RAPT: Rob's Audio Processing Toolkit - User Manual

Robin Schmidt

October 19, 2016

Chapter 1

Introduction

1.1 What is it

Rob's Audio Processing Toolkit (abbreviated as RAPT) is an open source C++ library for audio signal processing with a focus on musical signals. The lower levels of the library provide the numerical functionality that is relevant in this context (such as basic linear algebra, polynomial manipulation, Fourier transforms, interpolation, etc.). On top of that, a toolkit of audio processing specific algorithms is built. These algorithms are typically the ones, that client code will want to call, although it is also possible to reach down into the lower levels of the basic mathematical number crunching routines.

1.2 Copyright and Licensing

The copyright of the code is owned by the author, Robin Schmidt, and is released under the terms of the GNU General Public License (GPL). Basically, that means, if you want to use it somewhere, you have to open-source your project, too. If you want to use the code in a closed source product, you may contact me for negotiating the terms [...t.b.c]

1.3 Design Principles

1.3.1 Type Independence

Most of the functions and classes are written in a generic way in order to make them applicable to whatever kind of datatype you are dealing with. Technically, that means they are implemented as C++ templates. For example, if you have a class for representing multichannel audio data and that class provides appropriate operators for addition and multiplication (i.e. add/-multiply the individual channel data element-wise), you can apply a generic convolution routine that convolves two sequences of your multichannel datatype (assuming they have a compatible number of channels - but these compatibility requirements will depend on the implementation of the `+` and `*` operators of your class). You may also want to use convolutions on complex numbers and/or multiprecision datatypes. The genericity of the implementation allows for that.

1.3.2 Two Interface Levels

For certain tasks, the library supports 2 levels of interfacing - a more convenient high-level interface and a more performance efficient low-level interface. The idea is that you may write prototype code using the high-level interface which will result in readable code. Later, you may want translate that into production code using the low-level interface, which avoids certain overheads such as dynamic memory allocation. The low-level interface also gives you more options for tuning the runtime performance. For example, you may choose between different algorithms that perform the same task but have different numerical stability and efficiency properties. The low level interface will often operate on plain C-style arrays of your respective datatype. Some of those functions will assume that input and output arrays do not overlap while others may not care and therefore may be used 'in-place'. If in-place operation is possible, it will be said so in the documentation, otherwise assume it isn't.

1.3.3 Classes for Code Organization

The code of the library is organized in classes such as `Polynomial, Matrix, FilterDesign, Transforms`, etc. which group together related functionality. This organizing principle is also used in cases where the client code is not supposed to instantiate any objects of a class. For example, the class `IntegerFunctions` is merely a collection of functions that perform common operations on integer numbers, such as `IntegerFunctions::gcd(T x, T y)` for finding the greatest common divisor of the two numbers `x, y` (of a templated integer type `T`). This makes it easier to find the required functionality than in an unorganized collection of functions that all occupy the same namespace. The generated doxygen documentation will also show related functions under a common 'headline'.

1.3.4 Goals

The primary goal is to provide a collection of audio processing algorithms that is practically useful in actual audio software products. Thus, the code should be reasonably efficient. A secondary goal for the library is to have educational value. That means, the code should be written in a readable way and familiar conventions known from signal processing literature and other DSP software libraries (such as MatLab's signal processing toolbox) should be followed, where reasonable. Sometimes, these two goals are in conflict - the most efficient implementation is sometimes not the most readable. In these cases, efficiency is most often preferred and complications in the code will often be annotated by comments. However, the library does not try to squeeze the last bit of performance out of the computer. Specifically, all code is plain C/C++ and there are no assembler optimizations or use of intrinsic SIMD types and functions. This wouldn't make sense in a type independent template based library anyway. However, if client code wraps SIMD datatypes and operations into classes, the template implementations of RAPT may be instantiated for these class types as well, potentially leading to higher performance - but such considerations will be left to client code.

