`gmeteor` User's Manual

For version 0.95, 22 October 2007

Matteo Frigo

Copyright © 2000 Matteo Frigo.

Copyright © 2000 Biomedin s.r.l.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

1 Introduction

gmeteor is a tool for designing discrete-time equiripple filters with linear phase and a finite impulse response (FIR). gmeteor runs on systems that support the Guile extension language from the GNU project (for example, GNU systems running on a Linux kernel). gmeteor is free software distributed under the terms of the GNU General Public License (GPL). See Chapter 5 [License and Copyright], page 25, for details. gmeteor is a powerful filter-design tool because of the following three reasons.

1. gmeteor can design FIR filters with an arbitrary frequency response. You can even specify the frequency response analytically. In contrast, with other filter-design tools you are limited to piecewise-constant or piecewise-linear frequency responses.
2. gmeteor supports two filter design styles: the *approximation style* and the *limit style*. In the approximation style, you specify the desired frequency response of the filter, and gmeteor finds the “best” feasible approximation. (This style is used by the popular Parks-McClellan filter design program.) In the limit style, you specify upper and lower bounds on the frequency response, and gmeteor finds a frequency response that satisfies the bounds. With gmeteor, you can freely intermix the two styles whenever it makes sense.
3. Since gmeteor employs an extension language (specifically, the Scheme programming language), you can program it to solve your own filter-design problems. For example, I have used gmeteor to find the optimal minimum-length filter satisfying a given set of constraints, to produce many variations of a given filter, and to produce a ‘C’ source file that implements a given filter. Because a full programming language is always available in gmeteor, you can customize the tool to solve your own problems, rather than being limited to what the gmeteor’s author had in mind when he wrote the program.

A *filter*, for the purposes of this manual, is a linear time-invariant signal-processing system that is designed to modify certain frequencies relative to others. For example, a system that eliminates all frequencies above 3500 Hz and does not modify lower frequencies is a “low-pass” filter. (A similar system may be used in your phone.) This manual assumes that you are familiar with filters and discrete-time signal processing (also called “digital signal processing”). Otherwise, many textbooks on the subject are available, such as *Discrete-Time Signal Processing*, by Alan Oppenheim and Ronald Schafer. Without much signal processing knowledge, you can understand this manual and use gmeteor at an intuitive level, but in order to exploit gmeteor fully, a systematic training in the signal-processing field is desirable.

gmeteor is based on the algorithm described in the paper *METEOR: a Constraint-based FIR Filter Design Program,* by K. Steiglitz, T. W. Parks, and J. F. Kaiser, published in IEEE Trans. Signal Processing, vol. 40, no. 8, pp. 1901-1909, August 1992. (Both the paper and the METEOR program are available at http://www.music.princeton.edu/classes/). In this beautiful and very readable paper, Steiglitz et al. reduce the filter design problem to a linear programming problem, which in turn is solved by the simplex algorithm. This filter-design methodology is elegant and very general, but unfortunately, the METEOR program falls short of implementing the methodology in its full generality. For example, the algorithm in the paper can approximate arbitrary frequency responses, but METEOR is restricted to piecewise linear or exponential functions. gmeteor implements the ideas

described in the METEOR paper without imposing arbitrary restrictions on the METEOR algorithm.

`gmeteor` is user-friendly: simple filters can be designed easily, but complicated filters are possible. To design a filter with `gmeteor`, you must first prepare a *specification* file that describes the desired filter. If the specification file is called '`myfilter`' (for example), you design the filter by invoking '`gmeteor myfilter`'. A simple specification file could be the following:

```
(sampling-frequency 8000)
(filter-length 21)

(limit-= (band 0 2000) 1)
(limit-= (band 3000 4000) 0)

(go)
```

The previous specification file describes a low-pass filter of length 21. The sampling frequency is 8000 Hz, the frequency response has magnitude 1 for frequencies between 0 Hz and 2000 Hz, and the frequency response is 0 for frequencies between 3000 Hz and 4000 Hz.

The specification file is actually a program written in the Scheme programming language (which accounts for the many parentheses in the example). You need to know Scheme if you want to use `gmeteor`'s advanced features, but in most cases and for most users, no knowledge of Scheme is required.

The rest of this manual is organized as follows. The tutorial chapter, Chapter 2 [Tutorial], page 3, explains how to use `gmeteor` with a series of examples. The reference chapter, Chapter 3 [gmeteor Reference], page 17, documents `gmeteor` systematically. Chapter 4 [Installation], page 23 explains how to install `gmeteor` on your system. Chapter 5 [License and Copyright], page 25 gives license and copyright information.

2 Tutorial

The goal of this chapter is to show several examples of `gmeteor`'s usage. These examples should be sufficient to let you design simple filters quickly (i.e., within minutes.) Before showing the examples, however, in Section 2.1 [Filter Design], page 3 we briefly review basic concepts about filters and filter design.

2.1 Filter Design

In this section, we briefly review basic concepts about filters and filter design.

A *filter* is a signal-processing system that is meant to modify certain frequencies relative to others. For example, the "tone" control in your CD player is a filter that modifies the relative level of high and low frequencies. It is not fruitful for us to specify more precisely which systems are filters and which are not; the important point is that, in a filter, we mostly care about how the filter reacts to different frequencies.[1]

Researchers have identified many classes of useful filters, and in addition, any given filter can be implemented using a variety of technologies. For example, an old CD player might implement the tone control by means of resistors or capacitors, but a new CD player likely implements the filter in software. In this manual, we do not worry about the implementation technology, and we view the filter as a black box that inputs a certain signal $x(t)$ (a function of time), and outputs another signal $y(t)$.

`gmeteor` designs a specific class of filters, namely, linear time-invariant discrete-time linear-phase filters with a finite impulse response (FIR). For these filters, the time variable t is an integer, and the input and output signals can be viewed as equispaced samples of a continous signal. Other classes of filters are also important and useful, but we do not worry about them here, because `gmeteor` only knows about FIR filters.

