ELECTRONOTES 207

Newsletter of the Musical Engineering Group

1016 Hanshaw Road, Ithaca, New York 14850

Volume 22, Number 207 December 2011

GROUP ANNOUNCEMENTS

Contents of EN#207

TOOLS FOR INVESTIGATING KAUTZ FUNCTIONS

-by Bernie Hutchins

Likely few reading here are familiar with Kautz Functions. A search through many signal processing books yields nothing, and indeed the internet yields little material, although enough material to get going. I was fortunate enough to have them pointed out by Tom Parks at Cornell; many years in a class he taught, and sometime assisting him in that class. So I have played with them many times, but never all that seriously sat down and investigated them – particularly with regard to how we might use them for music instrument sound synthesis. This is overdue.

Our first hint about what Kautz functions are comes from the original paper : Kautz, William H., "Transient Synthesis in the Time Domain", IRE Transactions Circuit Theory, Sept. 1954. Clearly this is an analog-based discussion, and it does seem to be a title that grabs our attention. Percussive sounds, for example, are transients after all.

Here we will be discussing Kautz Functions in their discrete version. It will turn out that we will find very familiar ideas: things like modeling with poles, and digital filter structures. Matlab simulations will be easy to write.

BACKGROUND/REVIEW - RESONATOR

Suppose we have in mind a pole. For something more interesting, let's assume it is a conjugate pair of poles (more or less usual). Like a pair of poles inside the unit circle at a radius **r** (less than 1) and angles ±θ:

$$z_{p1} = r\cos(\theta) + j\,r\sin(\theta)j \qquad (1a)$$

and

$$z_{p2} = r\cos(\theta) - j\,r\sin(\theta) \qquad (1b)$$

These poles correspond to a second-order polynomial:

$$D(z) = (z-z_{p1})(z-z_{p2}) = z^2 - 2r\cos(\theta)\,z + r^2 \qquad (1c)$$

For example for a radius r=0.97 and at angles θ=±15° with respect to the real axis, the poles are:

$$z_{p1} = 0.97\cos(15) + j\,0.97\sin(15) = 0.9369 + 0.2511j \qquad (2a)$$

and

$$z_{p2} = 0.97\cos(15) - j\,0.97\sin(15) = 0.9369 - 0.2511j \qquad (2b)$$

These poles correspond to a second-order polynomial:

$$D(z) = (z-z_{p1})(z-z_{p2}) = z^2 - 1.8739z + 0.9409 \qquad (2c)$$

Nothing new here.

Likewise it is probably familiar [1] that the two-pole situation corresponds to a second-order network of the type shown in Fig. 1a. Here an impulse enters the network and an exponentially decaying sinwave emerges [the gain of sin(θ) in the output path is just to set the output to unity at the start]. Recall that D(z) from equation (1c) is the denominator of the network here. We can easily simulate this network (see Program 1 – Resonator). Using the numbers in equation (2c) as a specific example, we obtain an output as seen in Fig. 1b.

We use this as a familiar starting point. In later Matlab programs, it is convenient to use the Matlab ***filter*** function to generate test signals.

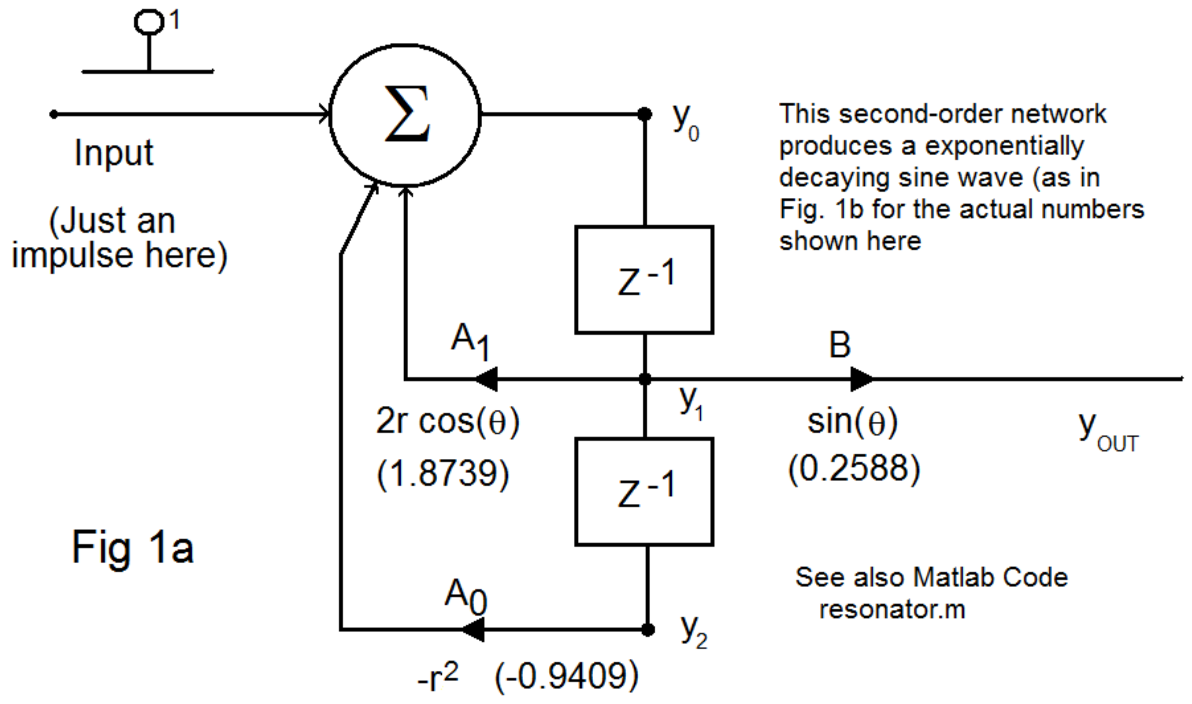


Fig 1a

This second-order network produces a exponentially decaying sine wave (as in Fig. 1b for the actual numbers shown here

See also Matlab Code resonator.m

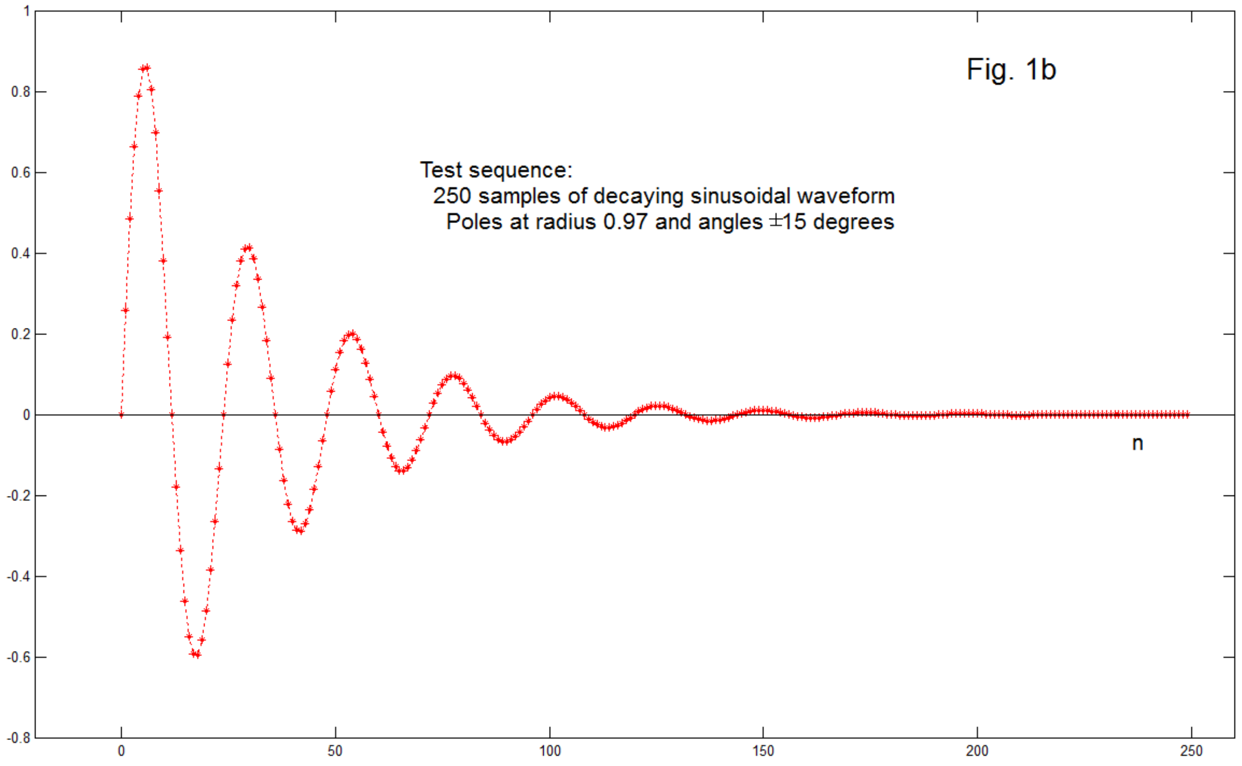


Fig. 1b

PROGRAM 1 - RESONATOR

```
function yout=resonator(rad,ang,N,fs)
% generates length N impulse response of a 2nd-order
% digital filter with poles at a given angle and radius.
% it also plays the response as a sound at sampling
% frequency fs.  Pole angles here in degrees.
% example, yout=resonator(0.97,15,250,10000)
% example, yout=resonator(0.999,30,10000,10000)


radang=ang*2*pi/360;
zp1= rad*cos(radang) + j*rad*sin(radang)
zp2= rad*cos(radang) - j*rad*sin(radang)
D=poly([zp1 zp2])
y1=0;
y0=0;
imp=1;
for n=1:N
   y2=y1;
   y1=y0;
   y0= 2*rad*cos(radang)*y1 - (rad^2)*y2 + imp;
   yout(n)=sin(radang)*y1;
   imp=0;
end
length(yout)
figure(1)
plot([0:N-1],yout,'*r')
hold on
plot([-100 1000],[0 0],'k')
hold off
axis([-20 260 -0.8 1])
sound(yout,fs)
```

Notes on this program:
This program calls Matlab's ***sound*** as a last instruction. In order to actually hear the output as sound, try the second example in the header. The 0.999 radius makes the sound last long enough to be clearly heard – but you won't get much detail in the plot.

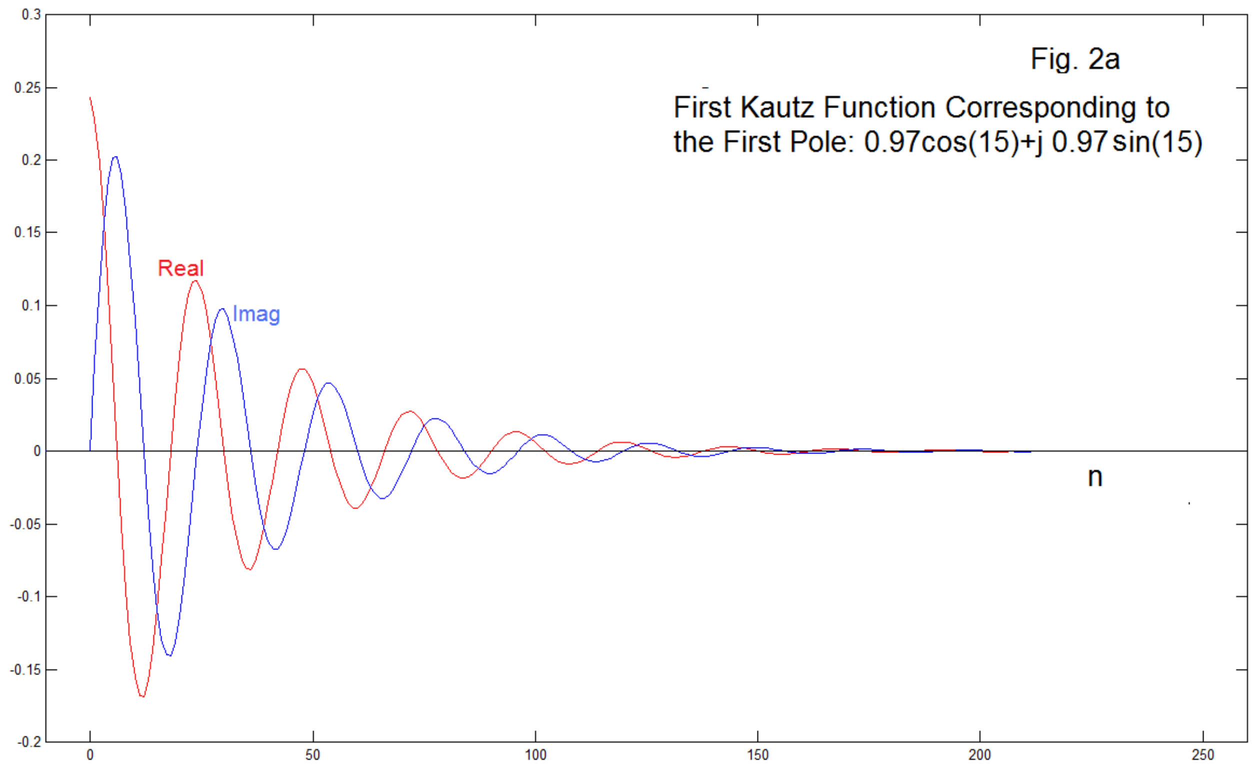
Here so far this is all background and review, and the establishment of a test sequence – the decaying sinusoidal sequence. What we have done is given a very simple description of the sinusoidal. We could describe it as samples of a sequence:

$$s(n) = Ae^{-\sigma nT} \cos(\omega nT + \varphi) \qquad (3)$$

where A is an amplitude, σ is a decay time constant, ω is a frequency, φ is a phase, and T is a sampling time (five numbers total). Equivalently, it can be described in terms of the parameters of the digital network of Fig. 1a, by giving A_0, A_1, B, $y_2(0)$, and $y_1(0)$. Again, five numbers. All this so far is old stuff.