1.4 Basic Examples

1.4.1 Butterworth Filter

Suppose, you want to compute the coefficients of a 5th order Butterworth filter with a cutoff frequency of 1 kHz operating at a samplerate of 44.1 kHz. With RAPT, you may do this like this:

```
double fs = 44100;     // samplerate
double fc =  1000;     // cutoff frequency
double b[6], a[6];     // filter coefficients
FilterDesign<double>::butterCoeffs(b, a, 5, 2*fc/fs);
```

The last line of this code snippet will fill the arrays `b, a` with the feedforward and feedback coefficients of a Butterworth filter in direct form, i.e. a filter that implements the difference equation:

$$y_n = \sum_{k=0}^{5} b_k x_{n-k} - \sum_{k=1}^{5} a_k y_{n-k} \tag{1.1}$$

Note that the frequency passed to the filter design routine is a normalized frequency, i.e. a frequency value between 0 and 1 where 1 corresponds to the Nyquist frequency (half of the samplerate). This convention is used for frequencies throughout the library and is in line with usage in MatLab/Octave/SciPy/etc. [verify]. Next, suppose you have an array `x` of length `N` and you want to filter that array using the Butterworth coefficients. The output shall be stored in another array `y` of length `M`. This can be done by:

```
Filter<double>::applyDF2(x, N, y, M, b, 5, a, 5);
```

this computes `M` output samples by applying the filter using a so called direct form II implementation structure. The length of the output `M` may be different from the length of the input `N`.

A Realtime Implementation

The code above applies the filter to the whole array `x` at once. This would be adequate only in situations, where you have the whole signal available, like in an audio editor application. But now imagine, for example, your product is a realtime audio plugin. In this case, you will typically have to write a subclass that is derived from a baseclass that comes from an audio plugin API. Such plugin APIs typically have some kind of audio callback function, often called `process`, that takes a short buffer of input samples as (pointer-type) parameter and is supposed to produce a corresponding buffer of output samples. The plugin programmer has to override this function in their derived subclass and do there whatever computations are needed to achieve the desired effect. For example, your `process` callback for a Butterworth filter plugin based on RAPT could look like:

```
void MyFilterPlugin::process(int numSamples, int numChannels,
                             float **in, float **out)
{
  assert(numChannels == 2);                   // handles only stereo signals
  filterL.process(in[0], out[0], numSamples); // process left channel
  filterR.process(in[1], out[1], numSamples); // process right channel
}
```

where `filterL, filterR` are objects of the class `Filter`. Many of RAPT's audio processing classes provide such a process function. As you can see from the difference equation 1.1, each filter output sample is computed as a weighted sum of the current input sample, past input samples and past output samples. An object of class `Filter` is responsible for keeping track of these in a realtime situation. In some situations, for example, when the user stops or restarts the audio playback in a digital audio workstation (DAW), it is appropriate to reset these state variables to their initial state (which is zero, in this case). The plugin interface will typically have another callback for you to override in these situations, for example called `suspend` in the VST API. There, you would call:

```
void MyFilterPlugin::suspend()
{
  filterL.reset();
  filterR.reset();
}
```

In addition to the process callback, realtime audio plugin APIs also provide a means to let the user set up the plugin's parameters, typically in the form a setParameter function that is supposed to be called whenever the user changes a parameter. In our case, the user might want to change the filter's cutoff frequency from a GUI. Whenever the cutoff frequency is changed, the filter coefficients have to be recomputed. Our implementation could look like:

```
void MyFilterPlugin::setParameter(int index, float value)
{
  switch( index )
  {
  case CUTOFF:
    {
      // User wants to change cutoff frequency.

      // Convert normalized parameter value to desired cutoff frequency:
      cutoff = RAPT::Conversions<float>::linToExp(value, 0, 1, 20, 20000);

      // Compute new filter coefficients:
      FilterDesign<float>::butterCoeffs(b, a, 5, 2*cutoff/samplerate);

      // Pass the new coefficients to the filter objects for left and right
      // channel:
      filterL.setCoeffs(b, a, 5);
      filterR.setCoeffs(b, a, 5);
    }
    break;
    // ... handle other parameter updates here
  }
}
```

