FIR Filter Optimization Toolbox

User's Guide Version 2.0

Jeff Coleman Dan Scholnik Josef Brandriss

March 8, 2004

Contents

Chapter 1

Introduction

1.1 Overview

This manual is a user's guide for the FIR Filter Optimization Toolbox (OPT, version 2.0)

1.1.1 OPT Directory Structure

Your OPT disk possesses the following files and subdirectories:

@CTProcess	Continuous time random process source files
@CTProcess/private	Private functions for CTProcess class
Doc	Documentation files
examples	Example scripts
@LinConstr	Linear constraint source files
@optArray	Optimizable array source files
@optGenSequence	Optimizable non-uniform sequence source files
@optQuad	Optimizable quadratic source files
@optQuadVector	Optimizable quadratic vector source files
@optSequence	Optimizable sequence source files
@optVector	Optimizable vector source files
@Process	Discrete-time random process source files
@Process/private	Private functions for Process class
@SOCConstr	Second-order cone constraint source files

1.2 Quick Reference

Tables 1.1 through 1.2 summarize operators in Opt.

notation	function
a+b	add
a-b	subtract
a*b	pointwise multiply
a/c	pointwise divide
a(n)	extract
real(a)	real part
imag(a)	imaginary part
conj(a)	complex conjugate
sum(a)	element sum

Table 1.1: Overloaded operators for objects of class `optVector`. `a` and `b` are of class `optVector`, `c` is a constant vector, and `n` is an integer vector.

notation	function
h.*g	convolve
h|M	shift
h./M	interpolate
h.\M	decimate
h.'	flip about origin
h'	flip and conjugate

Table 1.2: Additional operators for objects of class `optSequence`. `g` and `h` are of class `optSequence` and `M` is a positive integer.

1.3 Filter Design With OPT

A MATLAB session for Opt filter design typically takes the form in Fig. **??**. InitOpt is called once per session. An optimization space is created by newOptSpace, which returns a handle to a new set of optimization variables. A solution point in that space can be obtained thereafter by passing its handle to minimize with an *objective*, a list of *constraints*, and name of a solver routine. Impulse response variable h in Fig. **??** is an example of an Opt quantity that does not store numerical values but instead stores relationships to optimization variables. Numerical values are substituted into those relationships only when optimal evaluates the relationships at a solution point. This is the heart of Opt:

> Opt permits meaningful algebraic operations on quantities that, because optimization has not yet taken place, have no numeric values.

The last argument to minimize selects a solver. Here we use
'sedumi' (http://fewcal.kub.nl/sturm/software/sedumi.html),
a high-performance noncommerical (free) numeric solver for SOCPs. Other options include

the remarkably fast 'mosek' (www.mosek.com) and 'loqosoclp', an SOCP/LP interface to the LOQO (www.princeton.edu/ rvdb) package, which accepts a callback function for plotting intermediate results.

1.3.1 SOCP Specification

The SOCP specification in Fig. **??** generally takes this form:

- specification of an *impulse response*
- specification of the *constraints*
- specification of the *objective*

The *objective* is often simple and is just defined in minimize's argument list. This is occasionally true of the *constraints* also.

The SOCP *objective* and *constraints* in Fig. **??** are generally functions of the *impulse response* to be optimized. For information on specifying an impulse reponse, see chapter 2.

Construction of Error Measures

Frequency-Domain Grids Ref. ??? discusses the use of SOC constraints, one for each frequency, in a grid across a band of interest, to construct measures of frequency-domain errors in the L_∞ (or Chebyshev or minimax) sense, in the L_2 (or mean squared error) sense, and in the L_1 (mean absolute error) sense. Those techniques begin here:

```
f = linspace(f1, f2, (f2-f1) * 20 * length( h ) ) ;
H = fourier(h, f) ;
```

If h is an impulse response, H is the corresponding frequency response "evaluated" on the MATLAB vector f of frequencies. If h is optimizable then H is also, and its samples remain indeterminant because they depend on impulse-response coefficients that as yet have no numeric values. Here MATLAB's linspace function creates a vector with elements stretching between its first two arguments of length specified by the third argument, here set high enough for typical FIR design work. Factor 20 might be as low as 5 for quick experimentation or as high as 50 or more when the Fourier samples are to be used for precise approximation of an L_1 norm.

If h is linear phase, so that H is real by construction,

```
C = {-10 ∧ (-55/20) <   real(H)   , ...
                        real(H)   <10 ∧ (-55/20) } ;
```

sets C to a list containing two linear constraints for each frequency sample to bound H between $-\epsilon$ and ϵ, where $20\log_{10}\epsilon = -55$ dB. Taking the real part of H eliminates computational noise in imaginary components (which should be zero) in order that the "<" operator has real arguments as required. If h is instead nonlinear phase, so that H must be complex,

```
d = optVar( X ) ;
C = abs( H ) < d ;
```

sets C to a list of SOC constraints, one for each sample in H, that constrain the magnitute response by optimization variable d. A constant bound would be valid, but here d can be passed as the *objective* to minimize in order to minimize the peak magnitude of the samples in H. (Using abs(·) would also work for the linear-phase case, but a degenerate second-order cone would be used at each frequency instead of a linear-constraint pair as before.)

The forms just presented typically specify stopbands but apply to passbands as well if, when C is defined, H is replaced with (1 - H) to specify a desired complex passband gain of unity.

Random Processes as Drive Signals DSP often requires error measures of the form

$$MSE = \int |H(f) - D(f)|^2 W(f)\, df,$$

where $H(f)$ approximates, with error weighting $W(f)$, some desired function $D(f)$. If $\int W(f)\, df = 1$ with $W(f)$ taking only values $\{0, \alpha\}$ for some α, then this is the mean squared error (MSE) between $H(f)$ and $D(f)$ in the support of $W(f)$. But if $W(f)$ is the power spectral density (PSD) of a zero-mean random process driving filters $H(f)$ and $D(f)$, the integral is also the average power in the output-error signal (the difference between the filter outputs). An MSE specified as an error power can be derived by Opt automatically and to machine precision.

Chapter 2

Impulse Response Classes

2.1 `optVector` Class

The `optVector` class is the most fundamental in the toolbox, as it provides the basic representation of a quantity to be optimized. An `optVector` is an extension of the basic MATLAB vector, where each element of the vector is an affine (linear plus a constant) combination of the underlying optimization variables.

As per MATLAB convention, a length-N `optVector` `a` has first element `a(1)` and last element `a(N)`. A new `optVector` is created by calling

$$\mathsf{a} = \mathsf{optVector}(\mathsf{N},\ \mathsf{ov})$$

which allocates `N` new optimization variables from the pool `ov` and then assigns one variable to each element of `a`. When called with a single constant vector argument, optVector returns a constant `optVector`. This is rarely needed, since operators and functions treat a constant `optVector` the same as a standard MATLAB vector. Most of the usual MATLAB operators are overloaded to accept and return variables of class `optVector`, as listed in Table **??**. In addition to the usual MATLAB requirements on size compatibility, the multiply and divide operators have the restriction that one of the two arguments must be constant, so that the result is still affine in the optimization variables.

2.2 `optSequence` Class

The `optSequence` class builds on the `optVector` class, adding a time index n. This is used to represent FIR filters, as well as constant input signals and non-optimized filters. Additional operators defined for this class are shown in Table **??**. In addition, the function fourier(h, f) is defined, returning an `optGenSequence` in which the elements hold the frequence response of `h` at the normalized frequencies `f`.

2.2.1 Examples

Opt supports a finite-length-sequence data type incorporating both values and times of samples. Opt code

```
w = optSequence( vect ) ;
```

associates times $0, \ldots,$ length(vect)-1 with the elements of MATLAB vector vect, effectively making w a fixed, nonoptimizable impulse response. Alternate form

```
u = optSequence( N, X ) ;
```

adds N new variables to the set X of real optimization variables and creates a sequence in which samples $0, \ldots, N-1$ are associated with the new variables.

Here Opt variable u is the *optimizable* impulse response of a real nonlinear-phase filter. Such sequences also represent uniformly-spaced trains in continuous time; other optSequence forms place impulses nonuniformly and in multiple dimensions.

Overloaded MATLAB operators allow construction of structured impulse reponses. The prime operator ($'$) yields the time-reversed conjugate (match) and is used to impose a linear-phase response:

```
u = optSequence( N, X ) ;
v = u + u ' ;
```

Here v is a real impulse response with linear phase and support on $-(N-1), \ldots, N-1$. The creation instead of a complex linear-phase impulse response is straightforward using the delay operator (|) and simple arithmetic operations:

```
a = optSequence( N, X ) ;
b = optSequence( N, X ) ;
r = (a + j * b) | 1 ;
p = r + optSequence(1, X) + r ' ;
```

Here p depends on $2N+1$ distinct optimization variables. The center sample is real by construction, as required.

Opt's time-axis scale-up operator (./) used here makes a complex linear-phase impulse response be halfband:

```
u = optSequence( N, X ) ;
v = optSequence( N, X ) ;
c = optSequence( N, 1 ) ;
r = ((u + j * v) ./ 2 ) | 1 ;
h = r ' + c + r;
```

(Operator .\ would perform decimation.)

A convolution operator allows certain cascades to be defined. If two filters have impulse responses p and w, then

```
q = p .* w ;
```

makes q the impulse response of their cascade. There is an important restriction, however. Every sample in an Opt sequence must depend affinely (linearly plus a constant) on optimization variables. To avoid quadratic dependencies, which are not allowed, convolution and multiplication must have at least one "fixed" or nonoptimizable argument, like the fixed sequence w defined earlier. These optimizable-fixed cascades are used when designing a filter to meet specifications on a cascade of which it is a member

2.3 `optArray` Class

The `optArray` class similarly is an extension of the `optVector` class, with the addition of location information, used for the representation of multi-dimensional sequences. Location information is stored in a matrix, each row of which contains the coordinates of an individual location. For example, for a three-dimensional `optArray`, the locations matrix `locs` is

$$\texttt{locs} = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix}$$

The locations matrix is stored in sorted form; sorted by the first dimension first, then the second, and so on. A location can be any vector $\mathbf{x} \in \mathbb{R}^n$.

All of the operators defined for the one-dimension `optSequence` class are present in the `optArray` class, with some slight modifications, as well as the addition of a few others. The transpose (`.'`) and conjugate-transpose (`'`) operators flip a sequence about the origin; that is, each element is moved to the inverse of its location. The **flip** function inverts the sequence in a single dimension. The **rot(seq, ang, dims)** function rotates the sequence in a 2-dimensional subspace by applying a rotation matrix to the two columns of `locs` indicated by `dims`. A vector $\mathbf{x}$ is rotated by an angle θ by premultiplying it by a rotation matrix

$$\mathbf{Q}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

After the rotation matrix is applied the locations are resorted. For example,
rot`(seq, pi/4, [1 3])`
will rotate `seq` 45 degrees in the xz-plane (about the y-axis).

The **or** function, with its equivalent operator `|` is used as before as a shift operator, with the requirement that the offset operand be a vector of the same dimension as the sequence.

Convolution in multiple dimensions is expressed as

$$\begin{aligned} y(\mathbf{n}) &= x(\mathbf{n}) * h(\mathbf{n}) \\ &= (x * h)(\mathbf{n}) \\ &= \sum_{\mathbf{k}} x(\mathbf{k}) h(\mathbf{n} - \mathbf{k}) \end{aligned}$$

where $\mathbf{n}, \mathbf{k} \in \mathbb{R}^n$.

The Fourier transform of a multi-dimensional sequence $h(\mathbf{n})$ is

$$H(\mathbf{f}) = \sum_{\mathbf{n}} h(\mathbf{n}) e^{-j2\pi <\mathbf{f},\mathbf{n}>}$$

If soln contains a solution obtained by minimize, optimal(h, soln) returns a constant `optArray` of identical structure which is the optimal impulse response.

Chapter 3

Regions and Lattices

3.1 Creating `Region` Objects

The `Region` class in Opt provides a convenient way of describing regions in space. Regions are built from a selection of primitive shapes and using the various set operations.

A new `Region` object is created by calling

```
reg = Region(type, parameters)
```

`type` is one of the primitive shape names, which at this point include `sphere`, `halfspace` and `convpoly`. `parameters` is a structure whose fields are the various parameters that describe the shape. Each shape requires the field `dim` which indicates the dimension of the region.

In addition, `sphere` requires the fields `center` and `radius`. `center` is a vector of real numbers giving the coordinates of the center of the sphere object. The radius of the sphere is a real number given in `radius`.

A halfspace is the set described by

$$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}'\mathbf{x} \geq b\}$$

and thus requires the fields `a` and `b`.

`convpoly` is short for convex polytope. A `convpoly` object is created as the convex hull of the list of points given in `points`. `points` is a matrix with `dim` columns, with each row giving the coordinates for a separate point. The `convpoly` is more easily represented internally as an intersection of halfspaces, or the set

$$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\}$$

which is determined as follows.

First the convex hull is determined using MATLAB's `convhulln` routine, which returns a list of the facets which make up the surface of the polytope. A facet in N dimensions is an (N-1)-dimensional simplex, each of which lies on a hyperplane. A hyperplane is the set

$$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}'\mathbf{x} = b\}$$

or

$$\{\mathbf{x} \in \mathbb{R}^n \mid \sum_{i=1}^{N} a_i x_i = b\}$$

Let $\mathbf{x}^1 \ldots \mathbf{x}^N$ be the points which determine a facet, and thus its hyperplane. x_k^n is the kth coordinate of the nth point. Then the coefficients of the hyperplane are given by the following determinants:

$$a_1 = \begin{vmatrix} 1 & x_2^1 & \cdots & x_N^1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_2^N & \cdots & x_N^N \end{vmatrix} \qquad a_2 = \begin{vmatrix} x_1^1 & 1 & x_3^1 & \cdots & x_N^1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^N & 1 & x_3^N & \cdots & x_N^N \end{vmatrix} \qquad \cdots$$

$$a_N = \begin{vmatrix} x_1^1 & \cdots & x_{N-1}^1 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_1^N & \cdots & x_{N-1}^N & 1 \end{vmatrix} \qquad b = \begin{vmatrix} x_1^1 & \cdots & x_N^1 \\ \vdots & \ddots & \vdots \\ x_1^N & \cdots & x_N^N \end{vmatrix}$$

This hyperplane is the boundary of one of the halfspaces whose intersection describes the polytope. The direction of the inequality of the corresponding halfspace is determined by checking points in other facets of the polytope. It is possible that there are multiple coplanar facets of the polytope. Points of a coplanar facet should give an inconclusive result, but because of numerical error can give an incorrect answer. Because of this, all of the points on the surface of the convex hull are checked, and direction is chosen as that indicated by the majority of the points.

