NewtonSlides.mws

Slide Show of Newton's Method

Author: Carl Devore <devore@math.udel.edu>

15 April 2001

The author, Carl Devore, can be reached by email at <devore@math.udel.edu>, by paper mail at 655 E3 Lehigh Rd, Newark DE 19711 USA,
or by telephone at (302) 831-3176 or (302) 738-2606.

If you use this program, please send me a brief email, either to discuss it, or just to say "hi". I welcome any suggestions for improvement, and I will consider all suggestions for new programs.

Introduction

Newton's method (also known as the Newton-Raphson method) is a method for finding the roots or zeros of differentiable functions. You will find a discussion of the method in almost any calculus textbook, so I won't describe it in algebraic detail here. The purpose of this worksheet is to show *graphically* exactly what happens when Newton's method is applied to single-variable functions. Maple's animation commands are ideal for this, so this worksheet will help you understand Newton's method far better than any textbook can.

Given a curve y = f(x), we wish to find a solution to f(x) = 0. We start with a guess for the value of x. Call the guess xn. Graphically, the steps of Newton's method are

1. Is f(xn) reasonably close to 0?

If yes, then we are done.

Otherwise:

2. Draw the tangent line to the curve at xn.

3. Find the x-intercept of this tangent line.

4. Set xn equal to the x-intercept.

5. Go to step 1.

Note that several things can go wrong:

1. The tangent line might be horizontal and thus not have an x-intercept.

2. If the tangent line is close to horizontal, its x-intercept will be so far from the original guess that it is probably not worth continuing.

3. The values of f(xn) might not get closer to 0, or they may get closer to zero very slowly.

If so many things can go wrong, why use this method? The reason is that if the initial guess is chosen correctly, the method will almost always converge to the root extremely fast. Usually, the number of digits of accuracy of the root will double on each iteration. That's very fast. Convergence at that speed is called *quadratic* convergence. The animation procedure below attempts to detect the above problems, and advises you on how to avoid them. To do this, we need two more input parameters: We need a maximum for number of allowed iterations and an allowed range for the x-intercepts.

The comments in this worksheet are directed to the calculus student as one of their first exposures to Maple and to Newton's method; thus they are very detailed. But try running the procedure for several examples before attempting to read through it.

To get the best view, you should maximize both your Maple window and the worksheet window within the Maple window. Also consider directing your plot output to a separate window (but the advice will still print in the primary window). If the animations apperar to "shake", there are two ways to correct this: Make the plots larger, or make the fonts smaller. The plots can be made larger by sending them to a separate plot window or by clicking on the magnifying glasses on the toolbar.

It is usually a good idea to put a restart command at the beginning of a worksheet. This clears the values of all variables.

```
> restart;
```

This command sends me an email if this worksheet is run on a Unix computer

```
> if kernelopts(platform) = "unix" then system(`echo 'NewtonSlides' | mailx -sNewtonSlides devore@math.udel.edu`) fi:
```