This example is purely for instructional purposes and therefore kept simple. It's not the most efficient way to do it. Specifically, the call to FilterDesign<double>::butterCoeffs is computationally expensive. A rather elaborate mathematical machinery is invoked to compute these coefficients and they are always computed from scratch in such a call. It is possible to save some of the computational steps, by storing some intermediate results that are expensive to compute and don't change when the cutoff frequency is changed (namely the poles of the analog prototype filter). Furthermore, setParameter and process are typically called from different threads, so you may want to take some care about synchronization, for example by wrapping the two calls to setCoeffs and the two calls to process into a mutex lock. So, take this example code with a grain of salt - it shall just demonstrate the basic principles and is not production ready code. Note that you would also need to recompute the coefficients, when the samplerate changes. You will be informed about this in yet another callback.

Realtime and Latency The above scheme is typical for realtime audio plugins, but the designation as 'realtime' has to be taken with a grain of salt. The plugin's audio callback is called with a given buffer size B, i.e. a certain number of samples to be processed. From the perspective of the host application, that means, when it wants to produce an output sample at a given time instant n, it must know all samples from n up to $n + B - 1$, so it can feed them into plugin's process callback already at instant n. Sometimes the host can't know these future samples in advance, for example because the host itself receives them from a realtime input. In such a case, it must first accumulate B samples, then pass the buffer of B samples to the plugin, receive the output and pass it on further down the signal chain. This wait-and-accumulate strategy introduces a delay of $B - 1$ samples between input and output. Typically the buffer sizes are a couple of tens or hundreds of samples which translates to a couple of milliseconds or tens of milliseconds of delay. Delays below 10 ms are below the threshold of human perception, but it has to be kept in mind in certain situations, especially when similar signals are to be mixed. If the buffer size is just a single sample, there's no such delay and the mode of operation can be properly called 'realtime'. In some situations, such behavior is required, that's why the `Filter` class also provides a special `process` function which produces just a single output sample from a single input sample. An idiom, typically seen throughout the library in realtime capable classes, is that a buffer based `process` function actually calls a (possibly inlined) single sample based `process` function inside a loop that runs over all the samples in the buffer. If you are building a modular system with modules from RAPT and you want to introduce a feedback loop in a modular patch, you will probably want to use the the sample based processing functions. In this context, your feedback loop will then only have a single sample of delay - you can use the output of a module produced at instant n as input to the same module from instant $n+1$ onwards.

Zero Delay Feedback In cases where even that single sample of delay in the feedback path is too much, you can use special `processNoUpdate` functions which can produce the output sample without updating the internal state of the object. Using these functions, you might be able to set up a convergent iteration [t.b.c. ...].

Chapter 2

Mathematics

Digital signal processing is in general a very mathematical subject and audio- and musical signal processing is no exception to that rule. So it shouldn't come as a surprise, that many of the RAPT library functions rely on underlying algorithms that implement a considerable mathematical machinery. This chapter discusses some of the mathematical concepts that are relevant in the context of audio processing and shows how they are implemented in RAPT. The purpose here is mostly to explain, *what* the respective classes and functions do, not *how* they work internally. Most of the algorithms discussed in this chapter fall under the broad umbrella of numerical analysis. Detailed explanations and derivations can be found in the vast literature, specifically [Reference: Numerical Recipies] is a first class reference for such things.

2.1 Complex Numbers

For representing complex numbers, RAPT uses `std::complex` from the C++ standard library.