A FIR filter can be described by a sequence of l real numbers $h_0, h_1, \ldots, h_{l-1}$, called the *impulse response* of the filter. The number l is called the *filter length*. The relation between the input and the output is expressed by the following formula.

$$y_t = \sum_{i=0}^{l-1} h_i x_{t-i}$$

(This kind of expression is usually called a (linear) *convolution*.)

The *filter coefficients* h_i uniquely determine the *frequency response* $H(f)$ of the filter, which is a complex function of the frequency f. The frequency response is important because of the following two facts. First, every "interesting" signal can be decomposed into a (possibly infinite) sum of sine waves. Second, a FIR filter transforms a sine wave into a sine wave of the same frequency. If the input sine wave has amplitude 1 and frequency f, the output sine wave has frequency f, amplitude $|H(f)|$, and its phase is shifted by $\arg H(f)$ with respect to the input signal. Consequently, by selecting $H(f)$ appropriately, we can design a filter that modifies frequencies in any way we desire.

The function $H(f)$ is the *discrete-time Fourier transform* of the filter coefficients, and well-known ways exist to compute the frequency response from the filter coefficients and vice

[1] In the same way, the meat of a pig becomes "pork" only if you intend to eat it, and it is not fruitful to argue about the precise time when the pig becomes pork.

versa. Unfortunately, it turns out that most frequency responses can only be implemented with a filter of infinite length. If we are to implement these filters in practice, we must settle for an approximation of the desired frequency response. The filter design problem is therefore the problem of finding a "good" approximation to the desired $H(f)$ with a finite (and hopefully small) filter length.

With `gmeteor`, you address the filter design problem by specifying constraints on the desired frequency response and possibly on its first and second derivatives. `gmeteor` then finds the coefficients that best approximate these constraints. More precisely, `gmeteor` designs optimal filters in the Chebyshev sense. (The precise mathematical description the problem solved by `gmeteor` is given in Section 3.1 [Optimal Filters], page 17.) The rest of this chapter explains the various kinds of constraints that `gmeteor` supports, what they mean, and how to specify them.

2.2 Example 1

The purpose of this example is to teach you the basic usage of `gmeteor`. We start with a simple design, a low-pass filter of length 10. The sampling frequency is 60 Hz, the *passband* is [0..10], and the *stopband* is [20..30]. The desired frequency response is 1 in the passband, and 0 in the stopband. (The frequency response in the *transition band* [10..20] is not specified.) This is an instance of the *approximation design style*: we specify the ideal response, and let `gmeteor` approximate it.

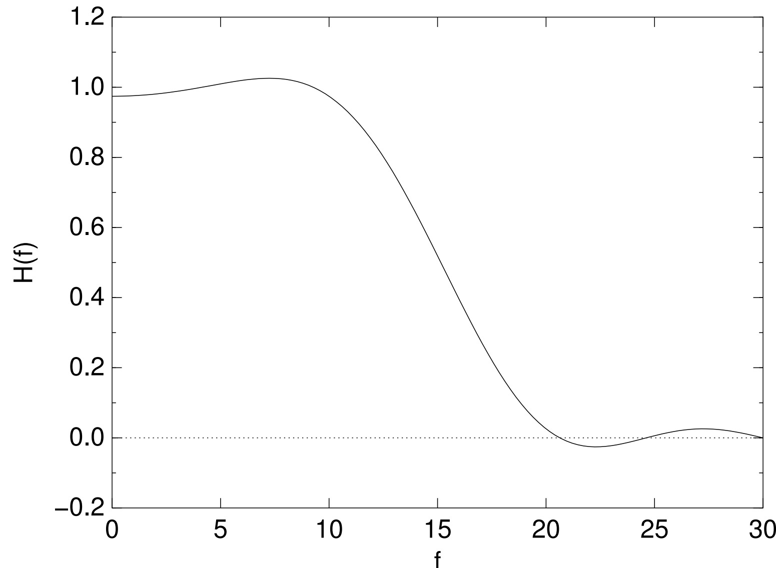
In order to design the filter, you must create a file called '`example-1.scm`' with the following contents.

```
; A simple filter
(title "A simple filter")
(verbose #t)
(cosine-symmetry)
(filter-length 10)
(sampling-frequency 60)

(limit-= (band 0 10) 1)
(limit-= (band 20 30) 0)

(output-file "example-1.coef")
(plot-file "example-1.plot")

(go)
```

Then, you must invoke `gmeteor` as follows.

```
gmeteor example-1.scm
```

You can plot the file '`example-1.plot`', obtaining a graph of the frequency response like the following.



As mentioned in Chapter 1 [Introduction], page 1, the specification file '`example-1.scm`' is a program written in the Scheme programming language, but do not worry if you do not know Scheme. All you need to know is that the file consists of a sequence of commands, and that each command is enclosed within a pair of parenteses, as in `(filter-length 10)`. Do not forget these parentheses, because Scheme requires them. In the rest of this section, we discuss the meaning of each command individually.

The first line

```
; A simple filter
```

is a comment. Comments are introduced by a single semicolon and last until the end of the line. You can add comments anywhere you wish.

The expression `(title "A simple filter")` specifies an optional title for the filter. The title does not affect the computation in any way, and it is used for documentation purposes. The title appears in the output file when `gmeteor` runs in verbose mode.

The command `(verbose #t)` tells `gmeteor` to run in verbose mode. In Scheme, the expression `#t` denotes the boolean true value, while `#f` denotes the false value. In non-verbose mode, `gmeteor` outputs the filter coefficients (and nothing else) to the output file. In verbose mode, `gmeteor` prints additional information such as the title, the sampling frequency, and so on.

The command `(cosine-symmetry)` specifies that the frequency response $H(f)$ is an even function. In linear-phase FIR filters—the only kind `gmeteor` knows about—$H(f)$ is either an even or an odd function of f. A function $H(f)$ is *even* whenever $H(f) = H(-f)$, and *odd* whenever $H(f) = -H(-f)$. To obtain an odd frequency response, use the command `(sine-symmetry)`. If neither symmetry is specified, `(cosine-symmetry)` is the default.