This procedure converts a univariate expression into a procedure. If it is already a procedure, nothing is done.

```
> makefunc:= proc(f :: {algebraic,procedure})
local x;
if f::procedure then
f
else
x:= remove(type, indets(f, name), realcons);
if nops(x) <> 1 then
ERROR(`More than 1 indeterminate in `, f)
else
codegen[makeproc](f, x[1])
fi
fi
end:
```

```
> Newt_slides:= proc(F :: {algebraic,procedure}
,x0 :: realcons
,xrange :: range(realcons)
,nsteps :: posint
)
# F is the function. It may be expressed as a function or as an expression.
#
# x0 is the starting x value for Newton's method.
#
# xrange is the range of x values for the plotting window.
# It is specified as a..b.
#
# nsteps is the max number of iterations of Newton's method to allow.

local
f # copy of F
,a, b # endpoints of the xrange
,ylow, yhigh # endpoints of the yrange
,n # number of Newton iteration we are currently on
,x # dummy function variable
,xn # nth Newton approximation to the root
,yn # f(xn)
,`f'` # derivative of f
,`f'(xn)` # `f'`(xn)
,T # Tangent line as a function
,txtx, txty # position where text is printed
,plotopts # user-supplied plot-options

#various pieces of the plot
,Curve #The graph of the function itself
,ToCurve #Line segment from x-axis to curve
,TanLine #Graph of the tangent line
,Xmarks #An "X" printed on the tangency point on the Curve
,XAxis #Simply a plot of the Xaxis itself
,Frame #table of animation frames
;

# op (short for operand) is a low-level command for taking things
# apart. In this case I am using it to extract the 1st and 2nd parts
# of the x-range.
a:= evalf(op(1,xrange));
b:= evalf(op(2,xrange));

if evalf(x0) <= a or evalf(x0) >= b then
```

```
ERROR(`Initial guess is not in the range.`)
fi;

# Convert optional arguments to internal form
plotopts:= `plot/options2d`(args[5..-1]);

f:= makefunc(F);
plot(f, xrange);
# Put the color spec, if any, inside the CURVES
Curve:= CURVES(op(op(1,%)), op(select(x->op(0,x)=COLOUR or op(0,x)=COLOR, %)));

# In this case, I am using op to strip the [] off of a list.
# This is required for the min and max commands.
yn:= op(remove(`=`, map(P->P[2], map(op, [op(select(type, Curve, listlist))])), FAIL));

# Get the y-range. I multiply by 1.1 to leave some margin.
yhigh:= 1.1*max(yn);
ylow:= 1.1*min(yn);

# Check if the plot "appears" to cross the x-axis.
if ylow*yhigh > 0 then
#"cat" conCATenates strings.
print
(cat
(`The function does not appear to cross the x-axis in`
,` this range.`
,` I will check for a tangential root anyway.`
)
);

# Expand the y-range to include the x-axis.
if ylow > 0 then
ylow:= -.1*yhigh
elif yhigh < 0 then
yhigh:= -.1*ylow
fi
fi;

# Compute coordinates of a point in the upper left of the plot.
txtx:= .9*a+.1*b;
txty:= .9*yhigh+.1*ylow;

# Don't want to print text too close to the x axis.
if abs(txty)/(yhigh-ylow) < .1 then txty:= .8*yhigh+.2*ylow fi;
```

```
# Plot the x-axis in black (RGB,0,0,0) as a line segment.
XAxis:= CURVES([[a,0], [b,0]], COLOR(RGB,0,0,0));

# Need to initialize the TanLine variable for the first frame.
# Although the x-axis is probably not a tangent line, it is going to
# be plotted anyway, so there's no harm in initializing TanLine this way.
TanLine:= XAxis;

# Now we begin the Newton's method part of the program.

# Get the derivative of f as a function.
`f`:= unapply(diff(f(x), x), x);

# Initialize the Newton sequence.
xn:= evalf(x0);

# This for loop is executed once for each Newton iteration. It is not
# necessarily executed nsteps times. The "break" statements might
# cause the loop to end early.

for n to nsteps do

# Evaluate the function at xn. The evalf makes f(xn) into an
# approximate decimal value. This prevents the expressions from
# getting too complicated.
yn:= evalf(f(xn));

# Plot a blue (RGB,0,0,1), dashed LINESTYLE(3) line segment between the x-axis
# and the curve. Note that a line segment can be plotted by
# specifying its endpoints.
ToCurve:= CURVES([[xn,0], [xn,yn]], COLOR(RGB,0,0,1), LINESTYLE(3));

# Plot an "X marks the spot" on the curve.
Xmarks:= TEXT([xn,yn], 'X');

# There are 2 animation frames plotted for each complete Newton iteration.
# The 1st tells and plots the coordinates of the tangent point. The second
# tells and plots the equation of the tangent line.
Frame[2*n-1]:=
[TanLine
,ToCurve
,Xmarks
# Print coordinates of (xn,yn)
,TEXT([txtx,txty]
```

```
,cat(`Tangent point is (`
# The second parameter to evalf is the number of digits
# to print. The computations are still done to
# "Digits" precision.
,convert(evalf(xn,4), name)
,`,`
,convert(evalf(yn,4), name)
,`).`
)
,ALIGNRIGHT
)
];

# Check for convergence. The fnormal command will convert a number
# to zero if it is "close to" zero relative to the setting of
# Digits.
if fnormal(yn) = 0 then
print
(`(Approximate) root x`[n]=
cat(convert(evalf(xn, Digits-2), symbol)
,` found. For greater accuracy, increase Digits.`
)
);
# In this case of successful convergence, I duplicate the
# 2nd-to-last frame because I am not interested in seeing the
# next tangent line.
Frame[2*n]:= Frame[2*n-1];

# The break statement means to jump out of the for loop
break
fi;

# Generate the equation of the tangent line in functional
# form and then plot it in green (RGB,0,1,0). Note how the derivative is
# evaluated at the point xn. Note how unapply is used to
# make an expression into a function.
`f'(xn)`:= evalf(`f'`(xn));
T:= unapply(yn+`f'(xn)`*(x-xn), x);
TanLine:= CURVES([[a,T(a)], [b,T(b)]], COLOR(RGB,0,1,0));

#Print the frame with the equation of the tangent line.
Frame[2*n]:=
[TanLine
,ToCurve
```

```
,Xmarks
,TEXT([txtx,txty], cat(`Tangent is y = `, convert(evalf(T(x), 4), name), `.`), ALIGNRIGHT)
];

# Check for a "wild" tangent
if fnormal(`f'(xn)`) = 0 then
print
(cat(`Tangent line is (almost) horizontal. Cannot continue.`
,` Try a different starting value. Play the slide`
,` show anyway.`
)
);
break
fi;

# Do the basic Newton iteration
xn:= xn - yn/`f'(xn)`;

# Check if we've left the allowed xrange.
if xn < a or xn > b then
print
(`x`[n]=
cat(convert(evalf(xn,4),symbol)
,` is outside the specified range. Try increasing the`
,` range. Play the slide show anyway.`
)
);
break
fi
od;

# Check if we've used up the max allowed number of iterations.
# If the loop ended because the max number of iterations was used,
# then n=nsteps+1. This is a quirk of Maple's for loops.
if n = nsteps+1 then
print
(`x`[n-1]= convert(evalf(xn,Digits-2), symbol)
,'f'(`x`[n-1])=
cat(convert(yn,symbol)
,`. No root (precisely) found.`
,` Try increasing the number of iterations. Play the`
,` slide show anyway.`
)
)
```

```
fi;

# Count the number of frames. Note that if the loop ended because the
# max number of iterations was used, then n=nsteps+1.
n:= min(2*n, 2*nsteps-1);

# plotopts will be everything that is added to every frame (the "background")
plotopts:= Curve, XAxis, VIEW(a..b, ylow..yhigh), AXESSTYLE(FRAME), plotopts;

# Return the animated plot structure
PLOT
(ANIMATE
(# Add "First frame" and "Last frame" messages to the 1st and last frames.
[op(Frame[1]), TEXT([.3*a+.7*b, .7*ylow+.3*yhigh], `First frame`, ALIGNRIGHT), plotopts]

# All the middle frames.
,seq([op(Frame[k]), plotopts], k= 2..n-1)

# The last frame.
,[op(Frame[n]), TEXT([.3*a+.7*b, .7*ylow+.3*yhigh], `Last frame`, ALIGNRIGHT), plotopts]
)
)
end:
```