AN EXAMPLE OF KAUTZ FUNCTIONS

The waveform of Fig. 1b can also be represented in terms of Kautz functions. At this point, we will just go ahead and show the results, discussing the theory, various programs, and applications later.



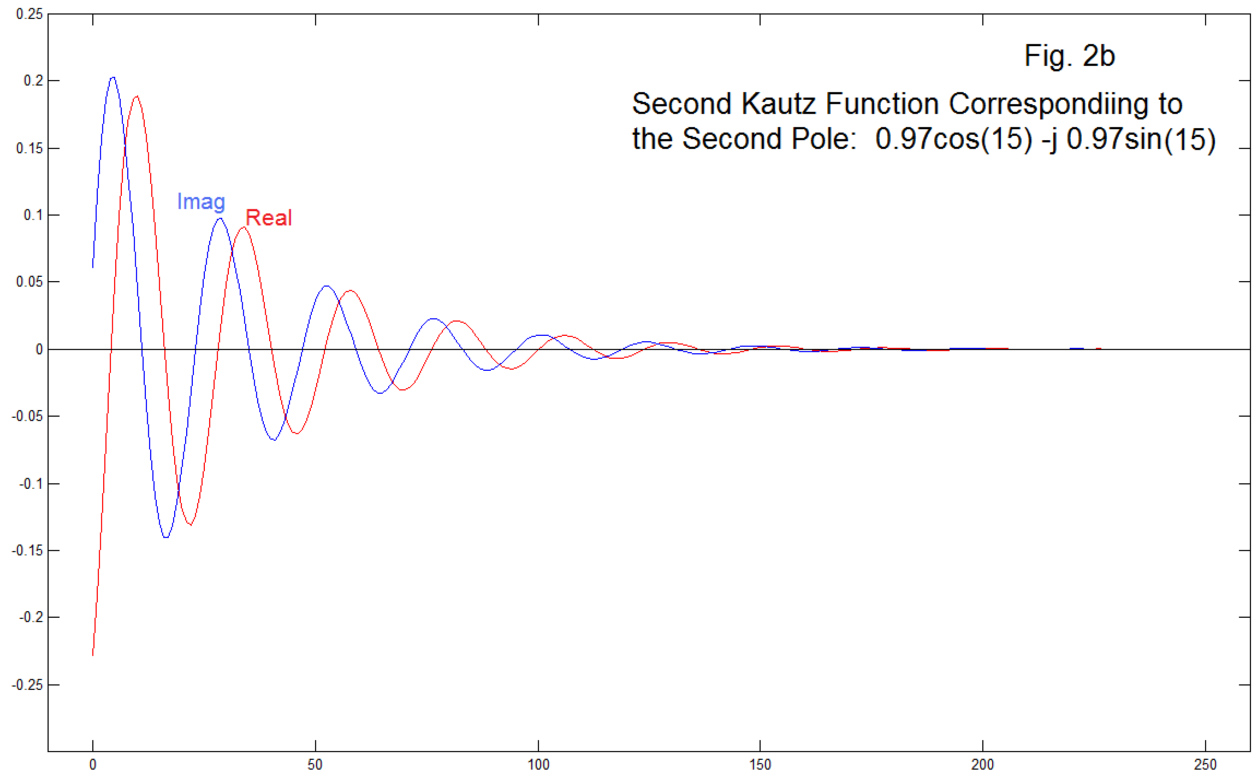


Fig. 2b
Second Kautz Function Correspondiing to the Second Pole: 0.97cos(15) -j 0.97sin(15)
Imag
Real
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
0
50
100
150
200
250

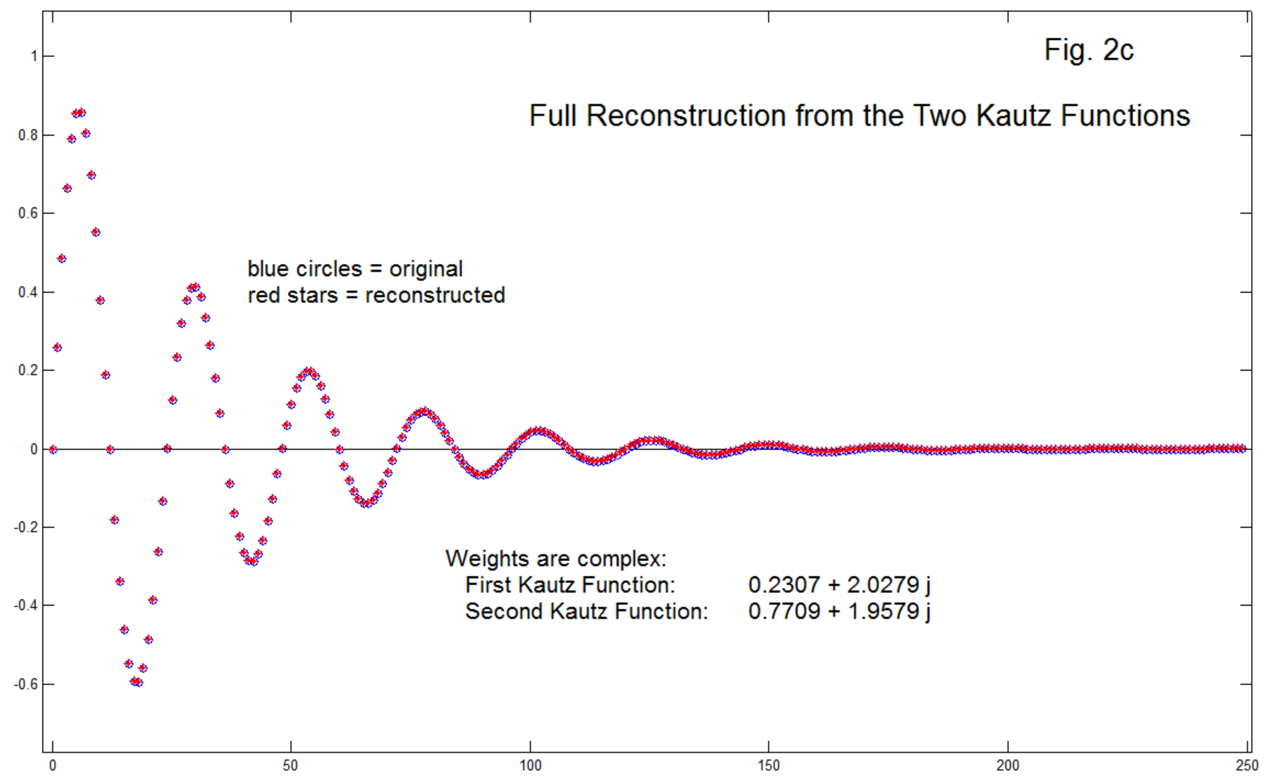


Fig. 2c
Full Reconstruction from the Two Kautz Functions
blue circles = original
red stars = reconstructed
Weights are complex:
First Kautz Function: 0.2307 + 2.0279 j
Second Kautz Function: 0.7709 + 1.9579 j
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
50
100
150
200
250

The example shown in the three parts (a, b, and c) of Fig. 2 show our resonator test signal constructed from Kautz functions. In this case, there are two Kautz functions (Fig. 2a and Fig. 2b) that form the “basis”, one for each of the two poles we have been using. Here we have a special case where we knew the poles that were involved in generating the test signal. Note that the basic functions are of a decaying sinusoidal form, and that they are complex (red part real, blue part imaginary). It turns out (see notes on Fig. 2c) that the corresponding weights are also complex, and the reconstruction (weighted sum of Fig. 2a and Fig. 2b) is real. As mentioned, here the reconstruction is perfect, because we cheated by knowing the correct poles. Had we guessed them (incorrectly in general) the reconstruction would be imperfect.

So this is a familiar case of an inner product space (not unlike a Fourier series or a DFT) where we have basis functions, and weights (coefficients of an expansion). We have three things to do. First, how do we generate the basis functions (Kautz functions), and what are their properties. Second, how do we compute the coefficients, perhaps by inner product, perhaps by digital filter (as in Goertzel’s algorithm for the DFT) or likely, both. Thirdly, are they good for anything.

THE KAUTZ FILTER BANK

It is convenient to show the structure of the Kautz filter because it looks very much like some digital all-pass and single-pole filters with which we are quite familiar. This filter bank is also easy to simulate for generating the corresponding Matlab programs. The impulse response of this filter will be the Kautz functions. Further, by time-reversing a signal to be analyzed, passing it through the filter, and stopping at the right time, we can obtain the coefficients.

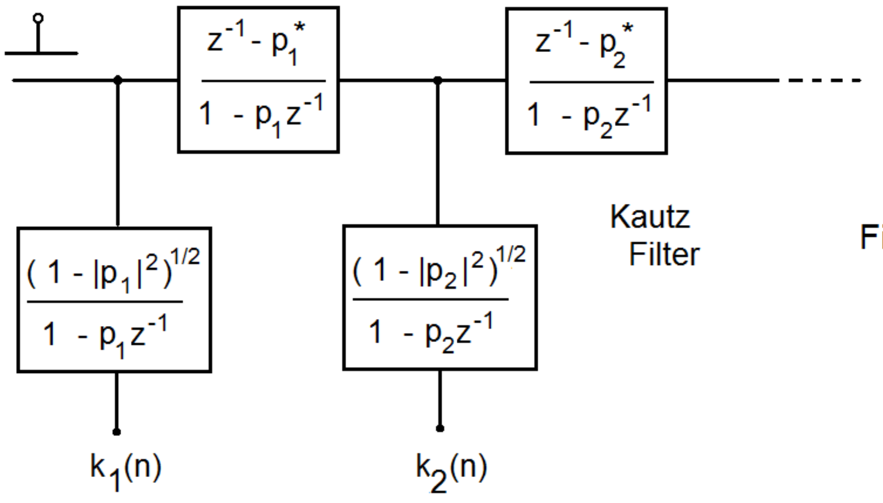

$\frac{z^{-1} - p_1^*}{1 - p_1 z^{-1}}$
$\frac{z^{-1} - p_2^*}{1 - p_2 z^{-1}}$
$\frac{(1 - |p_1|^2)^{1/2}}{1 - p_1 z^{-1}}$
$\frac{(1 - |p_2|^2)^{1/2}}{1 - p_2 z^{-1}}$
Kautz Filter
$k_1(n)$
$k_2(n)$

Fig. 3a

Fig. 3a shows the structure of the Kautz filter for length 2 (extended in the obvious way). It is similar to a "transversal filter" like an FIR filter, except instead of delay we have all-pass-like units, and instead of fixed taps, we have single pole units. Here the p's are the poles as we have been using them. Let's take a closer look at the two units in the boxes. Note that the signals here are going to be, in general, complex.

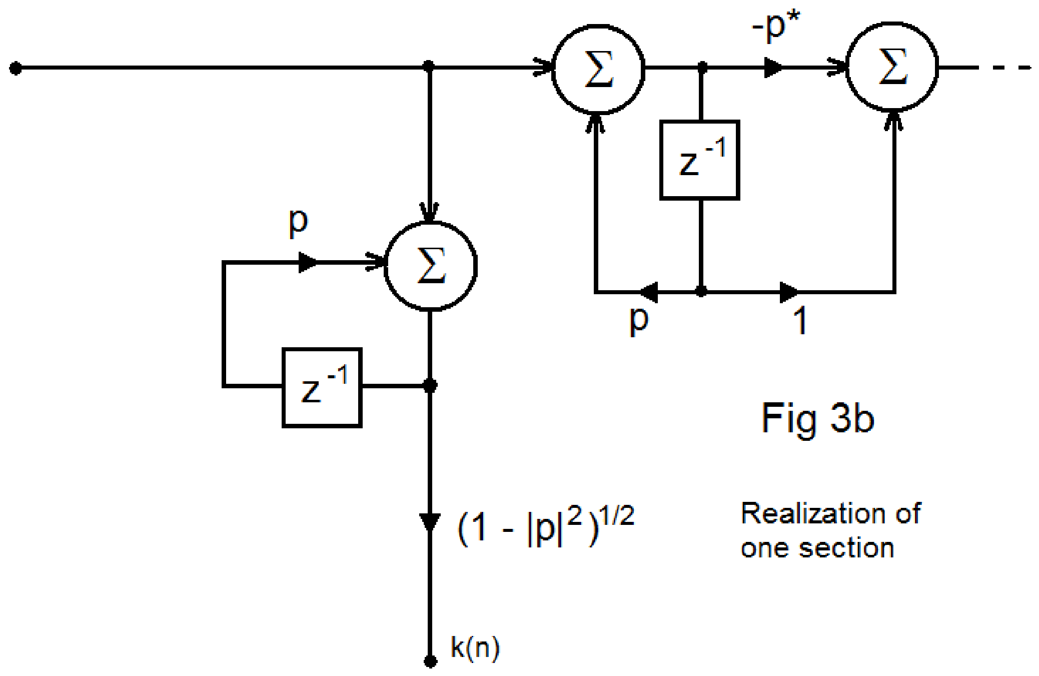


Fig 3b

Realization of one section

Fig. 3b corresponds exactly to half of Fig. 3a in an obvious way. Note that the denominator of the transfer function in the units going downward may appear complicated but they are just real constants and are realized as output multipliers $(1 - |p|^2|)^{1/2}$. We further note that the feedback section appears in both blocks and is thus redundant, and we will shortly remove this. We have called the horizontal blocks "all-pass-like" because they look like a first-order digital all-pass – except that a pole (inside the unit circle of course) is matched to a reciprocal zero outside and in the conjugate position.

