# Overview

When designing simulations, controllers, and sensor data filters, the use of linearized state space equations is huge. State space equations are one of the most widely used ways to represent physical systems in a condensed mathematical equation. The major drawback to a state space function is that it makes use of linear algebra; thus it cannot represent nonlinear systems. The often-used way of negating this issue is to linearize the physical system at the point of interest to approximate the nonlinear system as a linear system. This function approximation can be repeated as needed if the point of interest strays far away from the point being approximated.

This begs a question though: When is an equation so nonlinear that I need to resize my re-calculation bounds or re-evaluate my state space model as a whole? If you ask your professor or employer this question, they will likely answer, “It is up to the engineer to justify if their approximation is accurate enough to represent a system.”. While this is very true, it’s also not very helpful because it plants the go / no-go decision on subjective debate alone. It means that one engineer could think differently from another engineer and even crazier is they could both be right. It’s also impossible to teach a computer algorithm to make a subjective decision, because computers operate objectively. Scientists and engineers usually overcome this issue by learning how to quantify a debate and then standardizing the quantification for acceptability. For example, scientists and engineers have agreed to standardize 30 or greater as the number of sample points in a “large” sample size versus 29 or less representing a “small” sample size.

In this paper I will explain my derivation of an analytical and numerical algorithm for quantifying nonlinearity of an N-dimensional function with clearly defined boundaries. I will then use crowd-sourced data to propose a standardized set of rules for when a bounded set of variable domains is “too nonlinear” for a standard 1st order Taylor series to approximate. This paper will not attempt to standardize the acceptability of biased 1st order Taylor series.

# What is nonlinearity?

A function is linear when its’ entire domain for all dimensions (from -infinity to +infinity) has a constant slope. By extension, a nonlinear function has an inconsistent slope. In the world of pure mathematics that is all there is to it. However, in the world of practical mathematics there is room for caveats. Usually, we do not classify a function’s linearity from -infinity to +infinity, because that is impractical and unnecessary. Instead, we classify linearity based on a set of bounds that constrain the area of interest to a more practical set of values. Additionally, there are many equations which are technically nonlinear, but are so close to being linear under the described bounds, that it really doesn’t matter. To illustrate this, see Figure 1, based off Equation 1 as an example.

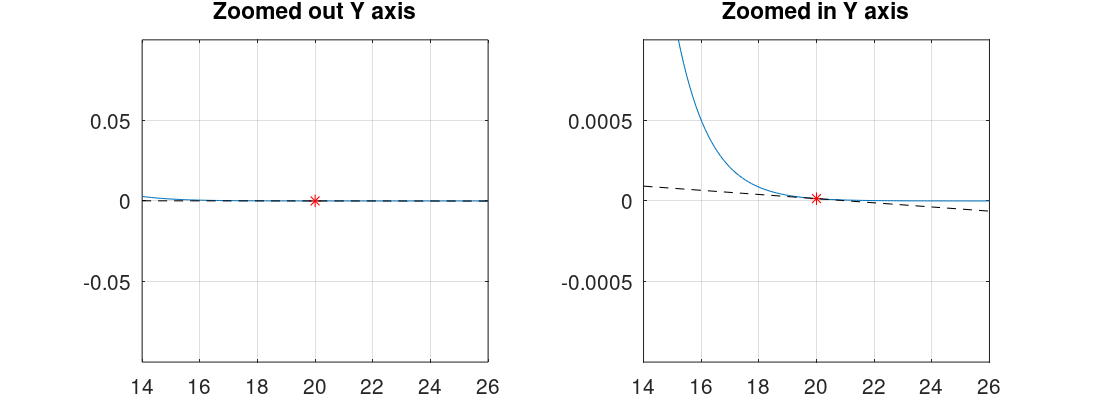
Chart, scatter chart

Description automatically generated

**Figure 1A, 1B, & 1C: Three different snapshots of the same equation but at different bounds**

As you can see, this equation is highly nonlinear for the bounds described in 1A, but for the bounds in 1B and 1C, it is very close to a linear system. All 3 plots are of the same equation, but for different bounds of interest. Figure 1B has the same center point as 1A, but is zoomed in to a smaller set of bounds. Figure 1C has the same scale as 1A, but at a different center point. Therefore it is not acceptable to rationalize the linearity and accuracy of an approximation if only given the center point or only the radius of interest. It is best to define bounds of the format [X1, X2]. This idea extrapolates to N dimensional equations of course.