To play the slide show, execute the Newt_slides command. When a picture appears, left-click on it. A new tool bar will appear on your screen. This new toolbar has icons that look like the icons on a VCR or audio CD player. One of the icons will play the show frame by frame. It is the 3rd icon from the left. Click this once for each slide. To replay the show, click on the backwards-direction icon (4th icon from the left), then click on the play icon (2nd from the left) to replay the whole show backwards, then click on the forward-direction icon (5th from the left), and you will be back where you started. All of these controls can also be accessed by RIGHT-clicking on the graph and using the pop-up animation menu.

Note that the *y* -axis has been placed on the left side so that it does not clutter up the rest of the picture. Note that small numbers are displayed in scientific notation. For example .6589e-1 means .6589 x 10^(-1) or .06589. This has nothing to do with the number *e* .

The Newt_slides command has 4 arguments. The first is the function. It may be in the form of a function or an expression. The second is the initial root guess x_0 .

The third is the x-range. It must be expressed in the form a..b. The fourth is the maximum number of iterations. You can also add any extra arguments that are valid options to a **display** command, for example, **scaling= constrained** or **axes= normal** or **font= [HELVETICA, 12].** .

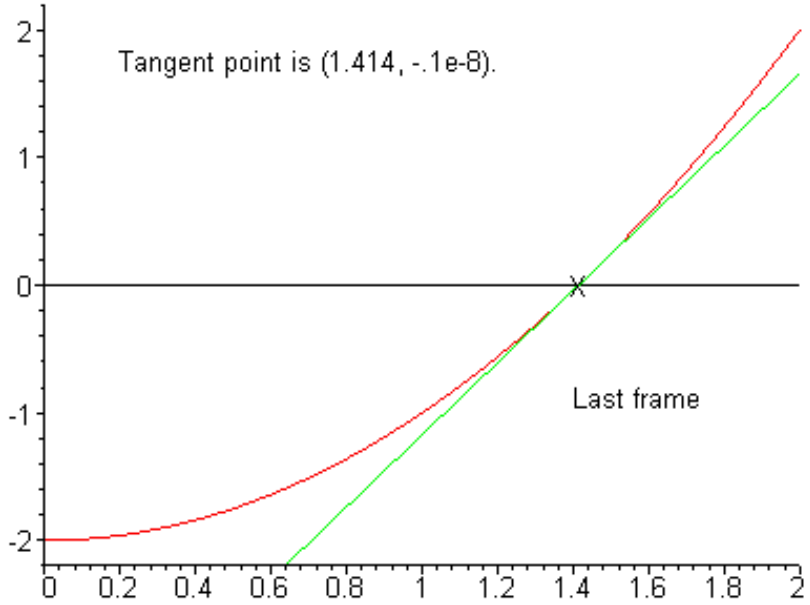
Usually the first example of using Newton's method is find the square root of 2. To apply the method, we need a function that has a root at $x = \sqrt{2}$. So what happens if we let $f(x) = x - \sqrt{2}$? The problem with using that function is that to compute its value we need to already know $\sqrt{2}$. What we need is a

computable function that has a root at $x = \sqrt{2}$. The function to use is $f(x) = x^2 - 2$.

