An Ultra Fast
Magic Sinewave Calculator

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

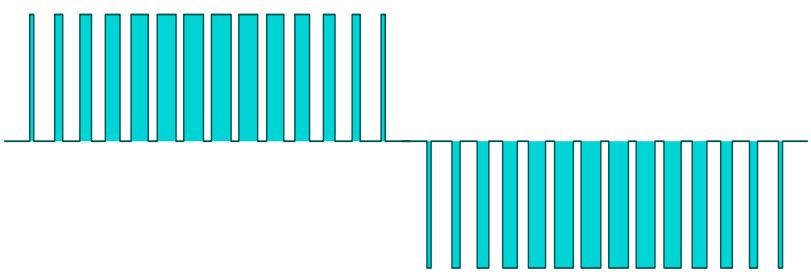
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Magic Sinewaves are a newly discovered class of mathematical functions that hold significant potential to dramatically improve the efficiency and power quality of solar energy synchronous inverters, electric hybrid automobiles, and industrial motor controls, among many others. An executive summary can be found **here**, a slideshow type intro presentation **here**, a development proposal **here**, the latest calculator **here**, and detailed additional tutorials and design info **here**.

Major goals of such digital sinewave generation including offering the **maximum possible efficiency** by using the fewest of simplest possible switching transitions; offering the **lowest possible distortion** by zeroing out a maximum number of low harmonics that impact power quality, whine, vibration, and circulating currents; and by using **all digital techniques** that are extremely low end microprocessor and/or microcontroller friendly.

Magic sinewaves have two remarkable properties: **Any number of desired low harmonics can be forced exactly to zero** in theory, and to astonishingly low levels when quantized to 8-bit compatible levels. And magic sinewaves use the **absolute minimum possible and simplest energy-robbing transitions** to achieve such harmonic suppression.

A typical **magic sinewave** might look something like this…



We see that this waveform is a variation on **PWM** or **pulse width modulation**.

Its highly unique characteristics are that it has far fewer energy robbing transitions than conventional PWM, that it is always exactly phase- and frequency locked to a fundamental, and uses half bridge rather than full bridge switching events for further efficiency improvment. Additional advantages include a 100 percent modulation depth allowing the carrier to **never** exceed the fundamental.

Plus, of course, zeroing out any chosen number of low harmonics and doing so with an absolute minimum of switching events.

There are several different types of Magic Sinewaves possible. Three of emerging interest are called **Best Efficiency**, **Bridged Best Efficiency**, and **Delta Friendly**. A **Best Efficiency Magic Sinewave** zeros out an additional two harmonics. When compared to conventional earlier solutions. Brought about by an invisible and zero integrated width pulse at zero degrees.

A bridged best efficiency is similar but is continuous at 90 degrees, And fills in with alternate values. A **delta friendly** magic sinewave meets the exacting special needs of three phase power systems. There are fewer of these at present, limited to **3, 7, 11, 15, ...** or more pulses per quadrant. They zero out somewhat fewer low harmonics but have a major advantage of needing only **one-half the storage** for amplitude data values.

Magic sinewaves are extremely exacting in their solutions. A typical equation set for a seven pulse per quadrant best efficiency **magic sinewave** might be…