The command `(filter-length 10)` specifies the length of the filter. In this case, the designed filter has 10 coefficients.

The command `(sampling-frequency 60)` sets the sampling frequency to 60 Hz.

The command `(limit-= (band 0 10) 1)` specifies the passband constraint: the frequency response is 1 for frequencies in the band [0..10]. The expression `(band A B)` creates a *band*, i.e., the interval of frequencies from `A` to `B`. The expression `(limit-= band value)` specifies that the frequency response should be equal to *value* in the band *band*. (`gmeteor` provides additional constructs `(limit-<= band value)` and `(limit->= band value)` to set upper and lower bounds to the frequency response, respectively. See Section 2.7 [Example 6], page 12.)

In a similar fashion, the command `(limit-= (band 20 30) 0)` specifies the stopband constraint, namely, the frequency response should be 0 in the stopband [20..30].

The command `(output-file "example-1.coef")` sets the name of the output file, which contains the filter coefficients. In verbose mode, the output file also contains additional information. If the output file is not specified, `gmeteor` prints to standard output. If the output file is '`#f`', `gmeteor` prints nothing.

The command `(plot-file "example-1.plot")` sets the name of the plot file, which contains a sequence of (x, y) pairs, where $y = H(x)$ and H is the frequency response. You can use '`gnuplot`' or '`graph`' to view the plot file. (See Info file '`plotutils`', node '`graph`'.) If the plot file is not specified, `gmeteor` does not produce a plot.

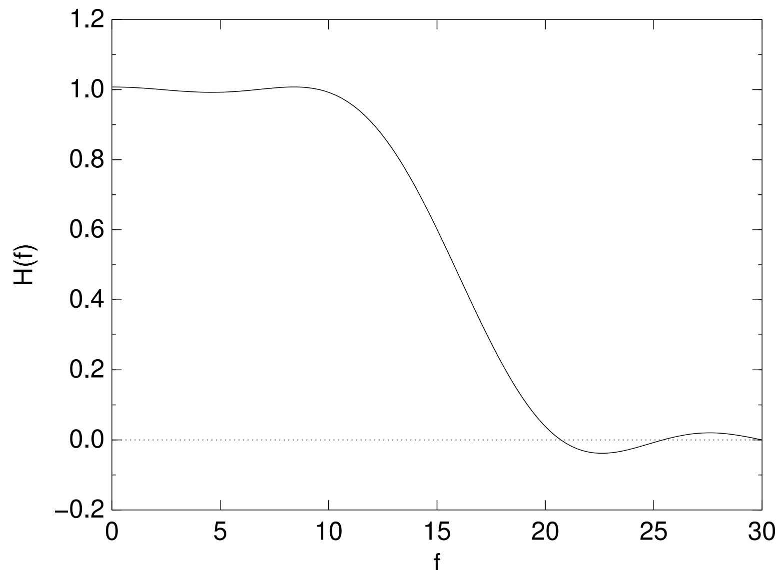
Finally, the command `(go)` tells `gmeteor` to compute the filter and produce the desired output files. Do not omit this line, otherwise `gmeteor` will exit silently without doing anything.

2.3 Example 2

In this example, we introduce the concept of *weight*. Recall that in most cases, the desired frequency response can only be approximated. It turns out that the "optimal" approximation exhibits a *ripple* around the desired frequency response. Using weights, you can decrease the amplitude of the ripple within certain bands at the cost of a larger ripple amplitude in other bands. Specifically in this example, we want to reduce the approximation error in the passband at the expense of a larger error in the stopband, in such a way that the maximum error in the passband is one fifth of the maximum error in the stopband. This goal can be attained with the following specification file.

```
; parameters as in Example 1
(title "A simple filter II")
(verbose #t)
(cosine-symmetry)
(filter-length 10)
(sampling-frequency 60)
(output-file "example-2.coef")
(plot-file "example-2.plot")

; new specifications
(limit-= (band 0 10) 1 .2)   ; .2 is the weight
(limit-= (band 20 30) 0)     ; no weight specified ==> weight = 1

(go)
```

A graph of the frequency response follows.



Intuitively, this specification says that the passband [0..10] has weight .2, and therefore the passband error will be one fifth of the error in the stopband [20..30], where the weight is 1. `gmeteor` always finds the filter that minimizes the error over the whole frequency range. Unless you specify weights explicitly, `gmeteor` distributes the error uniformly over all bands. If you do specify weights, however, `gmeteor` spreads the error in each band according to its weight.

The previous intuitive explanation suffices for simple cases, but we need a more precise definition of a *weight* in order to design more complicated filters. To this extent, we now discuss the operation of `gmeteor` in more detail.

`gmeteor`'s ultimate task is to find a frequency response $H(f)$ that satisfies certain upper- and lower-bound constraints. For example, we might specify that $H([0..10]) \leq 1.1$ and that $H([0..10]) \geq 0.9$. Whenever you specify equality constraints with the `limit-=` command, `gmeteor` internally converts them into a pair of inequality constraints. In our example, `gmeteor` converts the specifications into the four constraints $H([0..10]) \leq 1$, $H([0..10]) \geq 1$, $H([20..30]) \leq 0$, and $H([20..30]) \geq 0$. Since these constraints are impossible to satisfy exactly with a FIR filter, `gmeteor` somewhat relaxes these constraints in order to compute the "optimal" approximation.

The constraint-relaxation algorithm works by introducing a *deviation parameter*, denoted by y. Specifically in our example, `gmeteor` rewrites the four constraints in this way: $H([0..10])+w_1 y \leq 1$, $H([0..10])-w_2 y \geq 1$, $H([20..30])+w_3 y \leq 0$, and $H([20..30])-w_4 y \geq 0$. (Note that the sign in front of $w_i y$ depends on the sense of the inequality.) The parameters w_i are the weights. In our example, $w_1 = w_2 = .2$ and $w_3 = w_4 = 1$.

The *optimal* filter is defined as the one that maximizes y.