3.1.1 Forming Composite Regions

Composite `Region` objects may be formed by using the familiar set operations such as `union`, `intersect` and `not`. For ease of expressing unions and intersections of a series of objects, the operators `+` and `*` may be substituted for `union` and `intersect`, respectively. `setdiff(a,b)` returns the region which is the set difference a/b, or $a \cap \neg b$.

A composite region object is always formed as a result of a binary operation, and therefore contains the data indicating the type of operation and the two operands. Thus, a complicated composite `Region` can be thought of as a binary operation tree whose leaves are primitive shapes.

The operation `not(a)` or ∼`a` where `a` is a primitive `Region` returns a primitive region object identical to `a` but with the complement flag set. If, however, `a` is a `composite` region, the complement operation is applied recursively down through the operation tree.

3.1.2 Simple Region Transformations

Two functions are available to perform simple transformations on a `Region` object.

Offset

A region offset is done using the `or` function. This is more conveniently called using the equivalent operator `|`. The syntax, for an n-dimensional region, is

```
oreg = reg | offset
```

where `offset` is a vector of length n, giving the offset amount in each coordinate direction.

Shifting a `sphere` is straightforward; its center is simply adjusted by the offset amount.

A `halfspace`, as mentioned earlier, is described by

$$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}'\mathbf{x} \geq b\}$$

and has the hyperplane $\mathbf{a}'\mathbf{x} = b$ as its boundary. $\mathbf{a}$ can be thought of as the hyperplanes's normal vector, and b as the distance of the hyperplane from the origin along the normal vector $\mathbf{a}$. Shifting a hyperplane has the effect of sliding it along its normal vector. The distance slid is the length of the projection of the offset vector along the normal direction, or $\mathbf{a} \cdot \mathbf{offset}$. Therefore the b parameter must be adjusted as follows:

$$b_{\text{off}} = b + \mathbf{a} \cdot \mathbf{offset}$$

Since a `convpoly` is simply an intersection of several halfspaces, the procedure for shifting it is the same; each element of the `convpoly`'s $\mathbf{b}$ vector is adjusted as follows:

$$b_{k,\text{off}} = b_k + \mathbf{a_k} \cdot \mathbf{offset}$$

where each vector $\mathbf{a_k}$ is a row of the $\mathbf{A}$ matrix. In addition, the extreme points of the `convpoly` are shifted by the same vector, like the center point of a `sphere`.

Rotation

A region can be rotated using the `rot` function. The syntax is

```
rreg = rot(reg, ang, dims)
```

This function rotates `reg` by `ang` radians in the 2-dimensional subspace indicated by the 2-vector `dims`. For example, `rot(reg, pi/4, [1 3])` will rotate `reg` 45 degrees in the xz-subspace; that is, about the y-axis.

For a `sphere`, the center point is rotated by an angle θ by premultiplying the vector of coordinates in the specified 2-dimensional subspace by the rotation matrix

$$\mathbf{Q}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

For a `halfspace` with boundary hyperplane $\mathbf{a}'\mathbf{x} = b$, the rotation is accomplished similarly by premultiplying the components $\mathbf{a}$ vector corresponding to the specified 2-dimensional subspace by $\mathbf{Q}_\theta$. Since $\mathbf{a}$ is the normal vector of the hyperplane, it can be rotated like any other vector, while b, the distance of the hyperplane from the origin along the normal vector, remains constant.

For a `convpoly`, which is the intersection of halfspaces, each component halfspace is treated the same as above. In addition, the extreme points of the `convpoly` are rotated as well, like the center point of the `sphere`.

If either of these functions is called on a `composite` region, the operation is applied recursively down through the operation tree so that each primitive object is offset or rotated by the same amount.

3.2 Creating `Lattice` Objects

The `Lattice` class in Opt is used to specify a lattice, or pattern of points. A lattice in n dimensions is described by its basis matrix $\mathbf{M} \in \mathbb{R}^{n\times m}$. $\mathbf{M}$ is post-multiplied by an integer vector $\xi \in \mathbb{Z}^n$ to generate the coordinates of the lattice points in cartesian space. While m can be any integer greater than n, it is most convenient to require that $m = n$. A lattice can be scaled and offset as well. The full expression for a point on a lattice is then

$$\mathbf{x} = \alpha\mathbf{M}\xi + \mathbf{c}$$

where $\alpha \in \mathbb{R}$ and $\mathbf{c} \in \mathbb{R}^n$.

A `Lattice` object is created by calling

```
lat = Lattice(M)
```

where M is the $n \times n$ basis matrix. The Lattice object will also contain a scale factor, preset to 1, and an offset, preset to the zero-vector, both of which can be adjusted as follows.

A Lattice can be scaled by using the * operator. The syntax

```
slat = a * lat}
```

scales lat by a scalar a and returns the Lattice object slat. The object slat will contain the same basis matrix as well as the scale factor.

A Lattice can be offset by using the + operator. The syntax

```
olat = lat + off
```

shifts lat by the amounts in the vector off and returns the result in olat. As with the * operator, olat will have the same basis matrix as lat, with only the offset data adjusted.

3.3 Region and Lattice Operations

3.3.1 isin Function

The function isin takes a list of points and returns those located inside a Region. The syntax of the function is

```
result = isin(reg, points)
```

Both result and points are matrices in which each row gives the coordinates of a point.

If reg is a region primitive, the result is determined by checking each input point against the rules for that type of shape.

A point $\mathbf{x} = (x_1, \ldots, x_N)$ is inside a sphere of radius ρ and center $\mathbf{c}$ if

$$\sum_{i=1}^{N} (x_i - c_i)^2 \leq \rho^2$$

or

$$||\mathbf{x} - \mathbf{c}|| \leq \rho$$

A point $\mathbf{x}$ is in a `halfspace` if

$$\mathbf{a}'\mathbf{x} \geq b$$

A point $\mathbf{x}$ is in a `convpoly` if

$$\mathbf{A}\mathbf{x} \geq \mathbf{b}$$

where $\mathbf{A}$ and $\mathbf{b}$ are determined as described above.

If `reg` is a composite region, then it was formed as a result of set operations on primitive regions. Each of the two child regions is queried, and the result for a given point is computed. For example, if the two child regions were combined by an `intersect` operation, the a point is in the region only if the point is both of the child regions. If a child region is itself a composite, the process continues until a primitive region is queried. The results for the component primitive regions are then combined using set operations to produce the result for the composite region.

3.3.2 `msop` Function

The function `msop` takes a composite `Region` as input and determines a simplified sum-of-products (or union-of-intersections) equivalent. This is useful in determining whether a region is bounded, and if so, finding a bounding parallelpiped guaranteed to contain the entire region.

Any function on primitive regions $R_1, \ldots, R_m$ using the operations $\cup$, $\cap$ and $\neg$ can be reduced to a sum-of-products form, for example

$$f(R_1, \ldots, R_m) = \bigcup_{i=1}^{p} \bigcap_{j \in [1,m]} \delta_{i,j} R_m$$

Each composite `Region` object stores the identity of each unique primitive object that it is composed of. The function `probett` is then used to assemble the entire truth table of the Boolean function that the composite `Region` represents. The function `qm` is then used to find a reduced sum-of-products form for the truth table using the Quine-McCluskey method of finding prime implicants.

The reduced function is returned as a cell array, of which each element represents a product term. Each product term is represented as a vector of numbers, each of which is the index number of a primitive region in the composite. An example output is

```
fcn =

    [2 3 5] [1 2]
```

which indicates that the simplified representation of the composite region is

$$(R_2 \cap R_3 \cap R_5) \cup (R_1 \cap R_2)$$

`msop` is used by the function `bb`.

3.3.3 `bb` Function

The function `bb` determines whether a `Region` object is bounded, and if so, returns its bounding box. It is necessary for determing the lattice points which lie in a given region. Therefore, the function takes two arguments: a `Region` object and a `Lattice` object. The bounding box is returned not as Cartesian coordinates, but in terms of intervals along the lattice basis vectors. Thus the bounding box is in fact a parallelepiped which is guaranteed to circumscribe the `Region` in question.

For a `convpoly` region of dimension n, the bounding parallelepiped is determined by testing each of the region's extreme points.

A bounding interval must be found for each lattice basis vector. For each basis vector, any of the extreme points could contribute to the bounding interval, and as such each extreme point must be tested.

Once the bounding interval is found for a given basis vector, two faces of the bounding parallelepiped can be constructed. Each face of the bounding parallelepiped lies in the hyperplane spanned by all of the other basis vectors, and is itself a parallelepiped of dimension $n-1$. Since we require that the lattice basis matrix be full-rank, any $n-1$ basis vectors will span an $(n-1)$-dimensional subspace, and will have a 1-dimensional nullspace. This nullspace can be seen as the normal vector of the $(n-1)$-dimensional subspace. Together, the nullspace and the current basis vector lie in a 2-dimensional subspace (although they may be collinear, in which case they would not span the plane).

Each extreme point $\mathbf{p}$ is projected onto the orthonormal basis to the nullspace, $\mathbf{u}$, and the minimum and maximum projections determine the bounding interval. The projection for a given extreme point and lattice basis vector is

$$\text{proj}_{\mathbf{u}}\mathbf{p} = \frac{\mathbf{p}\cdot\mathbf{u}}{|\mathbf{u}|^2}\mathbf{u} = (\mathbf{p}\cdot\mathbf{u})\mathbf{u}$$

The angle θ between this projection and the current basis vector $\mathbf{v}$ is given by

$$\cos\theta = \frac{\text{proj}_{\mathbf{u}}\mathbf{p}\cdot\mathbf{v}}{|\text{proj}_{\mathbf{u}}\mathbf{p}||\mathbf{v}|}$$

Together, $\text{proj}_{\mathbf{u}}\mathbf{p}$ and θ determine the distance r' along the direction of the basis vector $\mathbf{v}$. By the definition of cosine,

$$\cos\theta = \frac{|\text{proj}_{\mathbf{u}}\mathbf{p}|}{r'}$$

and rearranging gives

$$\begin{aligned} r' &= \frac{|\text{proj}_{\mathbf{u}}\mathbf{p}|^2|\mathbf{v}|}{(\text{proj}_{\mathbf{u}}\mathbf{p}\cdot\mathbf{v})} \\ &= \frac{(\mathbf{p}\cdot\mathbf{u})^2|\mathbf{v}|}{[(\mathbf{p}\cdot\mathbf{u})\mathbf{u}]\cdot\mathbf{v}} \end{aligned}$$

r' must be scaled by the norm of the basis vector to determine its distance along that vector, giving

$$r = \frac{r'}{|\mathbf{v}|} = \frac{(\mathbf{p}\cdot\mathbf{u})^2}{[(\mathbf{p}\cdot\mathbf{u})\mathbf{u}]\cdot\mathbf{v}}$$

If it turns out that the projection onto the nullspace is the zero-vector, the distance along the lattice basis vector zero, and the computation is not done, or a division-by-zero error will result.

If the basis for the nullspace and the lattice basis vector are collinear, the angle between them is 0, making

$$r = \frac{|\text{proj}_{\mathbf{u}}\mathbf{p}|}{|\mathbf{v}|}$$

Because lattice points are determined by multiplying an integer vector by the basis matrix, the interval is rounded away from zero.

A `sphere` region's bounding box is most easily determined by inscribing it inside a hypercube, which has 2^n extreme points. The hypercube's bounding parallelepiped is then determined the same way as the `convpoly`'s.

A `halfspace`, as well as a complemented `sphere` and `convpoly` are always unbounded. However, the intersection of at least $n+1$ halfspaces is possible bounded. This possibility is covered by the treatment of `composite` regions.

The bounding parallelepiped of a `composite` region is found by properly combining the bounding parallelepipeds of the constituent regions. This is accomplished by first obtaining a sum-of-products (or union-of-intersections) representation of the `composite` region, using the `msop` function described in section 3.3.2.

Each product term is the intersection of primitive regions. If any of the product terms is unbounded, the entire region, which is the union of all of the product terms, will be unbounded. Otherwise, the bounding intervals are found by taking the minimum of the axis minima of each product term and the maximum of the axis maxima of each product term.

For an individual product term, the bounding intervals are found by taking the maximum of the axis minima of each primitive region, and the minimum of the axis maxima of each primitive region. If at least $n+1$ `halfspaces` are present in a product term, it is possible that their intersection is bounded.

A halfspace, and intersections of halfspaces, are convex polytopes. To determine whether a convex polytope is bounded, it is necessary to solve two linear programs, as follows.

For a convex polytope

$$P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\}$$

the *recession cone* is the set of all directions $\mathbf{d}$ along which we can move indefinitely away from a point inside P. The recession cone is the set

$$\{\mathbf{d} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{d} \geq \mathbf{0}\}$$

If P is bounded, the recession cone will consist of only the zero-vector, which is always a solution to $\mathbf{A}\mathbf{d} \geq \mathbf{0}$. Therefore, if either of the two linear programs

$$\begin{array}{rl} \min & \pm\mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{0} \end{array}$$

has a solution other than the zero-vector, a non-trivial recession cone exists, and the set P is unbounded. Conversely, if the only solution to both linear programs is the zero-vector, no non-trivial recession cone exists, and the set P is bounded.

It is possible for one choice of the cost vector $\mathbf{c}$ for a false indication of boundedness to result. If the unbounded direction is along the hyperplane $\mathbf{c}'\mathbf{y} = \mathbf{0}$ it is possible for a linear program solver to return the zero-vector as an optimal solution for both linear programs, as any solution along the hyperplane has minimum cost. However, if $\mathbf{c}$ is chosen randomly, the probability of this occurring is extremely small. If not for the finite precision of numbers on a computer, the probability would in fact be zero.

The linear program solvers recommended for use with `Opt` require that the linear program be given in *standard form*; that is,

$$\begin{array}{rlcl} \min & \mathbf{c}'\mathbf{x} & & \\ \text{subject to} & \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0} \end{array}$$

The duals of our linear programs are

$$\begin{array}{rlcl} \min & \mathbf{0} \cdot \mathbf{y} & & \\ \text{subject to} & \mathbf{A}'\mathbf{y} & = & \pm\mathbf{c} \\ & \mathbf{y} & \geq & \mathbf{0} \end{array}$$

which are in standard form. By duality theory, if the dual of a linear program is infeasible, the primal is either infeasible or unbounded. As we know that the primal in our case is feasible, it must be unbounded. If the dual is feasible and has a finite optimum, then the primal must also be feasible and have a finite optimum.