$$
\begin{aligned}
&\cos(1{*}p1s)-\cos(1{*}p1e)+\ldots+\cos(1{*}p7s)-\cos(1{*}p7e) = ampl{*}pi/4\\
&\cos(3{*}p1s)-\cos(3{*}p1e)+\ldots+\cos(3{*}p7s)-\cos(3{*}p7e) = 0\\
&\cos(5{*}p1s)-\cos(5{*}p1e)+\ldots+\cos(5{*}p7s)-\cos(5{*}p7e) = 0\\
&\cos(7{*}p1s)-\cos(7{*}p1e)+\ldots+\cos(7{*}p7s)-\cos(7{*}p7e) = 0\\
&\cos(9{*}p1s)-\cos(9{*}p1e)+\ldots+\cos(9{*}p7s)-\cos(9{*}p7e) = 0\\
&\cos(11{*}p1s)-\cos(11{*}p1e)+\ldots+\cos(11{*}p7s)-\cos(11{*}p7e) = 0\\
&\cos(13{*}p1s)-\cos(13{*}p1e)+\ldots+\cos(13{*}p7s)-\cos(13{*}p7e) = 0\\
&\cos(15{*}p1s)-\cos(15{*}p1e)+\ldots+\cos(15{*}p7s)-\cos(15{*}p7e) = 0\\
&\cos(17{*}p1s)-\cos(17{*}p1e)+\ldots+\cos(17{*}p7s)-\cos(17{*}p7e) = 0\\
&\cos(19{*}p1s)-\cos(19{*}p1e)+\ldots+\cos(19{*}p7s)-\cos(19{*}p7e) = 0\\
&\cos(21{*}p1s)-\cos(21{*}p1e)+\ldots+\cos(21{*}p7s)-\cos(21{*}p7e) = 0\\
&\cos(23{*}p1s)-\cos(23{*}p1e)+\ldots+\cos(23{*}p7s)-\cos(23{*}p7e) = 0\\
&\cos(25{*}p1s)-\cos(25{*}p1e)+\ldots+\cos(25{*}p7s)-\cos(25{*}p7e) = 0\\
&\cos(27{*}p1s)-\cos(27{*}p1e)+\ldots+\cos(27{*}p7s)-\cos(27{*}p7e) = 0
\end{aligned}
$$

Power polynomials of this complexity are unlikely to have a direct solution. Instead, **Newton's Method**, otherwise known as "**shake the box**" has proven to be an effective solution route. In which a **good guess** is made based on a previously useful result or a nearby amplitude. This is followed by one or more iterations of **improvement** to the good guess.

Such an initial guess **presupposes** one and only one solution for a given magic sinewave equation. Some experiments using **Monte Carlo Methods** do strongly suggest that single solutions are likely the case.

Per this **example code** and **this result**.

The general concept is to generate tens to hundreds of millions of random pulses, filter them to low distortions, and seek out any exceptions to the known solution set. Things rapidly get out of hand beyond **n=4**. But all of the lower order models strongly support uniqueness.

An extensive set of older JavaScript based interactive calculators is found **here**. These earlier calculators use a brute force iterative method that had demanded repeated trig calculations to seek the harmonic distortion minimums. While quite effective and useful, their initially slow computing times became excessive when many dozens or hundreds of harmonics are to be zeroed.

In **GuruGram #72**, some very preliminary and tentative work showed an improved and quasi-deterministic approach to **Magic Sinewave** solutions. However, these new solutions still remained quite slowly converging. Here we will explore some extensions to these techniques that has led to a brand new approach to **Magic Sinewave** calculations that is both exceptionally fast and quasi-deterministic.

Speedups beyond **1000:1** have been demonstrated. With typical calculation times of well under one second. As per this **current calculator demo**.