The optimal y is negative whenever constraints are violated, as in our example. Since the absolute value of y denotes the maximum weighted distance between the desired and the actual frequency responses, maximizing y produces a filter with the minimum error, which

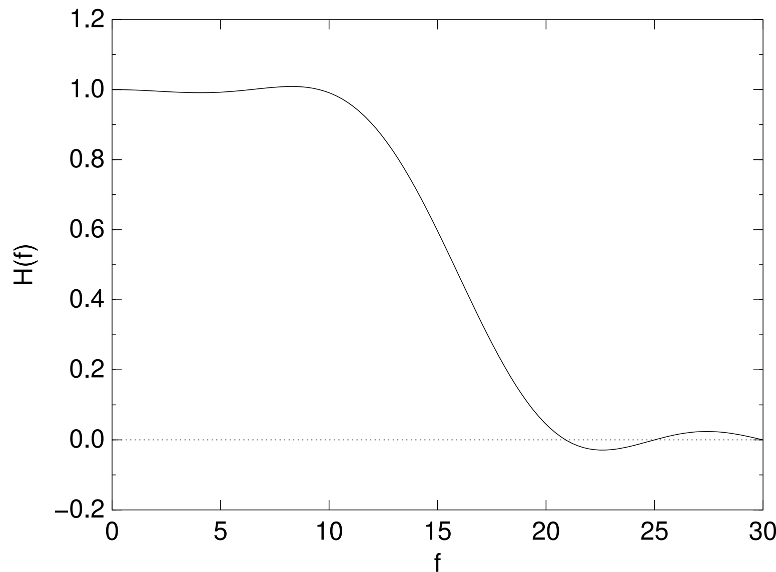
is what we want. Designs where the optimal y is negative are instances of the approximation design style. (In Section 2.7 [Example 6], page 12, we show an example of a limit-style design where the constraints are not violated and y is positive. Even in that case, maximizing y turns out to be the right design criterion.)

2.4 Example 3

This example introduces *point constraints*. Even though a frequency response can only be approximated over a whole band, it is usually possible to specify an exact value of $H(f)$ for a single frequency f, or for a finite set of frequencies. In this example, we design the same filter as in Section 2.3 [Example 2], page 6, but in addition, we specify that $H(0) = 1$ and $H(25) = 0$ exactly. These constraints are useful whenever we have noise at a known frequency (in our case, 25 Hz) that we want to eliminate completely. The following specification file describes this design.

```
(title "A simple filter III")
(verbose #t)
(cosine-symmetry)
(filter-length 10)
(sampling-frequency 60)
(limit-= (band 0 10) 1 .2)
(limit-= (band 20 30) 0)

;; We now constrain H(0) to be exactly 1.  We accomplish this effect
;; by setting weight = 0, and specifying the constraint for a point,
;; not for a band.
(limit-= 0 1 0)

;; another point constraint at 25 Hz.  H(25) = 0.
(limit-= 25 0 0)

(output-file "example-3.coef")
(plot-file "example-3.plot")
(go)
```

A graph of the frequency response follows.



A point constraint is specified as in the command `(limit-= 25 0 0)`. Note that we specify a single frequency 25 instead of a band. `gmeteor` interprets this single frequency as the band `(band 25 25)`. Second, we set the weight to 0, so that the constraint cannot be violated. (See Section 2.3 [Example 2], page 6, for a discussion about weights.) With this change, we have $|H(25)| = 0$ exactly, while in Example 2, we had $|H(25)| = 0.008$.

2.5 Example 4

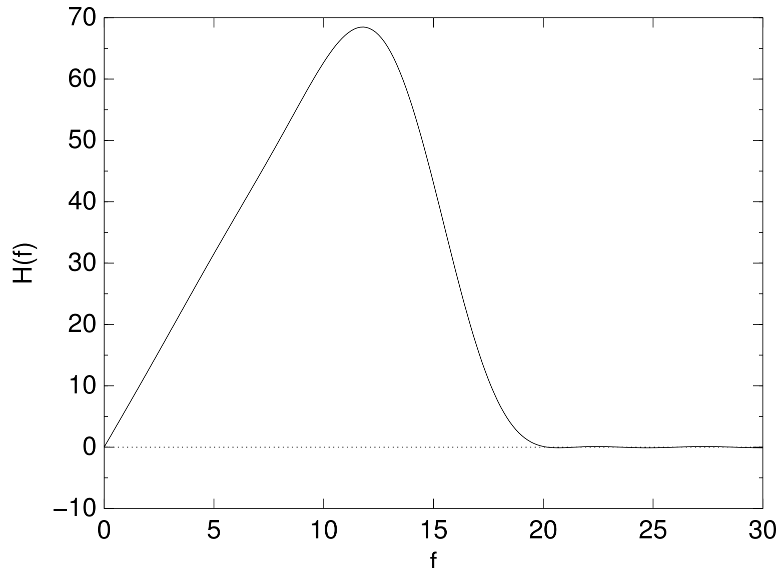
The goal of this example is to teach you some basics of the Scheme programming language, because `gmeteor` will be much more useful to you once you know Scheme. To learn Scheme, I recommend the book *Structure and Interpretation of Computer Programs*, by Hal Abelson and Gerry Sussman—probably the best Computer Science book ever written.

In this example, we want to design a *partial-band differentiator*, which is a filter with sine symmetry whose frequency response $H(f)$ is proportional to the frequency f in the band [0..10] of interest. In previous examples, the frequency response was constant within this band, but now we want to specify a nonconstant frequency response. To this extent, the bound in `limit-=` will be a *function* rather than a number. The following specification file implements the design.

```
(title "A simple filter IV")
(verbose #t)
(sine-symmetry)
(filter-length 10)
(sampling-frequency 60)
(define 2pi (* 8 (atan 1)))
(limit-= (band 0 10) (lambda (f) (* 2pi f)))
(limit-= (band 20 30) (lambda (f) 0))

(output-file "example-4.coef")
```

```
(plot-file "example-4.plot")
(go)
```

A graph of the frequency response follows.