Therefore, to test the boundedness of the intersection of halfspaces, we concatenate their **a** vectors to form the matrix **A**, and we concatenate their b's to form the vector **b**. We then run both of the dual linear programs. If either is infeasible, the polytope P formed by the intersection of halfspaces is unbounded. If both are feasible, and have finite optima, P is bounded. If it is determined that P is bounded, the extreme points of P are found, and we follow the procedure detailed above for finding the bounding parallelepiped of a convex polytope.

A vector **p** is an extreme point of P if

1. All equality constraints are active.
2. At least n linearly independent constraints are active at **p**.
3. All constraints are satisfied.

Our representation of the convex polytope contains no equality constraints, so item 1 is automatically satisfied. We test all possible combinations of n linearly independent constraints, and solve the linear system

$$\mathbf{a_i}'\mathbf{x} = b_i, \qquad i \in I_k$$

where the I_k is an index set over the kth combination of n linearly independent constraints. If the solution **x** satistfies all of the other constraints as well, it is an extreme point of the polytope.

One final point: If the `Lattice` given is offset, the offset will be applied to the extreme points before the bounding intervals are found. Therefore the bounding intervals which are returned by the function describe a parallelepiped which circumscribes a `Region` offset by the same amount.

3.3.4 `inbox` function

The function `inbox` returns a list of points on a given lattice that lie within a bounding box (parallelepiped). The syntax is

```
locs = inbox(lat, box)
```

where `lat` is a `Lattice` object and `box` is a vector of the form

```
box = [1min 1max 2min 2max ... ]
```

with entries which are the minimum and maximum extent of the parallelepiped along each of the lattice basis vectors. The points in `locs` are returned as a matrix in which each row lists the Cartesian coordinates of a single point.

To generate the list of points, we produce an integer grid within the bounding intervals. Each point on this integer grid is used as ξ in the expression

$$\mathbf{x} = \alpha \mathbf{M} \xi + \mathbf{c}$$

If the `Lattice` is offset, the offset $\mathbf{c}$ will be applied after the points are calculated, and so the bounding intervals must be given in terms of an offset region in space. This, as described above, is automatically taken into account by the `bb` function.

3.3.5 `points` function

The `points` function return a list of the points on a given lattice that lie within a given region. The syntax is

```
locs = points(reg, lat)
```

The function works by calling three previously-described functions as follows:

1. `bb` is called to determine the bounding parallelepiped for the region;
2. `inbox` is called to find the lattice points within the bounding parallelepiped;
3. `isin` is called to determine which of the points returned by `inbox` are in fact within the region

3.4 Usage of `Regions` and `Lattices` in `Opt`

Two useful applications of `Region` and `Lattice` to `Opt` are

- Easy formation of optimizable sequences in a variety of shapes and patterns
- Easy specification of frequency responses

An n-dimensional optimizable sequence can be created using the command

```
seq = optArray(lattice, region, pool)
```

where `pool` is a handle to a pool of optimizables. This command creates a sequence with support on lattice points within the given region. If the region and lattice given are not of the same dimensionality, or if the region given is unbounded, an error is returned. Otherwise, we compute the bounding parallelepiped, and then determine which points within that parallelepiped are inside `region`.

In addition, if `seq` has already been specified as an `optArray` , the command

```
sseq = seq(reg)
```

creates a subsequence `sseq` which contains only those points located within `reg`.

For frequency gridding, a convenient procedure is to define a region of the desired shape and a lattice of sufficient tightness, and use the `points` function to output a grid of point locations.

3.4.1 Examples

As an example, let us create a 2-D filter with its impulse response on a circular region of support. Such an impulse response pattern is useful when a circularly-symmetric frequency response is desired. In addition, the impulse response is to be real and symmetric about the origin so that the frequency response will be symmetric as well.

To produce an impulse response with the desired symmetry properties, we must take care that the coefficient at each point (n_1, n_2) depend on the same optimization variables as its reflection at $(-n_1, -n_2)$. First we define a square `Lattice` of points from which the impulse response locations will be chosen.

```
sqlat = Lattice(eye(2));
```

Then, we define a Region enclosing a single quadrant

```
p1.dim = 2;
p1.points = [0 0; 0 N; N 0; N N];
quadrantBoundary = Region('convpoly', p1);
```

where integer parameter `N` can be increased if better performance is desired. The following code defines a circular region centered at the origin with radius `N`.

```
p2.dim = 2;
p2.center = [0 0];
p2.radius = N;
circleReg = Region('sphere', p2);
```

Next, we create the quarter-circle region by intersecting region `circleReg` with `quadrantBoundary`.

```
qCircle = quadrantBoundary * circleReg;
```

The following creates two `optArray`s with support on the points on the lattice `sqlat` within the region `qCircle`, with `X` a space of optimization variables:

```
aa1 = optArray(sqlat, qCircle, X);
aa2 = optArray(sqlat, qCircle, X);
```

Finally, we combine various reflections of the two impulse responses to create the whole circular array:

```
aa = aa1 + aa1' + flip(aa2, 1) + flip(aa2, 2);
```

The command `flip(a, dim)` flips an `optArray` in dimension `dim`; that is, all point locations are changed from $x = \{\ldots, x_{\text{dim}}, \ldots\}$ to $\hat{x} = \{\ldots, -x_{\text{dim}}, \ldots\}$. So `aa1` is in the first quadrant, `aa1'` in the third, `flip(aa2, 1)` in the second and `flip(aa2, 2)` in the fourth. Although points along the axes will depend on two optimization variables each,

The array is shown in Fig. 3.1.

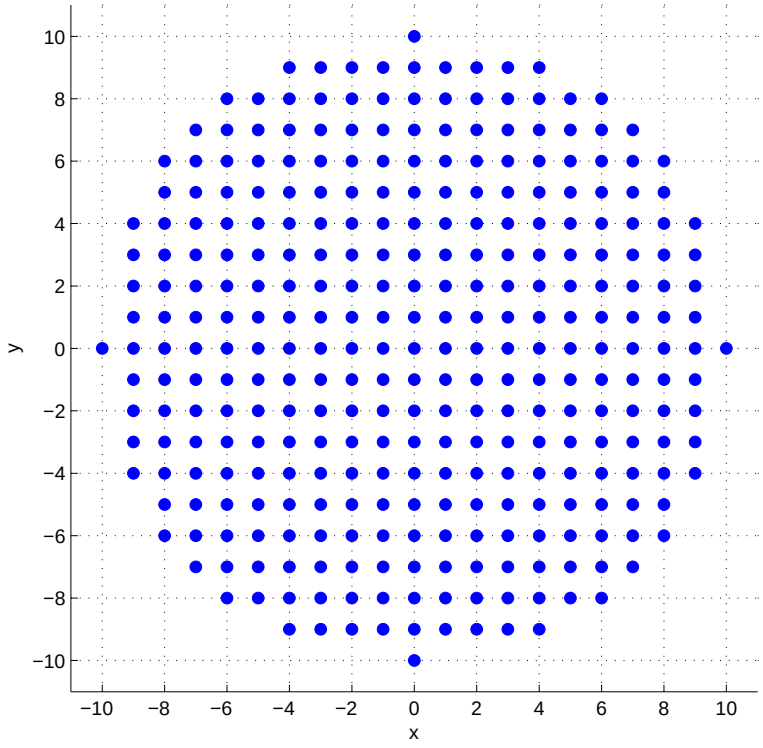

Figure 3.1: Circular Array.

Chapter 4

Random Processes

4.1 Processes in One Variable

A wide-sense-stationary random process $v(n)$ may be specified in terms of its power spectral density $S_v(f)$. The power spectral density, in turn, is expressed as the superposition of frequency-shifted copies of a basis function $\Phi(Mf)$:

$$S_v(f) = \sum_k w_k \Phi(Mf - \Delta - k)$$

where sequence w_k has period M, and Δ is a frequency offset. (Sequence w_k is binary valued)

For a process $z(n)$ driving a filter $h(n)$, and output process $u(n) = (z*h)(n)$, the output power is the value of the output autocorrelation function at offset 0. The output autocorrelation function is defined as

$$\mathcal{R}_u(n) = E[(h * z)(k)(z' * h')(n - k)]$$

Further manipulation results in

$$\begin{aligned} \mathcal{R}_u(n) &= \sum_\ell \sum_m h(\ell) E[z(k-\ell) z'(n-k-m)] h'(m) \\ &= \sum_\ell \sum_m h(\ell) E[z(w) z'(n-w-\ell-m)] h'(m) \end{aligned}$$

Letting $w = (k - \ell)$,

$$\begin{aligned} \mathcal{R}_u(n) &= \sum_\ell \sum_m h(\ell) \mathcal{R}_z(n-\ell-m) h'(m) \\ &= \sum_m (h * \mathcal{R}_z)(n-m) h'(m) \\ &= h * \mathcal{R}_z * h' \end{aligned}$$

and so the output power is

$$\begin{aligned} P_u &= \mathcal{R}_u(0) \\ &= (h * \mathcal{R}_z * h')(0) \\ &= \sum_{\ell} \sum_{m} h(\ell)\mathcal{R}_z(-\ell - m)h'(m) \end{aligned}$$

Opt random processes can be added, subtracted, scaled and convolved with impulse responses and can have their average powers obtained. Here, pwr(s) and pwr(r) return 0.75 and 0.5 respectively. If both h and d are fixed,

```
p = pwr( r .* ( h - d )) ;
```

sets p to the numeric MSE in (4.1) with $W(f)$ the PSD of r. There are no simulations or numerical integrals; calculation to machine precision is possible because available basis function exist internal to Opt as Fourier transform pairs, and the required integration is actually an inverse Fourier transform.

Since possible power spectrum basis functions and their Fourier transforms are known a priori, the autocorrelation function $\mathcal{R}_z(\tau)$ is easy to compute.

4.1.1 Examples

The following creates a zero-mean, discrete-time Opt random process:

```
rp = Process ( Φ , Δ, w ) ;
```

The three arguments to Process specify the PSD of s, where Φ is the *basis function*, Δ is the frequency offset and $\mathbf{w}$ is the weight vector. Opt constructs the PSD in steps:

1. Weight vector $\mathbf{w}$ is extended periodically to create a doubly-infinite sequence $w_k \triangleq \mathbf{w}_{1+(k \bmod M)}$,
2. the infinite weight sequence is applied to integer shifts of the supplied basis function $\Phi(f)$, creating a function $\sum_k w_k \Phi(f-k)$ with period M,
3. this function is offset upward in frequency by Δ, yielding $\sum_k w_k \Phi(f-\Delta-k)$, and finally

4. scaling the offset function down in frequency by M sets the period in normalized frequency f to unity:

$$\text{PSD} = \sum_k w_k \Phi(Mf - \Delta - k) \tag{4.1}$$

Here are two example random processes, s and r:

```
s = Process ( 'Box' , 0 , [1 1 0 1] ) ;
r = Process ( 'Triangle' , 0 , [1 1 3/4 1/4 0 0 0 1/4 3/4 1] ) ;
```

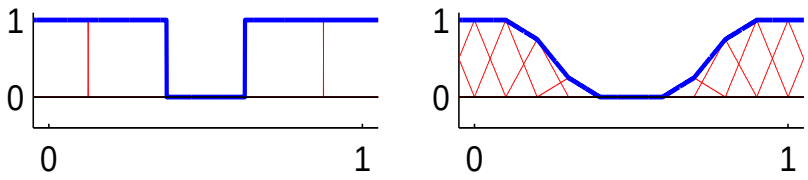

Figure 4.1: PSD of random processes.

The 'Box' basis function has unit width, so offset and scaled copies of it touch without overlap to create an ideal brick-wall spectrum. To instead create a spectrum linearly interpolated between samples, use the symmetric-triangle basis function supported on $[-1, 1]$. The PSD of r in Fig. 4.1 approximates a 60% raised-cosine spectrum at an oversampling rate of two. A longer weight vector would improve the approximation.

4.2 Processes in Several Variables

4.2.1 Average Output Power

The generalization of an Opt random process to several variables is straightforward. Let $z(\mathbf{n})$ be a wide-sense-stationary random process, where $\mathbf{n} = \{n_1 \ldots n_N\}$. Let $h(\mathbf{n})$ be the impulse response of a linear shift-invariant (LSI) system. If $z(\mathbf{n})$ is the input to this system, the output random process $u(\mathbf{n})$ is

$$\begin{aligned} u(\mathbf{n}) &= \sum_{k_1} \cdots \sum_{k_N} h(\mathbf{k}) z(\mathbf{n} - \mathbf{k}) \\ &= \sum_{\mathbf{k}} h(\mathbf{k}) z(\mathbf{n} - \mathbf{k}) \\ &= (h \star z)(\mathbf{n}) \end{aligned} \tag{4.2}$$

where the symbol $\mathbf{k}$ under the sum indicates an N-tuple sum over all possible values of the vector $\mathbf{k}$.

The average output power, $\mathcal{R}_u(\mathbf{0})$, is derived as follows, beginning with the definition

$$\mathcal{R}_u(\mathbf{n};\mathbf{k}) = E[u(\mathbf{n})u^*(\mathbf{k})].$$

Substituting (4.2), we obtain

$$\begin{aligned}\mathcal{R}_u(\mathbf{n};\mathbf{k}) &= E\left[\sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})z(\mathbf{n}-\mathbf{l})h^*(\mathbf{m})z^*(\mathbf{k}-\mathbf{m})\right] \\ &= \sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})h^*(\mathbf{m})E\left[z(\mathbf{n}-\mathbf{l})z^*(\mathbf{k}-\mathbf{m})\right]\end{aligned}$$

since the impulse response $h(\mathbf{n})$ is deterministic and can be removed from the expectation. Using the definition of an autocorrelation function we can substitute

$$\begin{aligned}\mathcal{R}_u(\mathbf{n};\mathbf{k}) &= \sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})h^*(\mathbf{m})\mathcal{R}_z(\mathbf{n}-\mathbf{l};\mathbf{k}-\mathbf{m}) \\ &= \sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})h^*(\mathbf{m})\mathcal{R}_z(\mathbf{n}-\mathbf{l}-\mathbf{k}+\mathbf{m})\end{aligned}$$

since $\mathcal{R}_z(\mathbf{n}) = E[x(\mathbf{k})x^*(\mathbf{k}-\mathbf{n})]$ for stationary z. Since $\mathcal{R}_u$ is shown to be a function of $\mathbf{n}-\mathbf{k}$, it as also stationary, and can be rewritten

$$\begin{aligned}\mathcal{R}_u(\mathbf{n}) &= \sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})h^*(\mathbf{m})\mathcal{R}_z(\mathbf{n}-\mathbf{l}+\mathbf{m}) \\ &= \mathcal{R}_z(\mathbf{n}) \star h(\mathbf{n}) \star h^*(-\mathbf{n}) \\ &= (h \star \mathcal{R}_z \star h')(\mathbf{n})\end{aligned}$$

with h' indicating a conjugate transpose of the sequence h.