The Approach

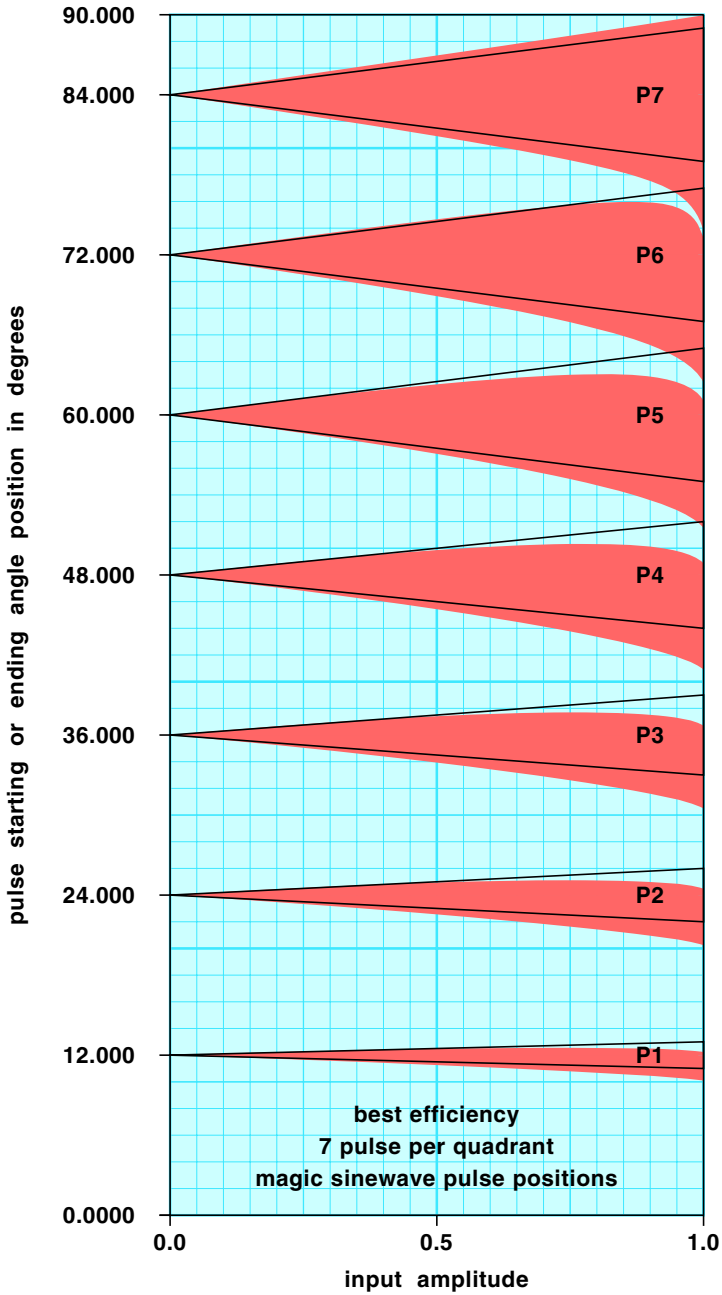
There is a fundamental mathematical proof that **no direct deterministic solutions exist for independent polynomial equation sets above order four**. But on the other hand, there are trigonometric identities that might somehow **indirectly relate** the variables in the above equations. And, as our results clearly prove, it certainly should be possible to modulate a carrier without distortion.

Whether a useful direct and deterministic solution to **Magic Sinewaves** exists remains an open question. The approach here uses a two step process of **a good guess** that is followed by **a fast converging improvement**. In some cases, a single iteration can give engineeringly useful results. And repeated iterations can end up amazingly fast. While converging to aesthetically and mathematically satisfying harmonics zeroed to well beyond fourteen decimal places.

As an additional bonus, the current technique converges **simultaneously** on the zeroed harmonics **and** on a chosen target amplitude.

Making Some Good Guesses

A better guess can start by **working backwards** from a known **Magic Sinewave** solution. While attempting to stay as close as possible to the "real" math. Here is how the **n=7** Best Efficiency Magic Sinewave angles vary with amplitude…


90.000
84.000
72.000
60.000
48.000
36.000
24.000
12.000
0.0000
pulse starting or ending angle position in degrees
P7
P6
P5
P4
P3
P2
P1
best efficiency
7 pulse per quadrant
magic sinewave pulse positions
0.0
0.5
1.0
input amplitude

We first note that very low amplitudes start off with a group of carefully locked **carrier phase impulses**. Having zero width and zero energy for zero amplitude. In the case of a best efficiency, seven pulse per quadrant magic sinewave, there will be impulses that start near **12.000**, **24.000**, **36.000**, **48.000**, **60.000**, **72.000**, and **84.000** degrees. These impulses will mirror over the **90** to **180** degree range and invert over the **180** to **360** degree range.