Fig. 3c shows the structure once the redundant feedback is removed. (We note on the figure that the summers in the middle here, and likewise between additional sections that may be present in the case of more than two poles, are also redundant but this probably saves nothing as far as programming is concerned).

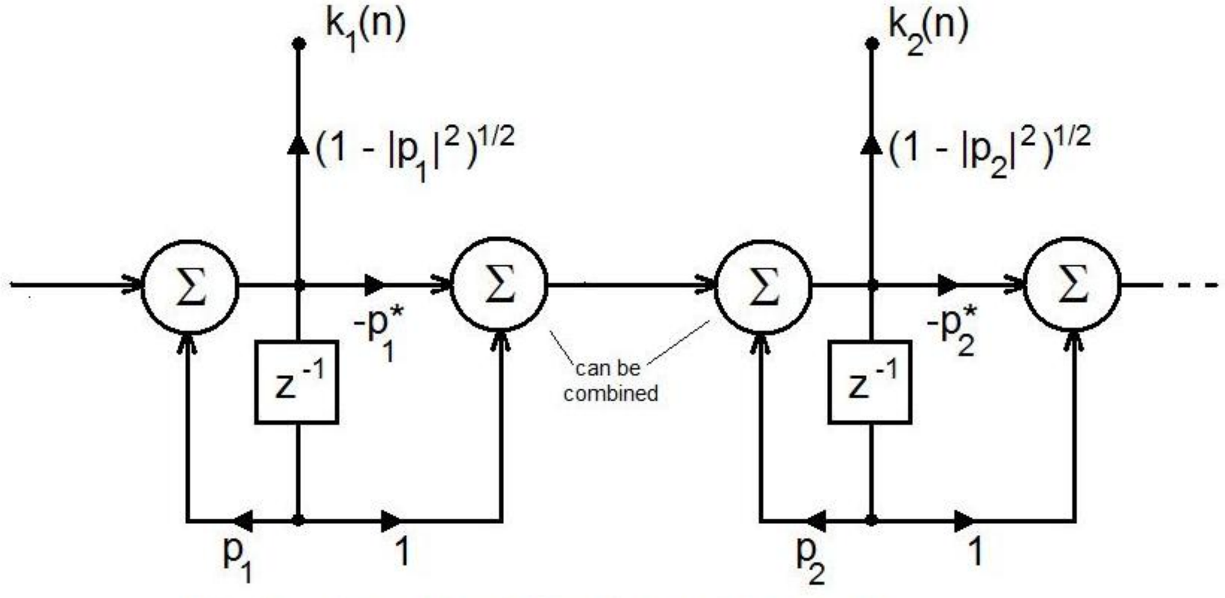


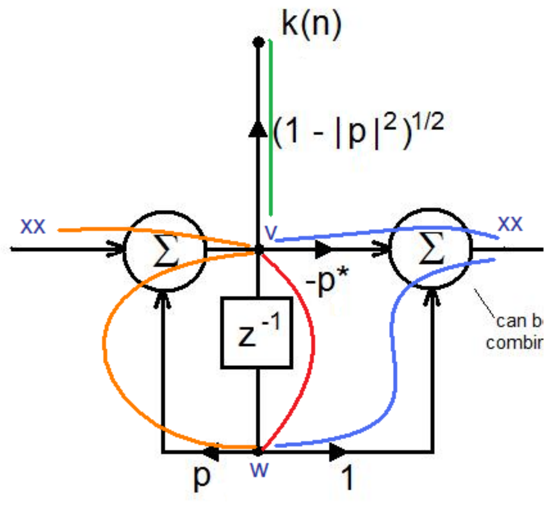
Fig. 3c Feedback Portions Combined

To be clear, the structure of Fig. 3c is for two poles, and can be extended in an obvious way for additional poles. We will be using this filter in two ways. By inputting an impulse, we generate the Kautz functions as the k(n). If we have a sequence to be analized, this can be input (time reversed) to give the Kautz coefficients. As a final note here, you can see (or redraw) Fig. 3c with all the poles p=0 and note that the structure is basically an FIR filter.

WRITING PROGRAM CODE

We need to write Matlab code to do our experiments. It is clear that the Kautz filter is central to all we are going to do, so it makes sense for to write this as a "function: which we can call ***kfilter***. This code is at first confusing to approach. We need to write it for a variable length signal and for a variable number of poles. And there are multiple outputs (the k's). It is convenient to look at a single section as in Fig. 4. What we do, for each time iteration (n), is to update each section (m) (Fig. 4 typical), left to right. This calculates the values for all the k's for the current time value n.

We update a section, as we probably should for all digital filter simulations, by first executing the delays. That is, we move the data from the input of ALL delays to the outputs. This overwrites the outputs, and we must then calculate new values for the delay inputs, using nothing that we have not updated already. This means we must



Step 1: Execute delay: contents of v move to w.
Note: v must now be updated.

Step 2: Update v as w feed back through p summed with input xx.

Step 3: Calculate k(n) by scaling updated v.

Step 4: Calculate xx for next stage. This is the sum of w and of the updated v multiplied by -p*.

Fig. 4

work left to right! . In Fig. 4, this means we move v to w (red line) as our first step, and concentrate on updating v as our second step. The node v is the output of the summer on the left. This sums the node w as multiplied by p with the latest input xx, and this becomes v (orange lines). Note: for the very first section, this xx comes from the input sequence x. For the rest of the sections, xx is fed to the following stage as a result of updating the current section. The third step (green) is simply to scale v to form the output k(n,m). This we are storing as a matrix for convenience. The last step (blue) is to calculate and feedforward the value of xx. This is the sum of w, and v as multiplied by –p*, giving to xx of the next stage. That's how the filter works.

PROGRAM 2 – kfilter.m

```
function k=kfilter(p,N,x)

np=length(p)
v=zeros(1,np);
w=zeros(1,np);

for n=1:N
   xx(1)=x(n);
   for m=1:np
      w(m) = v(m);
      v(m) = w(m)*p(m)+xx(m);
      k(n,m) = sqrt (1-abs(p(m)')^2)*v(m);
      xx(m+1)=w(m) - p(m)'*v(m);
   end
end
```

The function ***kfilter*** above is the core used for our investigations. Even if you are not good in Matlab you can likely understand the code just fine, referring to Fig. 4. It does the calculations specific to the Kautz functions. This is good. Most Matlab programs (most programs?), in terms of lines of code, are sparse in lines that actually calculate. Most of the code lines in a final program relate to manipulating and displaying results, and making things look good on a screen. Programmers who can never seem to finish a program are usually going back for one more user-friendly improvement.

Here in ***kfilter*** we have not done anything with the calculated results. You would need to plot the results by entering instructions individually. Or, you could embed ***kfilter*** in an application-specific program. Here we have decided to provide a program we will call ***ktest*** to "exercise", plot, and demonstrate the Kautz function properties. The program ***ktest*** produced the graphs of Fig. 2.

THE TEST PROGRAM

Below we list the code for ***ktest*** and its functions in terms of six "tasks" it performs. Recall that there were two applications of ***kfilter*** (see Fig. 5).

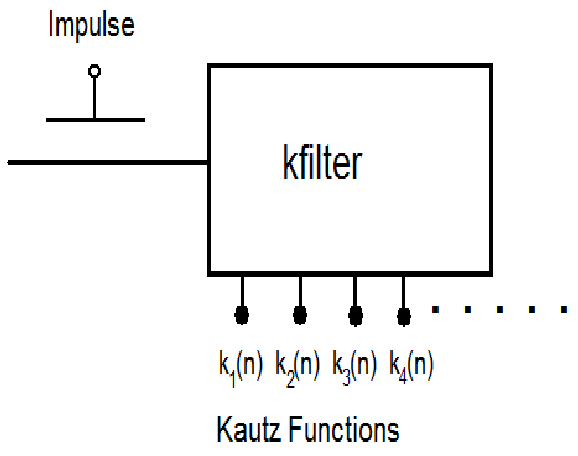


Fig. 5a

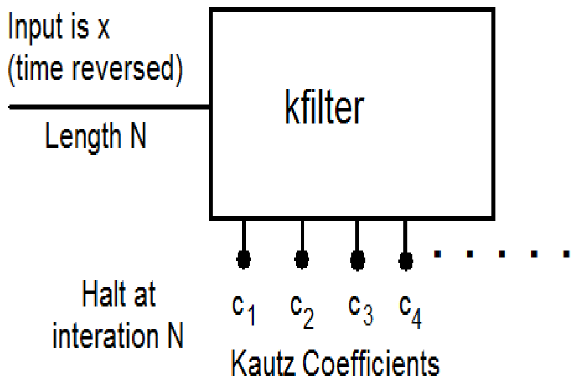


Fig. 5b

The inputs to the function ***ktest*** are the poles chosen, p, and the signal to be analyzed, x. Here (Task 1) we first set the length of the Kautz functions to be generated to be the length of x, and call this N. The function ***kfilter*** then is run with the input chosen as an impulse: a one at n=1 and zeros thereafter to a total length N. That is, we are using Fig. 5a. This gives us the Kautz functions as the columns of a matrix k. The matrix has a number of columns equal to the number of poles (length of p) and the rows at the successive time values of the Kautz functions.

PROGRAM CODE FOR KTEST

```
function [k,c,w]=ktest(p,x)
%  [c,w]=ktest(p,x)
%  p = poles
%  x = signal to be represented
%
%  k = Kautz functions
%  w = Kautz weights (by inner product)
%  c = Kautz filtering - last row = weights
%



% Task 1 - Generate Kautz functions from poles
N=length(x)
k=kfilter(p,N,[1,zeros(1,N-1)])
% check orthonormal
orthnorm=k'*k



% Task 2 - Plot the Kautz functions
for m=1:length(p)
   figure(m)
   plot([0:N-1],real(k(:,m)),'r')
   hold on
   plot([0:N-1],imag(k(:,m)),'b')
   hold off
end



% task 3 - Kautz weights by inner product
for m=1:length(p)
  w(m)=x*k(:,m);
end
```

```
% Task 4 - Sum the series
xr=zeros(1,N);
for m=1:length(p)
  xr=xr+w(m)*k(:,m)';
end
xr
xr=real(xr)
e=abs(x-xr)
```

```
% Task 5 - Plot reconstruction
figure(m+1)
plot([0:N-1],x,'ob')
hold on
plot([0:N-1],xr,'*r')
hold off
mx=max([max(x) max(xr)]);
mn=min([min(x) min(xr)]);
axis([-2 N+1 1.3*mn 1.3*mx])
figure(m+1)
```

```
% Task 6 - Kautz coeffs by filtering
% should be same as weights w
c=kfilter(p,N,(fliplr(x))');
```

The final part of Task 1 is to do a matrix inner product to check and see if the Kautz functions are orthogonal. We have not mentioned this, but they are indeed ortho<u>normal</u>, so the result of the orthonormal test should be a square identity matrix of size equal to the number of poles. It may not be that exactly. If so it is probably the case that the Kautz functions were not long enough to get small enough. That is, they correspond to decaying exponentials, but if the exponential tails are not small enough, the identity matrix may be approximate. The signal x can always be calculated to a longer length (if possible), padded with zeros, or simply, a value N can be made an input parameter as with ***kfilter***. We now have the Kautz functions in a matrix k, and we are ready to plot them, and this will be Task 2.

In Task 2, the variable m is the index of the pole (and the corresponding Kautz function). Note that the desired function is "extracted" from the matrix by the use of the colon to indicate a column: e.g., k(:,m) is the mth column. The functions are plotted individually as successive figures, the real (red) and imaginary parts (blue). If the function is purely real (as for a real pole) the horizontal axis is overplotted in blue. Here not attempt was made to automatically set an axis. Code could be added to do this, although it is also possible to do these by hand for optimum appearance (often we may want to add text as well). Fig. 2a and Fig. 2b were generated by ***ktest*** and tidied up.

In Task 3, we calculate the Kautz coefficients by the direct computation of the inner products of x with the individual basis functions, in the usual manner. We notice here that the program has already set the length of the Kautz functions to the length of x, as is required for this inner product. We call these coefficients w (for weights) here. The exact same numbers will be calculated in Task 7.

We are now at Task 4. This can be thought of as a "reconstruction" process, or better, just as a series sum. **If** the basis functions are long enough, and **if** the poles are chosen correctly, we can expect to get x back. Fig. 2c is such a case. But, and this is an essential point, we don't usually know the poles. Here x is an input to ***ktest***. It was generated or obtained ahead of time. For Fig. 2c, we selected the poles and ran the resonator program. This gave us x. Then when we ran ***ktest***, we chose the same poles we used in the resonator. That's cheating in the general case. On the other hand, we needed to be sure that that at least worked perfectly. The case where we have to guess poles is of major interest to us. The last three lines of Task 4 allow us to be sure we get a real result (assuming x was real) and to display the error.

Task 5 is a plot (as discrete samples)of the reconstruction (or attempted reconstruction). Note that this picks up the figure numbering at m+1. The original sequence is shown as blue circles and the reconstruction samples as red stars. This portion does attempt to set an axis, as shown. Fig. 2c is an example.

