399447)
-a1 double		coeffecicient a1 for y	(0.347656)
-a2 double		coeffecicient a2 for y	(0.855469)
-b1 double		coeffecicient b1 for x	(0.655273)
-b2 double		coeffecicient b2 for x	(0.631836)
-min double		lower limit of number range	(-1)
-max double		upper limit of number range	(1)
-wrap		wrap around node y using limits	<off>
-sat		saturate node y using limits	<off>
-zero		reset node y to zero while out of limits	<off>
-wrapmem		wrap around memory using limits	<off>
-satmem		saturate memory using limits	<off>
-zeromem		reset memory to zero while out of limits	<off>
-wrapout		wrap around output signal using limits	<off>
-satout		saturate output signal using limits	<off>
-zeroout		reset output signal to zero while out of limits	<off>
-allnodes		write all internal node to stream	<off>
-debug		generate internal debug information	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Finite Impulse Response Filter (fir)

Performs an FIR filtering operation on the input stream. The filter coefficients are either read from an external binary file (if a file name is given) or precomputed in several different ways.

The FIR-filter is applied in a way to calculate an output sample from its corresponding input sample and left and right neighbouring input samples. The number of left and right neighbours plus one must match the number of coefficients. If input samples before the first or beyond the last sample of the buffer are needed then the first or the last input sample is taken instead. The default setup with right = 0 and left = len - 1 creates the standard situation where the output is valid after a delay corresponding to the length of the coefficient vector - 1 while the last output sample is exact.

The Savitzky - Golay[12] convolution smoothes input data by calculating each output sample from its corresponding input sample and specified numbers of left and right neighbour samples. Coefficients for the Savitzky - Golay convolution are precomputed for up to 8 neighbour samples per side, for an order between one and four and for smoothing the curve (deriv = 0) or for calculating a smoothened derivative (deriv = 1).

[12] Numerical Recipes in C, Press/Teukolsky/Vetterling/Flannery, Cambridge University Press, page 650f

`Usage: fir [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-fname char*		file containing FIR coefficients	<off>
-len unsigned	[samples]	FIR filter length	<off>
-left unsigned	[samples]	leftside part of coefficients	<off>
-right unsigned	[samples]	rightside part of coefficients	<off>
-rectang		filter with rectangular window	<off>
-triang		filter with triangular window	<off>
-hamming		filter with hamming window	<off>
-ghamming alpha *alpha double*		filter with generalised hamming window *filter alpha value*	<off>
-hanning		filter with hanning window	<off>
-blackman		filter with blackman window	<off>
-nuttall		filter with nuttall window	<off>
-kaiser attn *attn double*	 [dB]	filter with kaiser window *filter attenuation*	<off>
-chebyshev *ripple double* *trans double*	 *[dB]* *[*fs]*	filter with chebychev window *filter ripple* *normalized filter transition width*	<off>
-savitzky *order int* *deriv int*		local smoothing filter *filter order* *filter derivation*	<off>
-lowpass *cutoff double*	 *[*fs]*	use low pass filter *normalized cut-off frequency*	<off>
-highpass *cutoff double*	 *[*fs]*	use high pass filter *normalized cut-off frequency*	<off>
-bandpass *low double* *high double*	 *[*fs]* *[*fs]*	use band pass filter *lower cut-off frequency* *higher cut-off frequency*	<off>
-bandrej *low double* *high double*	 *[*fs]* *[*fs]*	use band rejection filter *normalized lower cut-off frequency* *normalized higher cut-off frequency*	<off>
-printwin		print window to statistic	<off>
-printcoeff		print coeffs to statistic	<off>
-info		print filter characteristic to	statistic<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Cross Correlation (xcorr)

Cross correlates two data streams or auto-correlates one data stream yielding vector result. With MLS option auto correlation is compared with Dirac impulse yielding a scalar boolean result. An MLS sequence can be created using the stim command with the prnoise option.

The output vector of the correlation function is not scaled in any way. It is simply the sum of all products.

`Usage: xcorr [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-vector char*		binary file containing second vector operand (must have same length)	<off>
-MLS		compare auto correlation of stream with Dirac impulse	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Time Domain Analysis (tana)

Calculates signal characteristics and includes a selected subset in the output stream. The output stream can be either binary or textual and can be directed either to a file or to the standard output device. The order of the selected characteristics in the output stream does not depend on the order of the option statements. They always appear in the order shown below.

`Usage: tana [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-var		output variance of signal	<off>
-rms		output root mean square of signal	<off>
-dc		output mean value of signal	<off>
-min		output minimum of signal	<off>
-max		output maximum of signal	<off>
-peak		output maximum of abs (signal)	<off>
-imin		output position of first min	<off>
-imax		output position of first max	<off>
-ipeak		output position of first peak	<off>
-size		output number of sample	<off>
-len int		number of samples to process	<off>
-nl		append new line character	<off>
-db		output all voltages in db	<off>
-dboffs double	[dB]	zero dB value	(0 dB)
-negpos		output positions near the end relative to end	<off>
-fname char*		name of output file	(stdout)
-binary		create binary output	<off>
-term		don't propagate input data	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Frequency Domain Processing

Frequency Domain Analysis (fana)

Calculates a spectrum and includes a selected subset of frequency domain signal characteristics in the output stream. The output stream can be either binary or textual and can be directed either to a file or to the standard output device. The order of the selected characteristics in the output stream does not depend of the order of the option statements. They appear in the order shown below.

The selected frequency domain analysis results can be sent to the standard output pipe in binary format using the option -binary. In this case the -term is enforced which means that input data are not propagated from input to output.

If -binary is not included then textual output is directed to the file which must be specified using the -fname option. In that case it is possible to terminate the input stream or to propagate it, depending on the presence of the -term option.

The options -len, -freq, -lev, -adjust and -sample should match the corresponding options given to the stimulus generator which has generated the input stream. The fana command calculates the RMS power of three differently filtered versions of the input stream.

First, the whole AC RMS energy contained in the signal band limited by -noisehp and -noisebw is calculated (*rms*). Second, the AC RMS energy contained in the above signal band is determined after a perfect band-pass filter with a width of -stimbw around the stimulus frequency -freq has been applied (*sel*). Third, the AC RMS energy contained in the signal band is determined after a perfect band-reject filter with a width of -stimbw around the stimulus frequency -freq has been applied to eliminate the stimulus signal (*noi*). Using the option -weight one of two noise weighting filters can be turned on in that case.

From these three results the obtainable output measures are derived according to the table below:

option	*description*	*derivation*	*definition*
-rmswb	Deviation of wide band stimulus level	dB(*rms*) minus -dboffs minus -lev;	ITU O.132, bw=300..3400Hz
-rmssel	Deviation of selective stimulus level	dB (*sel*) minus -dboffs minus -lev;	ITU O.132, stim=1020Hz, bw=100Hz
-rmsnoi	Absolute level of weighted noise and quantisation products	dB (*noi*) minus -dboffs;	ITU O.132, bw=300..3400Hz minus stim
-distwb	Total distortion (wide band stimulus)	dB (*rms* / *noi*) minus cal;	ITU O.132, cal=10 log(3100Hz/3000Hz)
-distsel	Total distortion (selective stimulus)	dB (*sel* / *noi*) minus cal;	ITU O.133, cal=10 log(3100Hz/3000Hz)
-pwrwb	Wide band signal level	dB (*rms*) minus -dboffs;	ITU O.132, bw=300..3400Hz

An application of the fana command to analyze transmission characteristics of telecommunication circuits is described as one of the examples given in the Appendix B.

Usage: fana [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-len int		useful buffer length	(1024)
-freq double	[Hz]	stimulus frequency	(1020)
-lev double	[dBm0]	stimulus level	<off>
-adjust int		adjust frequency for 0=any, 1=int, 2=odd number of periods	(2)
-sample double	[Hz]	sampling rate	(8000)
-decim int		decimate before FFT (use 3rd order Comb)	(1)
-noisehp double	[Hz]	noise band lower bound	(0)
-noisebw double	[Hz]	noise band width	(4000)
-stimbw double	[Hz]	selectivity band width	<off>
-weight char*		spectral weights (psopho or Cmess)	(none)
-dboffs double	[dB]	zero dBm0 offset (absolute gain)	(0)
-rmswb		rms wideband (deviation from lev)	<off>
-rmssel		rms selective (deviation from lev)	<off>
-rmsnoi		rms noise	<off>
-distwb		distortion wideband	<off>
-distsel		distortion selective	<off>
-pwrwb		rms wideband (absolute)	<off>
-nl		append new line character	<off>
-fname opipe		name of output file	(stdout)
-binary		create binary output	<off>
-term		don't propagate input data	<off>
-debug		generate internal debug information	<off>
-trace char*		sub process output to trace file	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Sweep Analysis (swana)

Performs spectral analysis in a window which is shifted along the input stream. Creates a multi-column output stream containing base frequency and signal levels of all selected harmonic components for each window being processed. Many different strategies and algorithms are available in order to find the correct fundamental frequencies even if they are missing in the recorded sound.

The sweep analysis tool is applied to a time domain signal. Often the recorded response of a system to a sweep excitation signal is used as an input. Another important application is the analysis of sound recordings e.g. of musical instruments.

The main purpose of the program is to calculate the time varying signal power at certain frequencies of interest. A window of specified length (windowsize) moves stepwise (step) over the input stream. If the last windows are not complete then they are optionally (decay) padded with zeros until they are completely empty. For each window a spectral analysis is done and the requested output values are generated.

One line of output contains the fundamental frequency, the levels of a specified number (harm) of harmonics, the optional (inharmonics) cent-deviations of those harmonics, the optional centroid of all seleceted harmonics (centroid), the optional centroid of the complete spectrum (totcentroid) and the optional noise level (noise) which is the unweighted or weighted (psopho, Cmessage) RMS of the total spectrum with all or a selected number (partials) of harmonics zeroed out.

The fundamental frequencies for all window buffers can be read from an external binary file (fname), specified as a known sweep (sweep, start, rate) or they can be calculated using several methods containing different heuristics. Some options have a range parameter which limits their effect to a certain number of steps counted either from the beginning (positive range) or from the end of the signal (negative range). This allows to have three different strategies in three different sections of the signal (for example low register, middle register and high register of a musical scale).

The fundamental frequency detection tries to find a harmonic grid containing up to a user specified number of partials in the spectral buffer. Two different methods can be selected.

The grid method allows some harmonic frequency tolerance (tol) and takes only true peaks bigger than a specified threshold (floor) - defined relatively with respect to the spectral buffer's maximum value - into account. The peak amplitudes are completely neglected as long as peaks are bigger than the floor of a buffer, just their positions are of interest. It is not necessary that a fundamental actually exists. As long as there is a sufficient number of equidistant peaks a virtual fundamental frequency is defined by their distance. The score of a harmonic grid is determined from the number of peaks which are sufficiently close to a harmonic grid regardless of their level. It is the weighted sum of the harmonic indices of existing partials up to the limit specified with the partials option. The resulting fundamental frequency is the weighted geometric mean of the correct integer fractions of all partials taken into account. The weighting function is defined by the user specified coefficient w (weight) which is the exponent decaying harmonics with their index n: $\text{weight}_n = (1/n)^w$. With the default value w=0.0 no specific weight is applied.

The default method scores with weighted peak magnitudes. Here the virtual_f0 option determines if a true peak at the f0 frequency is a requirement for a valid fundamental frequency. The weighting function is the same as above and it is again specified by the partials option which also determines how many harmonics participate in the calculation of a score. The fundamental frequency with the best score is optionally interpolated parabolically (interpolate) if it is not virtual. Goertzel interpolation is not implemented yet.

If the fundamental frequency is known to change smoothly, like in sweep responses or musical scales, it is advisable to specify the continuos option with a first guess for the initial frequency (f0) and upper and lower limits (llim, rlim) for the frequency differences in [%] between adjacent spectral buffers. This specification puts constraints on the search range for fundamental frequencies. These ranges can be overwritten manually using the hint option. Hint frequencies and new tolerance limits can be introduced at any signal position.

`Usage: swana [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-windowsize uint	[samples]	analysis window size	(1024 samples)
-step uint	[samples]	analysis window step	(256 samples)
-decay		zero pad last incomplete windows	<off>
-sample double	[Hz]	sampling rate	(8000 Hz)
-interpolate int		interpolate between spectral lines (0 = parabolic, 1 = goertzel)	(0)
-operation int		infinitesimal operation on result (-1 = integ, +1 = diff)	(0)
-harmonic uint		include up to nth harmonic	(1)
-inharmonics *range uint* *floor double*	 *[Cents]* *[dB]*	include in-harmonic deviations of harmonics *maximum deviation* *ignore harmonics less than floor*	<off> *(150 Cents)* *(-40 dB)*
-centroid		append spectral centroid to each row *(of selected harmonics only)*	<off>
-totcentroid		append spectral centroid to each row *(of total spectrum)*	<off>
-noise		append non-harmonics to each row	<off>
-fname *char** *range int*		read frequencies from external file *filename and points to get with this method* *(neg counts from end)*	<off> *(0)*

-sweep		analyse synthetic sweep response	<off>
start double	*[Hz]*	*start frequency*	*(50 Hz)*
rate double	*[Dec/sec]*	*sweep rate*	
range int		*points to get with this method*	*(0)*
		(neg counts from end)	
-grid		use harmonic grid strategy	<off>
tol double	*[%]*	*harmonic tolerance*	*(5 %)*
floor double	*[dB]*	*ignore peaks smaller than floor*	*(-40 dB)*
range int		*points to get with this method*	*(0)*
		(neg counts from end)	
-hint double double		hints for fundamental detection	<off>
{ double double }		*alternate time [sec] and frequency [+ Hz]*	
		or time [sec] and range [- %]	
		(0 means full range)	
-partials		search for partials	<off>
n uint		*number of partials to search for*	*(1)*
		(incl. fundamental)	
weight double		*weighting function (1/n)^w*	*(0.0)*
-virtual_f0		detect virtual fundamentals	<off>
		(no actual peak in spectrum)	
-continuous		assume continuous stimulus frequencies	<off>
llim double	*[%]*	*left limit relative to previous f0*	*(-10 %)*
rlim double	*[%]*	*right limit relative to previous f0*	*(80 %)*
f0 double	*[Hz]*	*starting frequency*	*(0 Hz)*
-triang		filter with triangular window	<off>
-hamming		filter with hamming window	<off>
-ghamming		filter with generalised hamming window	<off>
alpha double		*filter alpha value*	
-hanning		filter with hanning window	<off>
-blackman		filter with blackman window	<off>
-nuttall		filter with nuttall window	<off>
-kaiser		filter with kaiser window	<off>
attn double	*[dB]*	*filter attenuation*	
-chebyshev		filter with chebychev window	<off>
ripple double	*[dB]*	*filter ripple*	
trans double	*[*fs]*	*filter transition width*	
-psopho		use psophometric noise weighting	<off>
-Cmessage		use C-message noise weighting	<off>
-printwin		print window to statistic	<off>
-stat char*		write program statistics to file	<off>
-debug		output debug information	<off>
verbose int		*amount of debug information*	*(1)*
from int		*trace range left limit*	*(1)*
to int		*trace range right limit*	*(0)*
*fname char**		*write spectral buffers to binary files*	*("spec")*
-? | -help		generates this help screen	<act>

Harmonic Analysis (dftpeaks)

Performs harmonic analysis in a window which is shifted along the input stream. Creates a multi-column output stream containing base frequency and signal levels of all selected harmonic components for each window being processed. Many different strategies and algorithms are available in order to find the correct fundamental frequencies even if they are missing in the recorded sound.

The harmonic analysis tool is applied to a time domain signal. It complements the swana functionality because there is no segmentation required and input sound does not need to sweep or step regularly.

The program tries to track sine-wave components which are specified using the -freq option even if their frequencies slowly change. Selected components do not need to be harmonics. They are related to the fundamental but do not need to be integer multiples of the base frequency.

The program outputs frequencies and magnitudes. Output data therefore contain two columns for each sound component which is part of the -freq list. If the -rms option is given then an additional column of data points is included in the output stream.

The input stream is analyzed using two moving windows. A long window giving the frequency resolution required to track component frequencies and a shorter window giving the time resolution to track magnitude variations. These windows are specified using the options -llen, -loffs, -slen, -soffs. If the stepping offset of a window is shorter than its length then subsequent windows will overlap. In order to window the analysis buffers before Fourier transform different window types can be selected (-kaiser, -nuttall, -hamming, -hanning, -blackman…).