There will also be two "invisible" carrier phase impulses you'll find at **0** and **180** degrees. **Whose very small and bipolar energy will integrate to zero**. And thus can be completely ignored. These invisible impulses are the key to a seven pulse per quadrant best efficiency magic sinewave being able to reject and zero all the harmonics through the **28th**. Or **two** more harmonics than would normally be expected. Because there really are **7-1/2** pulses per quadrant.

As the amplitudes increase, each of the carrier phase impulses will widen. This widening appears to be somewhat proportional to the **sine squared** of the carrier impulse phase angle. The fractional contribution of each carrier phase impulse can be found by summing the squares of the sines of all impulses and dividing.

Because of **Fourier Series** constant considerations, the sought amplitude will end up as **pi/4** or **0.785398163** of the **0** to **1** desired final amplitude.

As the carrier impulses fatten, they do **not** do so linearly. Instead, they will **trend downward** at very high amplitudes. Sadly, polynomials directly and accurately synthesizing these curves turn out to be incredibly complex and high order.

Instead, a "two step" guessing process is made. First a **linear expansion** get done based on **sine squared** cosine distributions. This is "good enough" for all but the highest amplitudes of certain magic sinewave solutions.

It is important to note that **these first guess angles expand as their cosines and NOT as degrees!**. Because you want just as much energy above the carrier pulse center as below. Should an amplitude fraction of **.007** be wanted, you can use…

```
starting angle =  acos ( cos(center angle) - .007)
ending angle   =  acos ( cos(center angle) + .007)
```

Another gotcha is forgetting that JavaScript works in **radians**, not **degrees**. The conversion constants are…

```
radians =  degrees * pi/180
degrees =  radians * 180/pi
```

To make sure the highest amplitudes converge, a **second guess** can be made that slightly tilts the highest amplitude angles downward…

correction = fudge * (amplitude)^4 * (angle/90)

… with a typical **fudge** value of **.02** or **.03** getting subtracted.

Summarizing, a good guess is made by first linear expanding to the sought amplitude in a sine squared weighted proportion. A second guess then slightly adjusts the highest amplitude values to guarantee convergence. Exact details can be found by using **view source** on the **calculator demo**.

Exploring a Trig Identity

It turns out the "improver" portion of our two-step algorithm is in fact **fully deterministic** when **very near** a given **Magic Sinewave** solution. To understand exactly why this is so, we can look at this **trig identity**…

cos(a + x) = cos(a) cos(x) - sin(a) sin(x)

This identity is true for all values of **a** and **x**. Useful simplifications can result if we are in the first quadrant and if **a** is much larger than **x**. If **x** is very nearly zero, its cosine will be close to one and its **radian** value will nearly equal its argument.

Which simplifies to…

cos(a+x) approximates cos(a)-xsin(a) if a >> x

This expression **exactly** matches that used by **Newton's Method**! Where you make a better approximation to a solution by multiplying its present error by the **slope** of the function and add this to the present value.

Note that the slope of the cosine is minus the sine. And also that the slope of **cos(nx)** is **-n*sin(nx)**.

It can also be of interest to find an even better approximation. The power **series definition** of sines and cosines are…

$$\sin(x) = x - x^3/3! + x^5/5! - \ldots$$
$$\cos(x) = 1 - x^2/2! + x^4/4! - \ldots$$

… which, when substituted in the original trig identity gives us a somewhat more precise approximation of…

$$\cos(a+x) \text{ closely equals } \cos(a)*(1-x^2/2)-\sin(a)*(x-x^3/6)$$

While this result is not needed for our current "improver" algorithm, it may prove highly useful for further refinements.

The "improver" algorithm

The "improver" algorithm ends up very close to fully deterministic when near a valid **Magic Sinewave** solution. It is based on taking our initial equations above and **substituting** each cosine value with **cos(lastguess + error)**. Rearranging constant and variable terms will leave fourteen **linear** equations in fourteen unknowns. These are easily and rapidly solved using **Gauss Jordan Elimination**.