Task 6 is redundant with Task 3 as far as results are concerned. Here we use the Fig. 5b use of ***kfilter***. The filter in this case computes the inner product. The input sequence x is time reversed, and following N iterations, the outputs are the coefficients. This is the same way that Gertzel's algorithm calculates DFT's. Note the fliplr is "flip left/right" and the use of the apostrophe here is conjugate transpose (only transpose since x is assumed real). [In general, we need to be careful about the apostrophe and the asterisk in Matlab. The "star" means multiply in Matlab, but we usually use it for a conjugate (as a raised star or asterisk) in our text. This often comes out printed as a asterisk in Matlab code, but it still means multiply. Now, the apostrophe in Matlab is a conjugate transpose – commonly needed in matrix algebra. It changes a row vector to a column (and vice versa) AND conjugates. For real vectors, it just transposes. For scalers, it just conjugates. In this example, it just transposes to get dimensions correct.

INVESTIGATIONS

Investigation 1: Ordering of Poles/Extra Poles/Unneeded Poles:

This investigation is a review of, and extension of the original case as in Fig. 2. This was the case where we generated x based on two poles chosen, and then used these exact same two poles for the Kautz expansion. Everything worked perfectly. So we will extend this a bit with certain "what if?" notions.

What if we reversed the order of the two complex conjugate poles? Well, it looks pretty much the same, except the imaginary parts of the Kautz functions are reversed in sign, and the imaginary parts of the coefficients are reversed in sign. Very reasonable.

What if we went to four poles by repeating the two poles that work perfectly. Well, we get two more Kautz functions, which are interesting to look at (different from the original two) but their coefficients are zero – they are not needed. Sounds reasonable too. The sections on the left grab all the business first. They don't, for example, share!

So the next move is to add another set of poles to the perfect poles. For example, follow the two original poles with a pair, 0.5 + 0.5j, and 0.5 - 0.5j. These two poles, too, are rejected in that their coefficients are zero. We can understand this as follows. With the two perfect poles, x is reconstructed and is a linear combination to two Kautz functions. In consequence, any additional Kautz functions that follow are orthogonal to the first two, and thus also to any linear combination of these (like x).

But what then happens if we choose the pair 0.5 + 0.5j, and 0.5 - 0.5j in the first two sections and then follow with the two perfect poles in positions three and four? In this case, we get an answer with four non-zero coefficients. A new solution. The first pair of poles, which were not a perfect solution, left part of x to be represented, which was picked up by the perfect poles.

As a final test in this series, we again try the case of using the perfect poles twice, but this time, the poles are not in order: pole, conjugate, pole, conjugate - but rather: pole, pole, conjugate, conjugate. Does the system choose, perhaps, the first pole, skip the second pole, choose the conjugate, and skip the second conjugate? Nope. It uses all four. And the Kautz functions are (of course?) different from the original case.

Incidentally, we probably have not said so explicitly, but all of these cases above are perfect reconstructions.

The lesson here is that the order of poles does matter.

<u>Investigation 2</u>: <u>Exponential Envelopes</u>

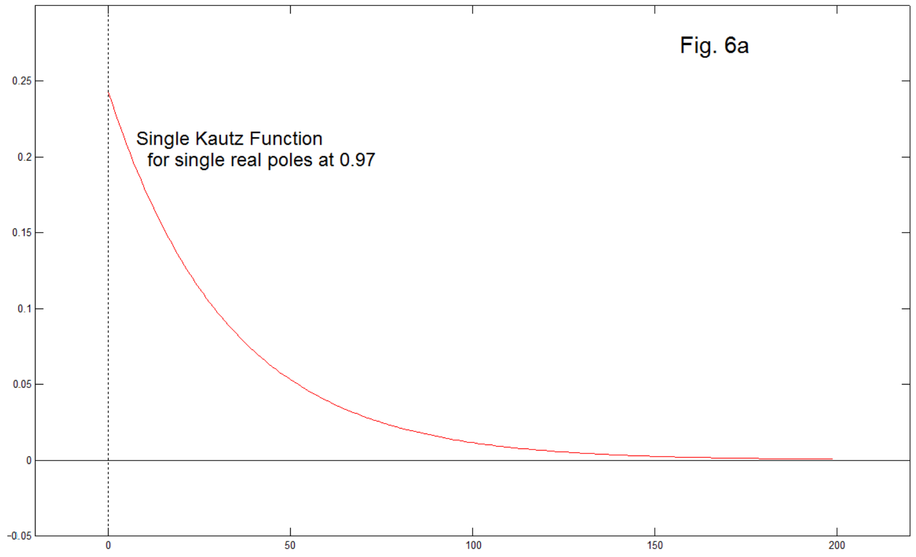
Here we drop back from complex exponentials (decaying sinusoidal waveforms) to real exponentials. Real exponentials are important in music sound synthesis as envelopes. In the simplest possible case, we have one real exponential, a real pole at some radius p. It is convenient to write real exponential decays directly in terms of this radius:

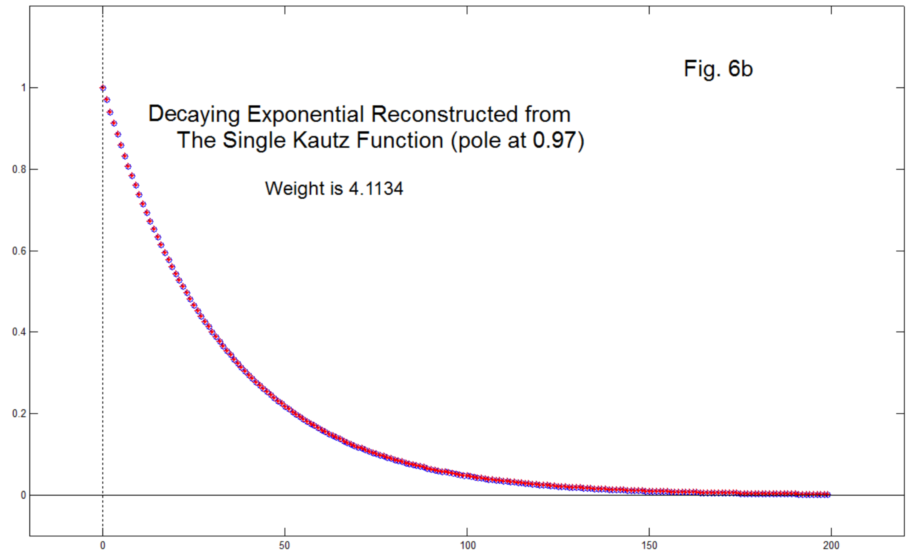
$$x = p^n = e^{-\sigma n} \qquad (4a)$$

where:

$$\sigma = -\ln(p) \qquad (4b)$$

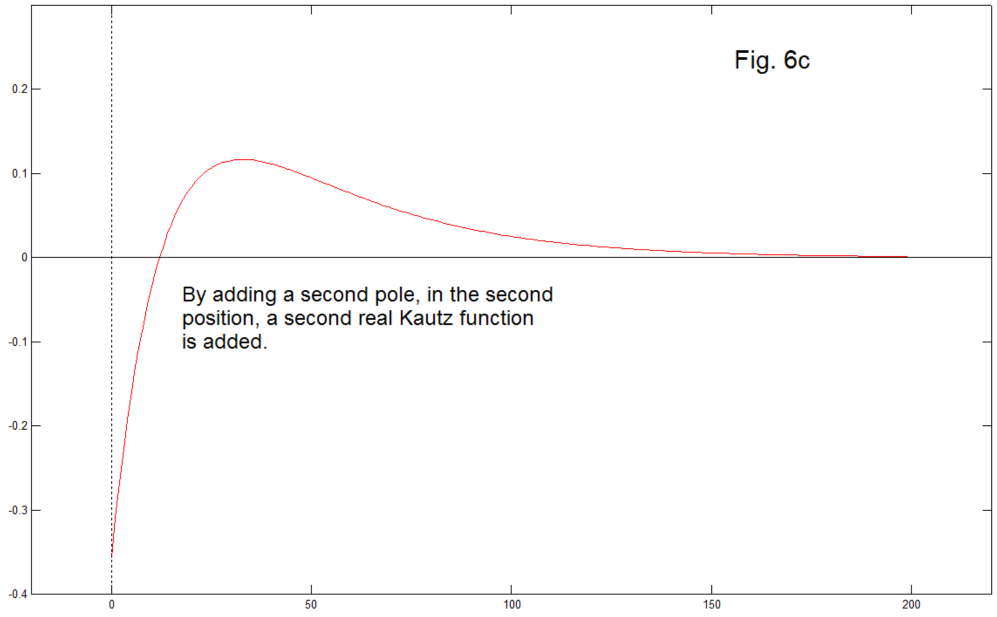
If for example we choose p=0.97, we can calculate x from equation (4a), and then use ***ktest*** with the same pole, as seen in Fig. 6a.

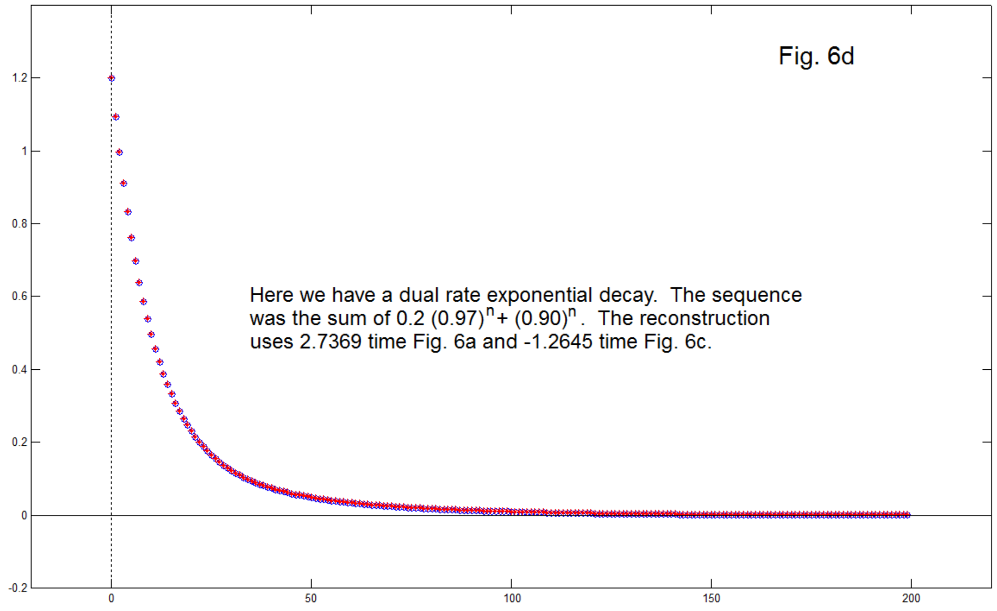




We see that the exponential decay is exactly what we expected and it is reconstructed perfectly. Perhaps if it was not already obvious, what is meant by the "weight" or "coefficient" is that Fig. 6a is multiplied by the weight 4.1134 to get Fig. 6b.

By adding a second pole to ***ktest***, this one at 0.9 (in addition to the one at 0.97) we obtain the second Kautz function shown in Fig. 6c. This does not resemble a typical exponential decay as the first one (Fig. 6a) did. When we run ***ktest*** with x being the single decay sequence, p=0.97, the weight requested for the second function (Fig. 6c) is zero, as we expect, and the reconstructed sum remains the same as Fig. 6b.





We are interested in envelopes for musical sounds that seem to have a dual rate of exponential decay: an initial fast decay followed by a slower rate of decay. For example, the case of coupled piano strings resulting in a "singing" slow decay [2]. We can form a typical two rate decay as:

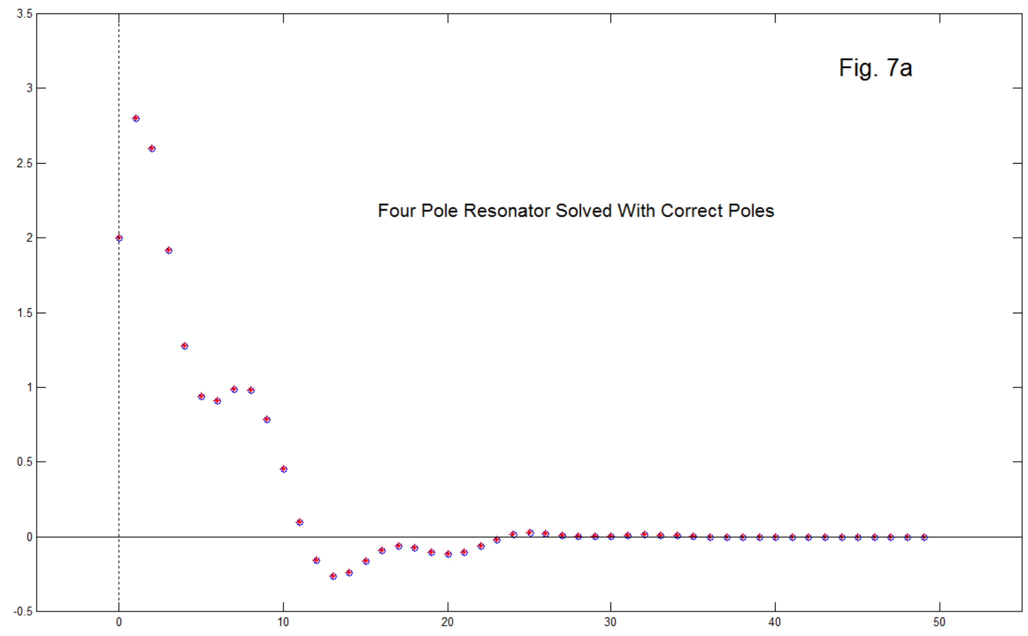
$$x = 0.2\,(0.97)^n + (0.9)^n \qquad (5)$$

Here we have reduced the amplitude of the slower decaying sequence to 0.2 relative to the faster decaying one. The first sample is 1.2, and the original and reconstruction as shown in Fig. 6d show the dual decay rates. Unlike the case of Fig. 6b, we now do require the second pole (the one at 0.9) for Fig. 6d. The weights are given in Fig. 6d.