The fundamental recognition works better when the search range is limited by specifying the options -fu and -fo. These limits specify a strict range for all fundamentals, not only for the initial one. The options -feps defines the tolerable frequency jitter of the harmonic grid. The option -frel specifies the fastest rate of change in fundamental frequency which has still to be tracked properly. If the detected fundamental frequency in an observation interval deviates more than the given percentage values from the previous one then the detected fundamental frequency is dropped and the one from the previous interval is used instead. The same is true if the running rms signal level drops below the threshold level specified using the -threshold option.

An example of how to use this program in a dynamic harmonic analysis is given in Appendix B.

```
Usage: dftpeaks [options] < infile > outfile
```

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-samples int	[Hz]	number of samples per second	(44100)
-llen int	[samples]	large window length	(32768)
-slen int	[samples]	small window length	(4096)
-lofs int	[samples]	large window move step size	(32768)
-sofs int	[samples]	small window move step size	(4096)
-fu double	[Hz]	lower frequency bound for base tone estimation	(100)
-fo double	[Hz]	higher frequency bound for base tone estimation	(2000)
-freq double { double }	 [Hz]	frequency grid where peaks are searched (have to be sorted by magnitude) list of frequency rates	<off>
-frel lower upper double lower double upper	 [%] [%]	maximum deviation factor of expected base tone's frequency (relative to last base tone) lower deviation factor upper deviation factor	<off> (0) (0)
-threshold double		minimum rms level for fundamental tone detection	(0)
-feps double	[%]	jitter of harmonic frequencies	(1)
-nearest		use only nearest peaks in jitter	<off>
-triang		use triangular window	<off>
-blackman		use blackman window	<off>
-nuttall		use nuttall window	<off>
-hanning		use hanning window	<off>
-hamming		use hamming window	<off>
-genhamming double alpha		use generalized hamming window alpha	<off> (0.54)
-kaiser attn double attn	 [dB]	use kaiser window desired attenuation	<off> (40.0)
-rms		append rms column to output	<off>
-debug		int debug level	(0)
-stat char*		write program statistics to file	<off>
-? |-help		generates this help screen	<act>

Time or Frequency Domain Processing

DFT Processor (dft)

Performs Discrete Fourier Transform or its inverse, taking real as well as complex values from the input stream. Real as well as complex output data are generated. Selectable co-ordinate systems for complex signals are the Cartesian (real/imaginary) and the Polar (magnitude/argument) co-ordinate system.

The DFT result is scaled in a way to equalize the root mean square of the time domain buffer and the frequency domain buffer. The number of elements in the frequency domain is the same as in the time domain. The first output sample is the DC-component, the second one (which has the same value as the last one) corresponds to the sampling frequency divided by the number of points. The sampling frequency itself as mirror image of the DC-value is not contained in the output spectrum.

If 2-dimensional (complex) output is turned on (-out2D), then the output stream consists of two columns. These columns normally contain magnitude and phase pairs. Cartesian representation (real part, imaginary part) can also be selected (-reim) and the order of the data pairs can be reversed (-swap) for the output. Note, that the co-ordinate system is valid for both input and output, while order swapping only concerns the output. The output order is especially important when 1-dimensional output (the default case) is selected, because then every second value is dropped. The default phase unit is radiant but degree units (-deg) are possible, too.

If a length is specified which is greater than the actual length of the input stream then the input stream is zero padded. If the specified length is shorter than the actual length of the input stream then a long term average spectrum (LTAS) is calculated. If the actual stream length is not an integer multiple of the given length then the last buffer will be zero padded. In LTAS mode no phase information is available.

The DFT buffer length should preferably but not necessarily be a power of two. Complex (two dimensional) input files (-inp2D) must have even length.

The option -smooth post-processes phase results in a way to avoid discontinuities. Near 360° (2×Pi) jumps are eliminated by extending the number range beyond the usual phase interval [-180..180]. Phase results can be trimmed by specifying a minimum magnitude level where non-zero phases are returned.

`Usage: dft [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-len int	[samples]	overrides DFT length	<off>
-reverse		invoke reverse fourier transformation	<off>
-inp2D		input data are complex	<off>
-out2D		request complex output	<off>
-reim		cartessian values instead of polar form	<off>
-swap		swap complex numbers before output	<off>
-min double	[dB]	minimum magnitude with nonzero phase	<-200 dB>
-smooth		reconstruct phase outside base circle	<off>
-deg		phase in degrees	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Windowing (window)

Applies a window function to an input stream. Typically used for time domain windowing or frequency domain filtering. Windows can also be applied periodically. Third octave band and octave band analysis applied to a spectrum returns a two column table with centre frequencies and sound intensities being the RMS of the related spectral bands. All other windows are being multiplied with the input stream and therefore do not alter the total stream length.

The windowing operation calculates a buffer usually containing as many weights as there are input samples and then applies these weighting factors to the input data. If a shorter window is explicitly specified using the -len option then this window is periodically applied until the end of the input stream is reached.

Applying a Nuttall or Kaiser window to a buffer with a sampled signal is required to eliminate spectral analysis errors caused by non-periodic signal components. Periodic windowing and periodic DFT analysis can be combined to obtain long term average spectra (LTAS) of audio data.

Psophometric or C-message weighting[13] is often applied to noise and distortion spectra in order to reflect the non-linear sensitivity characteristic of the human ear.

All general filter types (low-pass, high-pass, band-pass, band-reject) are available in an ideal (order = 0) and a real passive (order > 0) implementation. The default versions of these filters have been selected according to ITU O.132 recommendations. The inverse Sinc filter compensates the spectral effect of a decimation in the time domain using the Sinc operator. It is allowed to specify any meaningful combination of filters and weights in the same call.

[13] Requires buffer sampling rate

If the mirror option[14] is set then the filter rejects not only the selected band but also the mirror image of the selected band. The spectral weighting functions and the filters require specification of the buffer sampling rate.

To speed up execution time it sometimes can be desirable to save the calculated window in a binary file which is restored later on instead of recalculated. It is also possible to scale all output samples using a constant scale factor.

`Usage: window [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-nuttall		nuttall window	<off>
-kaiser double		kaiser window with given beta	(10)
-psopho		psophometric weights	<off>
-Cmess		C-message weights	<off>
-thirdoctavebands *res double*	 *[Hz]*	ISO third octave bands (12.5Hz-20kHz center frequency) *spectral resolution*	<off> *(1.0 Hz)*
-octavebands *res double*	 *[Hz]*	ISO octave bands (16Hz-16kHz center frequency) *spectral resolution*	<off> *(1.0 Hz)*
-db		output sound levels of band analysis in dB	<off>
-bp		standard bandpass (860 ... 1180 Hz, ideal)	<off>
-bandpass *fc double* *bw double* *order int*	 *[Hz]* *[Hz]*	bandpass *center frequency* *bandwidth* *filter order*	<off> *(1020.0 Hz)* *(320.0 Hz)* *(0)*
-br		standard bandreject (860 ... 1180 Hz, ideal)	<off>
-bandrej *fc double* *bw double* *order int*	 *[Hz]* *[Hz]*	bandreject *center frequency* *bandwidth* *filter order*	<off> *(1020.0 Hz)* *(320.0 Hz)* *(0)*
-lp		standard lowpass (3400Hz, ideal)	<off>
-lowpass *fc double* *order int*	 *[Hz]*	lowpass *cutoff frequency* *filter order*	<off> *(3400.0 Hz)* *(0)*
-hp		standard highpass (300Hz, ideal)	<off>
-highpass *fc double* *order int*	 *[Hz]*	highpass *cutoff frequency* *filter order*	<off> *(300.0 Hz)* *(0)*
-invsinc *order int* *rate int*		inverse characteristic of sinc *order of sinc function to compensate* *decimation rate*	<off>
-mirror		mirror window at window half length	<off>
-sample double	[Hz]	sampling rate	(8000 Hz)
-save char*		save window to file	<off>
-len int		apply window of this length periodically until end of stream	<off>
-restore char*		restore window from file	<off>
-originate		no input stream, only generate window	<off>
-scale double		constant scale factor	(1)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

[14] First value (DC) is not mirrored

Operations on Single Input Vector

Clip (clip)

Drops or keeps a specified number of leading and trailing signal samples when the input stream is copied to the output stream.

The main functionality of this operation is to clip a leading part, the prologue, or a trailing part, the epilogue, of a signal. The length of the prologue and epilogue to be clipped are specified using the -prolog and -epilog options. If the -len option us used to specify the length of the central part then one of the two other lengths is redundant and can be calculated from the actual length of the input stream.

The -keep option reverses the meaning and keeps rather than cuts off prologue and epilogue and drops the central part instead. The -fill option does not clip at all but instead it fills the specified signal parts with a constant value.

The -ac option has nothing to do with clipping the stream. It rather removes a DC part of a signal. The -insert option inserts a constant signal part right after the prolog which has a length as specified by the -len option.

The -fname option allows to specify a mask file which must match in length. For each nonzero element of the mask file the corresponding value of the input stream is copied to the output. With the -like option stream and file lengths will be matched. Either by truncation or by zero padding.

`Usage: clip [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-prolog int	[samples]	length of signal prologue	<off>
-epilog int	[samples]	length of signal epilogue	<off>
-length int	[samples]	length of signal center part	<off>
-keep		keep pro-,epilogue; drop,fill center	<off>
-insert double		insert constant signal section	(0)
-fill double		fill signal section with constant	(0)
-like char*		make length like other file	<off>
-ac		remove DC-component of result	<off>
-fname char*		template for picking up values	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Up- or Downsample by Integer Factor (resample)

Converts the sampling rate by picking out every nth sample from the input stream or by inserting a number of zeroes between every two input samples.

`Usage: resample [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-inc int		increase sampling rate by this factor	<off>
-red int		reduce sampling rate by this factor	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Difference (dif)

Calculates the differences between every two consecutive input samples (discrete derivative) or between samples read from the primary and a secondary input stream (vector difference) or between input samples and a constant (offset). The secondary input stream is usually connected to an external binary file.

`Usage: dif [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-fname char*		file to read second operands from	<off>
-offs double		offset	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Integrator (int)

Accumulates input samples and propagate intermediate accumulation result to output stream (discrete integration).

`Usage: int [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Moving Mean (movemean)

Calculates a moving mean over the input signal. Length of output stream is always equal to the length of the input stream. The last sample in the output stream is the mean of the last n samples of the input stream, which has been preceeded by n-1 zero samples.

`Usage: movemean [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-n int		length of moving mean	(1)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

dB (db)

The dB calculation operator uses the formula out = offs + a log10 (inp) - a log10 (inp[ref]). The parameter a is 20.0 or 10.0 (with -pow). If (inp <= zero) or (out < min) then out = min.

`Usage: db [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-ref int	[samples]	number of reference sample (0 dB value)	<off>
-offs double	[dB]	offset	(0 dB)
-min double	[dB]	minimum result	(-200 dB)
-pow		signal is power	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Inverse dB (invdb)

The Inverse dB calculation operator uses the formula out = 10^ ((in - offs) / a). The parameter a is 20.0 or 10.0 (with -pow).

`Usage: invdb [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-offs double	[dB]	Offset	<off>
-pow		signal is power	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Degree (deg)

The deg operator converts phase in radiant into phase in degree. Optionally it tries to reconstruct original phase values outside the base range ±PI. It is also possible to specify a reference phase which will be added to the output.

Usage: deg [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-ref int	[samples]	number of reference sample (0 deg value)	<off>
-offs double	[deg]	offset	<off>
-smooth		reconstruct phase outside base circle	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

General Function (func)

Applies a scalar function to all samples of the input data stream.

Usage: func [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-plus double		add value to samples	<off>
-minus double		subtract value from samples	<off>
-times double		multiply samples with value	<off>
-divide double		divide samples by value	<off>
-ratio num den *num double* *den double*		multiply samples with ratio *numerator* *denominator*	<off>
-abs		take absolute value of samples	<off>
-rct		rectify signal (zero negative samples)	<off>
-rcp		apply reciprocal (1/x) on samples	<off>
-sqrt		square root of samples	<off>
-sqr		square samples	<off>
-ln		take natural logarithm of samples	<off>
-log		take logarithm of samples	<off>
-exp		calculate e^sample	<off>
-exp10		calculate 10^sample	<off>
-pow double		calculate sample^value	<off>
-sin		calculate sine of samples	<off>
-cos		calculate cosine of samples	<off>
-tan		calculate tangens of samples	<off>
-att double		calculate sample * 10^(value/20)	<off>
-trunc int		keep digits after decimal point (negative allowed)	(0)
-sign		calculate sign of values	<off>
-lowlimit double		clip values lower than min	<off>
-highlimit double		clip values higher than max	<off>
-greater double		result greater than value	<off>
-less double		result less than value	<off>
-equal double		result equal to value	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Table Reshape (table)

Reads a single or multi column table from the primary input stream and creates another single or multi column table as output stream. Another suitable single or multi-column table may be present at the secondary input stream. Any combination of input columns of any of both source tables can be selected in any order to be included in the output table. Matrix transpose can also be performed. All related columns must have equal length.

Using the -remove option the indicated primary input columns will be deleted from the stream while using the -extract option all columns except the indicated ones will be deleted.

A table from the secondary input can be merged into the output stream by specifying columns using the -select option. A column position may be specified to indicate the insertion point. Column position 0 designates insertion in front of the first primary input stream column, which is the default. The column width of the primary input table must be specified using the -signals option, the column width of the optional secondary table is specified using the -width option. The secondary input stream can also be prepended or appended to the primary input stream using the -merge option.

The -xcol option works in conjunction with -extract and -select. It specifies one of the columns of the input stream (-xcol <= -signals) or input file (-xcol > -signals) to be included as first column of the output stream. If this column appears also in the -extract or -select list it is ignored there. The -reverse option operates only on the primary input stream. Any secondary input is ignored. It makes the last row the first one keeping the column order.

The -transpose option is another exclusive matrix operation. A standard matrix transposition is performed by exchanging rows and columns. Note that the -signals parameter must be updated in subsequent table operations. If actual matrix elements are groups of values, f.e. complex number pairs, then group size must be given as parameter to the transpose command. In that case, -signals should be set to the number of groups per row rather than values per row. The same is true for the -shift operation which moves all subgroups of size grp down within their column by a certain number of positions. Subgroups which are shifted out of the table are lost (if rot=0) or prepended to their column (rot=1). A global shift distance can be entered using the -position option, if all columns are to be shifted differently an external file with exactly one number per column has to be specified using -fname.

The combination of the options -sort -resample -border is used to create a linearly rising and equally spaced x-axes starting at the -border value. This is done by first sorting the rows according to the first column, applying a heuristic algorithm to find a proper x-axis grid and to calculate the other column’s center of distribution around these grid points. The -sort option must not receive any column parameter in that case.

Usage: table [options] < infile > outfile

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-signals int		number of columns in input stream	(1)
-remove int *{ int }*		remove these columns *more than one column may be given*	<off>
-extract int *{ int }*		extract these columns from input stream *more than one column may be given*	<off>
-transpose int		transpose rows and columns of subgroups *(parameter is subgroup length)*	(1)
-reverse		reverse rows (make last one first)	<off>
-merge int		merge stream and file (0..append, 1..prepend)	(0)
-fname char*		file to read second operands from	<off>
-width int		number of columns in input file	(1)
-select int *{ int }*		select these columns from input file *more than one column may be given*	<off>
-position int		position to insert selected columns	(0)
-xcol int		extract or select x-axes column (move in front)	<off>
-sort int *{ int }*		sort rows over these columns *more than one column may be given* *(negative numbers to sort down)*	<off> (1)
-resample		with sorting, makes first column linearily rising	<off>
-shift *grp int* *rot int*		shift columns down vertically[15] *subgroup length* *(0..shift out of table, 1..rotate, shift cyclically)*	<off> (1) (0)
-border double		with resampling, defines start point for resampling	(0)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

[15] requires -position (shift all columns down a constant number of samples) or -fname (with exactly one value per column)

Operations requiring Multiple Input Vectors

Pick Values according to Boolean Template (pick)

Pick out values from or insert values into the primary input stream according to masking information read from the secondary input stream.

Picks out and propagates to the output stream only those samples of the primary input stream whose corresponding boolean control flag, read from the secondary input stream is true (nonzero). The other input samples are simply skipped (omitted) and not copied to the output stream.

The -space option enforces separation between picked groups by inserting separation blocks as specified by length and padding value.

The -fill option instead fills up picked groups to make them equally long by padding them with the padding value.

If an -insert value is specified a different mode of operation is entered. Instead of skipping input samples when their corresponding mask value is zero, the given constant value is inserted into the output stream for each zero mask sample. Reading the input stream is on hold until another nonzero mask value is read. The insert mode is exclusive and does not support the space and fill features.