The "passband" [0..10] is specified by the expression

```
(limit-= (band 0 10) (lambda (f) (* 2pi f)))
```

This expression states that the frequency response must be equal to $2\pi f$ in the band [0..10]. (In previous examples, we used the constant value `1`.) `lambda` is the magic Scheme keyword that creates functions. In Scheme, the syntax `(lambda (`*var*`) `*body*`)` produces a function of the variable *var* that, when applied to an argument, evaluates *body* after binding the variable *var* to the argument. In our case, the variable is `f`, and the body `(* 2pi f)`.

Once you have a function, how do you apply it? If *fun* is a function and *expr* is any expression, the syntax `(`*fun expr*`)` denotes the application of *fun* to the value of *expr*. Functions in Scheme are not restricted to only one argument. For example, `sine-symmetry` is a function of zero arguments, and `limit-=` is a function of two arguments. Indeed, all expressions that we called *commands* in Section 2.2 [Example 1], page 4 are function applications.

You can now play with `gmeteor` and build your own frequency responses. To this extent, you need to know that `+`, `-`, `*`, and `/` are functions of two arguments, so that `(+ 1 2)` evaluates to `3`. For example, you can build a filter with response `(lambda (f) (* (* 2pi f) (* 2pi f)))` (a double differentiator). Scheme provides many other primitive functions that you can use, e.g., `sin`, `cos`, `log`, and `exp`. For a complete list of all Scheme primitives, see the paper *Revised^5 Report on the Algorithmic Language Scheme*, by Richard Kelsey, William Clinger, and Jonathan Rees, editors.

2.6 Example 5

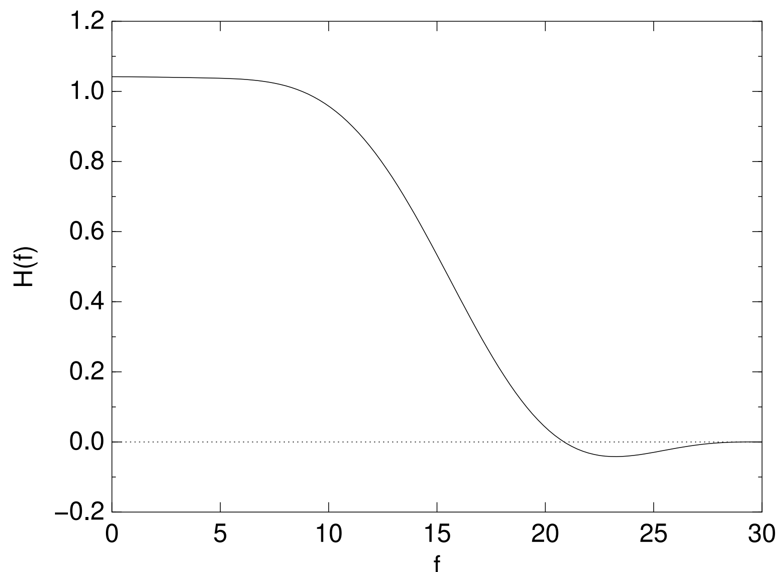
This example introduces *concavity constraints*. With `gmeteor`, you can impose constraints on the second derivative of the frequency response, in addition to constraints on the frequency response itself. These constraints are useful because a frequency response H that is

either concave-up (i.e., $H'' \geq 0$) or concave-down (i.e., $H'' \leq 0$) in a certain band tends to be "flat" in that band (i.e., it does not exhibit ripples). (You can also impose constraints on the first derivative, but they do not appear to be very useful.)

In this example, we design the low-pass filter from Section 2.2 [Example 1], page 4, but in addition we want a flat passband response. To accomplish this effect, we constrain the response to be concave-down in the passband.

```
(title "A simple filter V")
(verbose #t)
(cosine-symmetry)
(filter-length 10)
(sampling-frequency 60)

;; magnitude constraints
(limit-= (band 0 10) 1)
(limit-= (band 20 30) 0)

;; This command states that H(f) must be concave down in the passband.
;; In other words, we demand that H''(f) <= 0 for f in the passband.
(concave-down (band 0 10))

(output-file "example-5.coef")
(plot-file "example-5.plot")
(go)
```

A graph of the frequency response follows.



The expression `(concave-down band)` specifies that the frequency response should be concave-down in the band *band*. The analogous command '`concave-up`' does what you would expect. `gmeteor` also provides two commands '`downward`' and '`upward`' to impose constraints on $H'(f)$.

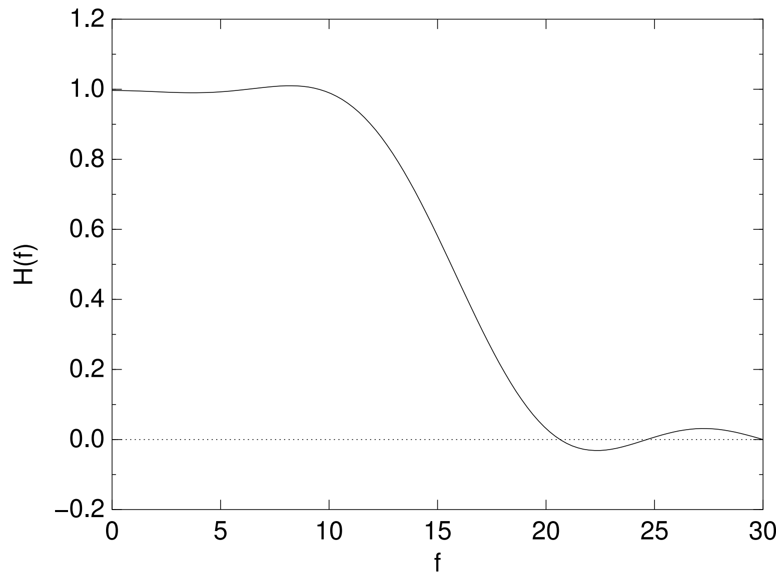
2.7 Example 6

This example discusses an instance of the limit style of filter design. In previous examples, we specified constraints on the frequency response that were impossible to satisfy. `gmeteor`'s goal was to find the feasible frequency response with the smallest deviation from the constraints. In this example, we change our approach: we specify constraints that *can* be satisfied, and let `gmeteor` find a frequency response that satisfies the constraints.