The average output power is then expressed in a conveniently-computable form:

$$\mathcal{R}_u(\mathbf{0}) = \sum_{\mathbf{l}}\sum_{\mathbf{m}} h(\mathbf{l})h^*(\mathbf{m})\mathcal{R}_z(\mathbf{m}-\mathbf{l})$$

4.2.2 Periodicity and Symmetry Issues

In the one-dimensional case, a discrete-time LTI system $h(n)$ with support on the integers has a frequency response $H(f)$ which is periodic with period 1. Similarly, in multiple dimensions, if an impulse response $h(\mathbf{n})$, $\mathbf{n} = \{n_1, \ldots, n_N\}$, has support on the square lattice $n = \mathbb{Z}^N$, its frequency response $H(\mathbf{f})$, $\mathbf{f} \in \mathbb{R}^N$, will be periodic with its *fundamental period* the hypercube with side length 1 centered at the origin.

4.2.3 Basis Functions

Box

As before, this expression for average power is useful since we have several PSD-autocorrelation transform pairs available. One such PSD basis function is the multi-dimensional generalization of the scaled 'Box' function $\square_{\mathbf{w}}(\mathbf{f}) = \prod_k \square_{w_k}(f_k)$, where each $\square_{w_k}(f_k)$ is a one-dimensional box function, defined as

$$\square_w(f) = \begin{cases} 1 & -w/2 \le f \le w/2 \\ 0 & \text{otherwise} \end{cases}$$

Several Box functions can be used together to map out regions of the frequency response to be used as passbands or stopbands.

Since the function $\square_{\mathbf{w}}(\mathbf{f})$ is separable, its inverse Fourier transform is the product of one-dimensional inverse Fourier transforms. So for a WSS random process with PSD $S(\mathbf{f}) = \square_{\mathbf{w}}(\mathbf{f})$, we have

$$\mathcal{R}(\tau) = \prod_{k=1}^{N} \frac{\sin(\pi w_k \tau_k)}{\pi \tau_k}$$

For an offset in frequency **off**, the transform pair becomes

$$\square_{\mathbf{w}}(\mathbf{f} - \mathbf{off}) \quad \longleftrightarrow \quad e^{j2\pi <\tau, \mathbf{off}>} \prod_k \frac{\sin(\pi w_k \tau_k)}{\pi \tau_k}$$

where $<,>$ indicates an inner (dot) product. For a PSD constructed from a superposition of shifted, scaled boxes, the autocorrelation is

$$\mathcal{R}(\tau) = \sum_\ell c_\ell e^{j2\pi <\tau, \mathbf{off}_\ell>} \prod_k \frac{\sin\left(\pi (w_\ell)_k \tau_k\right)}{\pi \tau_k} \tag{4.3}$$

where c_ℓ is the multiplier coefficient of the ℓth instance of the basis function.

Triangle

The 'Triangle' basis function, $\triangle_{\mathbf{w}}(\mathbf{f})$, is the multi-dimensional generalization of the one-dimensional

$$\triangle_w(f) = \begin{cases} 1 + f/w & -w \le f \le 0 \\ 1 - f/w & 0 \le f \le w \\ 0 & \text{otherwise} \end{cases}$$

which is the convolution of two one-dimensional Boxes. As such, a PSD made up of the superposition of shifted, scaled Triangles is

$$\mathcal{R}(\tau) = \sum_\ell c_\ell e^{j2\pi <\tau, \mathbf{off}_\ell>} \prod_k \left(\frac{\sin\left(\pi (w_\ell)_k \tau_k\right)}{\pi \tau_k} \right)^2 \tag{4.4}$$

Impulse

The 'Impulse' basis function, $\delta(\mathbf{f})$, is the multi-dimensional generalization of the one-dimensional $\delta(f)$. It is useful to use a grid of Impulses to map out an irregularly-shaped band of the desired filter's frequency response. For a PSD composed of a single Impulse, the PSD / autocorrelation pair is

$$S(\mathbf{f}) = \delta(\mathbf{f}) \quad \longleftrightarrow \quad \mathcal{R}(\tau) = 1$$

A PSD composed of the superposition of several shifted, scaled impulses therefore has the autocorrelation function

$$\mathcal{R}(\tau) = \sum_{\ell} c_\ell e^{j2\pi<\tau,\mathbf{off}>}$$

Circle

The 'Circle' basis function, available in a two-dimensional version, is defined as

$$\bigcirc_w (f_1, f_2) = \begin{cases} 1 & \sqrt{f_1^2 + f_2^2} \leq w \\ 0 & \text{otherwise} \end{cases}$$

This function has circular symmetry and can be written as a function of $\rho = \sqrt{f_1^2 + f_2^2}$, and therefore its inverse fourier transform can be found using the Fourier-Bessel transform formula,

$$g(r) = 2\pi \int_0^\infty \rho G(\rho) J_0(2\pi r\rho)\, d\rho$$

where $J_\nu(a)$ is a Bessel function of the first kind of order ν. The inverse transform, as a function of $r = \sqrt{\tau_1^2 + \tau_2^2}$, is

$$\begin{aligned} \mathcal{R}(r) &= 2\pi \int_0^\infty \rho \bigcirc_w (\rho) J_0(2\pi r\rho)\, d\rho \\ &= 2\pi \int_0^w \rho J_0(2\pi r\rho)\, d\rho \end{aligned}$$

Letting $u = 2\pi r\rho$, and thus $du = 2\pi r\, d\rho$, we have

$$\mathcal{R}(r) = \frac{1}{2\pi r^2} \int_0^{2\pi r w} J_0(u)\, u\, du$$

Using the identity

$$\int_0^u u' J_0(u')\, du' = u J_1(u),$$

we have

$$\mathcal{R}(r) = w \frac{J_1(2\pi r w)}{r}.$$

and thus

$$\mathcal{R}(\tau_1, \tau_2) = w \frac{J_1(2\pi w \sqrt{\tau_1^2 + \tau_2^2})}{\sqrt{\tau_1^2 + \tau_2^2}}$$

For a PSD composed of the superposition of shifted Circles of various radii the autocorrelation is

$$\mathcal{R}(\vec{\tau}) = \sum_\ell c_\ell w_\ell e^{j2\pi < \vec{\tau}, \mathbf{off} >} \frac{J_1\left(2\pi ||\vec{\tau}|| w_\ell\right)}{||\vec{\tau}||}$$

A useful limit is

$$\lim_{||\vec{\tau}|| \to 0} w \frac{J_1\left(2\pi ||\vec{\tau}|| w\right)}{||\vec{\tau}||} = \pi w^2$$

4.2.4 Examples

A multi-dimensional random process is created using

rp = NDProcess (basis, param, freqs, coeff) ;

Here rp is created with a PSD consisting of a superposition of basis functions at shifts given in freqs and with weights given in coeff. freqs is a matrix in which each row is a vector of frequency offsets; therefore, freqs should contain as many rows as shifted basis functions, and as many columns as the dimension of the random process. coeff is a vector of length equal to the number of rows of freqs.

If the PSD is to be made up entirely of a single type of basis function, basis should be the name of the basis function; for example, 'Box'. If the PSD is to be composed of several types of basis functions, basis must be a column vector cell array of length equal to the number of rows of freqs; that is, one entry for each shifted basis function. For example: { 'Box' ; 'Impulse' ; 'Impulse' } .

param contains the parameters for each shifted copy of a basis function. For a single-type PSD, param should be a matrix, each row of which containing the parameters for another copy of the basis function. For a multiple-type PSD, since different basis function types require different parameters to be specified, param must be a cell array column vector, in which each entry contains the parameters for another shifted basis function.

The basis function types 'Box' and 'Triangle' in N dimensions require one width parameter for each dimension. The 'Impulse' type requires no parameters, and the empty matrix [] should be used in place of any parameter. The 'Circle' basis function requires a radius parameter.

Here are two example random processes:

```
s = NDProcess ( {'Box'; 'Circle'; 'Impulse'} , {[1 1]; .2; []} , [0 0; 0 0; 0 0], [1; -1; 1]);
r = NDProcess ( 'Box' , [.1 .1 .1; .1 .1 .1] , [0 .25 .25; 0 .25 -.25] , [1 1] ) ;
```

Random process s is two-dimensional. It consists of three basis functions, a Box, a Circle and an Impulse. The Box spans the entire fundamental period; it has width 1 in each dimension and is centered at the origin. The Circle is of radius .2 and centered at the origin. Because the Circle's coefficient is -1 the effect is of a rectangle with a circle cut out of it. An impulse is added back in at the origin. Note that the param entries must be split into a cell array vector because the PSD consists of different basis function types.

Process r is three-dimensional, and is composed only of Box-type basis functions. Since it is a single-type PSD, the param entry is a matrix. The freqs matrix implies a 3-D process by its width, and that there are two copies of a Box by its height.

Chapter 5

Design Examples

5.1 A Simple Notch Filter

5.1.1 Gridded Formulation

An example using L_∞ error measure is a simple linear-phase FIR notch filter. To ensure linear phase, the impulse response is constrained to be symmetrical about the origin. Therefore, a length-N optSequence is allocated and shifted to the right, along with a center tap at the origin.

```
rt = optSequence(N, ov) | 1;
ct = optSequence(1, ov);
```

The filter impulse response is then constructed by

```
h = rt' + ct + rt;
```

The stopband and its Fourier transform are specified by

```
df = 1/(50*N);
fsb = [0.21:df:0.29];
Hsb = real(fourier(h, fsb));
```

where `df` is the discretization step Δf described above. Since $H(f)$ is real by construction, real is used to eliminate computational noise in the imaginary components. The passbands and their Fourier transforms are specified similarly by

```
fpb = [0:df:0.15 0.35:df:0.5];
Hpb = real(fourier(h, fpb));
```

The constraints are then assembled into a cell array, constraining the stopband by an auxiliary optimization variable delta and the passbands by a constant.

```
delta = optVar(ov);
lev = 0.1/20;
constr = {-delta<Hsb, Hsb<delta, ...
          Hpb<10^lev, 10^-lev<Hpb};
```

Finally minimize is called by

```
soln = minimize(delta, constr, ov, 'sedumi');
```

The impulse and frequency responses of the optimized filter, with $N = 15$, are shown in Fig. 5.1.

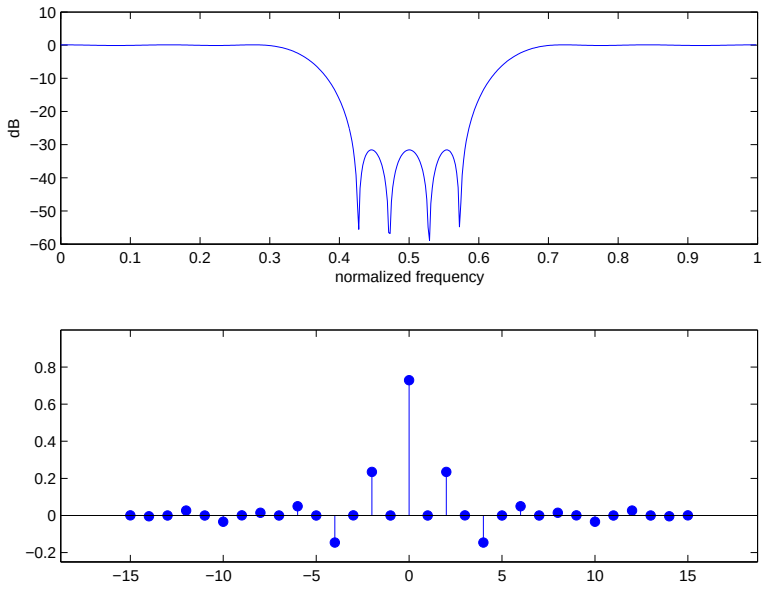

Figure 5.1: Notch Filter Impulse and Frequency Responses.

The optimal stopband supression can be found by examining the optimal value of delta:

```
db(double(optimal(delta, soln)))
```

which gives -31.6 dB, which is verified by a glance at Fig. 5.1.

5.1.2 Random Process Formulation

The notch filter response can also be expressed using a random process formulation. This entails specifying a new objective and set of constraints, while keeping the same impulse response specification as above.

The following creates two random processes; pp for the passband and ps for the stopband. Recall that a Process is created using the call Process(basis, offset, Coeff).

```
pp = Process('Triangle',0.5,[1 1 1 0 0 0 0 1 1 1 zeros(1,10)]);
ps = Process('Box',0.5,[0 0 0 0 1 1 0 0 0 0 zeros(1,10)]);
```

The coefficient vectors are of length 20, so a basis function corresponding to entry k of the coefficient vector is centered at frequency $k/20$. The offset of 0.5 means that the entire PSD is shifted by $0.5 \times 1/20$. The PSDs are shown in Fig. 5.2.

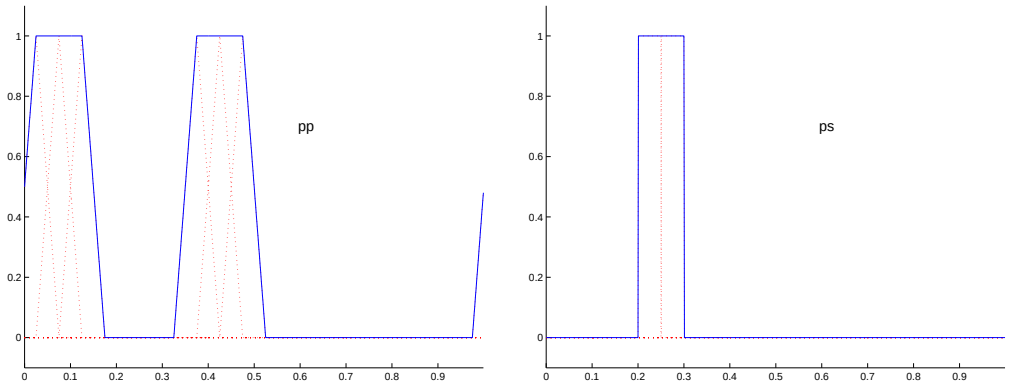

Figure 5.2: Random Process PSDs.