Periodic templates are supported in all modes using the -wrap option. In that case the control stream can be read multiple times as long as there are still input samples to process.

`Usage: pick [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-fname char*		template for picking out values	<off>
-space length value *length int* *value double*		insert space between picked groups *length of spacing* *padding value*	<off> *(0.0)*
-fill length value *length int* *value double*		fill picked groups for equal length *group length* *padding value*	<off> *(0.0)*
-wrap		wrap mask file around	<off>
-insert double		insert fill value where template is zero(0)	
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Resegment Binary Stream (reseg)

Creating a resegmentation mask for the pick command based on a signal's running RMS, peak value or slope and a given threshold. Spike and gap elimination, length trimming and segment augmentation. Can also be used to calculate the running signal parameters and to downsample.

The reseg function performs several operations related to resegmentation of audio files. Exactly one of the operation modes -rms, -peak, -slope, -compare, -despike, -merge, -augment and -trim must be selected.

The -rms, -peak, -slope and -compare modes operate directly on analog audio signal streams, while the other modes take boolean masking streams as their input which might have been created using the -compare mode before.

In -rms, -peak or -slope mode running RMS, peak or slope values are calculated over the input stream using the window length and stepping distance given by the options -windowsize and -step. The length of the output stream matches that of the input stream unless the -downsample option has been given. In that case only one output sample is created for each RMS value which has been calculated. The down-sampling ratio is therefore determined by the stepping distance. Without -downsample the calculated RMS and peak values are simply repeated in order to match the stream sizes. Slopes are calculated by linear regression. They are linearly interpolated between the breakpoints given by the stepping distance.

In -compare mode the running RMS, peak or slope values are compared with the given threshold. Instead of the calculated values the Boolean results of these comparisons are sent to the output stream. The -downsample option has exactly the same meaning as in -rms, -peak or -slope mode. Different on-threshold and off-threshold values can be specified.

Boolean mask streams created in the -compare mode can be post-processed by the remaining operation modes. The -despike mode can eliminate short spikes while the -merge mode does the same with short gaps. Spikes and gaps which are shorter than the specified minimum length are eliminated, that means, the Boolean masking values inside of those short regions are toggled in order to match their environment.

Other ways to post-process Boolean masks are the -augment and -trim operating modes. Augmentation shifts leading and trailing edges of selection regions by a specified number of samples in either direction.

Trimming tries to cut the selection regions to equal lengths. Handling of overlapping selection regions depends on the trimming mode. With “retrigger disabled”, a selection region starting when another one is already active will be ignored. With “retrigger prolongs”, both regions will be merged and longer regions might be created. With “retrigger aborts”, the first one will be truncated leaving a region behind which is shorter than specified. The later one will be trimmed correctly and it will be separated from the aborted one by one zero sample. Aborted regions can be removed like spikes, prolonged regions can be re-trimmed in a second pass. Processing a sound file containing double attacks, in the first two cases the start of the segmentation window will be placed in front of the first attack, while in the third case it will be placed in front of the second attack.

The created and post-processed mask stream is usually used as secondary input stream of a pick command which actually finishes the segmentation process.

`Usage: reseg [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-windowsize uint	[samples]	analysis window size	(1024 samples)
-step uint	[samples]	analysis window step	(256 samples)
-decay		zero pad last incomplete windows	<off>
-rms		calculate running RMS	<off>
-peaks		calculate local peak value	<off>
-slope		calculate local slope *(by linear regression)*	<off>
-downsample		output only one value per window	<off>
-compare *on_threshold double* *off_threshold double* *mode int*		compare signal with threshold levels *Signal level required to turn on the gate* *Signal level required to turn off the gate* *Signal property compared to threshold:* *0..running RMS* *1..peak value* *2..signed comparison* *3..slope*	<off> *(0.5)* *(0.5)* *(0)*
-despike *minlen int*	*[Samples]*	remove short spikes *minimum segment length*	<off> *(1024 Samples)*
-merge *minlen int*	*[Samples]*	remove short gaps *minimum pause length*	<off> *(1024 Samples)*
-augment *prolog int* *epilog int*	*[Samples]* *[Samples]*	augment by adding prolog and epilog *shift leading edges left(+), right(-)* *shift trailing edges right(+), left(-)*	<off> *(100 Samples)* *(100 Samples)*
-trim *fixedlen int* *mode int*	*[Samples]*	trim segment to fixed length *segment length* *0..retrigger disabled* *1..retrigger longens active phase* *2..retrigger aborts active phase*	<off> *(1024 Samples)* *(0)*
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

General Vector Function (vecfunc)

Applies a real or complex vector operation to the signals read from the primary and secondary in put stream. Samples flagged as invalid by another Boolean stream are interpolated.

The operations implemented on vector operands are -plus, -minus, -times and -divide. One of those options must be selected. They can be applied to vectors of real values and to complex vectors.

The complex mode is enabled by the option -complex. In this case both vectors must have exactly the same format. The value pairs can either contain real and imaginary parts (option -reim) or magnitude and argument. Arguments can either be degrees (option -degree) or radiants. The default format for complex numbers is magnitude and argument in radiants.

If the masking option is used a filename must be specified using the -valid option. The masking file must contain one Boolean flag value for each pair of operands. If the masking value corresponding to an operand pair is false (zero) then the arithmetic operation is not performed and the previous result is repeated instead.

```
Usage: vecfunc [options] < infile > outfile
```

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-plus		add vector to stream	<off>
-minus		subtract vector from stream	<off>
-times		multiply stream with vector	<off>
-divide		divide stream by vector	<off>
-vector char*		binary file containing second vector operand, must have same length or shorter (if vector is periodic)	<off>
-valid char*		binary file containing boolean vector identifying valid samples	<off>
-complex		perform operation on complex data	<off>
-reim		interpretation of inputs as (real,imag) not (mag,arg)	<off>
-degree		interpretation of argument is degree	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Vector Comparison (same)

Compares data read from primary and secondary input stream for identity.

Comparison can detect small phase shifts between operands and counts the number of matching samples. The option -full makes comparison more strict, the option -tolerance can exclude a prologue and epilogue of given length from the comparison.

The maximum phase shift between the two signals which is recognized properly is 2 samples ahead or behind in any direction. Matching results are output in textual format to the standard output device. No output is generated if perfect matching is detected.

`Usage: same [options] < infile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-fname char*		2nd file for compare	<off>
-tolerance int		number of samples to ignore at head and tail	<off>
-title char*		string to identify output	()
-full		strict comparison over full length	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Merge Multiple Streams (merge)

Merges several input streams into one output stream.

All input files must match in length. The interleaving length specified by the option -interleave defines the size of the data blocks which are read from each of the sources in turn. In case complex data streams are to be merged interleave should be set to 2. If the input files should simply be concatenated, interleave has to be set to the length of the streams.

`Usage: merge [options] > outfile`

`Options:`

-osignal opipe		write output signal to	(stdout)
-files char* *{ char* }*		names of input files *more than one columns may be specified*	<off>
-interleave int		sample interleave factor	(1)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Control Statements

Execute Script (do)

This program executes a TAP macro script.

The user who wants to execute a predefined macro script just might enter

```
    do -f macro.ana
```

to obtain further information about functionality and specific options of a macro script. TAP macro scripts should have the extension .ana and must be located in the current directory. The default script which is executed if no filename is specified is named makefile and should also be located in the current directory. It is part of the TAP distribution kit, just like two other predefined macro scripts named specdens.ana and harmonics.ana.

Macro specific options and macro specific targets can then be entered according to

```
    do -f macro.ana [option_assignments] [targets]
```

like in the example

```
    do -f specdens.ana WAV=myWav fu=60 fo=1800 Frequency Magnitude
```

The user who wants to change, extend or create macro scripts has to know, that TAP script processing is based on the nmake script processor. Macro scripts actually are makefiles for the GNU nmake utility. Syntax specification and programming instructions should be taken from the documentation delivered by the GNU usergroup.

`Usage: do [options]`

`Options:`

-f char*		name of the macro script to be executed	<off>
macro options		macro options depend on application	
-? | -help		nmake help screen	<off>

Execute Shell (run)

This primitive creates a shell command providing proper CPU-time accounting.

The UNIX syntax is emulated. Multiple commands can be separated by semicolons. The spawn operator & and the option -bg is not implemented yet. Pipes can be used in the command string.

```
Usage: run [options]
```

```
Options:
```

-sh char*		shell command	<off>
-bg		execute in background	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Map command line (map)

Create a window which is shifted along the input data stream and direct window data to the primary input stream of a shell which executes the specified command line. Compose output stream from partial streams created by the single shell processes. Let partial output streams overlap additively by given number of samples.

A window with a given number of samples (-windowsize option) is stepped along the input data. The step distance (-step option) can be smaller than the window size (overlapping input windows), it can be equal to the window size (contiguous input windows), and it can be greater than the window size (sparse sampling). If decay is specified (-decay option) then the input stream is padded with zeroes and the window is stepped until it does no longer contain any non-zero value. Just before that condition is reached the mapping process stops.

For each position of the input window the specified TAP command line (-cmd option) is executed and supplied with its input stream. Its output stream is collected and aligned sequentially without or with some specified overlap (-overlap option). If an overlap is specified then overlapping buffer segments are accumulated (added). A typical application for overlapping output buffers is a frequency domain filter with a Gaussian window. If such windows overlap by half of their lengths, their sum is always unity.

Place holders can be used in the TAP command string. “%d” will be replaced by the buffer index and “%e”,”%f”,”%g”,”%E” or ”%G” by the first consecutive input samples of each data frame. Up to four replacements are allowed. If the mapped command lines are signal sources which do not require any input data then the -originate option must be set. The -terminate option must be given if command lines are signal sinks not producing any output data.

`Usage: map [options] < infile > outfile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-windowsize uint	[samples]	analysis window size	(1024 samples)
-step uint	[samples]	analysis window step	(256 samples)
-overlap uint	[samples]	result window overlap	(0 samples)
-decay		zero pad last incomplete windows	<off>
-terminate		no output stream created by cmd	<off>
-originate		input stream not used in cmd	<off>
-cmd char*		command string to map	(func -att 10)
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Varying Parameters (vary)

This operator repeats a command line varying a parameter between certain limits. Special characters are used as place holders in the command line. A @-sign is replaced by the varying parameter. A #-sign is replaced by the channel number in case of multi-channel operation. A &-sign is replaced by the result of the reference run. The part enclosed by [ ] is executed only once for all channels. Normally it is preceded by the stimulus generator part and succeeded by the analysis part.

If in a multi-channel simulation the reference string contains an empty common part (like [ ]) then no separate run will be made and channel zero will be assigned to the reference simulation. The resulting value will be used as a reference for all channels. Otherwise a reference analysis will be made for each channel separately and these different references will be used for different channels. In the reference string xref will be substituted for all @-signs while &-signs are replaced by zero.

If the -xinput option is set then discrete input values are read from a binary input stream instead of looping between start to stop with increment. This is illustrated by the example Gain Tracking and Total Distortion Analysis found in appendix B.

`Usage: vary [options] < infile`

`Options:`

-isignal ipipe		read input signal from	(stdin)
-xinput		read external binary table	<off>
-x0 double		start value	(0)
-x1 double		stop value	(10)
-dx double		increment	(1)
-xref double		reference value	(0)
-chan int		number of channels	(1)
-dut char*		command line	(echo @)
-ref char*		command line for reference calculation	()
-debug		generate internal debug information	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Create Named Signal (signal)

This operator is used to name the signal of the pipe. It can be used to synchronise processes or to fork a signal to a second process.

```
Usage: signal [options] name < infile > outfile
  char* name              name of pipe file
```

Options:

-isignal ipipe		read input signal from	(stdin)
-osignal opipe		write output signal to	(stdout)
-? | -help		generates this help screen	<act>

Flag (flag)

This primitive creates one or more named flags in order to allow other processes to wait for proper input data.

To synchronize multiple processes the flag and cont operators can be used. One process calls flag to set a named flag as soon as it has finished to create a set of valid output information needed by another process.

The other process calls cont in order to wait until the required flag information is received. When cont returns it is save to access the data files created by the other process.

Using the -n option more than one flag can be generated simultaneously. The flags will get names created from the name specified using the option -o plus a postfix character from the set [0,1,2,…] up to the value of the -n option minus one.

The synchronization procedure is based on creating and polling for zero-sized disk-files having the indicated names.

```
Usage: flag [options]
```

Options:

-n int		number of flags to set	<off>
-o char*		flag name	(“flag”)
-? | -help		generates this help screen	<act>

Continue on flag (cont)

This primitive continuously polls for one or more named flags. It actually waits until other processes are signalling a certain execution state. Waiting for flags puts the process in a suspended state in order not to block the CPU.

This operation waits for one or more named flags created by another processes using the flag command.

Using the -n option it is possible to wait for more than one flag at once. Execution goes on as soon as all flags of the group are active. The names of the group members are derived by appending a number to the name specified using the -i option (name0, name1, name2...). This name conversion applies as soon as the -n option is specified even if it is set to one.

Once a flag has been sensed active it is reset automatically unless the -nodelete option is given. A time-out terminates the waiting state after a given number of seconds (-timeout) to prevent dead-lock.

`Usage: cont [options]`

`Options:`

-n int		number of flags to wait for	<off>
-i char*		flag name	(“flag”)
-nodelete		don't reset flag	<off>
-timeout int		number of seconds to wait	(1000)
-? | -help		generates this help screen	<act>

Create Textfile from Template (fill)

Vary text stream according to some directives contained in it. Variation is history dependent. Integer sequences, random integers, token sequences and random token selections are implemented.

Searches an textual input stream for directives which are replaced in the output stream by the result of their evaluation. The evaluation depends on history. History is stored in a temporary history file. Its name can be specified using the -tmpname option. Deleting this temporary file resets the history.

A seed for the random number generator may be specified using the -seed option. If this option is not used, random sequences will differ in different runs. Subsequent calls of fill on the same input stream will create different output if some variation directives are included in the input text. The recognized variation directives are:

{seq x0 x1 dx rx}	Generate one value of a periodic sequence {x0..x1} which increments by dx after every rx runs. The variables x0, x1, dx and rx represent integer numbers.
{ran x0 x1}	Generate random integer in range x0..x1 per run
{sel t1 t2...tn [rx]}	Sequential selection out of given list of tokens. The iterator is incremented after every rx runs. Valid tokens my contain any characters except a token delimiter, which can be freely chosen, the end of line character and the closing brace <}>. The character terminating the sel directive is accepted as token delimiter. A numeric value can be entered between the last token delimiter and the closing brace. If the closing brace is preceded by the token delimiter, rx defaults to one and a new selection is made in each single run.
{rsel t0 t1 t2...}	Random selection out of given list of tokens. A new random selection is made in each single run. Valid tokens my contain any characters except a token delimiter, which can be freely chosen, the end of line character and the closing brace <}>. The character terminating the rsel directive is accepted as token delimiter.