There is another factor to consider, however. Consider Equation 1 under the bounds of [14, 26]; such was Figure 1C. Depending on how we choose to scale the y axis, we can make the graph visually appear more or less linear. This is even though the center point, radius of interest, parent equation, and linearized equation remain unchanged.



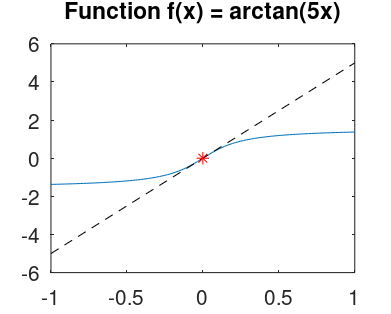
**Figure 2A & 2B: Two different snapshots of the same criteria, only differing in Y axis scale**

What does this mean, and if we can technically do this to any equation why do we accept any approximations? This is where an awareness of your system comes into play. If I am measuring the dimensions of my bedroom and my approximate measurements are off by 1 centimeter, that is not significant for arranging furniture in my room. However, if I am designing a mirror for interferometry or a telescope, a 1 centimeter measurement error is completely unacceptable. There are a couple of ways we can select a proper scale for the y axis. One way is we can mitigate this subjectiveness by converting our measurements from absolute to relative.

An absolute value or measurement is one which is taken relative to some global value, such as the X axis for example. A relative value or measurement is one which is taken relative to itself. Therefor a relative measurement eliminates the relevance of a global Y axis scale because all of the points of interest are taken relative to some other factor, like the center point of interest. In measurement theory, sensor uncertainty can be quantified as absolute or relative. An absolute measurement uncertainty would be something similar to +/- 1 cm error. And that error would be the same for any distance measurement you make. A relative measurement uncertainty would be something similar to +/- 0.5%. This is the possible percent error relative to the “true” measurement. 0.5% = (abs(measured-true)/true). In a similar way we can turn our measurement point from absolute to relative based on the center point of interest.

The method is not without flaw however. Take the lift coefficient as a function of angle of attack of an aircraft for example. It has been shown through potential flow analysis that a good approximation for all streamlined bodies is CL = sin(α) for values near α=0. So 0 degrees alpha is our point of interest. But if we try to use this point of interest we end up dividing by 0. A division by zero error is undefined in mathematics and infinity in MATLAB. Unfortunately using a relative approach for values near zero will almost always lead to bad outputs due to the division by zero. It is more practical in this case to use another factor as our relative point or to simply analyze the absolute error. We will revisit this phenomenon in our later derivation. From this, however we can also ascertain that the scale of our function outputs with respect to our system accuracy requirements also plays a role in the acceptability of our approximations.

Approximation error IS NOT the same thing as nonlinearity. This is a very important distinguishment. The two can be observed to be directly correlated however, there are some rare instances where a Taylor series approximation is awful for a relatively linear function, due to poor luck with which point of interest you linearize around.



**Figure 4: An example of a mostly linear function, which happens to have a poor Taylor series approximation with a lot of error.**

This is not to say that such functions are common, but to distinguish between approximation error and nonlinearity. A highly linear snapshot can have a poor linear approximation, but a highly nonlinear snapshot cannot have a good linear approximation. Quantifying how appropriate an approximation is, is an entirely separate art from quantifying nonlinearity. Luckily there are many alternative algorithms for calculating approximations besides the Taylor series, and as such further discussion of approximations is beyond the scope of this paper, which will focus on nonlinearity quantification.

# Quantifying nonlinearity

It is time to throw some equations at the problem. For the reader’s convenience I will show plots of the form f(x) and f(x,y) since both can be visually perceived well, but we will use derivations with respect to an N-dimensional function, such that the concepts shown for f(x) and f(x,y) can confidently be applied to functions with 3 or more inputs as well.