For both the stopband and passband, we wish to express a quantity that is to be bounded. In the case of the stopband, this is simply the average output power. For the passband, we instead use the power of the difference between the filtered process pp .* h and the unfiltered process pp. The powers are computed using the following:

```
Ppwr = pwr(pp.*h-pp)/pwr(pp);
Spwr = pwr(ps.*h)/pwr(ps);
```

These powers, being quadratic quantities, may only be bounded from above by a constant or another quadratic, to ensure that the optimization problem is convex. We bound the passband power by an auxiliary optimization variable, and we bound the stopband power by a constant:

```
delta = optVar(ov);
constr = {Ppwr < delta.^2, Spwr < 10^(-4) };
```

minimize is then called by

```
soln = minimize(delta, constr, ov, 'sedumi');
```

The impulse and frequency responses of the optimized filter are shown in Fig. 5.3.

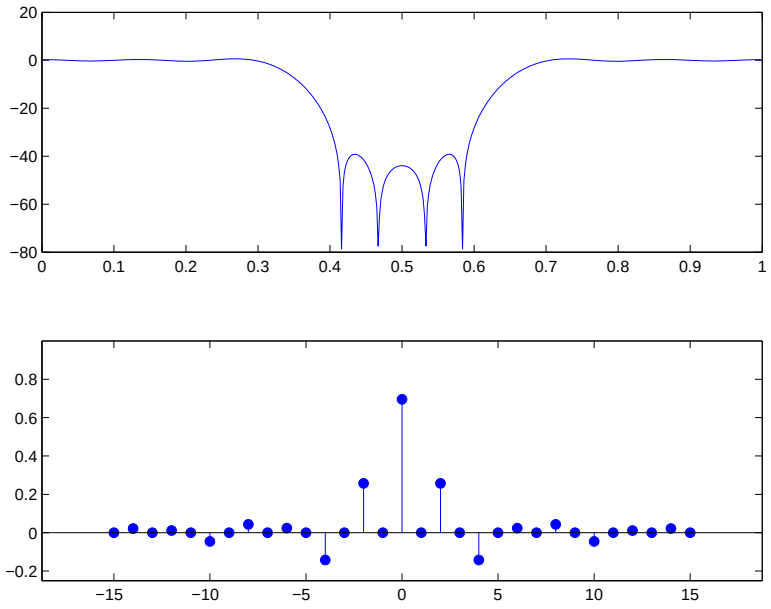

Figure 5.3: Notch Filter Impulse and Frequency Responses.

5.2 Annular Filter

Here we design a two-dimensional filter with an annular frequency response. Since the desired frequency response is circlarly symmetric, we can use the impulse response of the example in section 3.4.1 on page 20.

Using the same random process strategy as for the first example, we define passband and stopband processes:

```
pp = NDProcess('Circle', [.255;.245], [0 0; 0 0], [1; -1]);
ps = NDProcess({'Box'; 'Circle'; 'Circle'}, {[1 1]; .32; .18}, ...
       [0 0; 0 0; 0 0], [1; -1; 1]);
```

The desired passband is a thin annulus between radii .245 and .255. This is accomplished by subtracting a circle of the smaller radius from a circle of the larger radius, and hence the coefficient vector is [1; -1]. Both circles are centered at the origin.

The stopband is more complicated; starting with a box to indicate the entire fundamental period of the frequency response, we subtract a circle slightly larger than the outer ring of the annulus, and add back to the stopband a circle slightly smaller than the inner ring of the annulus.

Since this process is composed of different types of basis functions, the function call to NDProcess is a bit more complicated. The basis function types must now be specified for each instance of a basis function in a cell array column vector. A cell array is created using the and the entries are separated by semicolons.

Similarly, the parameters for each basis function instance must be given in separate entries of a cell array column vector. `'Box'` requires a width vector [1 1] which indicates a width of 1 in the x-direction and a width of 1 in the y-direction. `'Circle'` requires only a scalar for the radius.

The frequency shift matrix and coefficient vector are given as usual.

The various powers are computed as before. The power of the stopband process through the filter is to be constrained to be small. The passband is treated differently; in that case, we minimize the difference in power between the unfiltered and filtered passband process.

```
Ppwr = pwr(pp.*aa-pp);
Spwr = pwr(ps.*aa)/pwr(ps);
```

We bound the passband power by an auxiliary optimization variable, and we bound the stopband power by a constant:

```
delta = optVar(ov);
constr = {Ppwr < delta.^2, Spwr < 1e-3 };
```

minimize is then called by

```
soln = minimize(delta, constr, ov, 'sedumi');
```

The frequency response of the optimized filter is shown in Fig. 5.4.

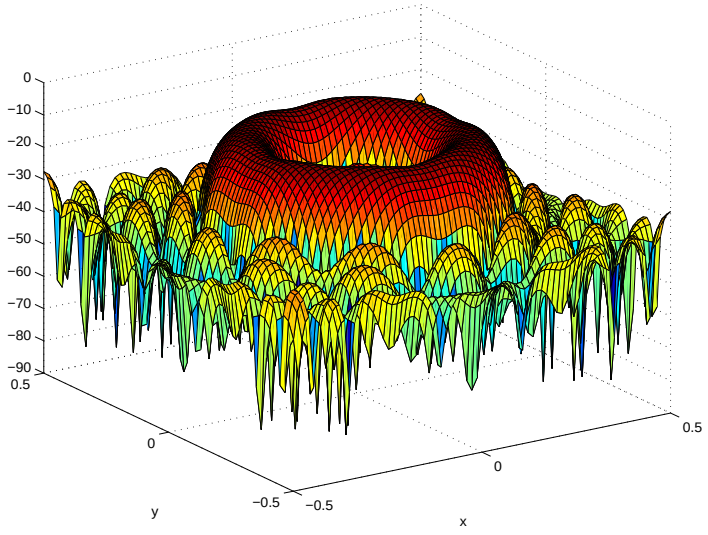

Figure 5.4: Annular Filter Frequency Response.

Chapter 6

OPT Reference

6.1 MATLAB Library

This section of the user's guide contains detailed descriptions of all the MATLAB-callable functions. On-line help is also available — if you execute `help` on one of the function names, MATLAB displays an abbreviated version of the reference entry.

6.1.1 Engine Functions

eqreduce

Purpose

Equality constraint reducer.

Syntax

```
[H, x0] = eqreduce(constr)
```

Description

Returns the nullspace `H` of equality constraints and vector `x0` such that
`x = H.'y + x0`
in terms of reduced-dimension vector `y`.

fixcase

Purpose

Case bug workaround.

Syntax

```
y = fixcase(x)
```

Description

Workaround for bug in PC/Windows version that requires lowercase references to class members. Currently has no effect on other platforms.

InitOpt

Purpose

Initialization script for OPT optimization toolbox.

Syntax

```
InitOpt
```

Description

`InitOpt` must be called by all design programs using the toolbox. `InitOpt` sets up the global data structure `OPT_DATA` which contains the following data:

`OPT_DATA.pools`	:	vector of optimization pool sizes
`OPT_DATA.procs`	:	array of base random processes
`OPT_DATA.procs().basis`	:	basis function
`OPT_DATA.procs().offset`	:	basis offset
`OPT_DATA.procs().Coeff`	:	basis frequency coefficients
`OPT_DATA.procs().coeff`	:	basis time coefficients
`OPT_DATA.ctprocs`	:	same as procs, but for c.t.
`OPT_DATA.casebug`	:	1 to workaround windows mixed-case classnamebug

See Also

`newOptSpace`	Allocate new pool of optimization variables.

mfactor

Purpose

Matrix factorization.

Syntax

```
S = mfactor(Q, eigtol)
```

Description

Factors a symmetric, positive definite matrix into `S'*S`.

minimize

Purpose

Interface to optimization engines.

Syntax

```
[soln] = minimize(obj, constr, ov, method)
```

Description

`minimize` uses an installed optimization engine to solve the problem.

`obj` is an `optVector`which specifies the objective function.

`constr` specifies the constraints. It is a cell array of `LinConstr` and `SOCConstr` type constraints.

`ov` specifies the pool (optSpace) of variables to optimize.

`method` specifies the optimization pacakage to be used, placed in single quotes. Currently, `eig`, `geneig`, `loqo`, `loqosocp`, `loqosoclp`, `boydsocp`, `sdppack` and `sedumi` are supported.

newOptSpace

Purpose

Allocate variable pool.

Syntax

```
pool = newOptSpace
```

Description

`pool = newOptSpace` allocates a new pool (optSpace) of optimization variables and returns a pointer to it in `pool`.

See Also

`InitOpt`	Initialization script for OPT optimization toolbox.
`optSpace`	Class of pools of optimization variables.

optVar

Purpose

Create a single optimized variable.

Syntax

```
opt = optVar(ov)
```

Description

Creates a single optimized variable in optSpace `ov`. This is an alias for `opt = optVector(1,ov)`.

See Also

`optVector`	Optimization variable vector constructor.

qm

Purpose

Minimal Sum-of-Products form.

Syntax

```
fcn = qm(minterms, numvars)
```

Description

Reduces the Boolean function expressed by `minterms` into the minimal sum-of-products form using the Quine-McCluskey method. `fcn` is a cell array of essential prime implicant terms. Terms are expressed as arrays of indices – so 1 indicates the least significant variable, and so on. A negative index indicates a complemented variable.

Examples

A result of `{[1 -4]; [-3]; [2 5]}` indicates the minimized function is $F(a, b, c, d, e) = ad' + c' + be$ for a being the LSV, and so on.

wherein

Purpose

Search function.

Syntax

```
m = wherein(x, y)
```

Description

If `x` and `y` are sets of unique numbers, `wherein(x, y)` returns a vector of the locations in `x` of the entries of `y`.

`wherein` returns `NaN` if an entry of `y` is not found in `x`.

6.1.2 Continuous-Time Processes

CTProcess

Purpose

Continuous-time process constructor.

Syntax

```
[rp] = CTProcess(basis, scale, freqs, Coeff)
```

Description

`CTProcess` initializes `rp` as a wide-sense-stationary zero-mean process whose spectrum is defined by shifted and weighted basis functions. The quadruple `(basis,scale,freqs,Coeff)` is placed in a global table and is referred to through its index, allowing multiple processes to refer to the same base process.

`basis` can be `'Box'` or `'Triangle'`.

`scale` sets the scaling of the basis function.

`freqs` is a vector of frequencies at which a basis function is added. `freqs` is sorted and checked for duplicate entries as well as non-real entries.

`Coeff` is a vector of coefficients which weight the basis functions. `Coeff` must be of the same length as `freqs`.

CTProcess/char, display

Purpose

Output functions for class CTProcess.

Syntax

```
c = char(rp)
display(rp)
```

Descripition

`char` converts class CTProcess to character for output. `display` is command window display of class CTProcess.

CTProcess/get_pool

Purpose

Pool extraction function.

Syntax

```
pool = get_pool(rp)
```

Description

`get_pool` returns the pool of the system component of the process `rp`.

CTProcess/minus

Purpose

Process subtraction.

Syntax

```
[rpmin] = minus(rpa, rpb)
```

Description

`minus` gives the difference of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpmin] = minus(rpa, rpb)` is called for the syntax '`rpa - rpb`'.

CTProcess/plot

Purpose

Graphical display of random process.

Syntax

```
hand = plot(rp, f, options)
```

Description

`plot(rp)` graphs the power spectrum of the processes making up `rp` over the set frequencies for which the power spectrum has support.

`plot(rp, f)` graphs the power spectrum of the processes making up `rp` at the frequencies given in the vector `f`.

`hand = plot(...)` returns the plot handle.

`options` can include:

- `'o'` or `'overlay'`: in addition to plotting the power spectrum, overlays a plot of the individual base processes.
- `'nooverlay'`: plots only the overall power spectrum. This is the default.
- `'p'` or `'print'`: uses options suitable for printer output. This option will override some user-set plotting defaults.
- `'noprint'`: uses default plotting options. This is the default.

CTProcess/plus

Purpose

Process addition.

Syntax

```
[rpsum] = plus(rpa, rpb)
```

Description

`plus` gives the sum of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpsum] = plus(rpa, rpb)` is called for the syntax '`rpa + rpb`'.

CTProcess/pwr

Purpose

Power in random process.

Syntax

```
[P] = pwr(rp)
```

Description

`pwr` returns the power in process `rp` as an optimizable quadratic quantity of type `optQuad`.

CTProcess/times

Purpose

Process convolution.

Syntax

```
[rpconv] = times(a, b)
```

Description

Result is convolution of a process with an affine sequence (system). One input is a random process, and the other must be a sequence.

`[rpconv] = times(a, b)` is called for the syntax '`a .* b`'.

CTProcess/uminus

Purpose

Process negation.

Syntax

```
[rpmin] = uminus(rp)
```

Description

Negates the system associated with process `rp`.

`[rpmin] = uminus(rp)` is called for the syntax '`-rp`'.

6.1.3 Lattices

Lattice

Purpose

Lattice object constructor.

Syntax

```
[lat] = Lattice()
[lat] = Lattice(M)
```

Description

`[lat] = Lattice()` creates an empty Lattice object.

`[lat] = Lattice(M)` creates a Lattice object with basis matrix `M`.

Lattice/char, display

Purpose

Output functions for class Lattice.

Syntax

```
c = char(lat)
display(lat)
```

Description

`char` converts class Lattice to character for output. `display` is command window display of class Lattice.

Lattice/get_M

Purpose

Lattice basis matrix extraction.

Syntax

```
M = get_M(lat)
```

Description

Returns the basis matrix of the lattice `lat`.

Lattice/get_off

Purpose

Lattice offset extraction.

Syntax

```
off = get_off(lat)
```

Description

Returns the offset vector of the lattice `lat`.

Lattice/get_scale

Purpose

Lattice scale extraction.

Syntax

```
scale = get_scale(lat)
```

Description

Returns the scale factor of the lattice `lat`.

Lattice/inbox

Purpose

Points within bounding box.