As the fundamental amplitude error is treated as an error **in the same way** as a nonharmonic zero error, the solution rapidly converges **both** on the desired amplitude **and** on totally zeroed harmonics. This completely eliminates the small amplitude errors of the **previous calculators**. And the need for repeat trips.

A functional and super fast demo Magic Sinewave calculator **appears here**.

Summarizing our "improver" rules…

Each cosine term in the basic Magic Sinewave equations gets substituted with cos(bestguess + error)

This gets approximated by cos(bestguess) - error*slope. Note that the slope of cos(nx) is -n*sin(nx).

Terms are rearranged, leaving an array of n linear equations in n unknowns.

The equations are solved, either using Gauss Elimination and back substitution. Or else Gauss-Jordan Elimination.

Errors are replaced using the cos (a+x) trig identity. Leaving a very close and nearly deterministic solution.

Let's look at some more detail. Our fundamental equation from above was…

$$\cos(1*p1s)-\cos(1*p1e)+\ldots-\ldots+$$
$$\cos(1*p7s)-\cos(1*p7e)=ampl*pi/4$$

Replace each cos with a sum of our known guess and unknown error **xn**…

```
cos(p1sg + x1)   - cos(p1eg + x2) +
cos(p2sg + x3)   - cos(p2eg + x4) + ... +
cos(p7sg + x14) - cos(p7eg + x14)  =  ampl*pi/4
```

Assume **xn** is very small and substitute its **cos (a+x)** approximation…

```
cos(p1sg)  - x1*sin (p1sg) -
cos(p1eg) + x1*sin (p1sg) + ... = ampl*pi/4
```

Note that **signs alternate** between starting and ending angles. Since **p1xg** is known, its sine and its cosine will be **constants**. Change sign and rearrange all constants to the right side of the equation…

```
x1*sin(p1sg) - x2*sin(p1eg) +
x3*sin(p2sg) - x4*sin(p2eg) + ... - ...  =

ampl*pi/4 + cos(p1sg) - cos(p1eg) +
cos(p2sg) - cos(p2eg) + ...
```

When all constants are substituted and combined, this becomes a fourteen term linear equation of form…

```
[j0,0](x1)+[j0,1](x2)+[j0,3](x3)+...+[j0,13](x14) = [k00]
```

Solving the harmonic equations are similar noting the slope of **cos(nx)** will be **-n*sin(nx)**. Giving us a **linear** equation set of 14 variables in 14 unknowns…

```
[j0,0](x1)  +[j0,1](x2)  +[j0,3](x3)  +...+[j0,13](x14) = [k00]
[j1,0](x1)  +[j1,1](x2)  +[j1,3](x3)  +...+[j1,13](x14) = [k01]
                  .  .  .  .  .  .  .  .  .  .  .  .
[j13,0](x1)+[j13,1](x2)+[j13,3](x3)+...+[j13,13](x14)=[k13]
```

Which, despite its apparent complexity, can easily be solved by either **Gaussian elimination** followed by **back substitution**. Or else by **Gauss-Jordan elimination**. The latter is preferable when expanding to larger magic sinewave solutions. Once the **x** errors are found, they are easily combined with the guess angles using the above exact **cos(a+x)** trig identity.

Convergence is amazingly rapid and speed appears at least a thousand times faster than the earlier calculations. Again, a demo can be found **here**.

Some Delta Friendly Considerations

If three phase loads are to be driven without needing rewiring and using only three half bridge drivers, special **delta friendly magic sinewaves** are required.

These are summarized in **this tutorial**.

Known three phase magic sinewave solutions are presently limited to **n=3**, **n=7**, **n=11**, and higher **(4x + 3)** pulses per quadrant. Because **all** triad harmonics **must** be explicitly cancelled, delta friendly magic sinewaves zero out a fewer number of low harmonics. But their benefits include having to solve only **one half** the usual number of linear equations and require only **one half** of the data storage.