Later we will look at the way in which a dual rate exponential envelope can be decomposed into two "Kautz envelopes" with the two envelopes separately controlling individually selected components of an additive synthesis.

IMPERFECT RECONSTRUCTION

We have good reasons to examine the Kautz functions while "cheating" by knowing the correct poles. This meant that we were able to attribute changes we observed to the different input conditions to the programs, without considering "errors". Here we will do a construction with one set of poles, and reconstruction with altered poles. We no longer anticipate a perfect reconstruction.



Our test signal will be generated from two pairs of complex poles. We could do this as in the Resonator program, but it is convenient to use the Matlab ***poly*** and ***filter*** functions such as:

```
x=filter(1,poly([.6+.6*j,.6-.6*j]),[1 zeros(1,49)])+ filter(1,poly([.8+.2*j,.8-.2*j]),[1 zeros(1,49)])
```

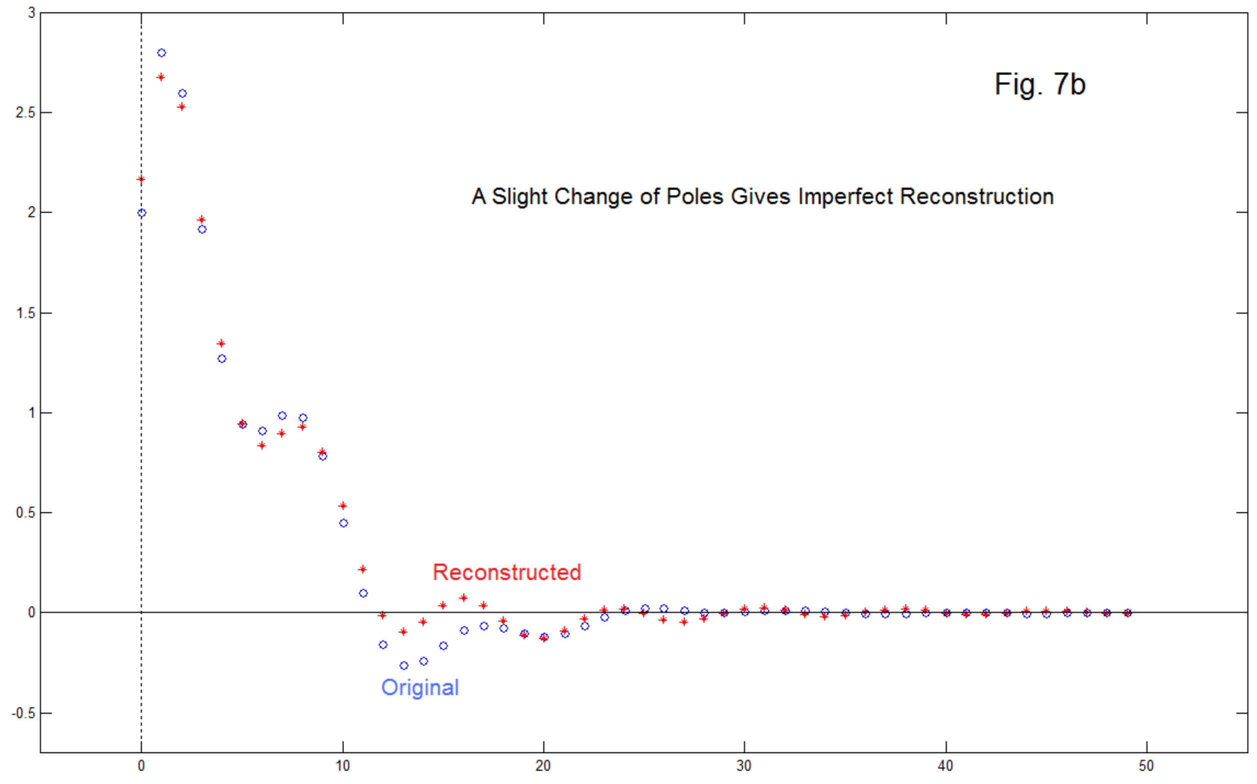
The perfect reconstruction is thus obtained with ***ktest*** as,

```
[k,c,w]=ktest([0.6+.6*j,0.6-0.6*j,0.8+0.2*j,0.8-0.2*j],x)
```

with the result shown in Fig. 7a.

So finally, we will try a reconstruction by changing the second set of pole slightly (changes in red). Fig. 7b shows the case of ***ktest*** with:

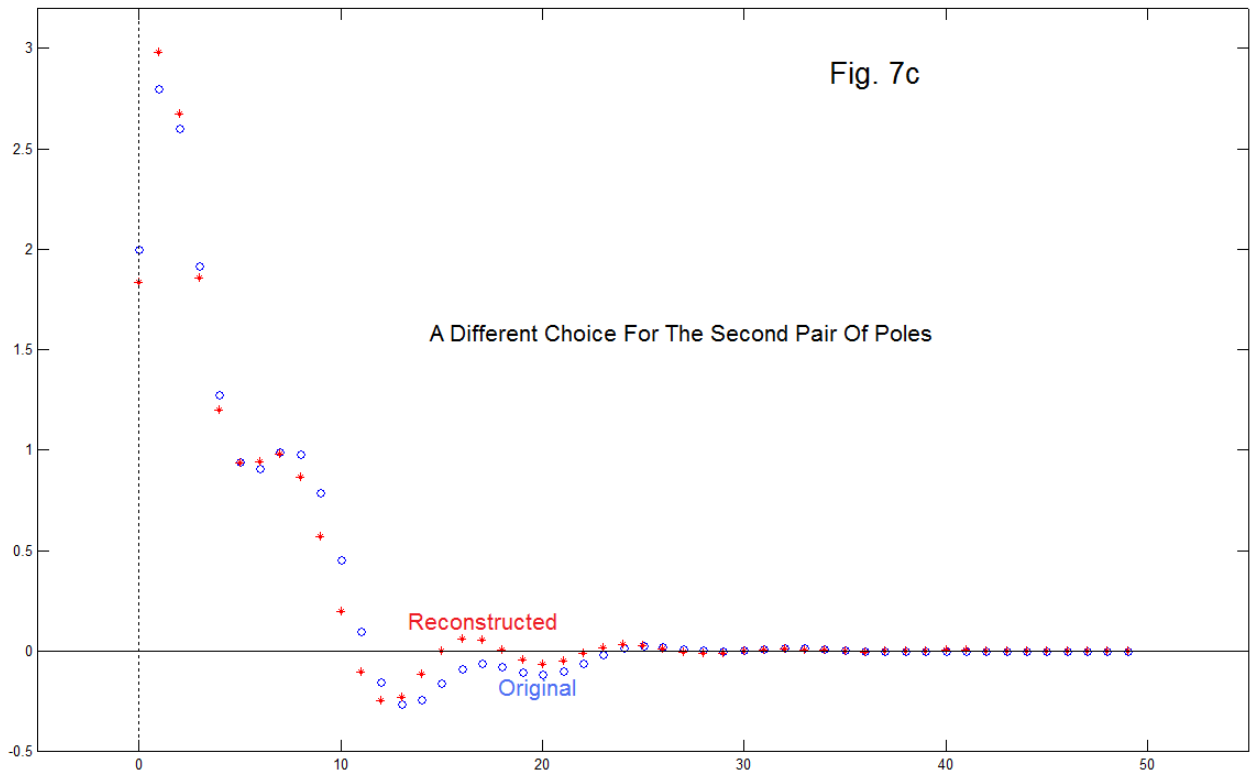
```
[k,c,w]=ktest([0.6+.6*j,0.6-0.6*j,0.8+0.15*j,0.8-0.15*j],x)
```



As we might expect, the reconstruction in Fig. 7b is not perfect, although it is not terribly wrong. That is, we changed the imaginary part of the second pair of poles from 0.2 to 0.15. It is clearly easy to make it worse, if not unrecognizable, by moving the poles further from their original positions To make it better, we can of course choose better poles, closer to the true ones that actually generated x. Alternatively we can consider including additional poles. While we generated the signal as a four-pole example, we can try poles in a larger number. Perhaps more poles working on the problem will help – in the same sense that adding more frequencies to a Fourier series help. We can start with a case (changed From Fig. 7a as shown in red):

```
 [k,c,w]=ktest([0.6+.6*j,0.6-0.6*j,0.7+0.2*j,0.7-0.2*j],x)
```

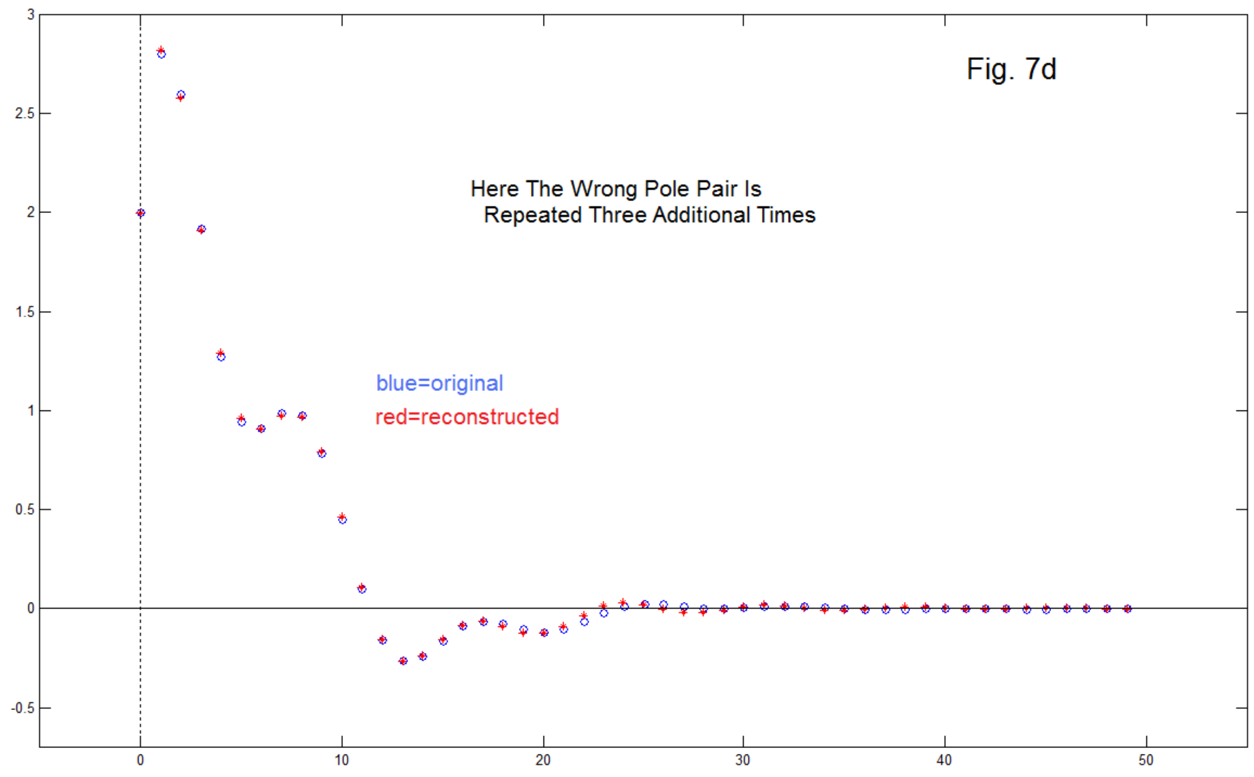
with the result as shown in Fig. 7c.



This is in one sense merely a second example of what happens when we choose the wrong poles. Here we are using it as a baseline case to which we intend to add more pole pairs. One simple thing we can do here is to add the new poles more than once – in fact three more times for a total of four. This is shown by:

```
[k,c,w]=ktest([0.6+.7*j,0.6-0.6*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j ],x)
```

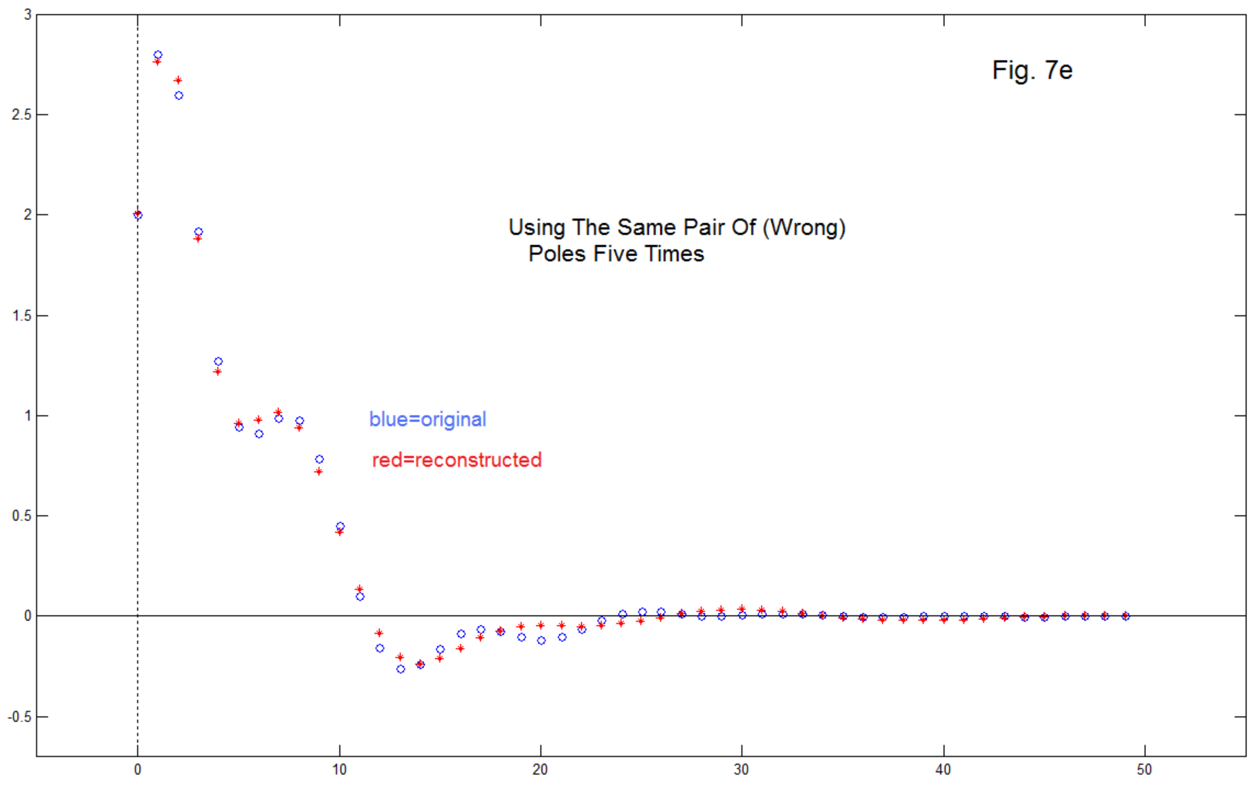
Here the three additional pole pairs (red), while identical to the those used (once) for Fig. 7c, do greatly improve the reconstruction, as seen in Fig. 7d.