Syntax

```
locs = inbox(lat, box)
```

Description

Returns the cartesian coordinates of the points on the lattice `lat` located within the bounding box `box`. `box` is expressed as a vector: $[xmin\, xmax\, ymin\, ymax\, \ldots]$ where the intervals are distances along the lattice basis vectors.

Lattice/mtimes

Purpose

Lattice scaling.

Syntax

```
slat = mtimes(lat, scale)
```

Description

Scales the lattice by a factor of `scale`.

`slat = mtimes(lat, scale)` is called for the syntax `lat * scale`.

Lattice/plus

Purpose

Lattice shifting.

Syntax

```
olat = plus(lat, off)
```

Description

Shifts the lattice by an offset; that is, any points on the base lattice will be shifted by the offset vector. The length of `off` must be the same as the dimension of `lat`.

`olat = plus(lat, off)` is called for the syntax `lat + off`.

6.1.4 Linear Constraints

LinConstr

Purpose

Linear constraint constructor.

Syntax

```
[lconstr] = LinConstr(rel, vect)
```

Description

`LinConstr` initializes `lconstr` as a linear constraint. A linear constraint is of the form

`vect < 0` or `vect == 0`

where `vect` is of type `optVector`.

The constraint is stored in the form

`A*x < b` or `A*x == b`

where `A` and `b` are the linear and constant terms of `vect`.

For the relation '`<`', imaginary parts of the constraint are ignored. Complex parts of an '`==`' constraint are converted to two real constraints.

LinConstr/char, display

Purpose

Output functions for class `LinConstr`.

Syntax

```
c = char(constr)
display(constr)
```

Descripition

`char` converts class `LinConstr` to character for output. `display` is command window display of class `LinConstr`.

LinConstr/get_A, get_b, get_rel

Purpose

`A`, `b` and `rel` extraction functions.

Syntax

```
A = get_A(lconstr)
b = get_b(lconstr)
rel = get_rel(lconstr)
```

Description

`A = get_A(lconstr)` returns `A` matrix of linear constraint `lconstr`.

`b = get_b(lconstr)` returns `b` vector of linear constraint `lconstr`.

`rel = get_rel(lconstr)` returns relation of linear constraint as character array. Can be '==' or '<'.

LinConstr/length

Purpose

Number of linear constraints.

Syntax

```
[len] = length(constr)
```

Description

Returns the number of linear constraints in `constr`.

6.1.5 Multi-Dimensional Processes

NDProcess

Purpose

Continuous-time process constructor.

Syntax

```
[rp] = NDProcess(basis, param, freqs, Coeff)
```

Description

`NDProcess` initializes `rp` as a wide-sense-stationary zero-mean process whose spectrum is defined by shifted and weighted basis functions. The quadruple `(basis,param,freqs,Coeff)` is placed in a global table and is referred to through its index, allowing multiple processes to refer to the same base process.

- `basis` : basis function for the process's PSD. Currently allowed are `'Box'`, `'Triangle'`, `'Impulse'` and `'Circle'`, if the PSD is to be composed of a single type of basis function. For a PSD composed of several types, basis must be a cell column vector with each individual basis function named. For example, 'Box'; 'Circle'; 'Circle' specifies a PSD composed of a 'Box' and two 'Circles'.
- `param` : parameters for basis functions. For PSD composed of a single type of basis function, param should be a matrix, each row of which contains the parameters for another basis function. For several types, param should be a cell column vector with each entry the parameters for an individual basis function.
  - `'Box'`, `'Triangle'` : width parameters are required, one positive, real width for each dimension. For example, if this is a one-type PSD, param should be a matrix, each row of which contains the widths for another basis function. For a several-type PSD, each `'Box'` or `'Triangle'` entry should be a row vector.
  - `'Circle'` : currently, only 2-D is supported. A positive, real radius parameter is required.
  - 'Impulse' : no parameters required. For one-type PSD, use the empty matrix []. For several-type PSD, each `'Impulse'` entry should be the empty matrix [].

- `freqs` : matrix of frequency shifts. Each row gives the shifts for another basis function. The PSD is the sum of basis functions at each of the frequency shifts in `freqs`. The width of the `freqs` matrix determines the dimension of the process.
- `Coeff` : vector of coefficients of basis functions. Must have length equal to the number of rows of `width` and `freqs`.

NDProcess/char, display

Purpose

Output functions for class NDProcess.

Syntax

```
c = char(rp)
display(rp)
```

Descripition

`char` converts class NDProcess to character for output. `display` is command window display of class NDProcess.

NDProcess/get_pool

Purpose

Pool extraction function.

Syntax

```
pool = get_pool(rp)
```

Description

`get_pool` returns the pool of the system component of the process `rp`.

NDProcess/minus

Purpose

Process subtraction.

Syntax

```
[rpmin] = minus(rpa, rpb)
```

Description

`minus` gives the difference of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpmin] = minus(rpa, rpb)` is called for the syntax '`rpa - rpb`'.

NDProcess/plot

Purpose

Graphical display of random process.

Syntax

```
hand = plot(rp, f, options)
```

Description

`plot(rp)` graphs the power spectrum of the processes making up `rp` over the set frequencies for which the power spectrum has support.

`plot(rp, f)` graphs the power spectrum of the processes making up `rp` at the frequencies given in the vector `f`.

`hand = plot(...)` returns the plot handle.

`options` can include:

- `'o'` or `'overlay'`: in addition to plotting the power spectrum, overlays a plot of the individual base processes.
- `'nooverlay'`: plots only the overall power spectrum. This is the default.
- `'p'` or `'print'`: uses options suitable for printer output. This option will override some user-set plotting defaults.
- `'noprint'`: uses default plotting options. This is the default.

NDProcess/plus

Purpose

Process addition.

Syntax

```
[rpsum] = plus(rpa, rpb)
```

Description

`plus` gives the sum of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpsum] = plus(rpa, rpb)` is called for the syntax '`rpa + rpb`'.

NDProcess/pwr

Purpose

Power in random process.

Syntax

```
[P] = pwr(rp)
```

Description

`pwr` returns the power in process `rp` as an optimizable quadratic quantity of type `optQuad`.

NDProcess/times

Purpose

Process convolution.

Syntax

```
[rpconv] = times(a, b)
```

Description

Result is convolution of a process with an affine sequence (system). One input is a random process, and the other must be a sequence.

`[rpconv] = times(a, b)` is called for the syntax '`a .* b`'.

NDProcess/uminus

Purpose

Process negation.

Syntax

```
[rpmin] = uminus(rp)
```

Description

Negates the system associated with process `rp`.

`[rpmin] = uminus(rp)` is called for the syntax '`-rp`'.

6.1.6 Optimizable Arrays

optArray

Purpose

Optimized affine multi-dimensional sequence constructor.

Syntax

```
[seq] = optArray(locs,pool)
[seq] = optArray(locs, vect)
[seq] = optArray(locs, vect, 'a')
[seq] = optArray(locs, x)
[seq] = optArray(locs, x, 'c')
[seq] = optArray(lattice, region, pool)
[seq] = optArray()
```

Description

`[seq] = optArray(locs,pool)` creates an optimized affine sequence at locations given in `locs` using pool of optimization variables `pool`.

`[seq] = optArray(locs, vect)`
`[seq] = optArray(locs, vect, 'a')` creates an optimized affine sequence from `optVector vect` at locations given in `locs`.

`[seq] = optArray(locs, x)`
`[seq] = optArray(locs, x, 'c')` creates a constant sequence from vector or sequence `x` at time indices given in `locs`.

`[seq] = optArray(lattice, region, pool)` creates an optimized affine sequence at locations on a `Lattice` within a `Region`.

`[seq] = optArray()` creates an empty sequence.

optArray/char, display

Purpose

Output functions for class `optArray`.

Syntax

```
c = char(seq)
display(seq)
```

Descripition

`char` converts class `optArray` to character for output. `display` is command window display of class `optArray`.

Displays the kernel, offset into variable pool, pool number and support locations.

optArray/ctranspose

Purpose

Conjugate flip.

Syntax

```
[tseq] = ctranspose(seq)
```

Description

Flips `seq` about the origin and conjugates it.

`[tseq] = ctranspose(seq)` is called for the syntax `seq'`.

optArray/flip

Purpose

Sequence flip.

Syntax

```
tseq = flip(seq, dim)
```

Description

Flips the sequence in dimension `dim`.

Examples

To flip about the y-axis in 2-D, use `flip(seq,1)`.
To flip about the xy-plane in 3-D, use `flip(seq,3)`.

optArray/fourier

Purpose

Sequence Fourier transform.

Syntax

```
G = fourier(g, f)
```

Description

Returns Fourier transform of `optArray g` at `f` as an `optArray`, `f` is a matrix in which each row indicates a point to evaluate the freqeuncy response.

optArray/get_dim, get_locs

Purpose

Dimension and locations extraction functions.

Syntax

```
dim = get_dim(seq)
locs = get_locs(seq)
```

Description

`dim = get_dim(seq)` returns the dimension of `seq`.

`locs = get_locs(seq)` returns the locations of support of `seq` as a matrix, in which each row contains the coordinates of a single point.

optArray/isempty

Purpose

True for empty `optArray`.

Syntax

```
[em] = isempty(seq)
```

Description

Returns 1 if `seq` is an empty `optArray`, and 0 otherwise.

optArray/minus

Purpose

Sequence subtraction.

Syntax

```
[mseq] = minus(a, b)
```

Description

Subtracts `optArray b` from `optArray a`.

`[mseq] = minus(a, b)` is called for the syntax `a - b`.

optArray/mtimes

Purpose

Sequence element-wise multiplication.

Syntax

```
[mseq] = mtimes(a, b)
```

Description

Muliplies `a` by `b`. It is legal to multiply a constant sequence by a constant sequence of the same dimension; and an optimized sequence by a constant sequence of the same dimension or a scalar.

`[mseq] = mtimes(a, b)` is called for the syntax `a * b`.

optArray/optimal

Purpose

Return optimal sequence.

Syntax

```
optseq = optimal(opt, soln)
```

Description

Returns optimal constant sequence as given by `soln`.

optSequence/or

Purpose

Sequence linear shift.

Syntax

```
[sseq] = or(seq, offset)
```

Description

Shifts the sequence by `offset`. `offset` must be a vector of the same dimension as `seq`.

`[sseq] = or(seq, offset)` is called for the syntax `seq | offset`.

optArray/plot

Purpose

Graphical representation of `optArray`.

Syntax

```
plot(seq)
plot(seq, ...)
```

Description

Plots `optArray seq`, which can be 1-, 2- or 3-dimensional.

`plot(seq, ...)` plots seq with comma-separated list of plotting options. options can be one or more of:

- `'color`, `'nocolor'`: whether to use color coding in plots; green if constant, blue if dependent on the optimization variables, and red if affine. Also, whether to plot a star for complex values and circle for real value, or only circles.
  Default: `'nocolor'`
- `'label'`, `'nolabel'`: whether to print labels of array elements.
  Default: `'nolabel'`
- `'print'`, `'noprint'`: whether to use formatting suitable for printout.
  Default: `'noprint'`
- `'stagger'`, `'nostagger'`: whether to stagger labels in 1-D plots readability
- `'FontSize'`: font size of labels. To be followed the font size value.
- `'LineWidth'`: Width of line used in stem plot. Followed by line width value.
- `'MarkerSize'`: Size of marker in stem plots.

optArray/plus

Purpose

Sequence addition.

Syntax

```
[pseq] = plus(a, b)
```

Description

Adds `optArrays` `a` and `b`.

`[pseq] = plus(a, b)` is called for the syntax `a + b`.

optArray/rot

Purpose

Sequence rotation.

Syntax

```
[tseq] = rot(seq, ang, dims)
```

Description

Rotates the sequence by `ang` radians in dimensions `dims`. `dims` is a length-2 vector indicating the subspace in which the rotation is performed. For example, `rot(seq, pi/4, [1 3])` will rotate `seq` 45 degrees in the xz-subspace.

optArray/set_locs

Purpose

Sequence locations insertion function.

Syntax

```
[tseq] = set_locs(seq, locs)
```

Description

Returns the sequence `seq` with support locations replaced by `locs`. `locs` must be a matrix with each row giving the coordinates of an individual point.

optArray/subsref

Purpose

Sequence subscripted reference.

Syntax

```
subs = seq(locs)
subs = seq([xmin xmax ymin ymax ...])
subs = seq(n, 'i')
subs = seq(reg)
```

Description

`seq(locs)` returns a subsequence at locations given in `locs`.

`subs = seq([xmin xmax ymin ymax ...])` returns a subsequence located in a hypercube which is described using the intervals $[x_{\min}, x_{\max}]$ and so forth.

`subs = seq(n, 'i')` returns a subsequence using 'Matlab' indices. For example, `seq(1:4, 'i')` returns a sequence containing the first through fourth elements of `seq`.

`subs = seq(reg)` returns a subsequence with only those elements within a region indicated by the `Region` object `reg`.

optArray/times

Purpose

Sequence convolution.

Syntax

```
[cseq] = times(a, b)
```

Description

Convolves `optArray`s `a` and `b`, one of which must not depend on the optimization variables (constant sequence).

`[cseq] = times(a, b)` is called for the syntax `a .* b`.

optArray/transpose

Purpose

Sequence flip.

Syntax

```
[tseq] = transpose(seq)
```

Description

Flips `seq` about the origin.

`[tseq] = transpose(seq)` is called for the syntax `seq.'`.

6.1.7 Quadratic Optimizables

optQuad

Purpose

Constructor for quadratic form optimization expressions.

Syntax

```
[quad] = optQuad(Q, xoffr, xoffc, pool)
[quad] = optQuad
```

Description

`[quad] = optQuad(Q, xoffr, xoffc, pool)` returns a fully specified quadratic with kernel matrix `Q` at offset (`xoffr`,`xoffc`) from optimization variable space `pool`.

`[quad] = optQuad` returns an empty quadratic.

optQuad/char, display

Purpose

Output functions for class `optQuad`.

Syntax

```
c = char(quad)
display(quad)
```

Descripition

`char` converts class `optQuad` to character for output. `display` is command window display of class `optQuad`.

Displays the kernel, offsets into variable pool, and pool number.

optQuad/eq

Purpose

Quadratic equality constraint.

Syntax

```
[eqconstr] = eq(a,b)
```

Description

Forms constraint of type `a == b`, where `a` is an `optQuad` and `b` is an `optQuad` or a scalar. The constraint is a second-order-cone constraint, of type `SOCConstr`.