2.2 Polynomials

A polynomial of a variable x is a weighted sum of integer powers of x, starting at $x^0 = 1$ and going up to x^N. Polynomials play a central role in the theory of analog and digital filters and they occur in the context of interpolation, curve fitting and function approximation - that's reason enough for RAPT to devote them a dedicated class - which is unsurprisingly called `Polynomial`. Mathematically, a polynomial $p(x)$ of order N is defined as:

$$p(x) = \sum_{n=0}^{N} a_n x^n \tag{2.1}$$

where the $a_n, n = 0, \ldots, N$ are called the coefficients. An Nth order polynomial has $N + 1$ coefficients, ranging from index 0 to index N. This is also the convention in which polynomial coefficient arrays are represented in RAPT - as length `N+1` arrays `a[0]`...`a[N]`. Consider the following code for multiplying the 2nd order polynomial $p(x) = 2 - 3x + x^2$ by the 1st order polynomial $q(x) = -3 + 4x$ and evaluating the resulting polynomial $r(x)$ at $x = 2$:

```
Polynomial<int> p(2, 2, -3, 1), q(1, -3, 4);  // allocation, initialization
Polynomial<int> r = p * q;                    // multiplication
int y = r(2);                                 // evaluation
```

This code uses the high-level interface of the RAPT library. The first line creates the two polynomials (with integer coefficients). The first parameter of the constructor of the `Polynomial` class is the order of the polynomial followed by an appropriate number of coefficients. The 2nd line multiplies the two polynomials which results in another polynomial which is stored in the variable `r`. The third line produces a numeric output by applying the product polynomial r to the value $x = 2$. The code looks reasonably straightforward and readable by C/C++ standards. However, if you have to do polynomial multiplication in a context where performance is critical, for example inside a realtime audio callback, you probably don't want to have any dynamic memory allocations. But that's what the constructors of the `Polynomial` class do. For performance and predictability reasons, you may want to operate on pre-allocated arrays, even if that means to uglify your code. In this case, your code could look something like this:

```
// pre-allocation and initialization:
int p[3] = { 2, -3, 1};                // 2nd order, 3 coeffs,  2 - 3x + x^2
int q[2] = {-3,  4};                   // 1st order, 2 coeffs, -3 + 4x
int r[4];                              // 3rd order, 4 coeffs

// multiplication and evaluation:
Polynomial<int>::mul(&p, 2, &q, 1, &r);  // multiply p, q, store result in r
int y = Polynomial<int>::eval(&r, 3, 2); // evaluate polynomial r at 2
```

First, we allocate the three arrays `p, q, r` of lengths `3, 2, 4` respectively. The order of the result polynomial `r` is given by the sum of the orders of the factors $2 + 1 = 3$, so we need space for $4 = 3 + 1$ coefficients for the `r` polynomial. The `p, q` arrays are also initialized in this process. Then we call static member functions `mul` and `eval` of the `Polynomial` class that operate on pre-allocated arrays of polynomial coefficients to perform the multiplication and evaluation. Arguably, the 2nd version is much less readable but these static member functions can be used whenever it is undesirable to have any dynamic memory allocations. Internally, the operators `*` and `()` of the `Polynomial` class call the low-level routines on internally managed coefficient array member variables. This idiom of static member functions that are internally called from the non-static functions and operators on the instance variables of an object will be used throughout the library. It allows the low- and high-level interface to share the same code.

2.3 Root Finding

Root finding is the problem of finding one or more solutions of the equation:

$$f(x) = 0 \tag{2.2}$$

where $f(x)$ is, in general, an arbitrary function. Any value of x that turns the equation into a true statement, when substituted for x, is called a solution or 'root' of the equation. In this problem setting, it is assumed that we have a means to evaluate the function at arbitrary values of x. In RAPT, the root finding algorithms must get such a means by passing them either function pointer to the respective function or a function object of class `Functor`. Functors are objects that provide a means to evaluate a function at some point x. What makes them more flexible than plain functions is the fact that they can have internal instance variables which may affect the function evaluation. They can be seen as a general way to define families of functions with an arbitrary set of parameters. A polynomial is an example for such a functor where the parameters are the coefficients.