`Usage: fill [options] < infile > outfile`

`Options:`

-tname char*		name of template file (stdin)	<off>
-oname char*		name of output file (stdout)	<off>
-tmpname char*		name of file keeping status info *(delete this file to reset history)*	(~history.tmp)
-logname char*		name of log file	<off>
-seed int		seed of random number generator	<off>
-stat char*		write program statistics to file	<off>
-? | -help		generates this help screen	<act>

Anhang B: Basic TAP Examples

Spectral Synthesis and Analysis

In the example below several sine-wave stimuli signals are merged additively, windowed, zero-padded and Fourier transformed before the logarithmic magnitude spectrum is sent to the plot function. The table operation is used to append a previously created zero string to the windowed signal. The db values are related to the 990th spectral line, which is the sine-wave component having 99 periods in the original, 10 times shorter buffer.

```
stim -freq 0 -len 9216 | signal zero.bin
stim -per 6 | stim -merg -per 22.4 | stim -merg -per 27 | stim -merg -per 99 | stim -merg -per 34 |
  window -nutt | table -fn zero.bin -merge 0 | dft | db -ref 990 |
  plot -grid -name "spectrum with 6, 22.4, 27, 34 and 99 periodes in buffer using nuttall window and
  zero padding" -title "Demonstrating stim, window, table, dft, db and plot commands"
```

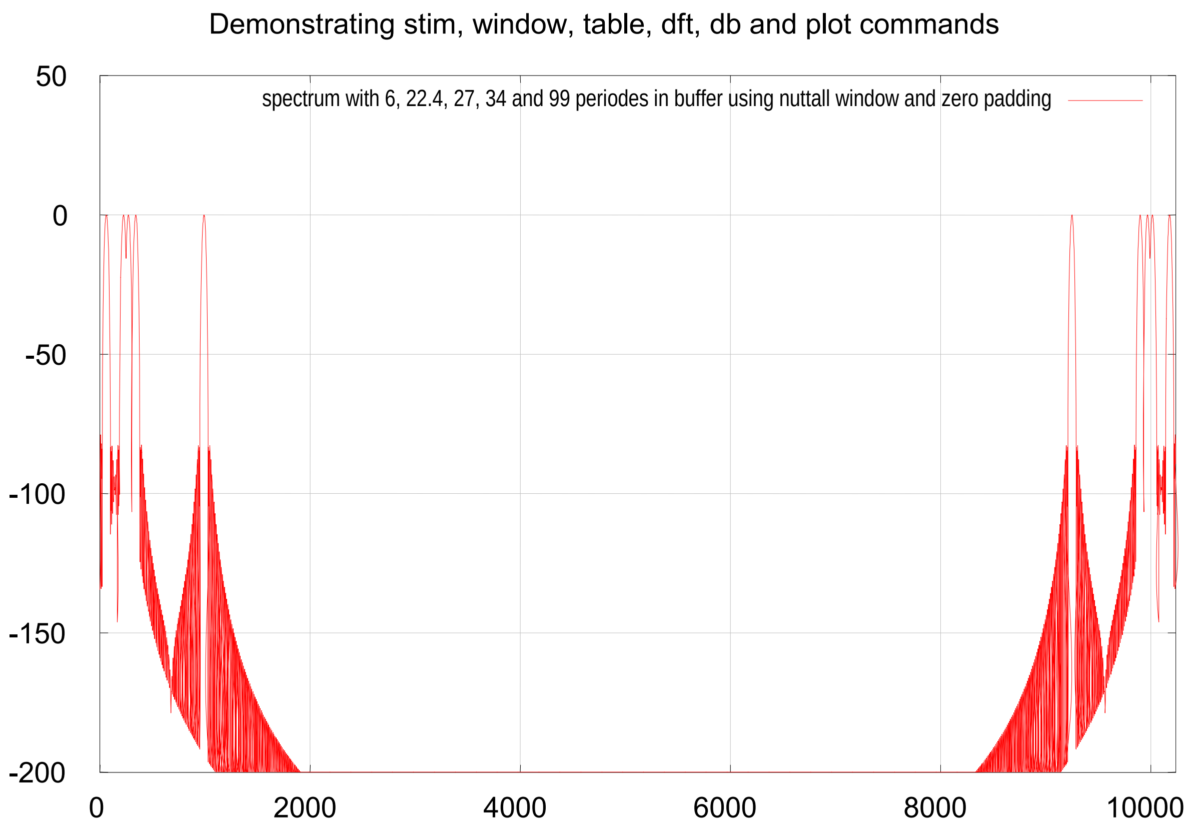


Figure 6: Spectral Synthesis and Analysis

Finding MLS Seeds

Uses the ramp generator to generate a vector of all possible 8-bit seeds. For all these seeds a maximum length pseudo random sequence is generated and its auto correlation is compared with the delta function.

From the initial seed vector all values are picked where the auto correlation matches the delta function. Intermediate results are stored in readable format in order to allow interruption of the lengthy process without loosing already calculated data.

```
stim -fre 0 -dc 1 -step 2 -len 128 > seed8.bin
vary -xin -dut "stim -len 255 -fre 0 -prn 8 @ | xcorr -MLS | bin2flo -nocnt" < seed8.bin > MLS8.flo
flo2bin -nocnt < MLS8.flo > MLS8.bin
pick -fn MLS8.bin <seed8.bin | bin2flo -nocnt > MLS8.txt
```

Create and Save Windowing Signal

The command line below creates a windowing signal which is the square-root of a Hamming window. It is needed in the following example when a spectral filtering operation is to be applied.

It uses the fact that the pulse response of the fir filter is the window it is based on. The Dirac pulse is created as spectrum of a single sine-wave signal. The stimulus contains 1024 sine-wave periods in the 4096 sample long buffer. Its Fourier transform is a Dirac pulse at the 1024th position and its mirror image which is clipped. This signal filtered by the fir term yields the hamming window itself. After clipping prolog and epilog the square root is calculated and saved.

```
stim -vp 0.03125 -len 4096 -per 1024 | dft | clip -epi 1024 | fir -hamming -len 1024 |
  clip -epi 1024 -pro 1024 | func -sqrt > sqrtham1024.bin
```

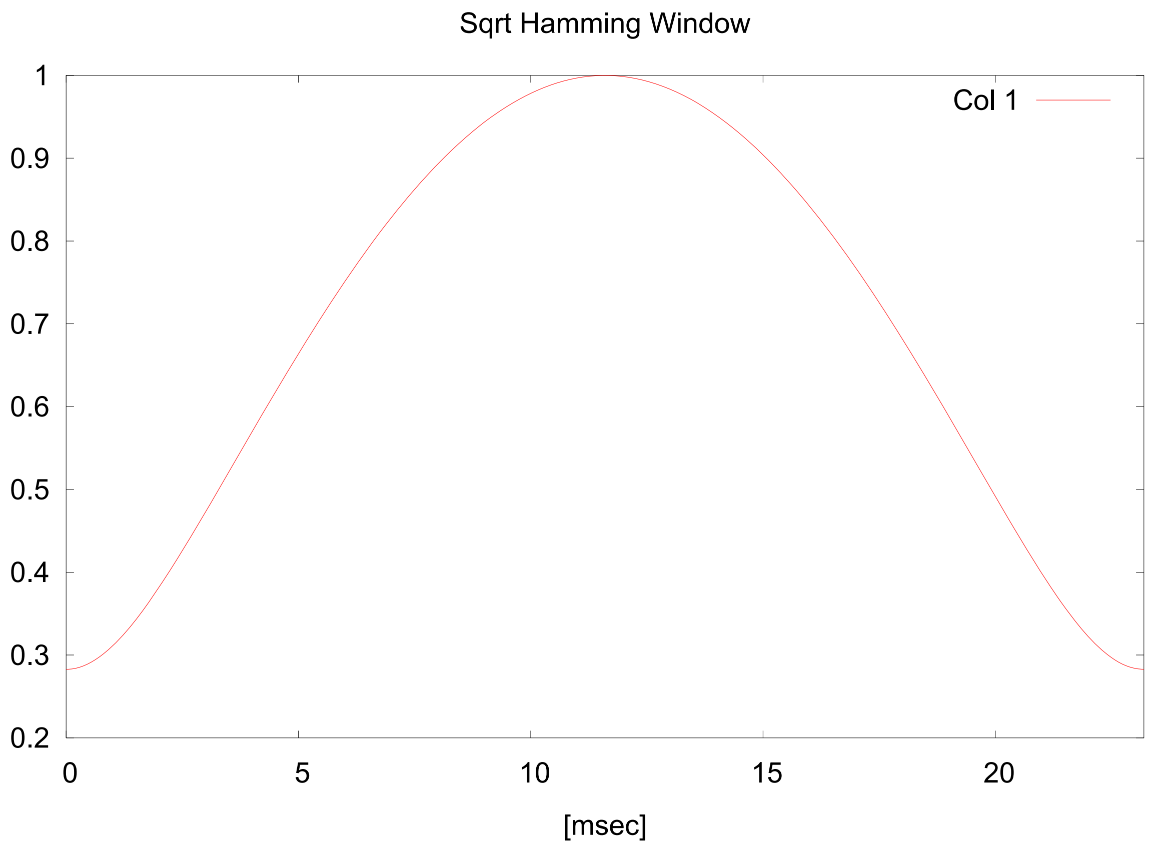


Figure 7: Sqare-root of Hamming Window

A filter which could be used for the following example can be created by the windowing function itself. The command line below creates a band-pass filter of 4th order with a pass-band centered around 1020 Hz with a 3 dB width of 320 Hz. The stimulus sets up a unity vector of proper length. Multiplied with the window it yields the window itself. The -mirror option makes it symmetrical. The sinc function of order one is used as a sample and hold to repeat each value two times. In the following example two identical multiplicators are required for real and imaginary part of the complex spectrum.

```
stim -dc 1 -freq 0 -len 1024 | window -bandp 1020 320 4 -sampl 44100 -mir |
   sinc -or 1 -n -2 | signal filter.bin
```

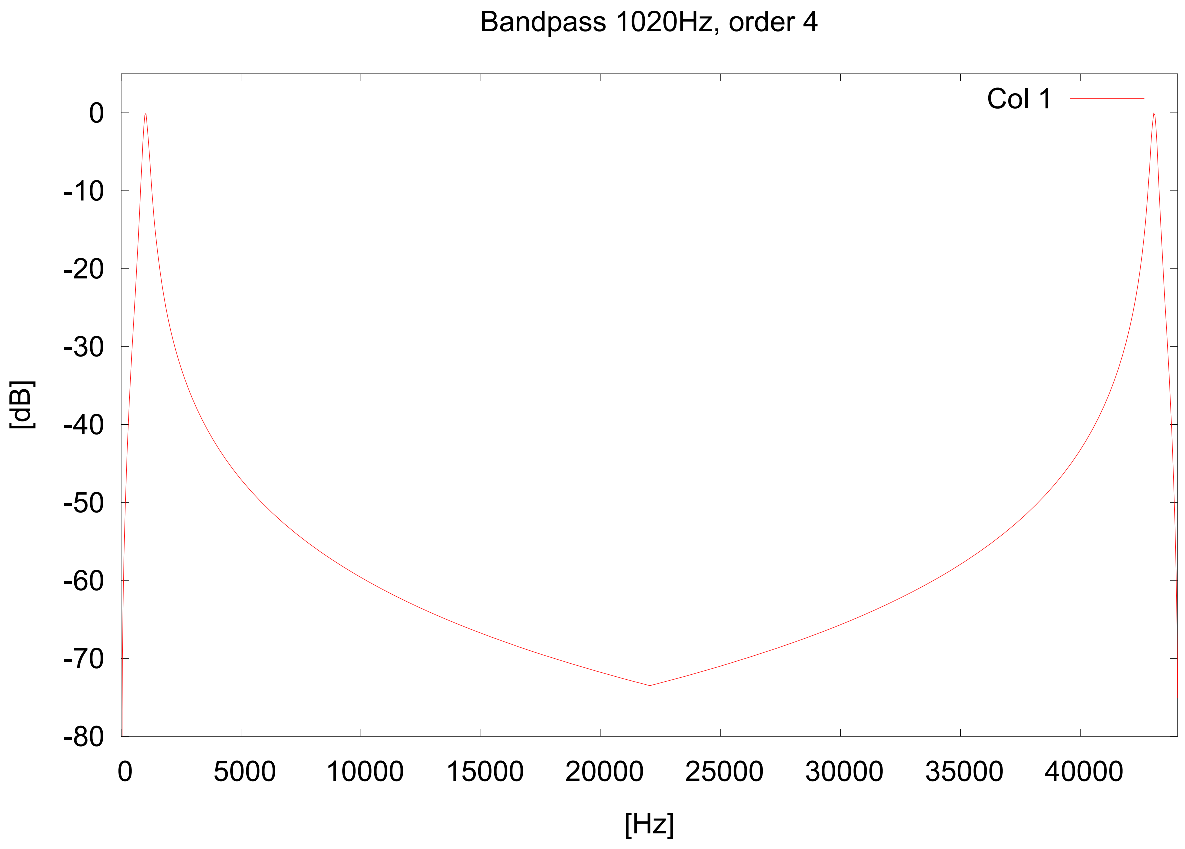


Figure 8: Bandpass Filter

Spectral Domain Filtering

Read input sound file and map a TAP command line performing a 1024 samples long frequency domain filter to overlapping portions of the input stream. The data window is 1024 samples long and is shifted by 256 samples after each step. The filter output windows which are overlapping by 768 samples are assembled back again to a contiguous output stream.

The window function which has to be applied twice - before and after the transition to the spectral domain - has been pre-computed and saved as sqrtham1024.bin in the previous example.

The command line which is mapped to the sound file segments applies the “square-root of hamming” window before the Fourier transformation and after the reverse Fourier transformation. The Fourier transformations (dft) are performed complex in rectangular coordinates. The data file filter.bin must be symmetric and must contain 1024 pairs of weighting coefficients. It is multiplied with the complex spectrum before this is scaled and retransformed:

```
wav2bin -fn myInFile.wav |
  map -win 1024 -step 256 -over 768 -cmd "window -rest sqrtham1024.bin |
  dft -reim -out2D | vecfunc -times -vector filter.bin | func -tim 0.25 |
  dft -rev -reim -inp2D | window -rest sqrtham1024.bin" |
  bin2wav -fn myOutFile.wav -like myInFile.wav
```

The weighting coefficients for a simple bandpass filter can be created according to:

```
stim -dc 1 -freq 0 -len 1024 | window -bandp 1020 320 4 -sampl 44100 -mir | sinc -or 1 -n -2 | sig-
  nal filter.bin
```

Gain Tracking and Total Distortion

Gain tracking and total distortion over input level are quality measures which are applied in telecommunications. The quantisation methods which are used are the Northern American µ-law and the European A-law. Both laws employ signal level dependent quantisation steps to maintain equal S/N ratio over a wide range of levels. This way linear bit resolutions of 12 to 13 bits are compressed to 8 bits.

The effect of the nonlinear quantisation scheme on gain over level (gain tracking) and on quantisation distortion over level is calculated by the following command lines. The tolerance masks for telecommunication systems have been taken from the ITU standards.

First a vector containing 650 level values is created to cover the level range from +3 dB down to -62 dB with a resolution of 0.1 dB. Next the vary operation is used to run the command line specified using the -dut option 650 times, replacing the place holders in the level statements by the values taken one by one from the input stream. The fana operation is used to generate gain and psophometrically weighted distortion values of its A-law stimulus with defaults for stimulus frequency (1020Hz), sampling frequency (8kHz) and buffer length (1024 samples). The resulting values are written in readable format to the text file alaw.flo

After conversion of the result file into a binary two-column signal, the initial levels vector is merged to create the first column of the resulting three-column signal. This is accomplished by the table command which can handle multi-column streams and files.

The plot commands are straight forward, and tolerance masks, axis labels and plot titles are specified. The first column is selected as x-axes column, while one of the remaining columns is chosen for display.

```
stim -dc 3 -fre 0 -step -0.1 -len 650 | signal levels.bin

signal levels.bin | vary -xinp -dut "stim -A -lev @ | fana -lev @ -db 60 -wei psopho -stimbw 120
 -rmswb -distwb -bin | bin2flo -noc" > alaw.flo

flo2bin -nocnt < alaw.flo | signal alaw.bin

table -sig 2 -fn levels.bin -sel 1 -pos 0 < alaw.bin |
 plot -sig 3 -xcol 1 -ycol 3 -mask td -dir tx -law a -xu [dB] -yu [dB] -tit "Total Distortion over
 Level, A-law Quantisation"

table -sig 2 -fn levels.bin -sel 1 -pos 0 < alaw.bin |
 plot -sig 3 -xcol 1 -ycol 2 -mask gt -dir tx -law a -xu [dB] -yu [dB] -tit "Gain Tracking, A-law
 Quantisation"
```