As mentioned previously, a perfectly linear snapshot of a function is one which maintains a constant slope in all directions. This is a standard line or plane equation, and for N dimensions can take the form of Equation 2 below:

Where:

* Xn – the nth input variable to the function f.
* Cn – A constant coefficient for the nth input variable. Can be any constant real number.
* B – Bias constant. Can be any constant real number.

Any equation which cannot be described in this manner is nonlinear in some way. Something unique to equation 2 is that if you calculate the gradient vector field of this equation, it is a uniform vector field.

As a quick refresher, a scalar field is a plot of scalar values. Scalar fields come from a function which accepts some kind of input (scalar, vector, tensor, phasor, etc.) and outputs a scalar value. A vector field on the other hand gives a vector as an output. Vector fields are useful for displaying the gradient of a function. A function gradient takes a scalar field and calculates a vector field, comprised of vectors pointing towards the “steepest” direction of change. In other words, it produces a vector that can tell you the derivatives in all axes. See the example below in Figure 5.

Chart

Description automatically generated

**Figure 5: Nonlinear gradient example versus linear gradient example**

Like I mentioned before, something unique about linear functions is that they have a uniform vector field. This is because no matter where on the plot you are, the slope would be constant, and therefore the gradient is also consistent at all points. Furthermore, if the vector field is uniform then the calculation of the vector derivative should equal 0. In other words, if the derivative describes a change in a vector as you analyze nearby local points, and if there is no change in the vector field because it is uniform, then naturally, there is 0 change, and the vector derivative is 0. Let’s describe these two conjectures mathematically.

As we will see, the vector derivative of all gradient terms will be a very recurring and very important array. Therefore, I have chosen to provide a layer of abstraction, and label it as vector field U. U can be represented as a matrix, in the fashion shown above, or as a 1-dimensional array. The way you represent the U vector field will become relevant when we convert this data from a vector field, back into a scalar field. Representing U as a matrix is akin to a Jacobian, but I don’t know anything about Jacobians really, so I’m not sure how to take full advantage of that fact yet…

Notice that when taking the vector derivative of a gradient, you can only do so with respect to one variable at a time. Therefore, for a function with n number of input variables, you must take n vector derivatives to capture the entire description of the vector field. If each individual vector derivative results in n number of equations to describe the resulting vector field, then after calculating all n vector derivatives with respect to all n input variables, you will have n2 results. This operation insinuates at least an O(n2) algorithm, which is typically considered highly undesirable. Luckily, due to Clairot’s theorem, we can simplify the above matrix as symmetric. Thus, the operation will require considerably less then O(n2) operations. Additionally most equations representing physical systems will not be absurdly high in order. Even a 20th order system would only be 400 equations if using an n2 approach. So when deriving a numerical approach it will be more important to optimize the constituent operations behind derivatives, integrals, and memory usage; than this symmetric matrix simplification. With that said, this simple fact can save a ton of time if you are calculating the result by hand, analytically.

You can analytically derive U for an equation, and the result will be a matrix of equations. If U is a matrix of zeros, then the equation is linear. If the equation is nonlinear, then the U matrix will contain a collective set of equations to calculate U for any point in the function domain. The U value for a nonlinear function varies depending on where in the function domain you calculate (i.e. what bounds you set for) U. The matrix of values you calculate for a nonlinear function is a direct representation of how nonlinear the function is at that point. The best way I’ve found thus far to represent nonlinearity is by taking the magnitude-squared of all points in the dedicated boundaries and integrating them to get a “sum of contributed nonlinearities”.

This gives us an absolute reference of how nonlinear a function is, and one can convert it to relative by dividing the absolute value by some measure of the overall output value variance, such as for example…

Revisiting Figure 2, we can see how taking the absolute value relative to the maximum and minimum function value described by the bounds solves our issue of how to scale the output axis in a visual approach, regardless of how close to 0 our point of interest is.