For instance, a **7** pulse per quadrant magic sinewave might use seven of its pulse edges to guarantee explicit triad cancellation, one pulse edge (used in obscure combination with the others) to set the amplitude, and the remaining six edges (again in combination) used to zero out harmonics **5**, **7**, **11**, **13**, **17**, and **19**. Since **21** is a triad harmonic and no even harmonics are present, the first uncontrolled harmonic would be the **23rd**. Compared to the **29th** for a single phase, seven pulse best efficiency magic sinewave.

Again for **n=7**, it is convenient to make the controllable edges **p4s**, **p4e**, **p5s**, **p6s**, **p6e**, **p7s**, and **p7e**. The other edges **must** be forced to obey this rule set…

```
p1s = 60 - p5s
p1e = p6e - 60
p2s = p7s - 60
p2e = 60 - p4e
p3s = 60 - p4s
p3e = p7e - 60
p5e = 120 - p6s
```

Instead of the usual **14** equations in **14** unknowns, we should be able to come up with only **7** equations in **7** unknowns instead. With each of the new variables representing a curious **vector sum** of the paired original edges…

```
cos (1*(p4s-30))* 1.732     - cos (1*(p4e-30))* 1.732 +
cos (1*(p5s-30))* 1.732 + cos (1*(p6s+30))* 1.732 -
cos (1*(p6e-30))* 1.732 + cos (1*(p7s-30))* 1.732 -
cos (1*(p6e-30))* 1.732 = amplitude * pi/4
```

Yes, these equations are truly bizarre. A complete derivation is included in the **demo**, which you can access through the usual "**view source**" route.

Note that the fourth term is different from the others. Because it relates a leading and trailing pulse edge. Also note that **1.732** more precisely is **2*sin(60)**.

The harmonic equations are similar to the above, except the "**1**" gets replaced by the non-triad harmonic numbers of **5**, **7**, **11**, **13**, **17**, and **19**. And **the output gets divided by the harmonic number**. Also, the overall harmonic signs invert for **5**, **7**, **17**, and **19**. Thus the equation for **5h** produces **minus** the actual fifth harmonic. Once again, a derivation appears in the **demo calculator**.

Calculator Design and Structure

The new ultra speed calculators differ dramatically from the earlier versions. Here are some of the key differences…

"N" INDEPENDENT CODE -- As many of the functions are made as independent of the pulse-per-quadrant and display box counts as possible. This enormously simplifies rewrites for different sizes of magic sinewaves.

NORMALIZATION -- Internal calcs are done with JavaScript preferred radian angles and Fourier rather than absolute amplitudes. Final values are limited to the display only.

ARRAY TECHNIQUES - A numerically accessed **Angles[x]** and a supporting **Harms[x]** array eliminates keeping track of fancy variable names and display positions.

CODE SPLITTING - The code is in two halves, an "analyze" portion that keeps the display happy and the "adjust" portion that provides newer and better values. Central to this is "pivoting" on the **Angles[x]** array. Which is the primary link between the two.

EXTENSIVE LOOPING - Used when and where possible to keep the code compact and to encourage "**n**" independence.

IMPROVED GAUSS-JORDAN - Latest versions of the required **n x n** linear equation solvers are ultra compact, amazingly fast, and fully "**n**" independent.

EXPORT AREAS - New cut and paste regions can greatly simplify extracting all angles for further use.

A Brief Gauss-Jordan Tutorial

Gaussian elimination is the process of playing around with some array values ahead to time to greatly simplify a final solution. Consider five linear equations in five unknowns...

A0*v + B0*w + C0*x +D0*y + E0*z = K0
A1*v + B1*w + C1*x +D1*y + E1*z = K1
A2*v + B2*w + C2*x +D2*y + E2*z = K2
A3*v + B3*w + C3*x +D3*y + E3*z = K3
A4*v + B4*w + C4*x +D4*y + E4*z = K4

While all sorts of solution methods exist, we seek one that is computationally efficient. If we dink around with some manipulations ahead of time, we can eventually end up with a solution that will be obvious by inspection!