`[eqconstr] = eq(a, b)` is called for the syntax '`a == b`'.

optQuad/get_kernel

Purpose

Kernel matrix extraction function.

Syntax

```
Q = get_kernel(quad, format)
```

Description

Extracts kernel matrix `Q` from `quad`. Quadratic is of the form

`[1;x].'*[k c.'/2;c/2 H]*[1;x] = x.'*H*x + c.'*x + k`

`format` can be: `'sparse'`, `'full'`, `'purequad'`, `'affine'`, `'linear'` or `'const'`.

`'sparse'`	returns nonzero portion as a sparse matrix
`'full'`	returns the whole kernel as a sparse matrix
`'purequad'`	returns `H`, the purely quadratic kernel
`'affine'`	returns `[k;c]`,the affine part of the quadratic
`'linear'`	returns `c`, the linear part of the quadratic
`'const'`	returns `k`, the constant part

optQuad/get_pool

Purpose

Pool extraction function.

Syntax

```
pool = get_pool(quad)
```

Description

Returns pool number of `quad`.

optQuad/get_xoffc

Purpose

`xoffc` extraction function.

Syntax

```
xoffc = get_xoffc(quad)
```

Description

Returns `xoffc`, offset of columns of kernel into the pool of optimizables.

optQuad/get_xoffr

Purpose

`xoffr` extraction function.

Syntax

```
xoffr = get_xoffr(quad)
```

Description

Returns `xoffr`, offset of rows of kernel matrix into the pool of optimizables.

optQuad/ispure

Purpose

Test if variable is purely quadratic.

Syntax

```
[pure] = ispure(quad)
```

Description

Returns true (1) if `quad` contains no constant or linear terms; that is, if `quad` is purely quadratic. Otherwise returns false (0).

optQuad/lt

Purpose

Quadratic inequality constraint.

Syntax

```
[constr] = lt(a, b)
```

Description

Forms constraint of type `a < b`, where `a` is an `optQuad` and `b` is an `optQuad` or a scalar. The constraint is a second-order-cone constraint, of type `SOCConstr`.

`[constr] = lt(a, b)` is called for the syntax '`a < b`'.

optQuad/mrdivide

Purpose

Scalar division.

Syntax

```
[c] = mrdivide(a, b)
```

Description

Divides `optQuad a`'s kernel matrix by scalar `b`.

`[c] = mrdivide(a, b)` is called for the syntax '`a / b`'.

optQuad/mtimes

Purpose

Scalar multiplication.

Syntax

```
[c] = mtimes(a, b)
```

Description

Multiplies `optQuad a`'s kernel matrix by scalar `b`.

`[c] = mtimes(a, b)` is called for the syntax '`a * b`'.

optQuad/optimal

Purpose

Optimal quadratic.

Syntax

```
optq = optimal(quad, soln)
```

Description

Given a solution `soln` returned by the `minimize` function, return optimal sequence.

optQuad/plus

Purpose

Quadratic addition.

Syntax

```
[c] = plus(a, b)
```

Description

Adds `optQuad`s `a` and `b`.

`[c] = plus(a, b)` is called for the syntax '`a + b`'.

optQuad/set_kernel

Purpose

Kernel insertion function.

Syntax

```
quadout = set_kernel(quad, kernel)
```

Description

Replaces `quad`'s kernel matrix with `kernel`.

optQuad/set_xoffc

Purpose

`xoffc` insertion function.

Syntax

```
quadout = set_xoffc(quad, xoffc)
```

Description

Replaces `quad`'s colunm offset with `xoffc`. Does not perform error checking.

optQuad/set_xoffr

Purpose

`xoffr` insertion function.

Syntax

```
quadout = set_xoffr(quad, xoffr)
```

Description

Replaces `quad`'s row offset with `xoffr`. Does not perform error checking.

6.1.8 Quadratic Optimizable Vectors

optQuadVector

Purpose

Optimized quadratic vector constructor.

Syntax

```
[qvect] = optQuadVector(aff, absflag, sqflag)
```

Description

Creates an optimized affine quadratic vector from `optVector aff`. `absflag` and `sqflag` are set to the values given.

optQuadVector/char, display

Purpose

Output functions for class `optQuadVector`.

Syntax

```
c = char(quadvec)
display(quadvec)
```

Descripition

`char` converts class `optQuadVector` to character for output. `display` is command window display of class `optQuadVector`.

Displays the kernel, offset into variable pool, pool number, and flags.

optQuadVector/get_absflag

Purpose

`absflag` extraction function.

Syntax

```
absflag = get_absflag(quadvec)
```

Description

Returns value of `absflag` for `quadvec`. This indicates whether `quadvec` was formed as the result of an `abs` operation.

optQuadVector/get_optVector

Purpose

`optVector` extraction function.

Syntax

```
vect = get_optVector(qvect)
```

Description

Returns only the `optVector`portion of `qvect`.

optQuadVector/get_sqflag

Purpose

`sqflag` extraction function.

Syntax

```
sqflag = get_sqflag(quadvec)
```

Description

Returns value of `sqflag` for `quadvec`. This indicates whether `quadvec` was formed as the result of a squaring operation.

optQuadVector/lt

Purpose

Quadratic inequality constraint.

Syntax

```
[constr] = lt(a, b)
```

Description

Forms constraint of type `a < b`, where `a` is an `optQuadVector` and `b` is an `optQuadVector`, an `optVector`, or a scalar. The constraint is a second-order-cone constraint, of type `SOCConstr`.

`[constr] = lt(a, b)` is called for the syntax '`a < b`'.

optQuadVector/power

Purpose

Elementwise power.

Syntax

```
[qvect] = power(a, b)
```

Description

Raises `a` to the power `b`. `a` can only be raised to the second power, and `a` cannot already have been squared.

`[constr] = power(a, b)` is called for the syntax '`a .^ b`'.

optQuadVector/set_absflag

Purpose

`absflag` insertion function.

Syntax

```
quadout = set_absflag(quadin, absflag)
```

Description

Replaces `absflag` with new value.

optQuadVector/set_sqflag

Purpose

`sqflag` insertion function.

Syntax

```
quadout = set_sqflag(quadin, sqflag)
```

Description

Replaces `sqflag` with new value.

optQuadVector/sum

Purpose

Sum of terms in quadratic vector.

Syntax

```
[quadsum] = sum(qvect)
```

Description

Returns `quadsum` of class `optQuad`, the sum of the terms in `qvect`.

6.1.9 Optimizable Affine Sequences

optSequence

Purpose

Optimized affine sequence constructor.

Syntax

```
[seq] = optSequence(n, pool)
[seq] = optSequence(vect)
[seq] = optSequence(x)
[seq] = optSequence
```

Description

`[seq] = optSequence(n, pool)` creates an optimized affine sequence of length `n`.

`[seq] = optSequence(vect)` creates an optimized affine sequence from `optVector vec`.

`[seq] = optSequence(x)` creates a constant sequence from vector `x`.

`[seq] = optSequence` creates an empty sequence.

optSequence/char, display

Purpose

Output functions for class `optSequence`.

Syntax

```
c = char(seq)
display(seq)
```

Descripition

`char` converts class `optSequence` to character for output. `display` is command window display of class `optSequence`.

Displays the kernel, offset into variable pool, pool number and time offset.

optSequence/ctranspose

Purpose

Conjugate flip.

Syntax

```
[tseq] = ctranspose(seq)
```

Description

Flips `seq` about the origin and conjugates it.

`[tseq] = ctranspose(seq)` is called for the syntax `seq'`.

optSequence/fourier

Purpose

Sequence Fourier transform.

Syntax

```
G = fourier(seq, nu)
```

Description

Returns Fourier transform of `seq` at frequencies `nu` as an `optVector`.

optSequence/get_noff

Purpose

`noff` extraction function.

Syntax

```
noff = get_noff(seq)
```

Description

Returns time offset of `seq`. The true position of the first sequence element is `noff + 1`.

optSequence/ldivide

Purpose

Sequence decimation.

Syntax

```
[dseq] = ldivide(seq, M)
```

Description

Returns `seq` decimated by `M`. `M` must be a scalar integer.
`[dseq] = ldivide(seq, M)` is called for the syntax `seq .\ M` .

optSequence/le

Purpose

Sequence linear shift.

Syntax

```
[sseq] = le(seq, nshift)
```

Description

Shifts `seq` left by `nshift`. `nshift` must be a scalar integer.

`[sseq] = le(seq, nshift)` is called for the syntax `seq <= nshift`.

optSequence/minus

Purpose

Sequence subtraction.

Syntax

```
[mseq] = minus(a, b)
```

Description

Subtracts `optSequence b` from `optSequence a`.

`[mseq] = minus(a, b)` is called for the syntax `a - b`.

optSequence/mtimes

Purpose

Sequence element-wise multiplication.

Syntax

```
[mseq] = mtimes(a, b)
```

Description

Muliplies `a` by `b`. It is legal to multiply a constant sequence by a constant sequence; and an optimized sequence by a scalar or a constant sequence.

`[mseq] = mtimes(a, b)` is called for the syntax `a * b`.

optSequence/optimal

Purpose

Return optimal sequence.

Syntax

```
optseq = optimal(opt, soln)
```

Description

Returns optimal constant sequence as given by `soln`.

optSequence/or

Purpose

Sequence linear shift.

Syntax

```
[sseq] = or(seq, nshift)
```

Description

Shifts the sequence right by (delays by) `nshift`. `nshift` must be a scalar integer.
`[sseq] = or(a, b)` is called for the syntax `a | b`.

optSequence/plus

Purpose

Sequence addition.

Syntax

```
[pseq] = plus(a, b)
```

Description

Adds `optSequences` `a` and `b`.

`[pseq] = plus(a, b)` is called for the syntax `a + b`.

optSequence/rdivide

Purpose

Sequence interpolation.

Syntax

```
[dseq] = rdivide(seq, M)
```

Description

Interpolates `seq` by `M`. `M` must be a scalar integer.

`[dseq] = rdivide(seq, M)` is called for the syntax `seq ./ M`.

optSequence/subsref

Purpose

Sequence subscripted reference.

Syntax

```
subs = seq(n)  subs = seq(n1, n2)
```

Description

`seq(n)` returns an optimized scalar at time index `n`.

`seq(n1, n2)` returns a subsequence from time index `n1` to `n2`.

optSequence/times

Purpose

Sequence convolution.

Syntax

```
[cseq] = times(a, b)
```

Description

Convolves `optSequence`s `a` and `b`, one of which must not depend on the optimization variables (constant sequence).

`[cseq] = times(a, b)` is called for the syntax `a .* b`.

optSequence/transpose

Purpose

Sequence flip.

Syntax

```
[tseq] = transpose(seq)
```

Description

Flips `seq` about the origin.

`[tseq] = transpose(seq)` is called for the syntax `seq.'`.

6.1.10 Optimization Spaces

optSpace

Purpose

Class of optimization variables pools (`optSpace`s).

Syntax

```
[ov] = optSpace(spacename)
[ov] = optSpace
```

Description

`[ov] = optSpace(spacename)` creates an `optSpace` with name `spacename`. `spacename` should be a string.

`[ov] = optSpace` creates an `optSpace` with default name assigned.

See Also

`newOptSpace`	allocates a new space (pool) of optimization variables

6.1.11 Optimizable Vectors

optVector

Purpose

Optimized vector constructor.

Syntax

```
[opt] = optVector(n, pool)  [opt] = optVector(svec)  [opt] = optVector(otheropt)
[opt] = optVector(h, pool, xoff)  [opt] = optVector
```

Description

`[opt] = optVector(n, pool)` creates a vector of `n` new optimization variables in optSpace `pool`.

`[opt] = optVector(svec)` creates a constant (non-optimized) vector from `svec`.

`[opt] = optVector(otheropt)` creates a copy of `otheropt`.

`[opt] = optVector(h, pool, xoff)` assembles an `optVector` from its parts in one step. Assumes that the variables are already allocated. `h` is the kernel matrix, `pool` is the number of the desired optSpace, and `xoff` is the index into space `pool` of the first variable in the `optVector`.

`[opt] = optVector` creates an empty constant vector.

optVector/abs

Purpose

Elementwise absolute value.

Syntax

```
aquad = abs(opt)
```

Description

Returns an `optQuadVector` which is the elementwise absolute value of `opt`.

optVector/char, display

Purpose

Output functions for class `optVector`.

Syntax

```
c = char(opt)
display(opt)
```

Descripition

`char` converts class `optVector` to character for output. `display` is command window display of class `optVector`.

Displays the kernel, offset into variable pool and pool number.

optVector/energy

Purpose

Energy (sum of squares).

Syntax

```
[q] = energy(opt)
```

Description

Returns quadratic form (`optQuad`) representing energy, or a scalar if `opt` does not depend on optimization variables.

optVector/eq

Purpose

Linear equality constraint.

Syntax

```
[eqconstr] = eq(vect1, vect2)
```

Description

Creates a linear equality constraint of type `LinConstr`. At least one of `vect1` and `vect2` is an `optVector`.

`[eqconstr] = eq(vect1, vect2)` is called for the syntax `vect1 == vect2`.

optVector/get_h, get_pool, get_xoff

Purpose

Kernel matrix, pool and offset extraction functions.

Syntax

```
h = get_h(opt, format)
pool = get_pool(opt)
xoff = get_xoff(opt)
```

Description

`h = get_h(opt, format)` returns kernel matrix of `opt`.

`format` can be `'sparse'`, `'full'`, `'linear'` or `'const'`.
`'sparse'` returns the nonzero portion as a sparse matrix.
`'full'` returns the entire kernel matrix as a sparse matrix.
`'linear'` and `'const'` return just the pure linear and the constant portions, respectively, of the kernel matrix.

`pool = get_pool(opt)` returns the number of the pool (optSpace) used by `opt`.

`xoff = get_xoff(opt)` returns `xoff`, offset of `opt`'s first variable into its variable pool.

optVector/imag

Purpose

Complex imaginary part.

Syntax

```
[iopt] = imag(opt)
```

Description

Returns an `optVector` which is the imaginary part of `opt`.

optVector/isconst

Purpose

True for constant.

Syntax

```
[cn] = isconst(opt)
```

Description

Returns 1 if `opt` is constant (does not depend on optimization variables) and 0 otherwise.

optVector/islinear

Purpose

True if no constant terms.

Syntax

```
[lin] = islinear(opt)
```

Description

Returns 1 if `opt` is linear in the optimization variables (no constant terms) and 0 otherwise.

optVector/isscalar

Purpose

True for scalar.