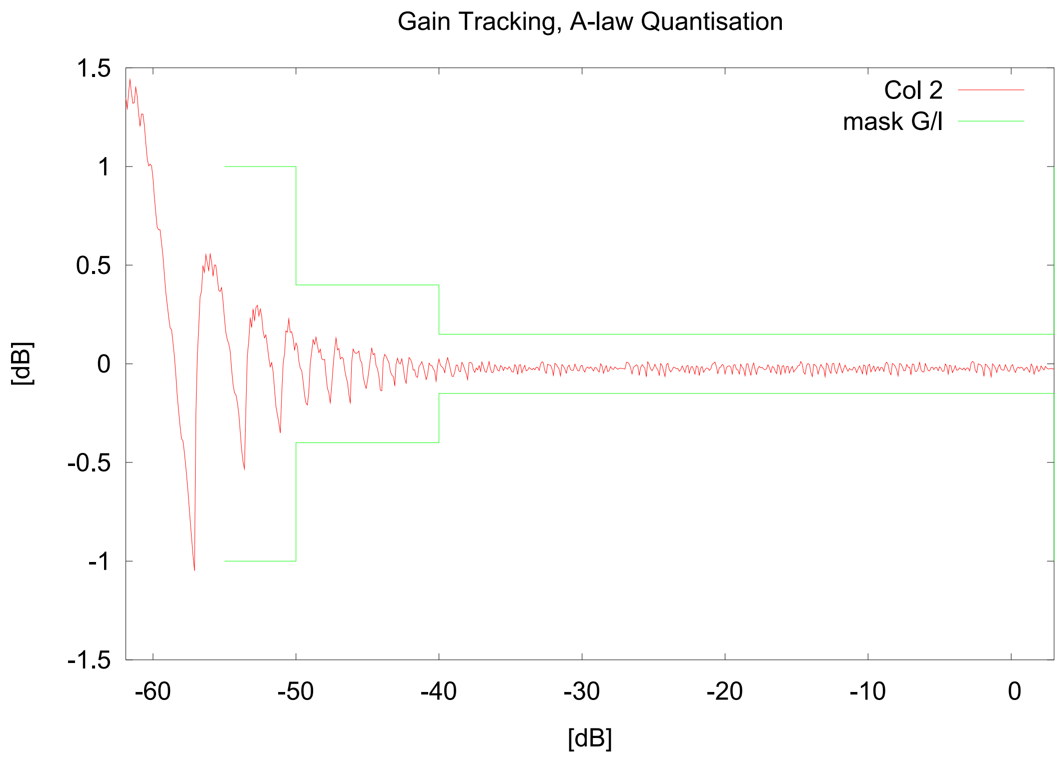


Figure 9: Gain Tracking, A-law

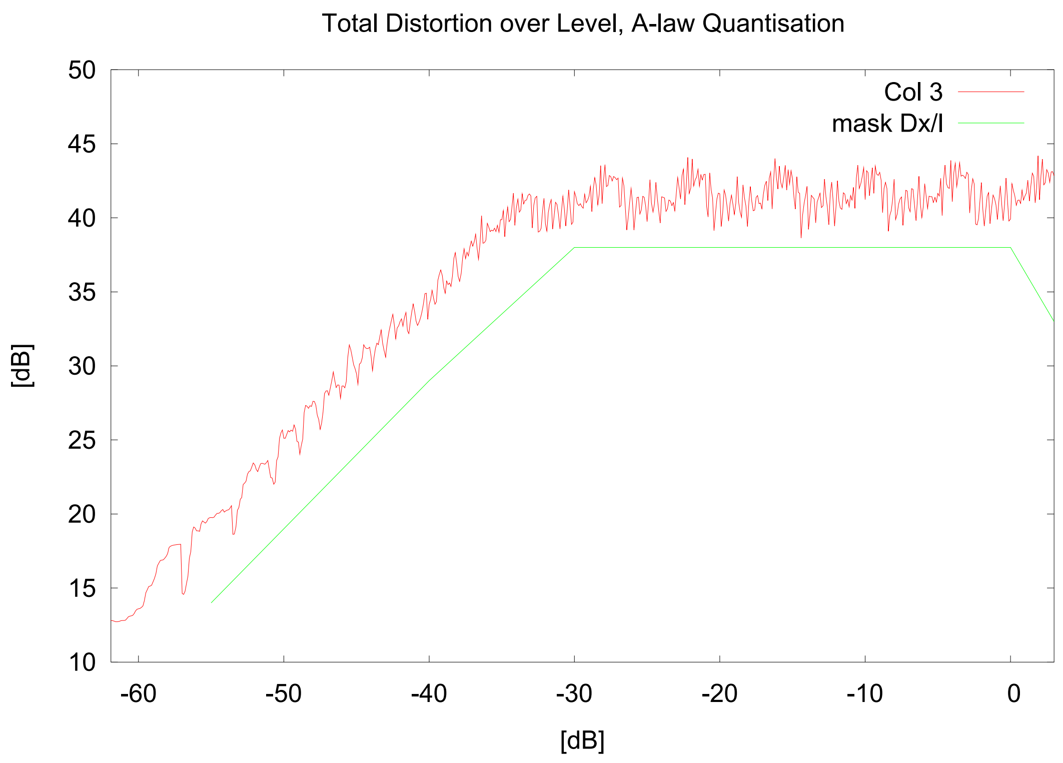


Figure 10: Total Distortion, A-law

Dynamic Harmonic Analysis

Dynamic harmonic analysis without segmentation can be performed using the dftpeaks command. In the first step ten seconds of a slightly distorted (μ-law quantisation at -40 dB level) sine-wave sweep is generated and stored in the wave file to be analysed.

```
stim -Ulaw -sampl 44100 -len 441000 -lev -40 -freq 100 -sweep 0.1 |
  bin2wav -bit 16 -sampl 44100 -chan 1 -fn mySig.wav
```

In the next step the analysis is started. In order to get smooth and accurate frequency curves the long window is stepped by the same amount as the small window. As we know a sweep signal is processed so the relative catch range -frel can be set to small percentages. The first nine harmonics are to be analyzed and the rms curve is to be added. The fundamental range lies between 90 and 1100 Hz. The last eventually incomplete frame is dropped before the results are saved.

```
wav2bin -fn mySig.wav | dftpeaks -slen 2048 -sofs 1024 -llen 8192 -lofs 1024 -feps 1 -frel 2 2
  -freq 1 2 3 4 5 6 7 8 9 -rms -nuttall -fu 90 -fo 1100 | clip -epi 19 | signal myResult.bin
```

The complete command lines to generate the following four analysis plots are shown below. The last one is more interesting because a fir filter with the savitzky smoothing kernel of order two and length 3+1+3 coefficients is applied to the whole data stream. The table commands do the matrix transposing (428x19 ⇒ 19x428 ⇒ 428x19) required in order to have the data vectors separated. It is tolerated here for the sake of simplicity that the filter is running over the joints of the columns.

```
signal myResult.bin | plot -sig 19 -ycol 1 3 5 7 9 11 13 15 17
  -title Frequencies -dx 0.02322 -xu [sec] -yu [Hz]

signal myResult.bin | plot -lny -sig 19 -xcol 1 -ycol 2
  -title Magnitude -name Fundamental -xu [Hz] -yu [dB]

signal myResult.bin | plot -lny -sig 19 -xcol 1 -ycol 19 2 4 6
  -title Magnitude -nam RMS H1 H2 H3 -xu [Hz]

signal myResult.bin | table -sig 19 -tra | fir -sav 2 -lef 3 -rig 3 | table -sig 428 -tra |
   plot -lny -sig 19 -xcol 1 -ycol 4 6 8 10 -title Magnitude -nam H2 H3 H4 H5 -xu [Hz]
```

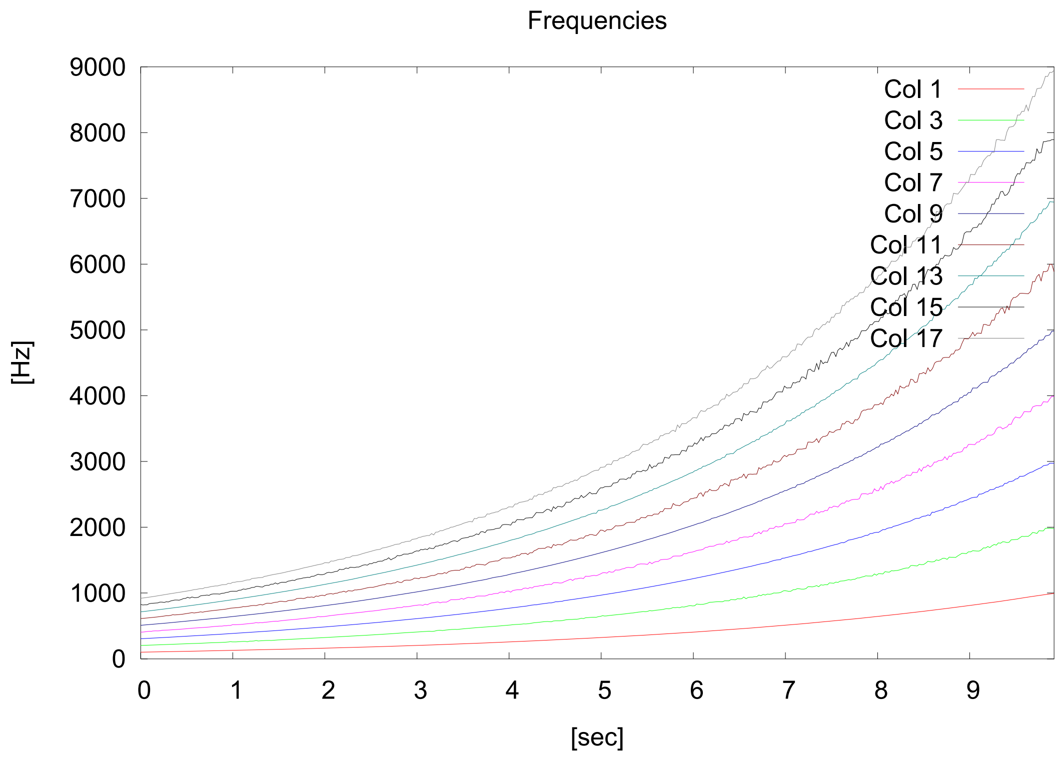


Figure 11: plot -sig 19 -ycol 1 3 5 7 9 11 13 15 17 -title Frequencies -dx 0.02322 -xu [sec] -yu [Hz]

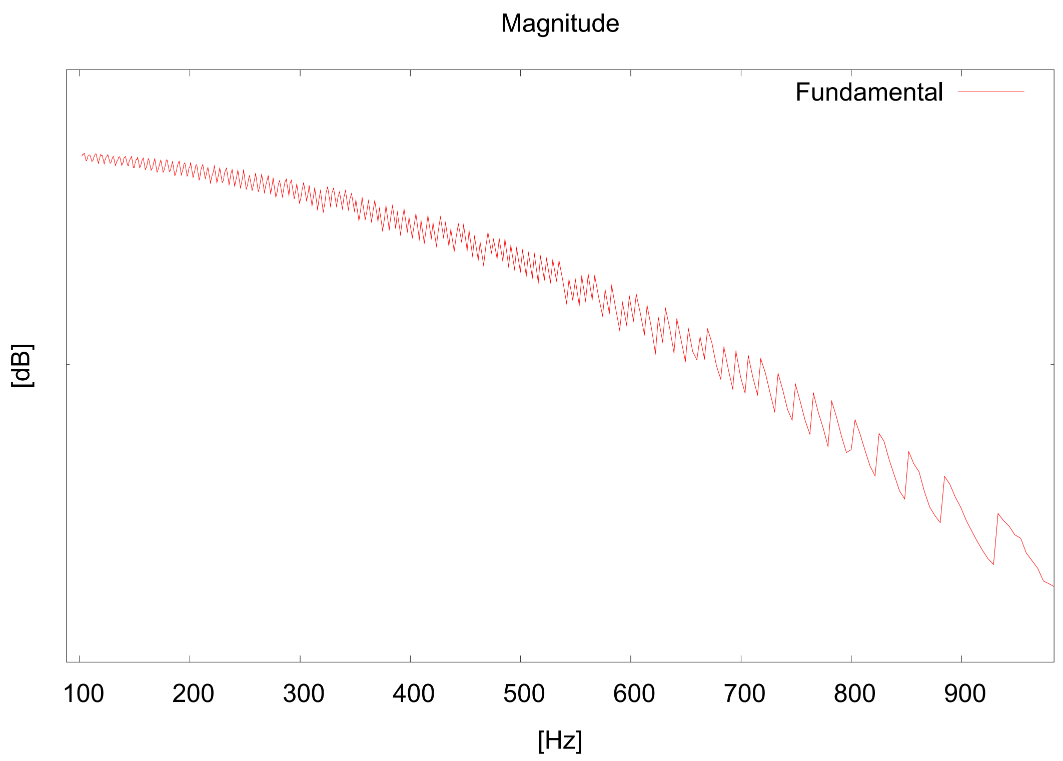


Figure 12: plot -lny -sig 19 -xcol 1 -ycol 2 -title Magnitude -name Fundamental -xu [Hz] -yu [dB]

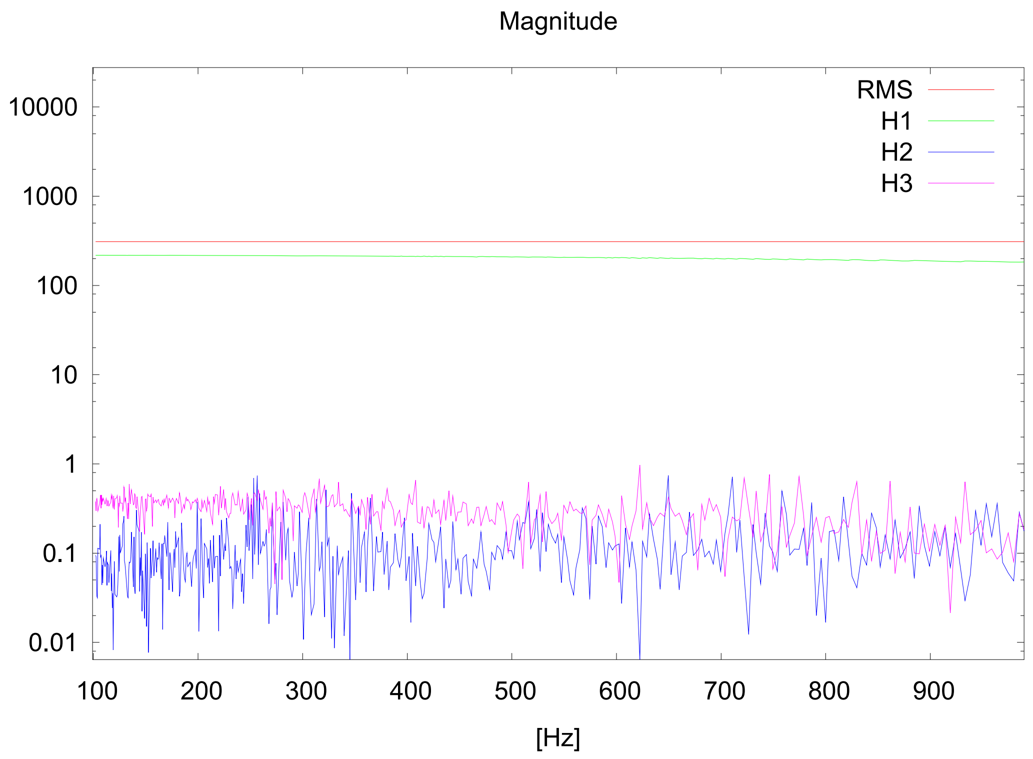


Figure 13: plot -lny -sig 19 -xcol 1 -ycol 19 2 4 6 -title Magnitude -nam RMS H1 H2 H3 -xu [Hz]

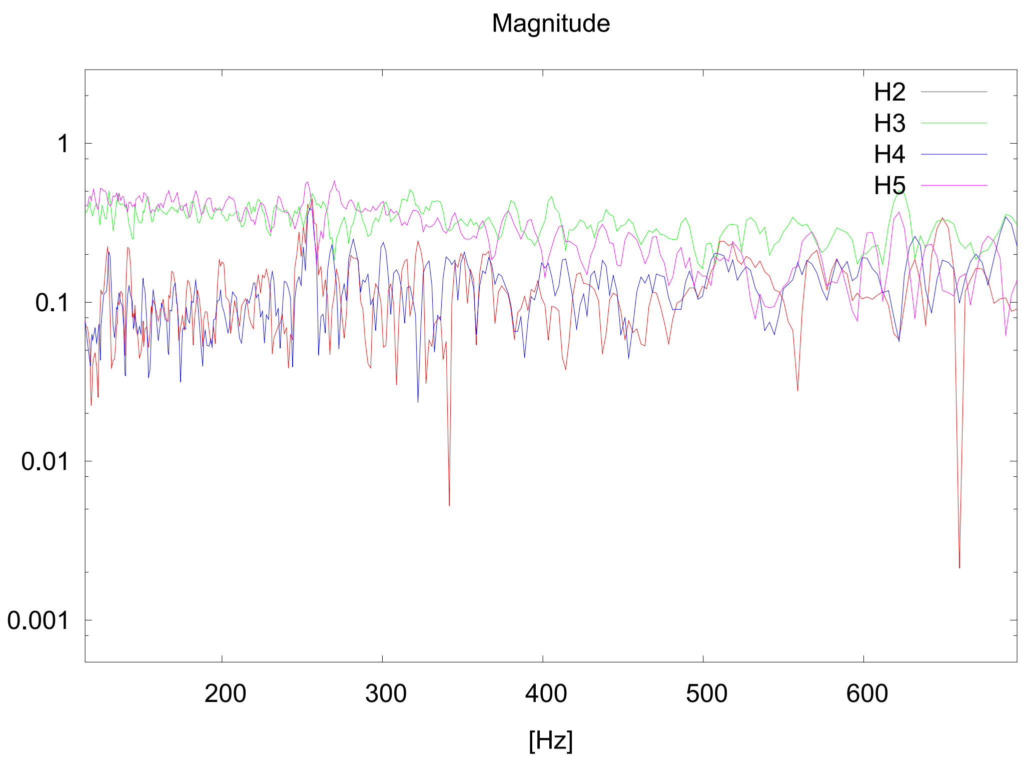


Figure 14: **plot -lny -sig 19 -xcol 1 -ycol 4 6 8 10 -title Magnitude -nam H2 H3 H4 H5 -xu [Hz]**

3D Spectrogram (Waterfall Chart)

The first command line creates ten seconds of a sine-wave sweep with some white noise superimposed on top of it. The sweep starts at 100 Hz and rises with 0.1 decades per second. Therfore it goes up to 1 kHz after 10 seconds.