2.3.1 Bisection

Suppose that, for two particular values $x = a$ and $x = b$, we know that $f(a) < 0$ and $f(b) > 0$. By assuming that the function is continuous between a and b, that means that a root, i.e. a zero crossing, must be present somewhere between a and b. The same is true, if $f(a) > 0$ and $f(b) < 0$. We say, the root is bracketed by the values a and b. We may now evaluate the function at the midpoint between a and b. If the value is equal to 0, we have found our root, if it's less than 0 or greater than 0, we have found a new bracketing interval either to the left or to the right of the midpoint. The new bracketing interval is only half as long as the old one. Iterating that procedure, the brackets move closer and closer to the root. When their difference reaches a threshold, typically related to the numeric precision of the floating point datatype, we stop the iteration. It is then converged to the root (up to an error given by our threshold). Since the interval halves at each iteration, the number of correct binary digits in our approximate solution (given by the current midpoint) increases by one. Because the number of correct digits is a linear function of the iteration number, we say that the algorithm has a linear order of convergence. In RAPT, root finding by bisection is implemented in `RootFinder::bisection`. As a subtle sidenote, it may make sense to choose the new evaluation point not exactly at the midpoint of the bracketing interval, but at [... tbc. Reference NR].

2.3.2 Newton Iteration

Newton iteration is a method to iteratively improve an initial guess x_0 for the solution by iteratively applying the update rule:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.3}$$

This formula results from approximating the function $f(x)$ as a straight line that goes through the point $(x_0, y_0 = f(x_0))$. This straight line is constructed to match the slope of the function $f(x_0)$ given by the derivative $f'(x_0)$ in the point (x_0, y_0). The resulting line equation is then solved for the zero crossing of the line, which becomes our new root estimate x_{n+1}. Thus, to actually implement it, we also need a means to evaluate the derivative of $f(x)$ for any x in addition to being able to evaluate the function $f(x)$ itself. The convergence of this iteration is typically quadratic, which means that in each iteration, the number of correct digits doubles [verify this] but in certain pathological cases it may be slower and in some cases, the iteration may even diverge. If a reasonable initial guess for the solution is available that is close to the true solution and the function is well behaved near the solution and we know how evaluate the derivative, Newton iteration is one of the methods of choice to converge to a true solution. In RAPT, Newton iteration is available in the function `RootFinder::newton` which takes two function pointers or functors - one for the function itself and one for the derivative and also takes an initial guess and returns the refined value.

2.3.3 Secant Method

The secant method works similar to the bisection method, just that we choose the new evaluation point not as midpoint of our current bracketing interval $[a, b]$, but instead, we fit a straight line to the points $(a, f(a)), (b, f(b))$ and solve for the root of this straight line. The order of convergence is given by the golden ratio: $\alpha = (1 + \sqrt{5})/2 \approx 1.618$. So, it converges faster than the bisection method but not quite as fast as the Newton iteration. It can be seen as an approximation to Newton iteration, where the derivative is replaced by a finite difference approximation. On the plus side, it does not need to evaluate the derivative, which might be expensive or sometimes even not feasible. The method is available in `RootFinder::secant`. Note that it may also diverge, since the new evaluation point may fall outside the initial bracketing interval.

2.3.4 False Position

Also known as 'regula falsi', this method is similar to the secant method, but it applies...

2.3.5 Roots of Polynomials

If the function $f(x)$ is a polynomial, there are specialized algorithms for finding its roots. One such algorithm is the Laguerre algorithm which is implemented in `Polynomial::roots(T *a, int N, complex<T> *r)`. It takes an array of `N+1` coefficients as parameter and returns the roots in the length `N` array `r`. The fundamental theorem of algebra tells us that each polynomial of order N has exactly N roots such that it can be expressed as:

$$p(x) = k \prod_{n=1}^{N} (x - r_n) \tag{2.4}$$

that is, a scaled product of the differences between x and the respective root r_n. Remember that a product is zero, if and only if (at least) one of its factors is zero, so if x is equal to one of the roots r_n, the corresponding factor $x - r_n$ in the product is zero, making the whole product zero - which verifies that r_n is a root, indeed. These roots, however, may not be unique - some of them may have a multiplicity, i.e. occur more than once in the above product. Some of the roots may also be complex. If the coefficients of the polynomial are real, complex roots will always come in pairs of complex conjugate numbers. In audio applications, we will mostly deal will polynomials that have real coefficients. For generality, however, there's also a version of the polynomial root finder that takes an array of complex coefficients. Internally, the same algorithm is used anyway, no matter if the coefficients are real or complex. For example, the roots of the 5th order polynomial $p(x) = 1 + 2x - 3x^2 + 3x^4 - x^5$ can be found via:

```
vector<complex<double>> r(5);
r = Polynomial<double>(5, 1., 2. -3., 0., 3., -1.).roots();
```