This is a significant result as it suggests that while “wrong” poles may not in themselves lead to a satisfactory representation, if they are used multiple times, or perhaps with other wrong poles, we might see continued improvement. The next logical step is to consider using only one wrong pole pair multiple times. This we conveniently do by using the case of Fig. 7d and replacing the first pole pair (which was a pair of the original four-pole resonator) with the wrong pair we used four times. That is, we use the wrong pair at total of five times now (changes again in red):

```
[k,c,w]=ktest([0.7+.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j],x)
```

The result shown in Fig. 7e, while clearly (as expected) not as good as that of Fig. 7d where we retained one pair of “correct” poles, is also clearly better than Fig. 7c. Accordingly there is merit to using additional poles. Does this mean that we can just choose a multiplicity of a more-or-less arbitrary pole pair, with any input signal, and achieve useful results?



To test the idea of representing an arbitrary time sequence with a large set of poles, here we try to represent a rectangular function with the same set of poles we used for Fig. 7e. That is:

```
    [k,c,w]=ktest ([0.7+.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,0.7+0.2*j,0.7-0.2*j,
            0.7+0.2*j, 0.7-0.2*j,0.7+0.2*j,0.7-0.2*j], [1 1 1 1 1 1 1 1 zeros(1,42)] )
```

The result seen in Fig. 8a shows 10 poles (the pair 0.7+0.2j and 0.7-0.2j) used five times, and it shows an attempt to represent the rectangle shown, quite reminiscent in fact of a Fourier series truncation. Our feelings that somehow we are dealing with a basis set of spanning functions is intensified. We can take another look by again using 10 poles, but this time, spreading them around. Our choice for this “array” is to place the poles at the same radius (0.74) as the 10 above for Fig. 8a, but this time place them at ±10°, ±20°, ±30°, ±40°, and ±50° with respect to the real axis. The result shown in Fig. 8b is somewhat similar to Fig. 8a. In fact, it is somewhat better when we look closely. The “ripple” is much the same, but the transition rate from high to low is significantly faster. The case where the 10 poles were five pairs all the same, the angle was about 16°. Accordingly we understand the faster transition rate as being due to the higher frequencies of many of the poles in the arc-like array.

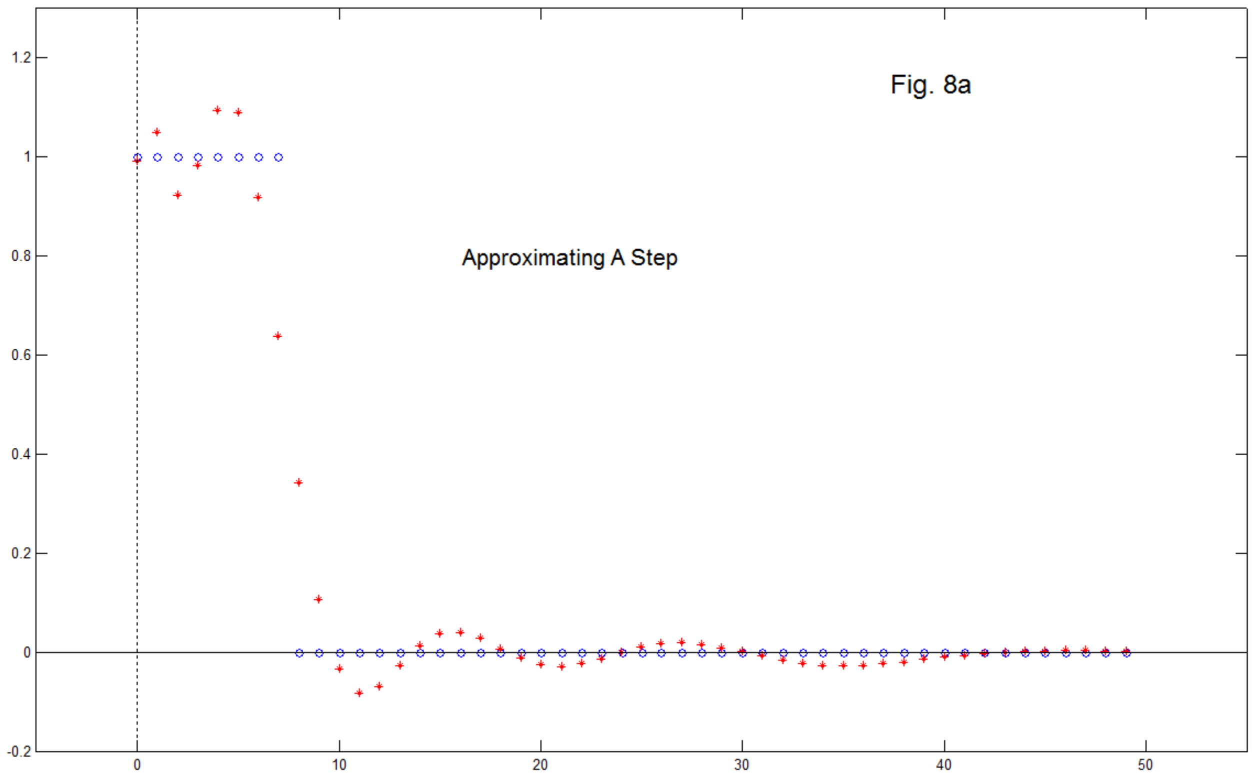


Fig. 8a
Approximating A Step
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
10
20
30
40
50

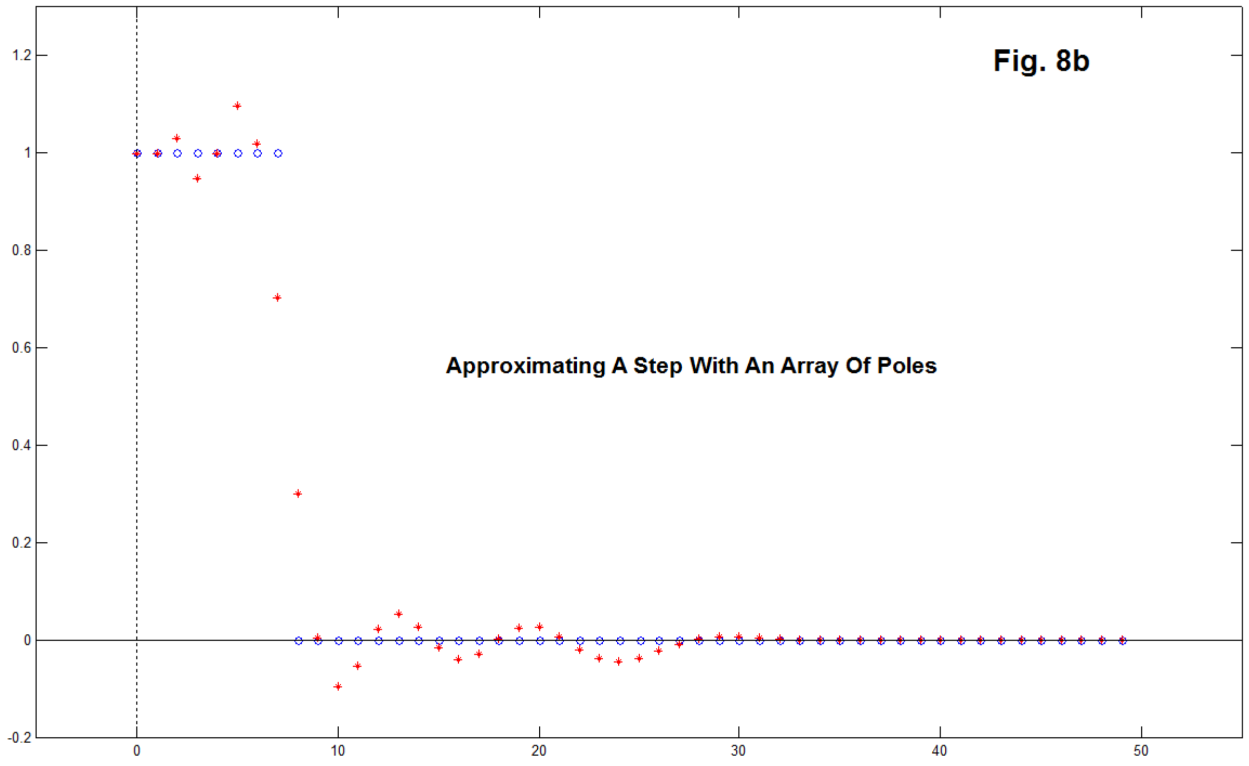


Fig. 8b
Approximating A Step With An Array Of Poles
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
10
20
30
40
50