Syntax

```
[sc] = isscalar(opt)
```

Description

Returns 1 if `opt` is a length-1 `optVector` (scalar), and 0 otherwise.

optVector/length

Purpose

Length of vector.

Syntax

```
[len] = length(opt)
```

Description

Returns the length of `opt`.

optVector/lt

Purpose

Linear inequality constraint.

Syntax

```
[lconstr] = lt(vect1, vect2)
```

Description

Creates a linear inequality constraint of type `LinConstr`. At least one of `vect1` and `vect2` must be an `optVector`.

`[lconstr] = eq(vect1, vect2)` is called for the syntax `vect1 < vect2`.

optVector/minus

Purpose

Affine subtraction.

Syntax

```
[maff] = minus(a, b)
```

Description

Returns an `optVector` which is the difference of `a` and `b`. Both `a` and `b` must be of type `optVector`.

`[maff] = minus(a, b)` is called for the syntax `a - b`.

optVector/mrdivide

Purpose

Scalar division.

Syntax

```
[dopt] = mrdivide(a, b)
```

Description

Divides `a` by `b`. `a` must be an `optVector`, and `b` must be a scalar.
`[dopt] = mrdivide(a, b)` is called for the syntax `a / b`.

optVector/mtimes

Purpose

Matrix-vector multiplication.

Syntax

```
[mopt] = mtimes(a, b)
```

Description

Multiplies `a` by `b`. Only one of `a` and `b` can depend on the optimization variables. `a` and `b` must be of the same length.

`[mopt] = mtimes(a, b)` is called for the syntax `a * b`.

optVector/numvars

Purpose

Number of optimization variables.

Syntax

```
nn = numvars(opt)
```

Description

Returns the number of optimization variables upon which `opt` depends (that is, the number of rows of `opt`'s kernel matrix).

optVector/optimal

Purpose

Optimal constant sequence.

Syntax

```
optvec = optimal(opt, soln)
```

Description

Returns optimal constant sequence based on `soln` furnished by `minimize`.

See Also

`minimize`	Interface to optimization engines.

optVector/plus

Purpose

Affine addition.

Syntax

```
[paff] = plus(a, b)
```

Description

Returns an `optVector` which is the sum of `a` and `b`. Both `a` and `b` must be of type `optVector`.

`[paff] = plus(a, b)` is called for the syntax `a + b`.

optVector/power

Purpose

Elementwise power.

Syntax

```
[aquad] = power(a, b)
```

Description

Returns `a` raised to the power `b` as an `optQuad`. `b` must be 2.
`[aquad] = power(a, b)` is called for the syntax `a .^ b`.

optVector/real

Purpose

Complex real part.

Syntax

```
[ropt] = real(opt)
```

Description

Returns an `optVector` which is the real part of `opt`.

optVector/set_h, set_pool, set_xoff

Purpose

Kernel matrix, pool and offset insertion functions.

Syntax

```
optout = set_h(opt, h)
optout = set_pool(opt, pool)
optout = set_xoff(opt, xoff)
```

Description

`optout = set_h(opt, h)` replaces kernel matrix of `opt` with `h`.

`optout = set_pool(opt, pool)` replaces variable pool number of `opt` with `pool`.

`optout = set_xoff(opt, xoff)` replaces index of `opt`'s first variable into variable pool with `xoff`. Does not check range of `xoff` or allocate additional variables.

optVector/subsref

Purpose

Vector subscripted reference.

Syntax

```
subvec = vec(s)
```

Description

Returns subvector with indices `s`.

optVector/sum

Purpose

Affine sum.

Syntax

```
[saff] = sum(aff)
```

Description

Returns sum of `aff` as type `optVector`.

optVector/times

Purpose

Element-wise multiplication.

Syntax

```
[mopt] = times(a, b)
```

Description

Returns element-wise product of `a` and `b` as an `optVector`. Only one of `a` and `b` can depend on the optimization variables.

`[mopt] = times(a, b)` is called for the syntax `a .* b`.

optVector/uminus

Purpose

Unary minus.

Syntax

```
[mopt] = uminus(opt)
```

Description

Negates the elements of `opt`.

`[mopt] = uminus(opt)` is called for the syntax `-opt`.

6.1.12 Discrete-Time Processes

Process

Purpose

Discrete-time process constructor.

Syntax

```
[rp] = Process(basis, offset, Coeff)
```

Description

`Process` initializes `rp` as a wide-sense-stationary zero-mean process whose spectrum is defined by shifted and weighted basis functions. The triple `(basis,offset,Coeff)` is placed in a global table and is referred to through its index, allowing multiple processes to refer to the same base process.

`basis` can be `'Box'` or `'Triangle'`.

`offset` determines the starting point in frequency for the process specification.

`Coeff` is a vector of coefficients which weight the basis functions. `Coeff` must be of the same length as `freqs`.

Process/char, display

Purpose

Output functions for class Process.

Syntax

```
c = char(rp)
display(rp)
```

Descripition

`char` converts class Process to character for output. `display` is command window display of class Process.

Process/get_pool

Purpose

Pool extraction function.

Syntax

```
pool = get_pool(rp)
```

Description

`get_pool` returns the pool of the system component of the process `rp`.

Process/minus

Purpose

Process subtraction.

Syntax

```
[rpmin] = minus(rpa, rpb)
```

Description

`minus` gives the difference of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpmin] = minus(rpa, rpb)` is called for the syntax '`rpa - rpb`'.

Process/plot

Purpose

Graphical display of random process.

Syntax

```
hand = plot(rp, f, options)
```

Description

`plot(rp)` graphs the power spectrum of the processes making up `rp` over the set frequencies for which the power spectrum has support.

`plot(rp, f)` graphs the power spectrum of the processes making up `rp` at the frequencies given in the vector `f`.

`hand = plot(...)` returns the plot handle.

`options` can include:

- `'o'` or `'overlay'`: in addition to plotting the power spectrum, overlays a plot of the individual base processes.
- `'nooverlay'`: plots only the overall power spectrum. This is the default.
- `'p'` or `'print'`: uses options suitable for printer output. This option will override some user-set plotting defaults.
- `'noprint'`: uses default plotting options. This is the default.

Process/plus

Purpose

Process addition.

Syntax

```
[rpsum] = plus(rpa, rpb)
```

Description

`plus` gives the sum of the two random processes. Any base processes shared by `rpa` and `rpb` have their systems combined.

`[rpsum] = plus(rpa, rpb)` is called for the syntax '`rpa + rpb`'.

Process/pwr

Purpose

Power in random process.

Syntax

```
[P] = pwr(rp)
```

Description

`pwr` returns the power in process `rp` as an optimizable quadratic quantity of type `optQuad`.

Process/times

Purpose

Process convolution.

Syntax

```
[rpconv] = times(a, b)
```

Description

Result is convolution of a process with an affine sequence (system). One input is a random process, and the other must be a sequence.

`[rpconv] = times(a, b)` is called for the syntax '`a .* b`'.

Process/uminus

Purpose

Process negation.

Syntax

```
[rpmin] = uminus(rp)
```

Description

Negates the system associated with process `rp`.

`[rpmin] = uminus(rp)` is called for the syntax '`-rp`'.

6.1.13 Regions

Region

Purpose

Constructor for `Region` objects.

Syntax

```
[reg] = Region()
[reg] = Region(index)
[reg] = Region(type, param)
[reg] = Region('composite', param, op1, op2, oper)
```

Description

`[reg] = Region()` creates an empty `Region` object.

`[reg] = Region(ind)` returns the previously-allocated `Region` object of index `ind`.

`[reg] = Region(type, param)` creates a `Region` object of the given type, with parameters given in the structure `param`.

`[reg] = Region('composite', param, op1, op2, oper)` creates a composite `Region` in `param.dim`-dimensional space, formed as a result of the operation `oper` of the two `Regions op1` and `op2`.

`type` can be one of

- `'halfspace'` is a halfspace, containing all points $\mathbf{x}$ such that $\mathbf{a}'\mathbf{x} >= b$. param must contain the fields `dim`, `a` and `b`, where `a` is a vector of length `dim`, and `b` is a scalar.
- 'sphere' is a hypersphere, containing all points $\mathbf{x}$ such that $|\mathbf{x}| <= r$. `param` must contain the fields `dim`, `center` and `radius`, where `center` is a vector of length `dim` which indicates the center point of the hypersphere, and `radius` is a positive, real scalar.
- `'convpoly'` is a bounded convex polytope. Its constructor requires `param` to contain the fields `dim` and `points`, where `points` is a list of points describing the polytope. The convex hull of the points is then determined, and fields `A` and `b` are added to `param`, which describe the intersecting halfspaces $\mathbf{a}'\mathbf{x} >= b$ which determine the polytope.

Region/bb

Purpose

Bounding box for `Region`.

Syntax

```
box = bb(reg, lat)
```

Description

Returns a bounding box for the `Region reg` in terms of the `Lattice lat`; that is, in terms of intervals along the lattice basis vectors which define a parallepiped that circumscribes the `Region`.

If the `Region` is unbounded, the function returns `inf`.

If `lat` is an offset `Lattice`, the intervals given are for a parallelepiped that circumscribes the `Region` shifted by the inverse of the lattice's offset.

Region/get_dim, get_type

Purpose

`Region` dimension and type extraction.

Syntax

```
dim = get_dim(reg)
type = get_type(reg)
```

Description

`dim = get_dim(reg)` returns the dimension of the `Region reg`.

`type = get_type(reg)` returns the type of the `Region reg`.

Region/intersect, mtimes

Purpose

Intersection of `Regions`.

Syntax

```
reg = intersect(a, b)
reg = mtimes(a, b)
```

Description

Returns the composite `Region` which is the intersection of `Regions` `a` and `b`.

`reg = intersect(a, b)` is called for the syntax `a * b`.

Region/isin

Purpose

Determine if points are in a `Region`.

Syntax

```
result = isin(reg, points)
[result, I] = isin(reg, points
```

Description

Returns the points from the list of points which are in `reg`. `points` is a matrix, in which each row contains the coordinates for an individual point. `result` is a matrix of the same structure.

`[result, I] = isin(reg, points)` also returns an index vector `I`, such that `result = points(I,:)` .

Region/msop

Purpose

Minimal sum-of-products representation.

Syntax

```
fcn = msop(reg)
[fcn, prims] = msop(reg)
```

Description

Returns a simplified sum-of-products representation of a composite `Region`. `fcn` is a cell array, of which each cell represents a product term. Each cell contains a vector of numbers which correspond to primitive `Region`s in the product term.

`[fcn, prims] = msop(reg)` also returns a list of the indices of primitive `Region`s in the composite `Region`.

Region/not

Purpose

Complement of region.

Syntax

```
reg = not(a)
```

Description

Returns a `Region` object which is the complement of `a`.
`reg = not(a)` is called for the syntax `~a`.

Region/or

Purpose

Region shift.

Syntax

```
rreg = or(reg, off)
```

Description

Shifts `reg` by the offset vector `off`.

`rreg = or(reg, off)` is called for the syntax `reg | off`.

Region/points

Purpose

Lattice points in region.

Syntax

```
[locs] = points(reg, lat)
```

Description

Returns the points in `reg` on the `Lattice lat`. `locs` is a matrix with each row containing the the cartesian coordinates of a point.

Region/probett

Purpose

Evaluate Boolean function.

Syntax

```
bit = probett(reg, prims, row)
```

Description

Evaluates the Boolean function over primitive regions specified by the `Region` `reg` and evaluated on the primitive variables given `prims`, a vector of indices. `row` is an integer which when converted to BCD specifies the binary value for each of the primitive values. The variables are assumed to be ordered from least significant to most significant. The result `bit` is a logical value which can be 0 or 1.

Examples

`probett(15, [1 2 3], 0)` evaluates

$$f_{15}(P_1 = 0, P_2 = 0, P_3 = 0).$$

`probett(15, [1 2 3], 6)` evaluates

$$f_{15}(P_1 = 1, P_2 = 1, P_3 = 0).$$

where f_{15} is the Boolean function specified by the `Region` of index 15.

Region/rot

Purpose

Rotate region.

Syntax

```
rreg = rot(reg, ang, dims)
```

Description

Rotates the region by `ang` radians in dimensions `dims`. `dims` is a length-2 vector indicating the subspace in which the rotation is performed. For example, `rot(reg, pi/4, [1 3])` will rotate `reg` 45 degrees in the xz-subspace.

Region/setdiff

Purpose

Set difference of regions.

Syntax

```
reg = setdiff(a, b)
```

Description

Returns `Region` whose members are in `a` but not `b`. `[reg] = setdiff(a, b)` has the same effect as `[reg] = intersect(a, b)`.

Region/union, plus

Purpose

Union of `Region`s.

Syntax

```
reg = union(a, b)
reg = plus(a, b)
```

Description

Returns the composite `Region` which is the union of `Region`s `a` and `b`.

`reg = union(a, b)` is called for the syntax `a + b`.

6.1.14 Second-order Cone Constraints

SOCConstr

Purpose

Second-order cone constraint constructor.

Syntax

```
[soc] = SOCConstr(relop, obj1, obj2)
```

Description

`SOCConstr` creates `soc` as a second-order cone optimization constraint. `relop` is the relational operator of the constraint, and can be `'<'` or `'=='` for inequality and equality constraints, respectively. `obj1` is the left side of the constraint, and can be an `optQuad` or an `optQuadVector`. `obj2` is the right side of the constraint, and can be an `optQuad`, an `optQuadVector`, or an `optVector`.

SOCConstr/get_A, get_b, get_c, get_d, get_Mrank

Purpose

`A`, `b`, `c`, `d` and `Mrank` extraction functions.

Syntax

```
A = get_A(soc)
A = get_A(soc, type)
A = get_A(soc, n)
b = get_b(soc)
b = get_b(soc, type)
b = get_b(soc, n)
c = get_c(soc)
c = get_c(soc, type)
c = get_c(soc, n)
d = get_d(soc)
d = get_d(soc,n)
Mrank = get_Mrank(soc)
```

Description

`get_A`, `get_b` and `get_c` return `A`, `b` and `c`, respectively. If they are called with only `soc` or `(soc, 'cell')`, they are returned as cell arrays. If called `(soc, 'matrix')` they are returned concatenated as a matrix. If called `(soc, n)`, only the `n`th element is returned.

`get_d` returns `d`. When called `get_d(soc, n)` only the `n`th element of `d` is returned.

`get_Mrank` returns `Mrank`.

6.1.15 Solutions