We design a low-pass filter with the following properties: $0.99 \leq H(f) \leq 1.01$ in the passband [0..10], and $|H(f)| \leq 0.1$ in the stopband [20..30]. The stopband ripple must be as small as possible, but we let the passband ripple touch the constraints if necessary. These goals are accomplished by the following specification file.

```
;; Constraint-based filter design.

(title "A simple filter VI")
(verbose #t)
(cosine-symmetry)
(filter-length 10)
(sampling-frequency 60)

;; the passband response is constrained within [0.99, 1.01], and it
;; is allowed to touch the constraints because the weight is 0.
(limit-<= (band 0 10) 1.01 0)
(limit->= (band 0 10) 0.99 0)

;; the stop response is constrained within [-0.1, 0.1], and it
;; is pushed away from the constraints as far as possible, because
;; the weight (1) is nonzero.
(limit-<= (band 20 30) 0.1)
(limit->= (band 20 30) -0.1)

(output-file "example-6.coef")
(plot-file "example-6.plot")
(go)
```

A graph of the frequency response follows.



The expression `(limit-<= band value)` specifies that the frequency response should be at most *value* in the band *band*. Similarly, the expression `(limit-> band value)` establishes a lower bound.

How does `gmeteor` react to this specification file? Recall, from the discussion about weights in Section 2.3 [Example 2], page 6, that `gmeteor` converts the specifications into the four inequalities $H([0..10]) \leq 1.01$, $H([0..10]) \geq 0.99$, $H([20..30]) + y \leq 0.1$, and $H([20..30]) - y \geq -0.1$. (The weight is zero for the first two inequalities, and therefore the deviation parameter y does not appear in them.) Recall also that `gmeteor` designs $H(f)$ so as to maximize the deviation parameter y. In our case, y can be interpreted as "distance from the stopband bounds," and its maximum value is $y = 0.069$ (as shown in the output file). Because the passband weights are 0, `gmeteor` does not attempt to push $H(f)$ away from the bounds in the passband, and the passband ripple amplitude turns out to be exactly 0.01.

This style of constraint-based filter design is the one described in the METEOR paper. In fact, METEOR does not allow equality constraints, nor does it allow negative values of y. I find the approximation style easier in many cases, because it requires only one constraint instead of a pair of upper and lower bounds. On the other hand, the limit approach is more powerful, as shown in the METEOR paper. Consequently, `gmeteor` supports both styles, and you can choose the design approach that best suits your problem.

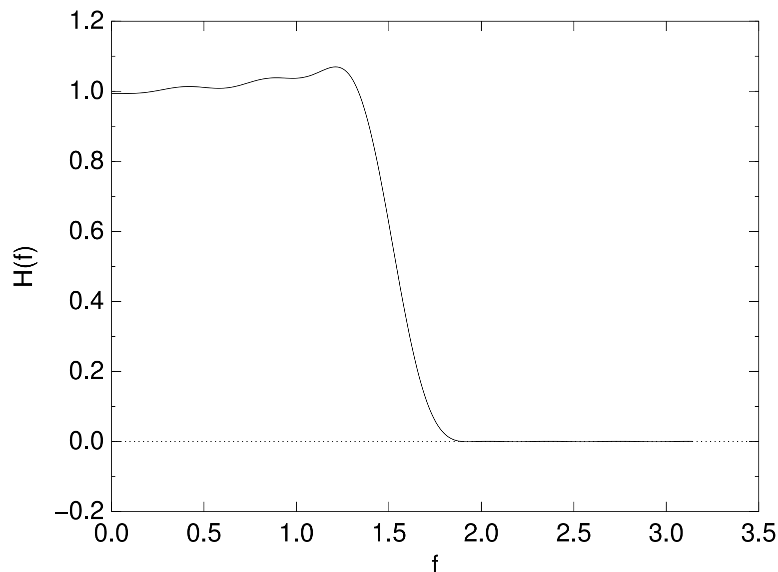
2.8 Example 7

In this example, we design the zero-order-hold compensator example from Oppenheim and Schafer, *Discrete-Time Signal Processing*, 1st ed., Section 7.7.2.

The specifications are as follows. The sampling frequency is 2π, the passband is $[0..0.4\pi]$, the stopband is $[0.6\pi..\pi]$. The frequency response is not flat in the passband, but it has the form $H(f) = (f/2)/(\sin(f/2))$. (See Oppenheim and Schafer for why you may want such a filter.) The stopband error is 1/10 of the passband error. The specification file follows.

```
(title "Compensation for Zero-Order Hold")

(cosine-symmetry)
(filter-length 29)

(define pi (* 4 (atan 1)))   ; pi = 3.14...
(define 2pi (* 2 pi))        ; 2pi = 2 * pi
(define (*2pi x) (* 2pi x))  ; a function that multiplies x by 2pi

(sampling-frequency (*2pi 1))
(define passband (band 0 (*2pi 0.2)))
(define stopband (band (*2pi 0.3) (*2pi 0.5)))

(limit-= passband (lambda (f)
                    (if (= f 0)
                        1
                        (/ (/ f 2) (sin (/ f 2))))))

(limit-= stopband 0 .1)

(output-file "example-7.coef")
(plot-file "example-7.plot")

(go)
```

A graph of the frequency response follows.



This example does not really introduce any new `gmeteor` concept, but it shows some more useful Scheme constructs.

The expression `(define var value)` defines a new variable *var* with value *value*. The similar expression `(define (var arg) body)` defines a function, and it is equivalent to `(define var (lambda (arg) body))`.

A variable can contain an arbitrary object. In particular, the expression

```
(define passband (band 0 (*2pi 0.2)))
```

defines the variable `passband` and sets it value to the *band* `(band 0 (*2pi 0.2))`. This construct is useful if the passband is used in many places.

The expression `(if test if-expr else-expr)` denotes a conditional expression. In our example, we use it to avoid division by 0.

3 gmeteor Reference

[This chapter is in progress.]