The second command line generates a 3D surface plot by mapping a spectral analysis command line to the original input stream. For this purpose an analysis window of 4096 samples is stepped over the input data stream with a stepping distance of 4410 samples (corresponds to 0.1 seconds). The command string argument of the -cmd option is then executed for each analysis frame and the resulting spectra are concatenated.

The plot function gets a 100x100 data grid. The -surface option initiates the 3D surface plot and the first dimension is specified using the -signals option. Surface mash is turned off in the -options string.

```
stim -sampl 44100 -len 441000 -freq 100 -sweep 0.1 -noise 0.01 | signal mySig.bin
```

```
signal mySig.bin | map -win 4096 -step 4410 -cmd "window -nutt | dft | clip -keep -pro 100 | db" |
   plot -surf -sig 100 -opt "set pm3d; set view 20,10,1.5,1; set nosurface"
```

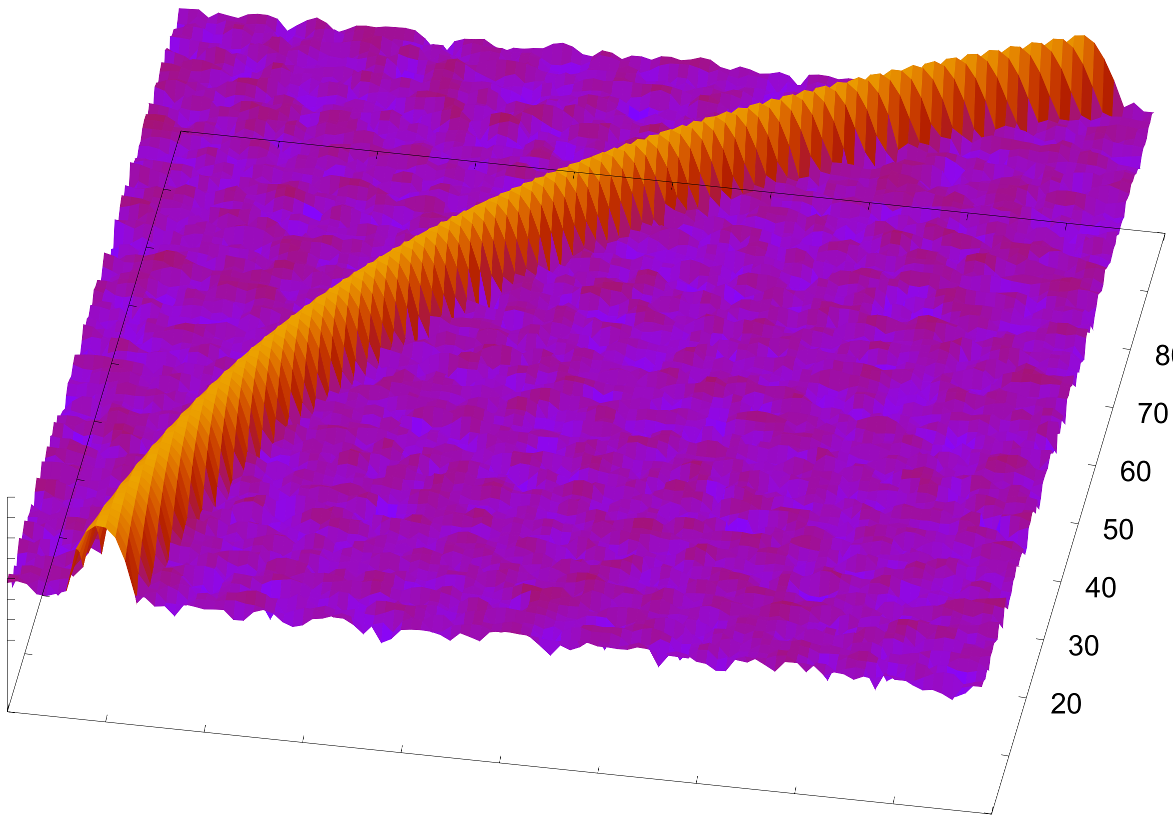


Figure 15: Waterfall Spectrogram of Sweep

Simulation of Nine Level CMOS DAC with Dynamic Averaging

Multi-bit CMOS D/A Converter with Integrated Low-Pass Filter

In CMOS technology a multi-bit D/A converter can most efficiently be implemented as a switched capacitor structure. This way it is even possible to include a low-pass filter with the D/A converter for the only additional cost of one capacitor. A switched capacitor summing stage with a continuous time feed-back, often referred to as lossy integrator stage, is well suited for this purpose. Its input is permanently connected to the reference voltage and the steady state gain of that stage which is defined by a capacitance ratio will determine the DAC output voltage.

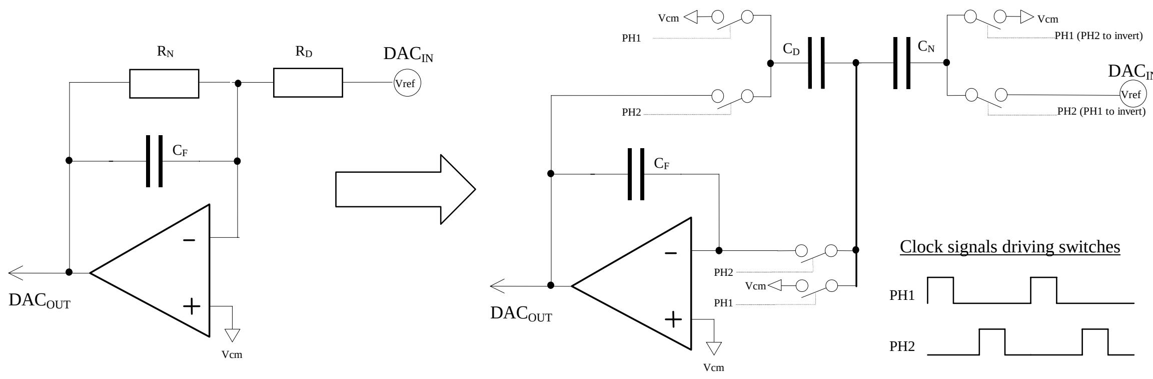


Figure 16: DAC with Low-Pass Function, Voltage defined by Capacitor Ratio

To understand its operation a simple summing amplifier as shown in Figure 16 should be considered. The steady state output voltage is defined by the resistance ratio R_N/R_D. If resistors are replaced by switched capacitors their equivalent resistance $1/(f_S\ C)$ has to be inserted and the steady state output voltage $V_{DAC} = V_{ref} \times C_N / C_D$.

The signals PH1 and PH2 in the switched capacitor circuit are two non-overlapping clock signals both running at the DAC's sampling frequency (e.g. 4 MHz) which never are active at the same time (refer to Figure 16). By swapping the two clock phases driving the switches to the right of C_N (as indicated in the figure) the steady state output voltage will reverse its polarity.

This leads to a practical implementation of a nine level DAC where C_N consists of up to four unit capacitors which are switched in parallel contributing to the numerator of the fractional expression for the output voltage - C_N is the capacitor group between circuit input and summing node. Four other unit capacitors, switched in parallel, generate C_D, the denominator value - the group is connected to the amplifier output and to summing node. The input to the circuit is the reference voltage for non-zero magnitudes and the common mode voltage (analogue ground) for a zero magnitude. The polarity bit of the signal value drives a multiplexer which exchanges the clock phases PH1 and PH2 in the indicated positions.

Topologically the whole array of eight unit capacitors has to be connected together at the capacitors' top plates. The reason is the parasitic capacitance between capacitor plates and chip substrate (ground). Bottom plates have parasitic ground capacitances which are very big. They always have to be connected to low impedance voltage sources, like OPAMP outputs or supply voltages. High impedance nodes, like the summing node of an amplifier, must be connected to the capacitor's top plate, which has a very small parasitic capacitance to ground, because it is shielded completely by the bottom plate.

The common top plate is switched to ground at PH1 (the precharge phase) and switched to the inverting amplifier input at PH2 (the evaluation phase).

One group of four unit capacitors (their total value is the denominator of the fractional expression for the output voltage) is switched to ground at PH1 and switched to the amplifier output at PH2. The other group of 1..4 capacitors (depending on the magnitude of the signal value) is switched either to the reference voltage or to ground. In case of a positive sample the capacitor subset defined by the signal magnitude is switched to ground at PH1 and to V_{ref} at PH2. In case that a negative output voltage is required this group is switched to V_{ref} at PH1 and to ground at PH2.

All eight elements are involved in the generation of four steps. The step voltages are given by

$$\left\{V_{ref} \times \frac{C_1}{C_5+C_6+C_7+C_8},\ V_{ref} \times \frac{C_1+C_2}{C_5+C_6+C_7+C_8},\ V_{ref} \times \frac{C_1+C_2+C_3}{C_5+C_6+C_7+C_8},\ V_{ref} \times \frac{C_1+C_2+C_3+C_4}{C_5+C_6+C_7+C_8}\right\}$$

or in the ideal case

$$\left\{V_{ref} \times \frac{1}{4},\ V_{ref} \times \frac{2}{4},\ V_{ref} \times \frac{3}{4},\ V_{ref} \times \frac{4}{4}\right\}$$

which means that V_{ref} directly defines the maximum output level.

The case of zero magnitude is a special case. Instead of disabling all capacitors of the numerator group for the charge transfer it is better to generate the zero output voltage by activating the complete numerator group but connecting the circuit input to ground instead of V_{ref}. This way circuit parasitics like OPAMP offset voltage, channel charges of CMOS switches, gate-drain capacitances of MOS transistors and bottom plate capacitances do not effect the accuracy of the DAC output voltage as much.

Dynamic Averaging

There is an essential drawback connected to multi-bit D/A converters. Bad linearity or symmetry properties have a dramatic effect on the overall converter performance and much of the accuracy gained by a ΣΔ-approach is lost again. In order not to loose a significant part of the accuracy provided by ΣΔ-converters matching of unit resistors or capacitors in the DAC must be better than what is currently achievable by technology. This has considerably limited the use of multi-bit ΣΔ-converters in the past.

As already outlined a multi-bit CMOS D/A converter can be designed in a way, that its output voltage $V_{DAC} = V_{ref} \times C_N / C_D$ is proportional to the ratio of two capacitances C_N and C_D. Both capacitances are composed from unit capacitors which are turned on and off by activating and deactivating the clock signals which connect these elements to the circuit. Without much circuit overhead any fixed assignment of a unit capacitor to a certain DAC output value can be avoided. It is even possible to allow that a unit capacitor can be used to contribute to the numerator of the voltage term as well as to its denominator.

It is also known that the inversion of the output voltage of switched capacitor circuits can easily and accurately be achieved by swapping the two clock phases controlling one of the capacitor groups. This means that the same group of capacitors is responsible for both the gain in the positive and in the negative half of the operating range. This way perfect symmetry of output voltages can be achieved which does not depend on capacitor matching.

The method called 'dynamic element selection' or 'dynamic averaging' makes now use of these properties to eliminate linearity problems caused by non ideal matching of unit elements. It uses the flexibility in the circuit topology to select the unit capacitors contributing to the DAC output voltage dynamically and algorithmically in order to average their mismatch over time.

Just as the spectral distribution of quantisation noise is shaped by a ΣΔ-converter in order to move most of the noise power up into a frequency range where a simple post filter can get rid of it, dynamic element selection shapes the noise spectrum which is connected to bad DAC element matching and moves most if it up, where it is removed by the post-filter in the same way. With dynamic averaging the effect of DAC element mismatch of up to 3% and more is eliminated completely from the base-band signal. 3% capacitor matching can easily be achieved in production even on less silicon area which is usually spent for unit capacitors.

A multi-bit ΣΔ-converter in conjunction with dynamic element selection is now the way to put minimum requirements on all analogue circuit parts while modern production technologies are allowing high density digital circuits occupying less and less silicon area. For all these reasons a 3 bit ΣΔ-converter in conjunction with a 9 level DAC using an improved data-weighted dynamic element selection algorithm has been selected for the actual application example.

This new and enhanced DWA (data weighted averaging) concept allows not only to make linearity but also gain independent of capacitor matching which is important because another source of the total gain error is the reference voltage generator - an analogue sub-circuit. If the total transmission gain is no longer influenced by the DAC component values there is more margin for the reference voltage itself.

In order to meet the very stringent absolute gain specifications of ±0.3 dB over all temperatures, supply voltages and process conditions component value trimming was commonly applied to the reference voltage. If the reference voltage becomes the only source of gain errors this trimming can eventually be omitted and an expensive production step can be saved.

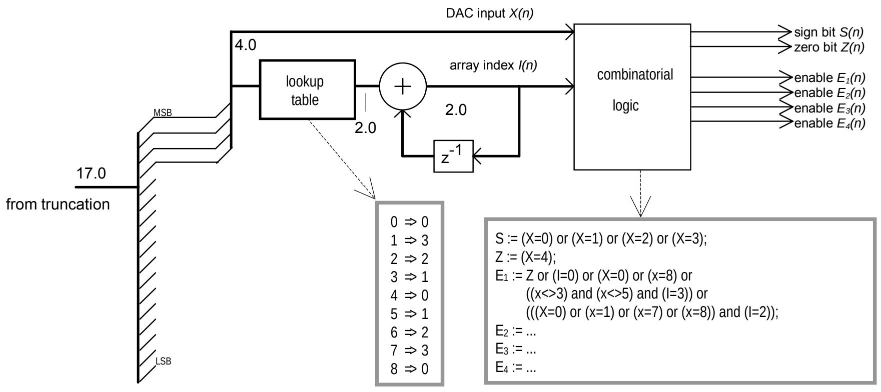
Data Weighted Dynamic Element Selection

The principle of data weighted dynamic element selection is to use the whole set of DAC elements (in our special case capacitors) as often as possible[16]. To do this an array index *I(n)* always pointing to the next DAC element in turn is introduced. Depending on the DAC value *X(n)* a certain number of DAC control lines *$E_i(n)$* starting at the index position is activated and the index is incremented by the number of elements used. Modulo arithmetic is used of course. As an illustration a possible code sequence for a 9 level DAC (value range from 0..8) together with the result of a data weighted dynamic element selection (8 selectable elements with indices 0..7) is shown in Table 3.

DAC input X(n)	*array index I(n)*	*index increment D(n)*	*selected elements $E_i(n)$*
1	0	1	0
0	1	0	none
2	1	2	1, 2
3	3	3	3,4,5
2	6	2	6,7
4	0	4	0,1,2,3
3	4	3	4,5,6
5	7	5	7,0,1,2,3
6	4	6	4,5,6,7,0,1
7	2	0	2,3,4,5,6,7,0,1
0	2	0	none
...	...	...	...

Table 3: Data Weighted Dynamic Element Selection (Example Code Sequence, 8 Elements)

A real implementation which generates four enable signals *$E_i(n)$* for any kind of DAC elements, a signal *S(n)* representing the sign information of the analogue output value and a signal *Z(n)* indicating that a zero value has to be generated is shown in Figure 17.



```
S := (X=0) or (X=1) or (X=2) or (X=3);
Z := (X=4);
E1 := Z or (I=0) or (X=0) or (x=8) or
        ((x<>3) and (x<>5) and (I=3)) or
        (((X=0) or (x=1) or (x=7) or (x=8)) and (I=2));
E2 := ...
E3 := ...
E4 := ...
```

Figure 17: Data Weighted Averaging (Typical Implementation)

The lookup table for the input value *X(n)* takes care of the unsigned to signed conversion required to transform the ΣΔ-demodulator output range [0..8] into the range [-4..+4] which is used by the DAC. It delivers the number of positions the array index *I(n)* has to advance after a sample has

[16] **Baird, Fiez;** *Lin. Enhancem. of Multibit ΔΣ A/D and D/A Conv. Using Data Weighted Averaging*; IEEE Trans. Circuits & Syst. II, vol. 42, pp. 753-762, Dec.'95

been processed. This increment is 0 when no or when all DAC elements are used to compose the output value. The combinatorial logic block is typically defined by a lot of straight forward boolean equations which will be optimised by a synthesis tool and finally implemented as array logic.