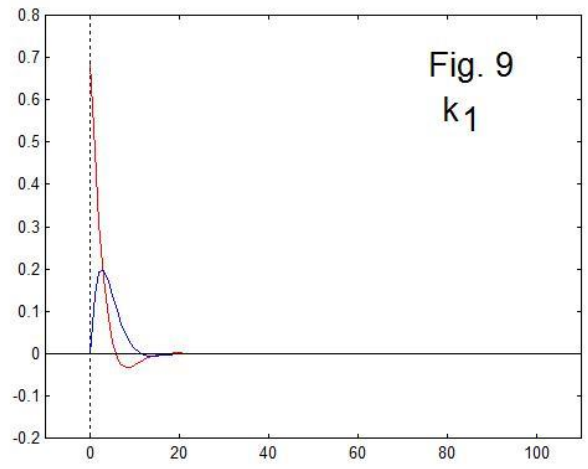


Fig. 9
k_1

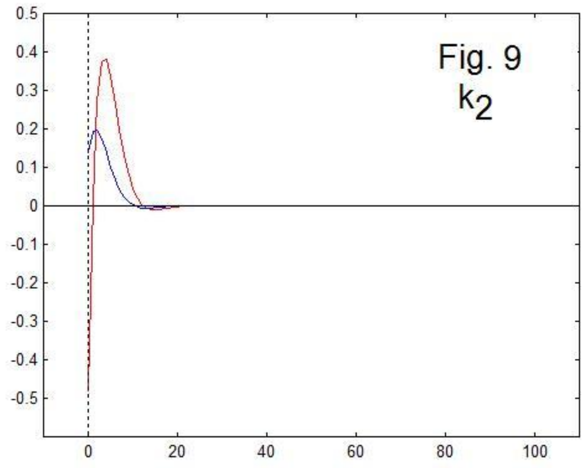


Fig. 9
k_2

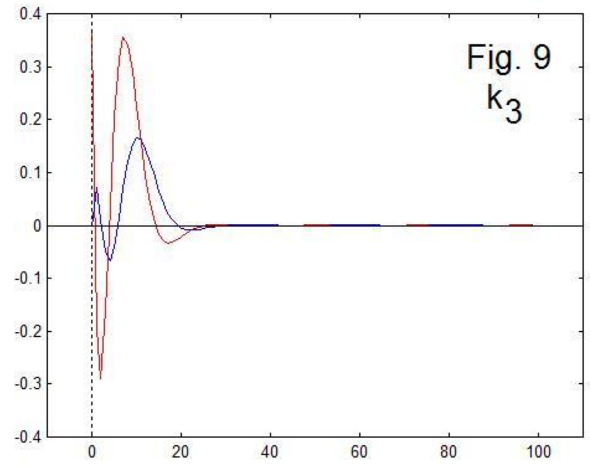


Fig. 9
k_3

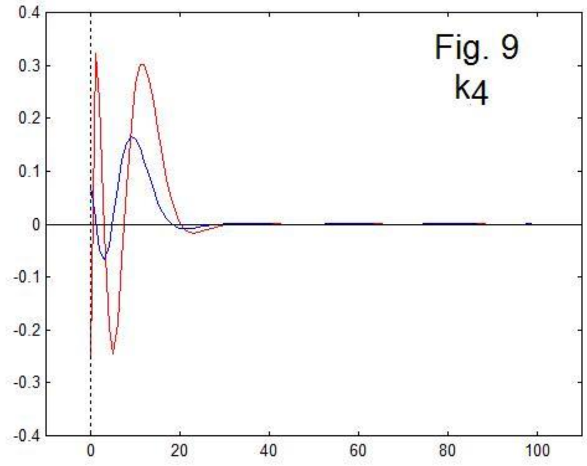


Fig. 9
k_4

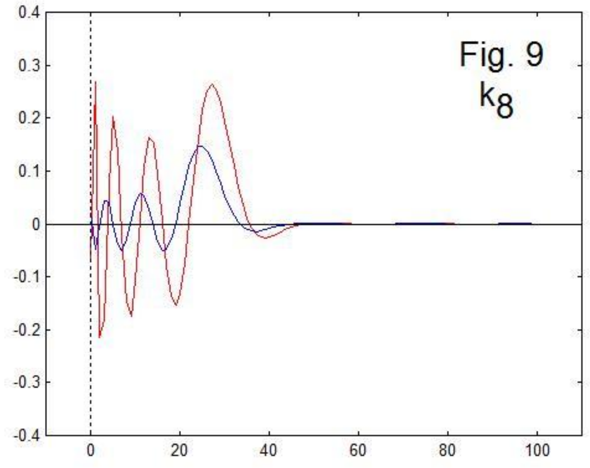


Fig. 9
k_8

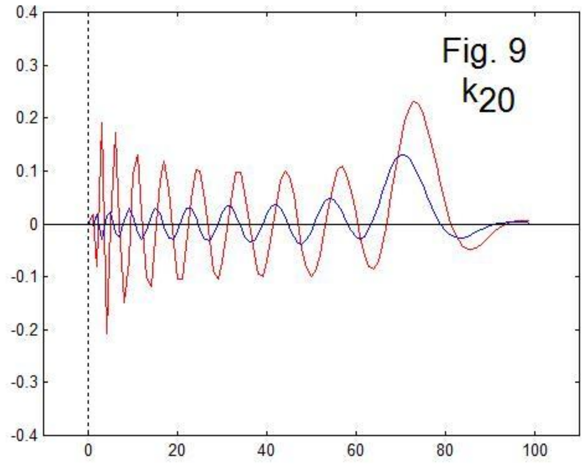


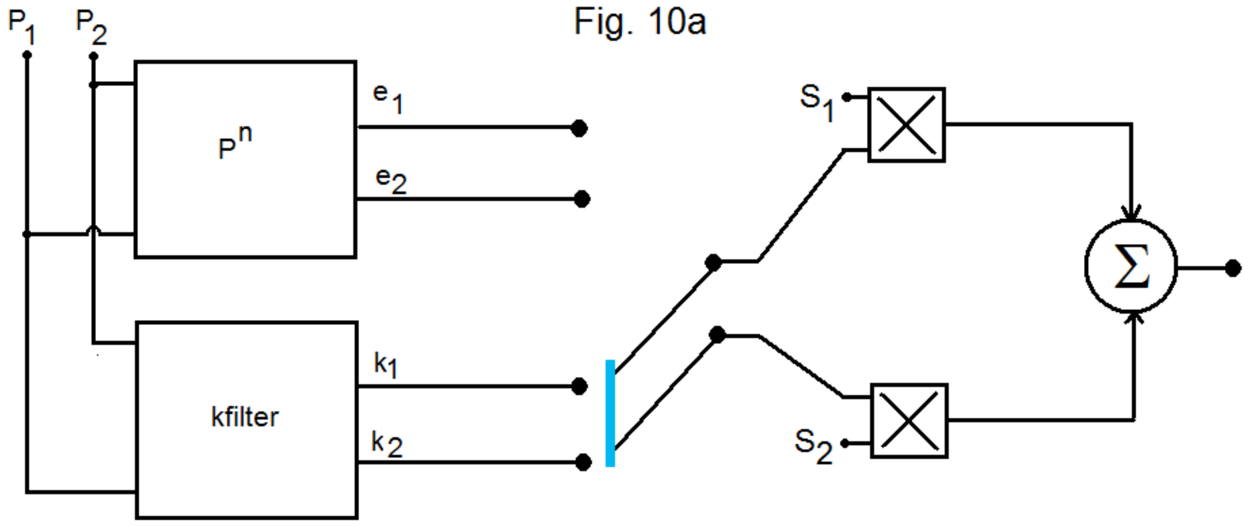
Fig. 9
k_{20}

The reader has noted that the Kautz functions are not a fixed set of functions (as DFT time-domain sequences are, for example) but depend on the choice of poles. Yet we do seem to achieve a useful orthonormal expansion, and we might be served by seeing a fuller set of the waveforms if some coherent viewpoint can be established. The case where all the poles are in the same point offers such an opportunity, and this is shown in Fig. 9 where we show the first four Kautz functions, the eighth, and the 20th. We can follow the series, in fact showing increasing ripple with order. In this case, the 20 poles are in 10 conjugate pairs at 0.6±0.2j. According to the 1954 Kautz paper, this case of only one pole pair, used multiple times, is the Laguerre case, the Kautz functions being the Laguerre polynomials with an added exponential decay factor. It's always nice to know that things are tied together. In fact, in his 1954 paper Kautz did choose some familiar pole sets (such as Butterworth and Chebyshev) to work with.

A SOUND EXAMPLE

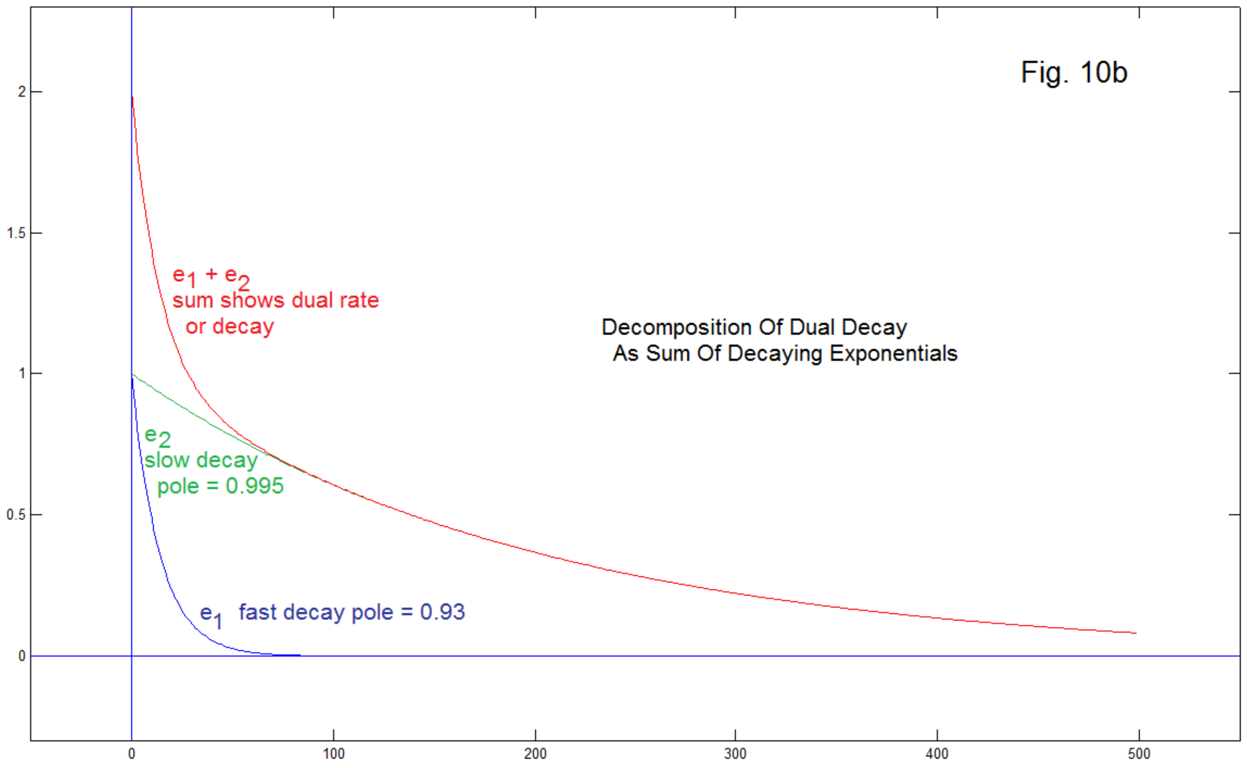
At this point it is interesting to try to find out what things sound like. We can easily listen to the Kautz functions, such as the K20 case of Fig. 9, and as many readers suspect, you don't hear much since there are still relatively few cycles. Instead, we will follow up on the suggestion made earlier (discussion around Fig. 6) to use the Kautz functions as envelopes.

Fig. 10a

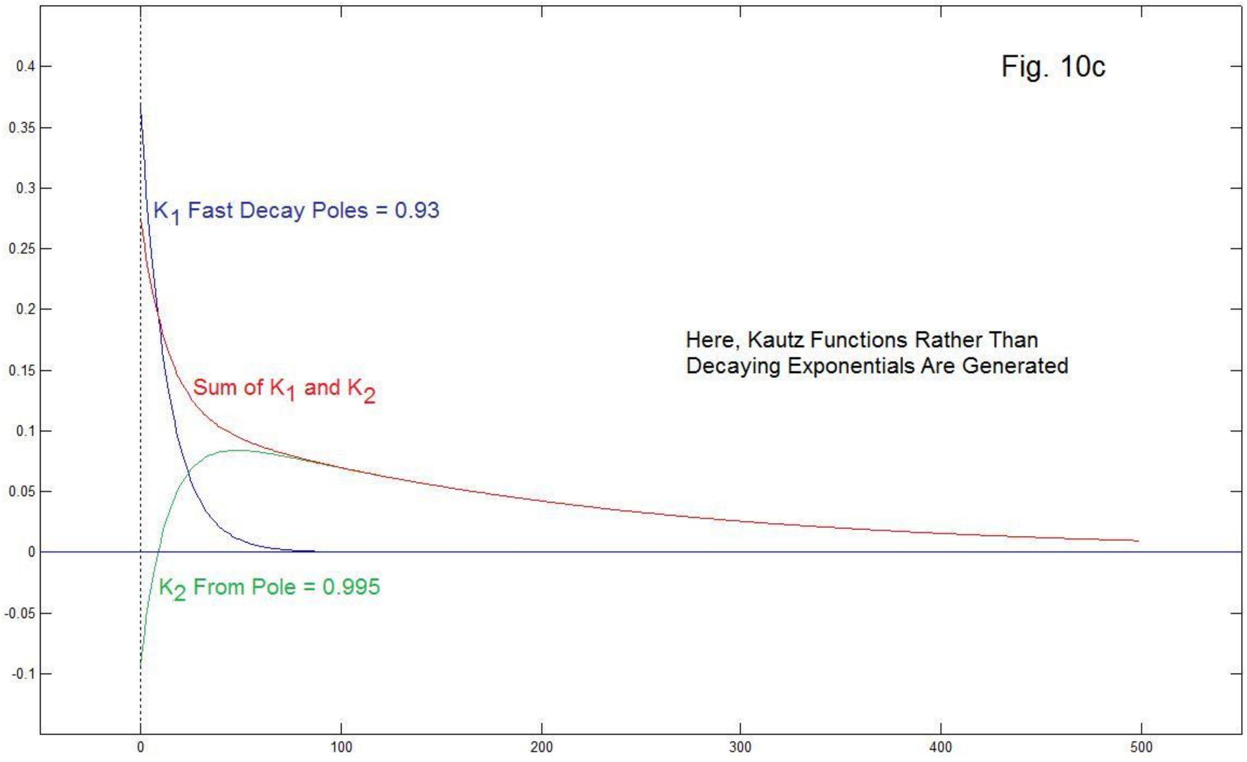


In this experiment, diagrammed in Fig. 10a, we propose to listen to percussive sounds generated from ordinary exponential decay envelopes, and Kautz envelopes. A more-or-less standard "additive synthesis" scheme is shown here. We have two sources (S_1 and S_2) that are sinusoidal waveforms. One of these S_1 is thought of as the attack transient source, while the second S_2 is more the tone itself. The amplitudes of these two are set by envelopes that are either exponentials, or Kautz functions, according to the switch shown. The basic idea is that we want a percussive sound such as a wooden marimba – a resonant block of wood struck by a hard mallet. That is, a "plink" sound. Such a sound is basically an exponentially decaying sinewave (with an interesting overtone structure). At the very beginning, there is the transient of the strike. Perhaps it is some rapidly decaling modes of the wood bar (like side to side rather than longitudinal) or perhaps a mode of the mallet itself. In any case, it make a big difference as to whether the sound is natural or not. We thus think of the envelope for S_1 as being at the very beginning and relatively short (and S_1 as being a higher frequency). The envelope of S_2 should be much longer, and certainly at least asymptotically exponential.

The most natural approach is to use two decaying exponentials (e_1 and e_2) as shown in Fig. 10b (switch of 10a UP). Here there is the rapidly decaying (blue) curve and the more slowly decaying green curve (which soon merges with the sum in red). We might choose S_2 as a sine wave and S_1 as a third harmonic, or something more at random. This is a successful approach. The alternative is to use the Kautz envelopes k_1 and k_2 (switch in Fig. 10a DOWN). The k_2 envelope is quite different from the e_2



envelope (see Fig. 10c). It does start negative (this is probably not important). It then goes through zero and turns the S_2 signal on, in fact cross-fading with S_1 as controlled by k_1. The envelope k_1 is just a decaying exponential. Thus there is a naturally produced delay. What do we hear?



The experiment here is conducted with Matlab using the ***sound*** function. The program code is given below as ***ktone***. Figures 10b and 10c were produced by ktone, and the output cases are seen in Fig. 10d and Fig. 10e. The ***sound*** commands allow us to listen to the individual decays as well as the sums.