This chapter documents all the functionality of `gmeteor`.

To run `gmeteor`, you must prepare a *specification file*, which is a Scheme program that calls `gmeteor` as a library. `gmeteor` provides two alternative interfaces that the specification file can use. The *simple* interface (the one described in Chapter 2 [Tutorial], page 3) is easy to use, but it lets you design only one filter per invocation of `gmeteor`. You will probably use this interface in most cases. The *message-passing* interface is more complicated, but it lets you design many filters at the same time. In this interface, each filter is an abstract data type (an "object", if you wish) that reacts to messages.

The rest of this chapter is organized as follows. Section 3.1 [Optimal Filters], page 17 describes the filter-design problem solved by `gmeteor` precisely. Section 3.2 [Simple Interface], page 18 documents `gmeteor`'s simple interface. Section 3.3 [Invoking gmeteor], page 20 describes `gmeteor`'s command-line options, which are also part of the simple interface. Finally, Section 3.4 [Message-Passing Interface], page 21 documents the message-passing interface.

3.1 Optimal Filters

This section describes the algorithm used by `gmeteor` to design a filter. This algorithm was originally described in the paper *METEOR: a Constraint-based FIR Filter Design Program*, by K. Steiglitz, T. W. Parks, and J. F. Kaiser, published in IEEE Trans. Signal Processing, vol. 40, no. 8, pp. 1901-1909, August 1992.

The inputs to the algorithm are the *filter length* l, the *symmetry* (either sine or cosine symmetry), the *sampling frequency*, the *grid density*, and a list of *specifications*. Each specification consists of a *band* B, a *weight* $w(f)$, and a *constraint* that has one of the following six forms: $H(f) \leq u(f)$, $H(f) \geq u(f)$, $H'(f) \leq 0$, $H'(f) \geq 0$, $H''(f) \leq 0$, or $H''(f) \geq 0$. We think of each specification as a compact representation for an infinite number of constraints, one for each frequency f in the band B.

The algorithm begins by rewriting the specifications as follows. Constraints of the form $H(f) \leq u(f)$ are rewritten as $H(f) + w(f)y \leq u(f)$. Constraints of the form $H(f) \geq u(f)$ are rewritten as $H(f) - w(f)y \geq u(f)$. In the other four cases (specifications on the derivatives of $H(f)$) the constraint is left unchanged and the weight is ignored.

Next, from the filter length, symmetry, and sampling frequency, `gmeteor` derives an analytic expression for $H(f)$ in terms of the *filter coefficients* h_i. This analytic expression is the discrete-time Fourier transform of the h_i's, but without diving into too many details, we can simply state that $H(f)$ has the general form $H(f) = \sum_i h_i t(i, f)$, where t is some function. The important property is that, for a fixed f, $H(f)$ is a linear combination of the filter coefficients.

At this point, `gmeteor` solves the following optimization problem: find the filter coefficients h_i's so as to maximize y, subject to the constraints rewritten as discussed above. This optimization problem has an infinite number of constraints. To solve it, `gmeteor` samples the constraints on a finite grid of frequencies. The grid is determined by the `grid-density` parameter, which specifies the number of grid points in the range $[0..F/2]$. After this sampling, the filter-design problem has been reduced a linear-programming problem, which `gmeteor` solves by means of the dual simplex algorithm. (I believe it is possible to

solve the unsampled continuous problem using symbolic algebra and a modification of the revised dual simplex algorithm, but the finite algorithm works well already.)

3.2 Simple Interface

This section describes `gmeteor`'s *simple interface*. The interface consists of a library of Scheme functions that you can call, and of some global variables that you can access. For ease of reference, we subdivide the functions into three sets. The first set of functions (see Section 3.2.1 [Filter Parameters], page 18) lets you specify filter parameters such as the filter length, symmetry, and sampling frequency. The second set (see Section 3.2.2 [Specifications], page 18) is used to specify the desired bounds on the frequency response. The third set (see Section 3.2.3 [Miscellaneous Commands], page 19) contains miscellaneous functions that control the format of the output and similar parameters.

3.2.1 Filter Parameters

`filter-length` *n* [Command]

This command sets the filter length to *n*. The filter length is the number of filter coefficients.

`sampling-frequency` *f* [Command]

This command sets the sampling frequency to *f*. All other frequencies in a specification file must belong to the band $[0..f/2]$.

`sine-symmetry` [Command]
`cosine-symmetry` [Command]

The command `(sine-symmetry)` specifies that the frequency response $H(f)$ is an odd function, i.e., $H(-f) = -H(f)$. The command `(cosine-symmetry)` specifies that the frequency response $H(f)$ is an even function, i.e., $H(f) = H(f)$. The default is `(cosine-symmetry)`.

`title` *t* [Command]

Set the filter title to *t*, which is a string. The title is used for documentation purposes only. If unspecified, the title equals the string `"UNTITLED"`.

3.2.2 Specifications

`band` *a b* [Function]

Create a *band*, i.e., a representation for the interval of frequencies $[a..b]$.

`limit-<` *band u* [Command]
`limit-<` *band u w* [Command]
`limit-<=` *band u* [Command]
`limit-<=` *band u w* [Command]

These commands establish a specification of the form $H(f) \leq u(f)$ in the band *band*. There is no difference between `limit-<` and `limit-<=`. The weight *w* is an optional function of the frequency f. The default weight is 1. If *band* is a number, it is converted internally into the band consisting of the single frequency *band*. If *u* is a number, it is converted internally into the constant function `(lambda (f) u)`. A similar conversion applies to the weight.

`limit->` *band u* [Command]
`limit->` *band u w* [Command]
`limit->=` *band u* [Command]
`limit->=` *band u w* [Command]

These commands establish a specification of the form $H(f) \geq u(f)$ in the band *band*. The remaining parameters are as in the command `limit-<`.

`limit-=` *band u* [Command]
`limit-=` *band u w* [Command]

Shortcut for the sequence of commands `(limit-< band u w)` and `(limit-> band u w)`.