# Programming a 3-dimensional approach

The realm of R3 is the highest dimension we can visibly perceive using computer generated imagery and is a good basis for experimenting with an algorithm to execute this equation. Below is some MATLAB/Octave code I wrote for an exclusively R3 domain and range. Note that the algorithm takes advantage of very basic numerical methods for integration and derivation, and has not been optimized to high end, nor does it need optimization. The realm of R3 has very few required operations, and if any improvement were to be made, it would stand to be in accuracy, not runtime.

%nonlinearity measurement of N-dimensional equations

clear all

close all

clc

%define inputs and equation

f = @(x,y) x.^4+y.^4+0.1;%x.\*y.\*y+6\*x-y.\*sqrt(x);

%umag2 = @(x,y) (y.\*y./x./x./x/16)+(8\*y.\*y)+(-4\*y./sqrt(x))+(1/2./x)+(4\*x.\*x);

p = [0,0];

rx = 1;

ry = 1;

x1 = p(1)-rx;

x2 = p(1)+rx;

y1 = p(2)-ry;

y2 = p(2)+ry;

dx = 0.05;

dy = 0.05;

%step 1, Get points in the range as described

x\_arr = x1:dx:x2;

y\_arr = y1:dy:y2;

%[xx,yy] = meshgrid(x\_arr,y\_arr);

%zz = f(xx,yy);

%dzz = umag2(xx,yy);

%surf(xx,yy,dzz)

%step 2, Do the partial derivatives vector for all points

n = 1;

for ii = 2:(numel(x\_arr)-1)

for jj = 2:(numel(y\_arr)-1)

%dxdx

f1 = f(x\_arr(ii+1),y\_arr(jj));

f2 = 2\*f(x\_arr(ii),y\_arr(jj));

f3 = f(x\_arr(ii-1),y\_arr(jj));

dxdx = (f1-f2+f3)/(dx\*dx);

%dxdy

f1 = f(x\_arr(ii+1),y\_arr(jj+1));

f2 = -f(x\_arr(ii+1),y\_arr(jj-1));

f3 = -f(x\_arr(ii-1),y\_arr(jj+1));

f4 = f(x\_arr(ii-1),y\_arr(jj-1));

dxdy = (f1+f2+f3+f4)/(4\*dx\*dy);

%dydx

dydx = dxdy;

%dydy

f1 = f(x\_arr(ii),y\_arr(jj+1));

f2 = 2\*f(x\_arr(ii),y\_arr(jj));

f3 = f(x\_arr(ii),y\_arr(jj-1));

dydy = (f1-f2+f3)/(dy\*dy);

dxdx\_mat(ii-1,jj-1) = dxdx;

dxdy\_mat(ii-1,jj-1) = dxdy;

dydx\_mat(ii-1,jj-1) = dydx;

dydy\_mat(ii-1,jj-1) = dydy;

u\_mag\_mat = [dxdx,dxdy;dydx,dydy];

[V,L] = eig(u\_mag\_mat);

eig\_mag(jj-1,ii-1) = (max(abs(L(:,1)))).^2+(max(abs(L(:,1)))).^2;

n = n+1;

u\_mag(jj-1,ii-1) = abs(dxdx)^2+abs(dxdy)^2+abs(dydx)^2+abs(dydy)^2;

endfor

endfor

%u\_mag\_mat = [dxdx\_mat,dxdy\_mat;

% dydx\_mat,dydy\_mat];

sum(eig\_mag)

hold on

[xxx,yyy] = meshgrid(x\_arr(2:end-1),y\_arr(2:end-1));

surf(xxx,yyy,u\_mag)

surf(xxx,yyy,(eig\_mag))

for ii = 1:(numel(x\_arr)-2)

%integrate over the Y axis before the x axis

u\_mag\_focus = u\_mag(:,ii);

u\_mag\_int(ii) = trapz(y\_arr(2:end-1),u\_mag\_focus);

endfor

%integrate over x axis

integrand = trapz(x\_arr(2:end-1),u\_mag\_int);

f0 = f(p(1),p(2));

eta = integrand/f0/f0

# Programming an N-dimensional approach

For such a nonlinearity calculator to be truly useful we need a computer program which can run any N-dimensional function for all possible bounds with some specifiable resolution.

# Standardizing acceptance

Blah blah blah