or, alternatively using the low-level routine that operates on pre-allocated arrays:

```
double a[6] = { 1., 2. -3., 0., 3., -1. };
complex<double> r[5];
Polynomial<double>::roots(a, 5, &r);
```

A function for getting from the roots back to the coefficients is also available via the function `Polynomial::fromRoots`. In addition to the N roots, it also needs the value of the overall scaling constant k in Eq. 2.4. This is simply the Nth coefficient, but in case you forget that, the `Polynomial` class has also the function `productFormScaler(int N, T a)` which trivially just returns `a[N]`. Since the `Polynomial` class is a subclass of class `Functor`, the general root finders from above can be applied for polynomial root finding as well. They can, however, find only one root at a time and you need an initial guess or bracketing interval.

2.4 Rational Functions

A rational function is a quotient or 'ratio' of two polynomials:

$$r(x) = \frac{\sum_{m=0}^{M} b_m x^m}{\sum_{n=0}^{N} a_n x^n} \tag{2.5}$$

where M is the order of the numerator and N is the order of the denominator. All realizable analog and digital filter transfer functions are functions of this very type - which explains the special significance of rational functions, and hence polynomials, in the field of signal processing. The roots of the numerator are called the zeros of the transfer function and the roots of the denominator are called the poles. Since the denominator is zero at these poles, the value of the rational function at these points goes to infinity, unless there's a corresponding zero in the numerator at exactly the same point to cancel the pole. RAPT provides the class `RationalFunction` to deal with rational functions.

2.4.1 Partial Fraction Decomposition

Any rational function can be expanded into the form of a sum of a polynomial and several first order rational functions, in which the numerator is a constant (called the residue which is possibly complex) and the denominator is a linear function of x. That is:

$$r(x) = \frac{\sum_{m=0}^{M} b_m x^m}{\sum_{n=0}^{N} a_n x^n} = \sum ... \tag{2.6}$$

If the coefficients of the original numerator and denominator polynomial are both real, complex values for the numerators and denominators in the expanded form come in conjugate pairs, such that 1st order terms can be combined pairwise into 2nd order terms with real coefficients. In signal processing applications, such decompositions amount to decomposing a direct form filter into a parallel connection of the respective 1st or 2nd order filters and an FIR part, where the latter results from the polynomial part. Because the individual terms are easier to understand and analyze, partial fraction decompositions play a role in certain filter analysis applications. For example, they are required for finding closed form expressions of a filter's impulse response. This impulse response is given by the inverse z-transform of its transfer function, but inverse z-transforms are only available for low-order terms - which is precisely what the decomposition delivers. Partial fraction decomposition is available in RAPT via `RationalFunction::partialFractions`.

2.5 Vectors

2.6 Matrices

2.7 Transforms

Transforms are - together with filters - one of the mainstays in the field of signal processing. Consider an N dimensional vector $\mathbf{x}$ which may represent a (segment of a) signal. A general transform would take that vector as input and produce a new vector $\mathbf{y}$ as output. If the transform is designed well, the transformed vector $\mathbf{y}$ may reveal relevant features of the signal that are hard to see in the original representation. An important concern is invertibility - we want to be able to reconstruct the original vector $\mathbf{x}$ from the transformed vector $\mathbf{y}$ by some kind of associated inverse transform. Given such a pair of a transform and its inverse, we could transform an input vector, modify the transformed representation and apply the inverse transform to get a modified output vector/signal. A good transform in an audio processing setting would be one, that allows for convenient modifications that closely relate to modifications of perceptual, auditory attributes of the signal. We will consider mainly linear transforms. These are the class of operations on our input vector $\mathbf{x}$ that can be expressed as a matrix-vector multiplication of an $N \times N$ matrix $\mathbf{A}$ with our input vector $\mathbf{x}$:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{2.7}$$