`upward` *band* [Command]
`downward` *band* [Command]

These commands establish a specification of the form $H'(f) \geq u(f)$ and $H'(f) \leq u(f)$ (respectively) in the band *band*.

`concave-up` *band* [Command]
`concave-down` *band* [Command]

These commands establish a specification of the form $H''(f) \geq u(f)$ and $H''(f) \leq u(f)$ (respectively) in the band *band*.

3.2.3 Miscellaneous Commands

`output-file` *n* [Command]

This command sets the name of the output file to *n*. If *n* is the false value `#f`, gmeteor produces no output. If *n* is the string `"-"`, gmeteor outputs to standard output. The default value is `"-"`.

The output file contains the filter coefficients, and additional information when the verbose mode is on.

`plot-file` *n* [Command]

This command sets the name of the plot file to *n*. If *n* is the false value `#f`, gmeteor produces no plot file. If *n* is the string `"-"`, gmeteor outputs the plot file to standard output. The default value is `#f`.

The plot file contains a series of lines of the form $f H(f)$, for various frequencies f equispaced between 0 and half the sampling frequency. The spacing is controlled by the plot grid density parameter.

`plot-absolute` *b* [Command]

If *b* is the true value `#t`, then `gmeteor` plots the absolute value of the frequency response in the plot file. The default is to plot the signed value, which corresponds to `(plot-absolute #f)`.

`verbose` *b* [Command]

If *b* is the true value `#t`, then `gmeteor` prints the design parameters to the output file, in addition to the filter coefficients. The default is not to print additional information, which corresponds to `(verbose #f)`.

The additional information includes the title, symmetry, filter length, grid density, and the list of specifications.

`grid-density` *n* [Command]

Set the grid density for the filter. `gmeteor` samples all frequencies on a finite grid consisting of *n* frequencies between 0 and half the sampling frequencies. If unspecified, the grid density is about ten times the filter length.

`plot-grid-density` *n* [Command]

Set the plot grid density for the filter. For the purposes of producing a plot file, `gmeteor` samples all frequencies on a finite grid consisting of *n* frequencies between 0 and half the sampling frequencies. If unspecified, the plot grid density equals the grid density.

`go` [Command]

Compute the optimal filter and print the output files. `gmeteor` does not compute anything unless this command is given.

The rationale for an explicit `(go)` command is that you may want to perform some processing after the filter has been computed. For example, you may want to output the filter coefficients using your own routine written in Scheme. Without `(go)`, `gmeteor` would not know whether you are still specifying the filter or whether you need to know the result.

3.2.4 Variables

After the `(go)` command, `gmeteor` stores the result of the computation into certain global variables, which you can access for further processing.

`*result*` [Variable]

The result of the filter optimization problem. This variable contains one of the symbols `'optimal`, `'infeasible`, or `'unbounded`. If the filter was computed correctly, the value is `'optimal`. Otherwise, either there is no solution (`'infeasible`) or there are too many solutions (`'unbounded`).

`*coefficients*` [Variable]

A Scheme vector containing the filter coefficients.

`*y*` [Variable]

The final value of the deviation parameter y.

3.3 Invoking `gmeteor`

The general usage of `gmeteor` is

```
Usage: gmeteor [OPTION...] INPUT-FILE
```

For your convenience, `gmeteor` accepts some parameters both from the command line and from the specification file. In case of conflict, the command-line value takes precedence. A list of options follows.

`-A`
`--plot-absolute`
Same as `(plot-absolute #t)`.

`-d` *n*
`--plot-grid-density=`*n*
Same as `(plot-grid-density` *n*`)`.

`-g` *n*
`--grid-density=`*n*
Same as `(grid-density` *n*`)`.

`-n` *n*
`--filter-length=`*n*
Same as `(filter-length` *n*`)`.

`-o` *file*
`--output-file=`*file*
Same as `(output-file` *file*`)`.

`-p` *file*
`--plot-file=`*file*
Same as `(plot-file` *file*`)`.

`-s` *n*
`--sampling-frequency=`*n*
Same as `(sampling-frequency` *n*`)`.

`-S`
`--plot-signed`
Same as `(plot-absolute #f)`.

`-v`
`--verbose`
Same as `(verbose #t)`.

`-q`
`--quiet` Same as `(verbose #f)`.

`--help` Print a list of options.

`-V`

`--version`
Print `gmeteor`'s version.

3.4 Message-Passing Interface

[To be written]

4 Installation

[This chapter is very preliminary. Please provide feedback if `gmeteor` works for you.]

To install `gmeteor`, you need `guile-1.3` or newer installed in your system. Then type

```
bash$ ./configure
bash$ make
bash$ make install
```

`gmeteor` *works only if it has been installed*, because it needs to load auxiliary Scheme files and a shared library. Sorry about that.

`gmeteor` has been tried succesfully on RedHat Linux 5.2 (with `guile-1.3`), RedHat Linux 6.0 (with `guile-1.3.4`), and Linux PPC (with `guile-1.3.4`).

5 License and Copyright

gmeteor is copyright © 2000 Matteo Frigo, © 2000 Biomedin s.r.l.

gmeteor is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. You can also find the GPL on the GNU web site.

When you design a filter with gmeteor, neither the filter specification nor the filter coefficients consitute a work based on gmeteor. Consequently, the General Public License does not impose any restrictions on the use and distribution of said filter specification and coefficients.

6 Acknowledgments

Thanks to Giuseppe Torresin (my boss) for letting me work on `gmeteor` and release it.

`gmeteor` uses a simplex solver called LPPRIM, which has been made available in the public domain by the Division of Information Technology (DoIT) at the University of Wisconsin, Madison.

The examples in the tutorial chapter were for the most part inspired by the METEOR paper by Steiglitz et al.

The file `f2c.h` comes from the `f2c` distribution, by S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer.

`gmeteor` uses many tools from the GNU project, including Guile, `autoconf`, `automake`, `texinfo`, and `libtool`.

Thanks to Prosa/Linuxcare for hosting the `gmeteor` web and ftp sites.

Thanks to Steven G. Johnson and to Sandra for reviewing this manuscript. All mistakes are my own, however.