The result is that there is a significant difference between the exponential envelopes and the Kautz envelopes. It is not overwhelming, but it can be easily heard if one experiments with the program a bit (the results in Fig. 1d and Fig. 10e correspond to the example command line in the header comments of ***ktone***. In fact, by comparing Fig. 10d to Fig. 10e we see a clear difference. Fig. 10d still looks exponential, and barely like the dual rate decay we were hoping for. In Fig. 10e, we see a dual rate, and more of a detachment between the attack transient and the main portion (in fact, there is a region of rise. What we hear is in some ways more interesting. Perhaps I can describe it as a “ker-plink” (Fig. 10e) instead of just a “plink”. Either sounds better than just a single source and one exponential envelope.

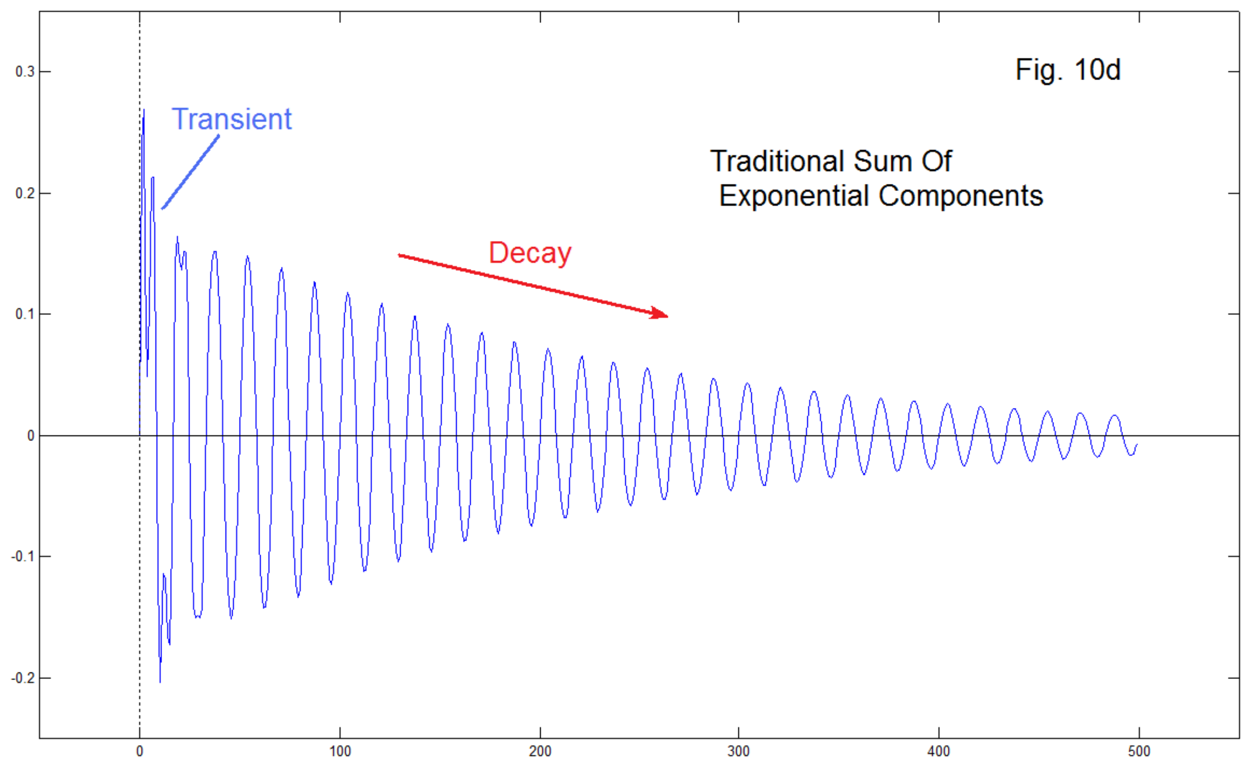


Fig. 10d
Transient
Traditional Sum Of Exponential Components
Decay
0.3
0.2
0.1
0
-0.1
-0.2
0
100
200
300
400
500

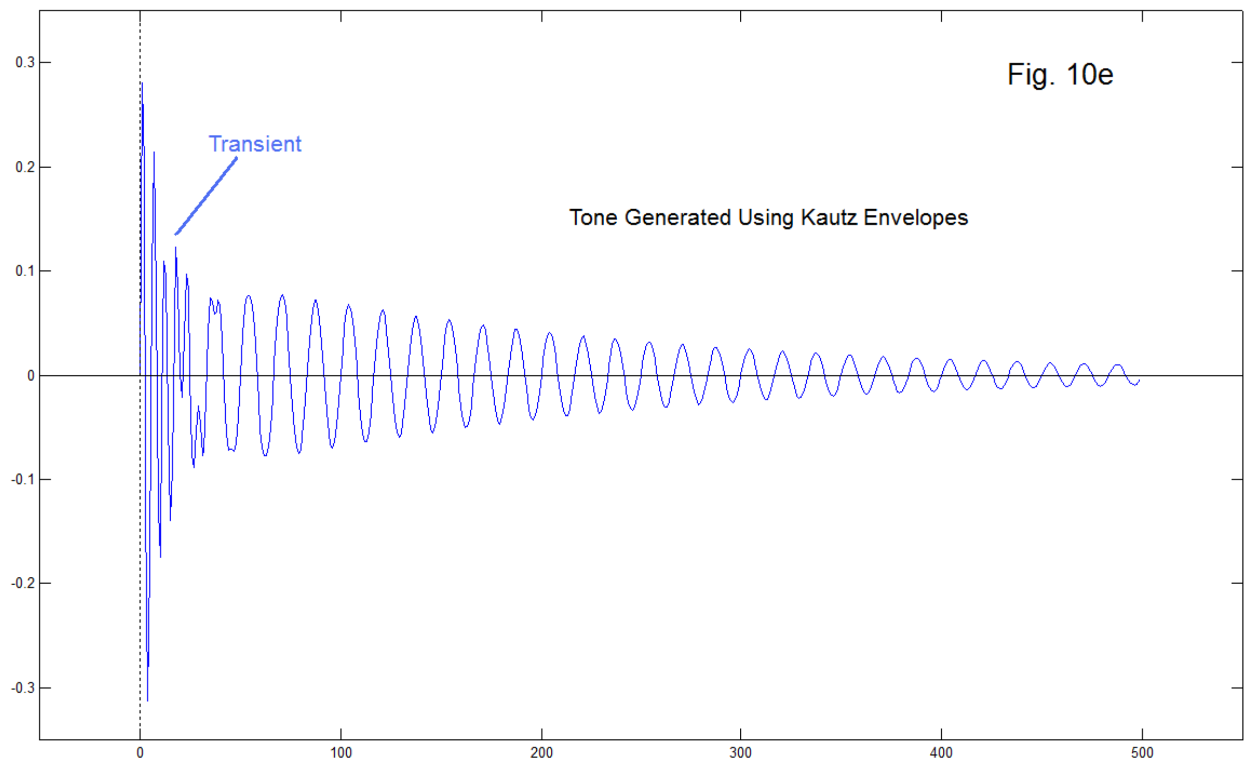


Fig. 10e
Transient
Tone Generated Using Kautz Envelopes
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
0
100
200
300
400
500

Program ktone

```
function ktone(p1,p2,f1,f2,N,fs)
% p1, p2 are real poles ---> K1, k2 Kautz functions
% f1, f2 frequencies to be enveloped by k1, k2
% N = number of samples
% fs = playback samplign rate
% example    ktone(.93,.995,1800,600,500,10000)

k=kfilter([p1,p2],N,[1 zeros(1,(N-1))]);
k1=k(:,1);
k2=k(:,2);
figure(1)
plot([0:(N-1)],k1,'b')
hold on
plot([0:(N-1)],k2,'g')
plot([0:(N-1)],(k1+k2),'r')
hold off

y1=sin(2*pi*f1*[0:(N-1)]/fs).*k1';
y2=sin(2*pi*f2*[0:(N-1)]/fs).*k2';
e1=p1.^(0:(N-1));
e2=p2.^(0:(N-1));
y3=e1.*sin(2*pi*f1*[0:(N-1)]/fs)+e2.*sin(2*pi*f2*[0:(N-1)]/fs);
y3=y3/5;

figure(2)
plot([0:(N-1)],e1,'b')
hold on
plot([0:(N-1)],e2,'g')
plot([0:(N-1)],(e1+e2),'r')
hold off

figure(3)
plot([0:(N-1)],y3)
figure(4)
plot([0:(N-1)],y1+y2)

sound(y1,fs)
pause
sound(y2,fs)
pause
sound(y3,fs)
pause
sound((y1+y2),fs)
```

ANALOG POSSIBILITIES

Everything we have done so far has been digital rather than analog, except for the mention that Kautz's 1954 paper was of course analog. It would appear that the analog is very similar. Fig. 11a shows an analog circuit where we have basically a first-order all-pass (upper left) feeding a low-pass on the right. This produces the k_2 Kautz function, and trivially, we use the op-amp follower to tap off the k_1 function. This envelope generator needs to be initiated with an impulse (sharp rectangle at least), not with a traditional gate. But of course, it is the "trigger" rather than the "gate" that is of interest in the generation of percussive sounds.

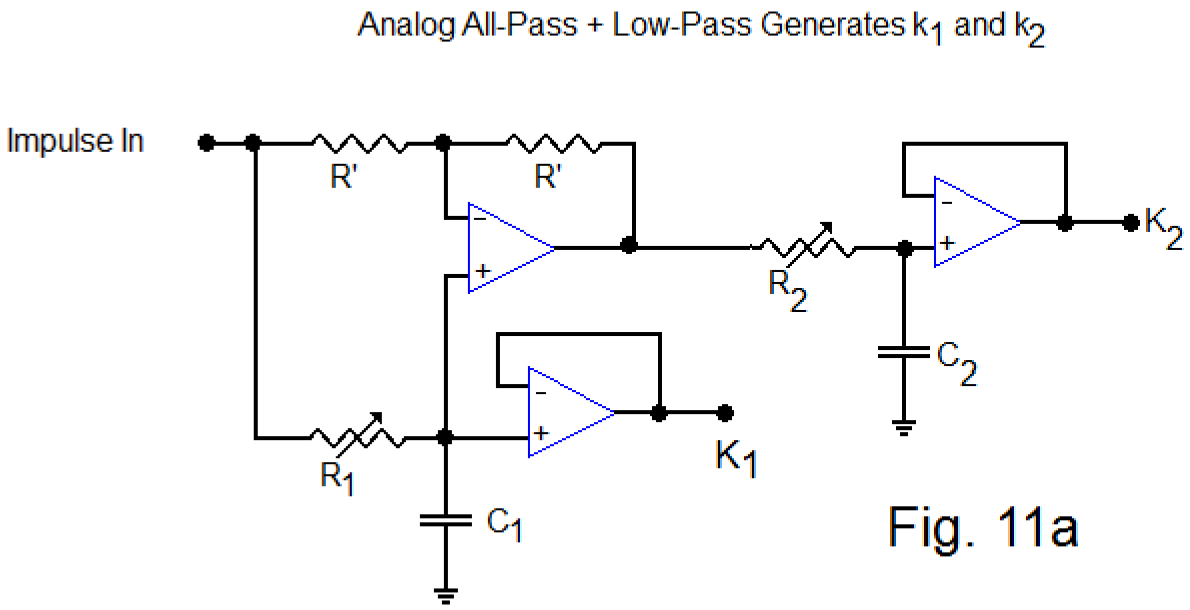


Fig. 11a

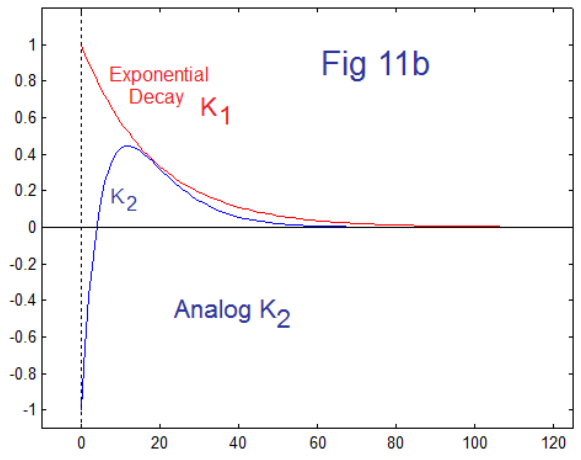


Figure 11b shows the (Matlab) calculated analog impulse responses for k_1 and k_2. No particular scaling is imposed – both start at magnitude 1. It is clear however that we can get the desired crossfade here. No analog synthesis test have been conducted on this, although it looks like a very doable design and experiment. The basic circuit of Fig. 11a was breadboarded with 10 second time constants to experimentally verify the basic waveshapes of Fig. 11b.

SUMMARY

Here we have looked at Kautz functions that are not as well knows as perhaps they should be. Accordingly we have introduced the functions and certain tools, including an easy-to-understand digital filter that generates them. Based on very preliminary work, It is likely that our main interest will be in the use of the Kautz functions as control functions rather than sounds themselves. It is further shown (Fig. 11) that an analog implementation leads to a very simple dual envelope generator where the two envelopes have a natural "cross-over" relationship summing to a natural exponential decay (while keeping source components separate until after enveloping). This would seem to be an interesting new project in analog synthesis.

REFERENCES

[1] B. Hutchins, "Design Considerations for Digital Sine-Wave Oscillators", Electronotes Application Note 299, Sept/Oct. 1987

[2] G. Weinreich, "Coupled Motion of Piano Strings", Scientific American, January 1979; and in "Coupled Piano Strings", J. Acoustical Soc. Amer., Vol. 62, No. 6, pp 1474-1484 (1977)

* * * * * * * * * *