Enhanced Data Weighted Averaging (EDWA)

In the switched capacitor DAC according to Figure 16 the output voltage is given by $V_{DAC} = V_{ref} \times C_N / C_D$ as was already outlined above. C_N defines the magnitude and is composed from 0 .. 4 unit capacitors referred to as the numerator elements. C_D is the reference capacitor and it is always composed from 4 unit capacitors, the denominator elements. The sign of the output voltage can be inverted directly as was shown so just the magnitude is of interest when an element selection algorithm is applied.

The conventional dynamic element selection circuit as described above would algorithmically select as many unit capacitors as are needed for the actual output voltage aiming to average their mismatches over time. This way a perfect linearity can be achieved but any mismatch between the mean unit capacitance of the reference group and the one of the working group will cause an absolute gain error.

Therefore the basic principle of data weighted averaging has been modified by including the reference capacitance in the averaging process. Actually two different groups of unit capacitors are swapped after each frame. In even frames the first group serves as the pool which a data dependent number of elements is selected from while the complete other group serves as reference capacitance. In odd frames the two groups change their place.

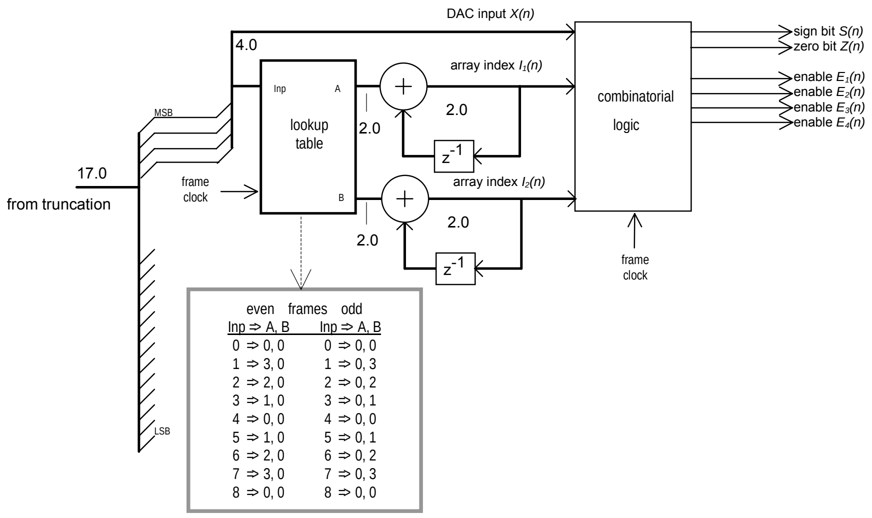
In a DWA algorithm it is required to keep track of the index of the last element which has been used to ensure that each element is used exactly the same number of times thus equally contributing to the output voltage. To do this a modulo 4 accumulator is required integrating all data inputs which have to be processed. Its output is the array index $I(n)$ which points into the capacitor array indicating where to take the next element from. According to the data value $X(n)$ a certain number of elements is then enabled by control lines $E_i(n)$ which contribute to the numerator of the gain term before the pointer $I(n)$ is updated.

A zero magnitude must be treated slightly differently due to the structure of the used DAC. Instead of disabling all numerator elements ($E_i(n)$=0) it is better to enable all of them ($E_i(n)$=1) but at the same time to make the value of the reference voltage temporarily to zero. This is done by means of the control signal $Z(n)$. When this is active $I(n)$ does not increment because no element has contributed to the output voltage.

In the enhanced algorithm there are two independent groups of capacitors so two different DWA pointers *$I_1(n)$* and *$I_2(n)$* must be used. A circuit which generates the required control signals for the D/A converter taking group swapping into account is shown in Figure 18.

The lookup table takes care of properly incrementing the index pointers depending on how many elements of each group have to be used. Usage of 0 elements does not increment the corresponding pointer and usage of 4 elements would move the pointer a complete turn again resulting in no index movement.

The least significant bit of the frame counter indicates which group of unit capacitors is active in a frame and contributes to the magnitude of the DAC output value. All capacitors of the other group are used as feedback capacitances. The least significant frame counter bit is also used to toggle between the two array pointers *$I_1(n)$*and *$I_2(n)$*.



even frames Inp ⇒ A, B	odd frames Inp ⇒ A, B
0 ⇒ 0, 0	0 ⇒ 0, 0
1 ⇒ 3, 0	1 ⇒ 0, 3
2 ⇒ 2, 0	2 ⇒ 0, 2
3 ⇒ 1, 0	3 ⇒ 0, 1
4 ⇒ 0, 0	4 ⇒ 0, 0
5 ⇒ 1, 0	5 ⇒ 0, 1
6 ⇒ 2, 0	6 ⇒ 0, 2
7 ⇒ 3, 0	7 ⇒ 0, 3
8 ⇒ 0, 0	8 ⇒ 0, 0

Figure 18: Enhanced Data Weighted Averaging (Element Selection Circuit)

The correspondence between the EDWA circuit state *$I_1(n)$*, *$I_2(n)$* and frame clock, the input data *$X(n)$* and the output signals sign *$S(n)$*, zero *$Z(n)$* and enable 1..4 *$E_i(n)$* is defined by a page of boolean equations written in VHDL. These equations are very similar to those of the standard DWA circuit but toggling between odd and even frames has to be taken into account. A synthesis tool implements this set of equations as a combinatorial matrix which is optimised very efficiently. This leads to a very compact and area efficient digital circuit implementation especially suitable for multi-channel devices.

The proposed kind of data weighted averaging which eliminates not only the effect of mismatch on linearity but also on absolute gain invalidates the usually very stringent specifications for capacitor matching. This allows the usage of very small unity capacitors even in the feedback path of the D/A converter. In system simulations a capacitor mismatch of up to 5% has been simulated and it did not have a significant influence on the signal distortion. The absolute gain showed a very little sensitivity to capacitance mismatch, too.

However, it has to be noted that group swapping cannot eliminate the effect of group mismatch on the DAC gain completely. If the mismatch between the two groups is very big then higher order terms of the error propagation function cannot be neglected. They are not identical for the numerator and denominator of a quotient.

Two simple examples: (3/4 + 4/3) / 2 = 1.041666666667, (99/100 + 100/99) / 2 = 1.000050505051. In the first case a group mismatch of 33% is reduced by a factor of 8 to a gain mismatch of about 4% by group swapping. In the second case 1% group mismatch is reduced to 0.005% gain error, an improvement by a factor of 200. Therefore a remaining gain error can be neglected as long as the group matching is not extremely bad.

Gain and Linearity Compensated D/A Converter with Integrated Low-Pass Filter

The actual DAC circuit which is controlled by the EDWA circuit described above is very similar to the one shown already in Figure 16. But there are some differences. In order to swap the two capacitor groups in odd and even clock frames more switches are required. Each capacitor bottom plate must be connected to three switches. One connects it to ground (normally activated at PH1) the second connected to the amplifier output (activated at PH2 if the group is the denominator group) and the third connected to V_{ref} (for positive values activated at PH2 if the group is the numerator group).

Figure 19 shows this stage in a single ended configuration. A corresponding differential version is obvious. The output voltage DAC_{out} is defined by the sequence and activity of its clock inputs PHA_i, PHB_i, PHC_i. In total these are 24 clock lines which are generated by an optimised array logic created by the synthesis tool.

The correspondence between these 24 control lines $PHA_{1..8}$, $PHB_{1..8}$ and $PHC_{1..8}$ and the master clock phases PH1 and PH2 (refer to Figure 16) depending on data enables $E_i(n)$, zero flag $Z(n)$, sign bit $S(n)$ and LSB of frame counter is shown in Table 4 and Table 5. The DWA unit which generates the control signals taking group swapping into account is shown in Figure 18.

The clock signals $PHA_{1..8}$ are used to connect the capacitors to V_{cm} (analogue ground). During normal operation with positive input values they all are connected to PH1, the pre-charge clock phase. The clock signals $PHB_{1..8}$ are used to connect to V_{ref} (input voltage of summing stage), while signals $PHC_{1..8}$ are used to connect to the OPAMP output closing the feedback loop.

During normal operation with positive input values all PHC signals of a complete group are connected to PH2, the evaluation clock phase. This makes the involved capacitors feedback capacitors, their values contributing to the denominator of the gain term.

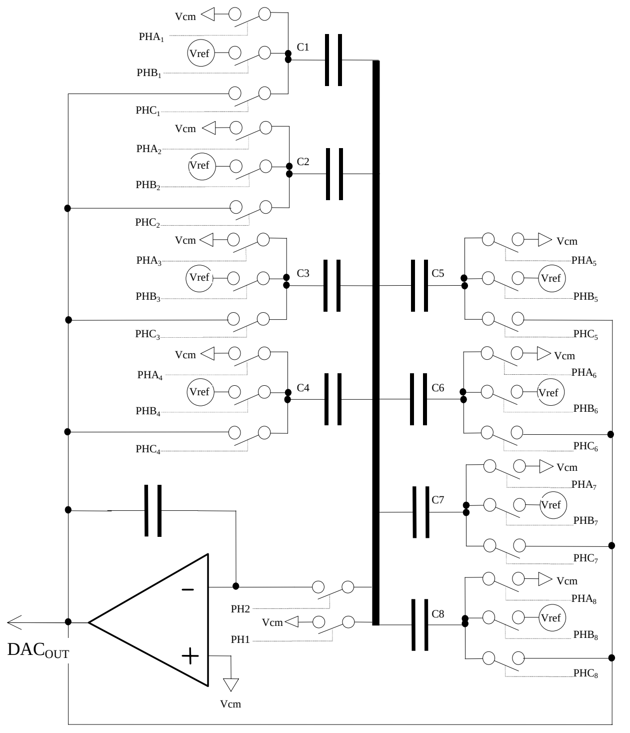


Figure 19: Nine level DAC with Low-Pass Function

At the same time PHB signals of a data dependent number of capacitors of the other group are also connected to PH2. This makes a data dependent number of unit capacitors the input group which contributes to the numerator of the gain term. For negative data values pre-charge and evaluation phase connected to this input group capacitors are swapped.

The Table 4 and Table 5 below contain the complete connection maps taking the special zero case (with two alternatives) and group swapping into account. Table 4 and Table 5 are identical except that group 1..4 and group 5..8 are exchanged. One is applied in even clock frames the other one in odd.

The tables can directly be used to generate the CASE statements with the signal assignments used to describe such a structure in VHDL. Schematic diagrams or clock timing sheets would be voluminous and confusing and have therefore not been used.

	positive (S=0)	negative (S=1)	zero (Z=1)
$PHA_{1..4}$	on during PH1	on during PH1	on during PH1
$PHA_{5..8}$	on during PH1	on during PH2 if $E_{1..4}$	on during PH1 (also PH2)
$PHB_{1..4}$	off	off	off
$PHB_{5..8}$	on during PH2 if $E_{1..4}$	on during PH1	off
$PHC_{1..4}$	on during PH2	on during PH2	on during PH2
$PHC_{5..8}$	off	off	off

Table 4: SC-Clock Assignment in Even Frames

	positive (S=0)	negative (S=1)	zero (Z=1)
$PHA_{1..4}$	on during PH1	on during PH2 if $E_{1..4}$	on during PH1 (also PH2)
$PHA_{5..8}$	on during PH1	on during PH1	on during PH1
$PHB_{1..4}$	on during PH2 if $E_{1..4}$	on during PH1	off
$PHB_{5..8}$	off	off	off
$PHC_{1..4}$	off	off	off
$PHC_{5..8}$	on during PH2	on during PH2	on during PH2

Table 5: SC-Clock Assignment in Odd Frames

During even frames C1 .. C4 are switched into the feedback branch while a programmable selection of C5 .. C8 creates the actual output value. During odd frames two capacitor groups are swapped. C5 .. C8 are switched into the feedback branch while a selection of C1 .. C4 defines the output value.

The zero case can be viewed as a special positive case with zero input magnitude. During PH1 the numerator group (C1..C4 in Table 5, C5..C8 in Table 4) is discharged (by PHA_i) like in the positive case. At PH2 the enabled elements of this group are normally connected to the reference voltage (by PHB_i) to divide it by the proper factor. In the zero case the complete group should again be connected to ground instead (by PHA_i) which is indicated by the expression given in parenthesis. This action is not mandatory because no charges should be transferred in the ideal case but it is recommended because it compensates clock feed-through of other switches. The table entry therefore means $PHA_{1..4}$ (in odd frames) respective $PHA_{5..8}$ (in even frames) should be connected to a logical or of PH1 and PH2 in the zero case.

PH1 and PH2 can most easily be derived from an 8 MHz master clock with a duty cycle of 25% each. Alternatively these clocks could be generated from the 4 MHz sampling clock using certain cross-coupled gating to enforce non overlap at slightly less than 50% duty cycle.

Using this DAC in conjunction with the enhanced DWA (Data-Weighted-Averaging) algorithm described in the previous section all capacitor mismatches are eliminated. Not only the linearity is improved by the averaging but also the absolute gain. The average value of the feed back capacitor is exactly four times the mean value of all eight unit capacitors making the gain of the DAC perfectly unity.

Simulation Example

The command line below was used to generate the simulation results shown in Figure 20. The contents of the input file dac2bin.dat are listed in Table 6. To keep the table small linefeeds have been replaced by spaces.

```
dac2bin -noc -vref 1 -fb 10 < dac2bin.dat | plot
```

The file was generated by a C-program implementing Table 4 and Table 5. It is listed in Table 7. To keep it simple, no dynamic averaging other than group swapping was implemented. The main program generates one period of a 1000 Hz sine wave sampled at 64 kHz. The low-pass filter has been readjusted by the -fb option while the reference voltage -vref has been set to 1V.

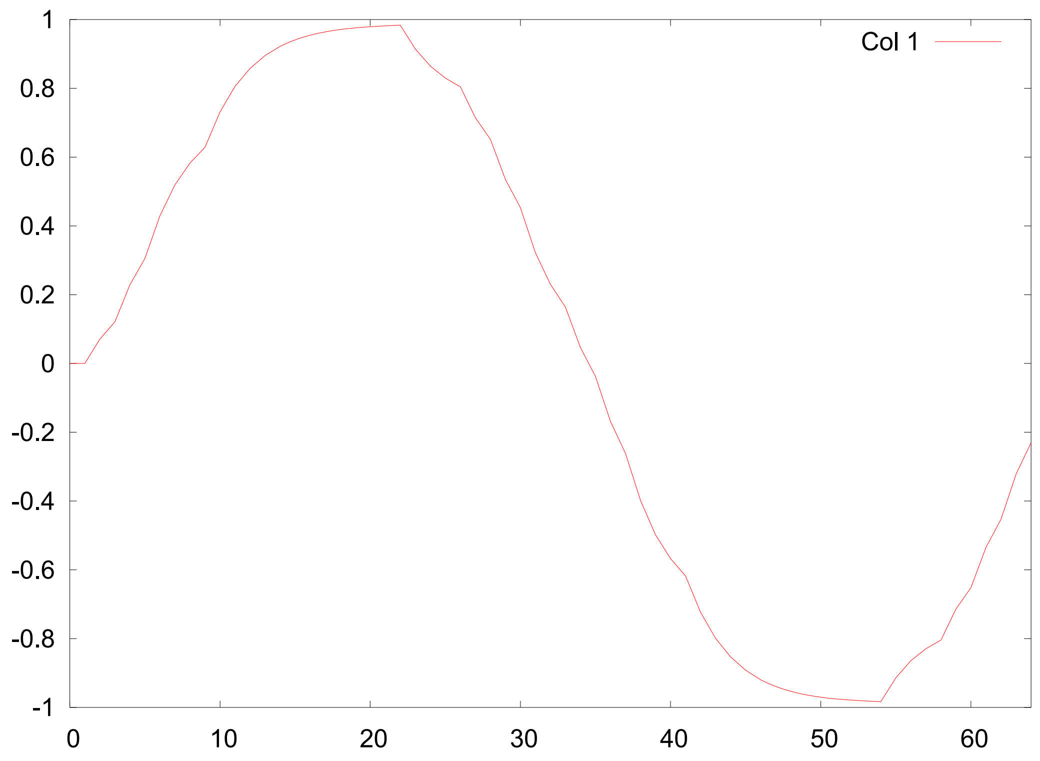


Figure 20: Simulation of Switched-Capacitor DAC

```
ff00001 0f00f00 ff00001 f0000f0 ff00001 0001f00 ff00001 00100f0 ff00001 0003f00 ff00001 00300f0
ff00001 0007f00 ff00001 00700f0 ff00001 0007f00 ff00001 00700f0 ff00001 000ff00 ff00001 00f00f0
ff00001 000ff00 ff00001 00f00f0 ff00001 000ff00 ff00001 00f00f0 ff00001 000ff00 ff00001 00f00f0
ff00001 000ff00 ff00001 00f00f0 ff00001 000ff00 ff00001 00f00f0 ff00001 000ff00 ff00001 00700f0
ff00001 0007f00 ff00001 00700f0 ff00001 0007f00 ff00001 00300f0 ff00001 0003f00 ff00001 00100f0
ff00001 0001f00 ff00001 f0000f0 ff00001 0f00f00 ff00001 f0000f0 f00f001 0100f00 0ff0001 10000f0
f00f001 0300f00 0ff0001 30000f0 f00f001 0700f00 0ff0001 70000f0 f00f001 0700f00 0ff0001 70000f0
f00f001 0f00f00 0ff0001 f0000f0 f00f001 0f00f00 0ff0001 f0000f0 f00f001 0f00f00 0ff0001 f0000f0
f00f001 0f00f00 0ff0001 f0000f0 f00f001 0f00f00 0ff0001 f0000f0 f00f001 0f00f00 0ff0001 f0000f0
f00f001 0f00f00 0ff0001 70000f0 f00f001 0700f00 0ff0001 70000f0 f00f001 0700f00 0ff0001 30000f0
f00f001 0300f00 0ff0001 10000f0 f00f001 0100f00 ff00001 f0000f0 ff00001 0f00f00
```