Arrange the coefficients into a group of arrays...

[A0 B0 C0 D0 E0 K0]
[A1 B1 C1 D1 E1 K1]
[A2 B2 C2 D2 E2 K2]
[A3 B3 C3 D3 E3 K3]
[A4 B4 C4 D4 E4 K4]

The rules for our "Gauss" part of rearrangement are that **any row can be scaled by any constant term by term without changing the results**. And that **any row can be subtracted from any other row term by term and substituted**. Again without changing the results.

In interests of sanity, let "~" be any coefficient that resulted from any and all previous manipulation. Scale the top row by dividing by its initial value...

[1 ~ ~ ~ ~ ~]
[A1 B1 C1 D1 E1 K1]
[A2 B2 C2 D2 E2 K2]
[A3 B3 C3 D3 E3 K3]
[A4 B4 C4 D4 E4 K4]

Scale the top row by A1 and subtract it from the next row down and replacing...

[1 ~ ~ ~ ~ ~]
[0 ~ ~ ~ ~ ~]
[A2 B2 C2 D2 E2 K2]
[A3 B3 C3 D3 E3 K3]
[A4 B4 C4 D4 E4 K4]

Similarly, scale the top row by A2 and subtract it from the middle row. Then scale by A3 for row 3 and A4 for row4...

```
[ 1  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
```

Now, scale the **second** row down by its first nonzero coefficient...

```
[ 1  ~  ~  ~  ~  ~ ]
[ 0  1  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
[ 0  ~  ~  ~  ~  ~ ]
```

Next, force zeros in the second column the same as we did with the first, but using the **second** row for subtraction and substitution...

```
[ 1  ~  ~  ~  ~  ~ ]
[ 0  1  ~  ~  ~  ~ ]
[ 0  0  ~  ~  ~  ~ ]
[ 0  0  ~  ~  ~  ~ ]
[ 0  0  ~  ~  ~  ~ ]
```

Keep working your way through the array, this time scaling the **third** row down by its first nonzero term and then using scaled subtractions to zero out everything below in the same column.

Eventually, you should end up with...

```
[ 1  ~  ~  ~  ~  ~ ]
[ 0  1  ~  ~  ~  ~ ]
[ 0  0  1  ~  ~  ~ ]
[ 0  0  0  1  ~  ~ ]
[ 0  0  0  0  1  ~ ]
```

This completes the Gauss part of the process. The lower right squiggle will be **z** by inspection!

Relabel the above array…

```
[ 1   c01  c02  c03  c04  j05 ]
[ 0   1    c12  c13  c14  j15 ]
[ 0   0    1    c23  c24  j25 ]
[ 0   0    0    1    c34  j35 ]
[ 0   0    0    0    1    z   ]
```

where **cxx** is the row and column coefficient for the left side equation terms, and **jxx** is the similar row and column coefficient for the right side equation term.

The traditional way to solve this was by **back substitution**. You can start off with **y = j35 - z*c34** and so on. And then work your way up a row at a time, making more complex calculations until you have **v** through **z** all solved.

The Jordan approach starts off the same way, but **it works one column at a time**, greatly simplifying computer programming. Especially when more than one **n x n** equation set size is to be accommmodated. The new rule is that **any constant can be subtracted from one term in the left side of the equation as long as that same constant get subtracted from the right side of the equation**.

Subtract **z*c34** from row 4...

```
[ 1   c01  c02  c03  c04  j05 ]
[ 0   1    c12  c13  c14  j15 ]
[ 0   0    1    c23  c24  j25 ]
[ 0   0    0    1    0    y   ]
[ 0   0    0    0    1    z   ]
```

So far, this is the same as the usual back substitution. We now can observe **y** by inspection The difference with Jordan is to continue by **working columns** instead of rows. Modify the rows by subtracting **z*c24**, **z*c14**, and **z*c04** to get...

```
[ 1   c01  c02  c03  0   j05 ]
[ 0   1    c12  c13  0   j15 ]
[ 0   0    1    c23  0   j25 ]
[ 0   0    0    1    0   y   ]
[ 0   0    0    0    1   z   ]
```