If the matrix $\mathbf{A}$ is non-singular, there will be an inverse matrix $\mathbf{A}^{-1}$ and hence, there will be an inverse transform, given by:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} \tag{2.8}$$

Alternatively to thinking in terms of an N dimensional vector $\mathbf{x}$, we may also consider the inputs and outputs as sequences of length N, which, in this context, can be seen as merely a different name for the same thing - a finite, ordered set of N numbers.

2.7.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) of a length N sequence $x_0, \dots, x_{N-1}$ is another length N sequence $X_0, \dots, X_{N-1}$ in which each element X_k can be computed by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, \qquad \text{where } W_N = e^{-j\frac{2\pi}{N}} \tag{2.9}$$

expressed as matrix-vector product, this looks like:

$$\dots \tag{2.10}$$

Computing a product of an $N \times N$ matrix and an N vector in general requires $\mathcal{O}(N^2)$ operations, because we have to form N^2 individual products between numbers. In the DFT matrix, however, not all elements are distinct. This redundancy of the matrix elements allows for an algorithmic shortcut, known as the fast Fourier transform (FFT) which requires only $\mathcal{O}(N \cdot \log N)$ operations. Fast Fourier transforms are particularly efficient when N is a power of two - these transforms are known as radix-2 FFTs. There are also $\mathcal{O}(N \cdot \log N)$ algorithms for arbitrary N, for example, the Bluestein FFT algorithm which uses a radix-2 algorithm on a zero-padded sequence internally. In RAPT, both radix-2 and Bluestein FFTs are available via the functions `Transforms::fftRadix2` and `Transforms::fftBluestein`. There's also the function `Transforms::fft` which tests, if N is a power of two and then dispatches between the radix-2 and the Bluestein algorithm accordingly.

Inverse Transform

2.7.2 Short Time Fourier Transforms

2.7.3 Wavelet Transforms

2.8 Statistics

Statistical signal measures such as the mean value, the autocorrelation sequence or the cross-correlation sequence (between two signals) play an important role in many signal analysis algorithms and in adaptive filters. ...

2.9 Interpolation

Assume that we have a certain number N of pairs of abscissa/ordinate values $(x_n, y_n), n = 0, \dots, N-1$ represented as length `N` arrays `x` and `y`. We imagine these data points as a table that represents a continuous function and we assume that the x-values are in strictly ascending order. The goal of interpolation is to create a continuous function from the discrete data. That means, interpolation would connect the points by assigning a y-value to *any* x-value between `x[0]` and `x[N-1]`. If we try to find a y-value for an x-value outside this range, it's called extrapolation. In RAPT, a continuous interpolating (and extrapolating) function is represented as an `Interpolator` object which itself is subclassed from `Functor`. This implies that the root-finding algorithms can be applied to interpolating functions in the same way as they are applied to regular functions. In the context of audio processing, interpolation may be required for obtaining signal values between the actual sample values or for creating smooth curves of control-data for which values are available only at certain time instants but not at each sample instant.

2.9.1 Linear Interpolation

A very simple way of interpolating between datapoints is to connect the datapoints with straight lines.

2.9.2 Lagrange Interpolation

Lagrange interpolation takes a given number N of datapoints and constructs an Nth order polynomial that passes exactly through these points. Such an interpolating polynomial tends to oscillate wildly between the actual datapoints, especially if the order of the polynomial is high. In most application, such oscillations are undesirable and that's why Lagrange interpolation is mostly used only with low orders. In certain audio applications, however, these oscillations might actually be exactly what is needed - as the order of the Lagrange interpolator goes up, it approaches the optimal sinc-interpolator [Reference: 'Elephant interpolators']

2.9.3 Spline Interpolation

2.9.4 Sinc Interpolation

2.10 Numerical Differentiation