Table 6: dac2bin.dat

```
#include <stdio.h>
#include <math.h>

int PHAL (int c, int e) {
 if (!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?15:((e<0)?15:15); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?15:((e<0)?0:15); else/*PH2*/ return (e>0)?0:((e<0)?abs(e):15);}};

int PHAH (int c, int e) {
 if(!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?15:((e<0)?0:15); else/*PH2*/ return (e>0)?0:((e<0)?abs(e):15);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?15:((e<0)?15:15); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}};

int PHBL (int c, int e) {
 if(!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?15:0); else/*PH2*/ return (e>0)?abs(e):((e<0)?0:0);}};

int PHBH (int c, int e) {
 if(!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?15:0); else/*PH2*/ return (e>0)?abs(e):((e<0)?0:0);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}};

int PHCL (int c, int e) {
 if(!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?15:((e<0)?15:15);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}};

int PHCH (int c, int e) {
 if(!((c>>1)&1)) { // even frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?0:((e<0)?0:0);}
 else { // odd frames
  if (!(c&1))/*PH1*/ return (e>0)?0:((e<0)?0:0); else/*PH2*/ return (e>0)?15:((e<0)?15:15);}};

void DACLine (int c, int e, FILE* fout) {
 fprintf(fout,"%x%x%x%x%x%x%x\n",
  PHAL(c,e),PHAH(c,e),PHBL(c,e),PHBH(c,e),PHCL(c,e),PHCH(c,e),!(c&1));};

void DACPoint (int* LineCnt, int e, FILE* fout) {
 if (e == -4)      e = -15;
 else if (e == -3) e = -7;
 else if (e == -2) e = -3;
 else if (e == 2)  e = 3;
 else if (e == 3)  e = 7;
 else if (e == 4)  e = 15;
 DACLine((*LineCnt)++, e, fout);
 DACLine((*LineCnt)++, e, fout);};

int main() {
 int LineCnt = 0;
 double t = 0;
 while (t < 0.001) {
  DACPoint(&LineCnt, (int)floor(0.5+4.5 * sin(2.0*3.141592*1000.0*t)), stdout);
  t += (1./64000.);}
 return(0);}
```

Table 7: C-program to generate ramp

Anhang C: TAP Macro Scripts

Harmonic Analysis on Scales (harmonics.ana)

This macro script performs analysis of harmonic content over time or tone step. Available targets are for segmentation, overload recognition, static and dynamic partial analysis and plot generation.

The ***RMS*** target creates and displays the running RMS which is the base of subsequent segmentation. The RMS display is useful in determining a suitable segmentation threshold which has to be entered using the *min=* option when the actual segmentation has to be performed.

The ***Overflow*** target creates a display to check if the input wave file is overloaded and contains clipping in some regions. Harmonic analysis results should be neglected in regions marked as overloaded.

The ***Mask*** target displays the running RMS together with the segmentation mask where the level exceeds the specified minimum. The Mask display is useful to crosscheck the segmentation mask.

The targets ***SegmentRMS*** and ***SegmentOverflow*** are similar to the corresponding targets above but the segmentation has already been performed and only the masked regions are displayed.

The targets ***Stat*** and ***Dyn*** display the actual static and dynamic analysis results. The static plots which can also be displayed specifying separate targets are ***StatF0***, ***StatHarm*** and ***StatInharm***.

StatF0 shows the detected fundamental frequency with harmonic and spectral centroid over the tone number. ***StatHarm*** shows the average harmonic levels over the tone number and ***StatInharm*** shows the average inharmonicity over the tone number.

The dynamic analysis targets are ***DynF0***, ***DynHarmLog***, ***DynHarmLin*** and ***DynFreq.***

DynF0 shows the detected fundamental frequency with the harmonic now over time rather than over tone number. ***DynHarmLog*** and ***DynHarmLin*** show the harmonic levels dynamically over time and ***DynFreq*** shows the actual frequencies of the harmonics over time.

The targets ***Clean*** or ***Clear*** remove all intermediate results and the target ***Help*** displays the table with options and targets below.

The most essential option is the ***WAV=*** option which specifies the sound file which is to be analyzed. No extension should be given, because all intermediate result files will get that file name with different terminations and extensions. The ***min=*** option has already been mentioned. It specifies the segmentation threshold amplitude and has to be entered correctly when the actual masking is performed.

The ***f0=*** option guides the fundamental recognition and should be set to the estimated fundamental frequency at the beginning of the sound file. Together with the option ***df=*** which specifies the maximum frequency deviation in % (in upward direction) between two subsequent notes these two values are passed to the -continuous option of the swana command. The macro is setup for rising scales or sweeps only and had to be changed if descending scales are to be analysed.

The options ***hg=*** and ***ex=*** are passed to the -partials option of the swana command. It guides the search for a harmonic grid by specifying how many partials should take part in this search and how their contribution should be weighted. Refer to the swana command in Appendix A for a detailed description of the effect of the -partials option.

The options ***sw=*** and ***dw=*** specify the static and dynamic window size which are used by the Fourier transform in the static and dynamic mode. Especially the static window must be shorter than the actual tone duration in order not to cross the border to the next tone. The static window is stepped tone by tone by the distance related to the ***cut=*** option. The cut specification is used during the segmentation process but it also defines the window stepping distances in static and dynamic mode. In static mode the stepping distance is 100x the cut input, in dynamic mode the stepping distance is the cut input itself. A corresponding ***dx=*** definition must be given in order to scale the x-axes of all dynamic plots properly.

The options ***opt=*** and ***pltopt=*** are forwarded to all plot commands and can be used to select a plot device, to define plot titles, specific plot ranges or other plot related settings. The combination *opt*= “set term postscript landscape color solid; set output '\\\\PDC\\HPLJ4500'" and *pltopt*=-nowait will send the plots to the printer without showing them

- **TAP - Transfer Analysis Package - Harmonic Analysis on Scales** - Version 1.0 by W. Kausel, IWK (**harmonics.ana)**

  Displays harmonic content over time or tone step

Options:
WAV= wave file to process (without .wav extension!)
min= threshold level for segmentation (def=1000)
f0= expected initial base tone frequency
df= maximum frequency deviation in [%] (def=20)
hg= harmonic grid size (def=7)
ex= exponent of weight function (def=0.2)
sw= static window size (def=32768)
dw= dynamic window size (def=4096)
cut= note length in samples / 100 (def=441)
dx= note length in sec / 100 (def=0.01)
opt= plot command option string
pltopt= additional plot options

Targets:
help: displays this help page
clean / clear: remove intermediate/all files
RMS: displays the running RMS over time
Overflow: displays regions with overflowing (> 32766) input levels
SegmentRMS: displays the running RMS over time
SegmentOverflow: displays regions with overflowing input levels
Mask : displays segmentation mask with RMS > min
Stat: static result plots (statf0,statharm,statinharm)
Dyn: dynamic result plots (dynf0,dynharmlog,dynharmlin,dynfreq)
All: static and dynamic result plots

Spectral Dynamic Analysis (specdens.ana)

This macro script primarily analyses the harmonic content over the dynamic level, but it also can do analysis of harmonic content over time without any segmentation on any kind of sound file input. No scale or sweep must be present in the recording. Available targets are for calculation and display of partial frequency variation, partial magnitude variation, partial harmonicity variation and spectral dynamic analysis.

The ***Frequency*** target performs a fundamental extraction by searching for a harmonic grid. The extracted fundamental frequencies together with the frequencies of the partials are plotted over time.

The ***Harmonicity*** target creates a plot where partial frequencies are related to their fundamental frequency so that harmonicity can be seen as harmonic ratios over time.

The ***Magnitude*** target displays the intensity of the partials over time.

The ***SpectralDynamic*** target sorts all harmonic results according to the magnitude of the fundamental and creates a plot which shows harmonic content over the dynamic level. For this analysis a recording of a single note played at varying dynamic levels will be most suitable.

The ***Batch*** target does all the lengthy processing without generating any graphical output. Subsequent calls to the other targets will then be able to immediately present the requested the plots.

The targets ***Clean*** or ***Clear*** remove all intermediate results and the target ***Help*** displays the table with options and targets below.

The most essential option is again the ***WAV=*** option which specifies the sound file which is to be analyzed. No extension should be given, because all intermediate result files will get that file name with different terminations and extensions.

The ***fu=*** and ***fo=*** options are guiding the fundamental recognition. They limit the frequency range where fundamentals are to be expected. They are directly passed to *dftpeaks*. By specifying maximum relative deviations using the options ***lu=*** and ***lo=*** discontinuities in the fundamental frequency curve can be avoided. These limits can be set to small percentage values if constant pitch or slowly varying sweeps are analyzed. They are passed to *dftpeaks -frel lower upper*.

Another option effecting fundamental recognition is the ***min=*** value. It is the threshold for the running rms signal level which must be exceeded in order to enable fundamental recognition. If the signal is below that level, fundamental detection is skipped and the last valid fundamental frequency is used instead. The option value is passed to the *dftpeaks -threshold* option.

The options ***lwin=***, ***lstep=***, ***swin=*** and **sstep=** define the long and short window sizes and stepping distances. The corresponding *dftpeaks* options are *-llen, -lofs, -slen* and *-sofs*. The long window should provide the required frequency resolution while the small window should be small enough for tracking the spectral content in time.

The ***minlev=*** option is used for the SpectralDynamic target to define the left x-axes limit of the displayed level range. The ***dx=*** option is the x-axis resolution of all time domain plots. It has to be set to *sstep / sample*. The sampling rate can be changed using the ***sample=*** option. Its default is 44100 Hz. The option string ***opt=*** is passed to the GNUPlot utility and can be a string like “set terminal postscript landscape color solid; set output '\\\\PDC\\HPLJ4500'". The ***pltopt=*** string is passed to the plot function and could be a string like “-nowait” in case of batch printing.

- **TAP - Transfer Analysis Package - Spectral Dynamic Analysis** - Version 1.0 by H. Nachtnebel, IWK (**specdens.ana)**

  Displays the spectral density over the base tone power.

Options:

WAV=...	wave file to process (without .wav extension!)
sample=...	sampling rate of wave file [44100Hz]
fu=...	lower limit of base tone frequency range
fo=...	upper limit of base tone frequency range
lu=...	rel lower limit of next base tone frequency deviation [3%]
lo=...	rel upper limit of next base tone frequency deviation [15%]
lwin=...	long window size for base tone detection [8192]
lstep=...	long window step for base tone detection [4096]
swin=...	short window size for harmonics detection [2048]
sstep=...	short window step for harmonics detection [1024]
dx=...	X-axis scaling (sstep / sample) [1024/44100sec]
min=...	minimum rms power to calculate f0 [0]
minlev=...	minimum db level for SpectralDynamic
opt=...	plot command option string
pltopt=...	additional plot options

Targets:

Help: displays this help page
Clean / Clear: remove intermediate files
Batch: does the lengthy part without any plot afterwards
Frequency: displays the spectral frequencies found over time
Harmonicity: displays the harmonic ratios found over time
Magnitude: displays the spectral magnitude found over time
SpectralDynamic: displays the spectral density over the base tone power

Sound File Segmentation (segmentation.ana)

This macro script reads an audio file in windows .wav format and detects sound events and periods of silence in order to repartition the file. The script can eliminate all periods of silence creating a sound file with all significant sound events concatenated. The single segments can be trimmed to a given length. It is also possible to cut the input wave file into separate short wave files each containing exactly one significant sound event.

Detecting significant sound events requires an analysis of the RMS sound level first which can be plotted using the ***RMS*** target. The ***Overflow*** target can be used to check whether input data do not exceed the valid audio range.

The most essential option needed by all targets is the ***WAV=*** option which specifies the sound file which is to be analyzed. No extension should be given, because all intermediate result files will get that file name with different terminations and extensions.

Using the options ***on=*** and ***off=*** threshold levels can be specified for the detection of the sound events. The levels should be given such that no significant event fails to reach the threshold but background noise should always be below this level. The RMS plot is a useful tool to determine a suitable threshold level. The off threshold is optional and is set to the on threshold by default.

The ***Mask*** target again plots the running RMS but now with the binary segmentation mask superimposed. This way the segmentation thresholds can easily be checked before further processing. The ***ymax=*** option can be used to adjust the range of all plots. Portions where the segmentation mask is zero will later on be eliminated.

The running RMS is calculated by stepping a user definable window over the audio data stream. The step size is 441 samples corresponding to a time resolution of 10ms or 100 points per second. The window size may be specified using the ***win=*** parameter. Usually the default of 1000 will work well in most cases. For very low frequencies a bigger value might be required if a smooth RMS curve is desired. Shorter windows will better reflect sharp transitions of the signal energy and yield more accurate slew-rates during attack.

Segmentation can be controlled using several options to modify the selected intervals. Using the ***spike=*** and ***gap=*** option minimum segment lengths and minimum segment distances can be specified. All hits which consist of fewer samples than 100 times the given spike length will be ignored and all gaps which consist of fewer samples than 100 times the given gap size will be eliminated from the mask. Detected segments can be augmented using the ***aug=*** option and trimmed to equal length by using the ***cut=*** option. Again the parameter is a number of samples divided by 100.

The ***opt=*** and ***pltopt=*** options allow to forward option strings to the plot command for e.g. PDF printing, scaling or setting plot title, axis labels, display grid etc. The option string ***opt=*** is passed to the GNUPlot utility and can be a string like “set terminal postscript landscape color solid; set output '\\\\PDC\\HPLJ4500'". The ***pltopt=*** string is passed to the plot function and could be a string like “-nowait” in case of batch printing or “-title SegmentationPlot -grid -xunit [Hz]”.

- **TAP - Transfer Analysis Package - <u>Segmentation of wave files</u>** - Version 1.0 by W. Kausel, IWK **(segmentation.ana)**

  Performs wave file segmentation based on sound power detection.

<u>Options</u>:

WAV=...	wave file to process (without .wav extension!)
on=...	on threshold level for segmentation (def=1500)
off=...	off threshold level for segmentation (def=200)
aug=...	augment segments by 100*aug samples (def=10)
spike=...	eliminate spikes shorter than 100*samples (def=20)
gap=...	eliminate gaps shorter than 100*samples (def=20)
win=...	processing window length in audio samples (def=1000)
cut=...	segment length in samples / 100 (def=441, this is 1 sec)
ymax=...	y plot range for rms plots
opt=...	plot command option string
pltopt=...	additional plot options

<u>Targets</u>:

Help: displays this help page
Clean / Clear: remove intermediate files
RMS: displays the running RMS over time
Overflow: displays regions with overflowing (> 32766) input levels
SegmentRMS: displays the running RMS over time after segmentation
SegmentOverflow: displays regions with overflowing input levels
Mask: displays segmentation mask with RMS > on
SegmentWAV: perform segmentation on wave file (time consuming)
CutWAV: store segments in separate wave files
NormaliseWAV: normalise all segments in segmented wave file
CutNorm: store normalised segments in separate wave files
All: perform all operations