Next, modify column **three** by subtracting **y*c23**, **y*c13**, and **y*c03**. And then column **two** by subtracting **x*c12** and **x*c02**. And finally column one by subtracting **w*c01** to get...

```
[ 1  0  0  0  0  v ]
[ 0  1  0  0  0  w ]
[ 0  0  1  0  0  x ]
[ 0  0  0  1  0  y ]
[ 0  0  0  0  1  z ]
```

Your values **v** through **z** are now instantly readable by inspection!

Once again, the Jordan method takes just as many calculations as does a back substitution, but it greatly simplifies computation. In that loops do not have any multiple calculations or complicated cross-coefficients in them.

This is especially handy when it comes to making the working code independent of **n** .

A Code Example

Here's a JavaScript program that solves **nxn** linear equations. It is amazingly compact, offers **64 bit** arithmetic, and works for most any sane value of n. But it does not yet trap out any div0's or accomodate wildly varying coefficients.

Here is the main proc...

```
function solveGaussJordan() {
  gjNsize = eqns.length ;
  for (var iii = 0; iii <=(gjNsize-1); iii++){
  normaLize ( eqns[iii],iii ) ;
     for (var jjj = iii; jjj <=(gjNsize-2); jjj++) {
     subScaled (eqns[iii],eqns[(jjj+1)],iii)} } ;
  normaLize ( eqns [(gjNsize-1)],(gjNsize-1) ) ;
  jorDanify () } ;
```

It needs these three support subs...

```
function normaLize (bb,cc) { xx = bb[cc] ;
   for (var ii = 0; ii <= gjNsize; ii++)
        { bb[ii] = (bb[ii]/xx) } } ;

function subScaled (aa,bb,cc) { xx = bb[cc] ;
   for (var ii = cc; ii <=gjNsize; ii++)
        { bb[ii] -= aa[ii] *xx } } ;

function jorDanify() {
   for (var i3 = (gjNsize-1); i3 >=1; i3--){
       zz = eqns[i3][gjNsize] ;
      for (var i4 = (i3-1); i4 >=0 ; i4--)
         eqns[i4][gjNsize] -= eqns [i4][i3]*zz
         eqns[i4][i3] = 0 } } } ;
```

And here is how you would use it...

```
eq0 = [ 4, 3, -2, 1 , 22 ]     eq1 = [ 2, 1, -2, 2,  9 ]
eq2 = [ 1,-1, 1, 5 ,  8 ]      eq3 = [ 3, 1, 3, 1 , 22 ]

eqns = [ eq0, eq1, eq2, eq3] ;
solveGaussJordan () ;
```

eq0 represents **4w + 3x - 2y + z = 22**. There is an implicit equals sign before the rightmost column.

Reals as well as integers can be used. Processing time increases sharply with increasing **n**. But is well under one second for **n = 30x30**.

Returned via Gauss-Jordan elimination is ...

```
eq0 = [ 1, 0, 0, 0, w ]
eq1 = [ 0, 1, 0, 0, x ]
eq2 = [ 0, 0, 1, 0, y ]
eq3 = [ 0, 0, 0, 1, z ]
```

...and for the above example, **w=4**, **x=3**, **y=2** and **x=1**.

For Additional Assistance

Obvious next steps are expanding **the calculators** for other types and orders of magic sinewaves. Of particular interest should be suppressing **1000** or more zeros. Which now should be possible with the dramatic speedups. Also of interest is finding whether in fact a fully deterministic solution can be found. Or an accurate interpolation scheme.

These further explorations require your participation as a Synergetics **partner** or **associate**.

To proceed, view the many **Magic Sinewave** tutorial files and JavaScript calculators you'll find at **http://www.tinaja.com/magsn01.asp**.

Or you can email **don@tinaja.com**. Or call **(928) 428-4073**.