Let's use an inital guess of 1. Obviously, the answer is in the range [0,2]. I will allow a maximum of 20 iterations.

> **Newt_slides(x^2-2, 1, 0..2, 20);**

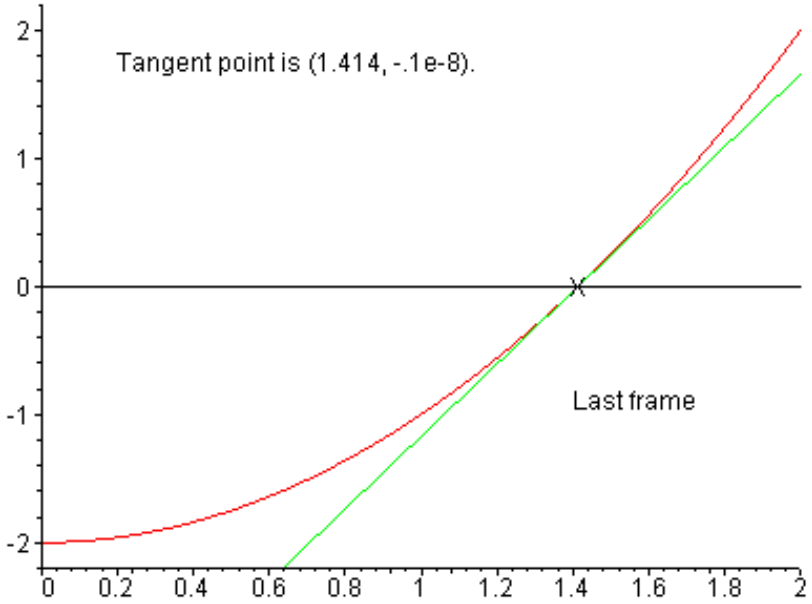
(Approximate) root $x_5 = 1.4142136$ found. For greater accuracy, increase Digits.



If that font is not a good size for your monitor, try something like this

> **Newt_slides(x^2-2, 1, 0..2, 20, font= [HELVETICA, 9]);**

(Approximate) root $x_5 = 1.4142136$ found. For greater accuracy, increase Digits.



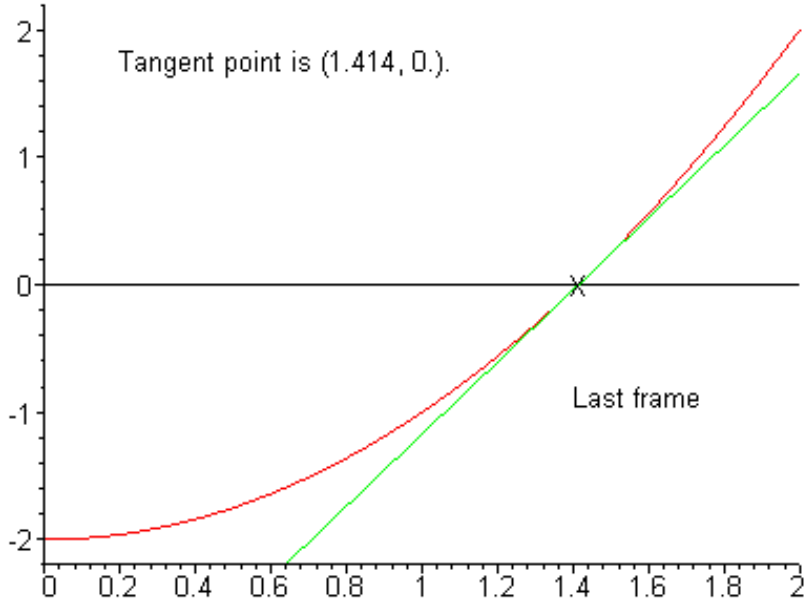
Now you try changing the parameters (the function, the initial point, the range of x-values, the maximum number of iterations, or the value of **Digits**). Be sure to try a case where the x-axis is itself a tangent line to the curve. Newton's method converges more slowly in that case. Be sure to also try a case where the derivative is zero (the tangent line is horizontal) at your initial guess.

> **Digits:= 20;**

$$Digits := 20$$

> **Newt_slides(x^2-2, 1, 0..2, 20);**

$$\text{(Approximate) root } x_6 = 1.41421356237309505 \text{ found. For greater accuracy, increase Digits.}$$

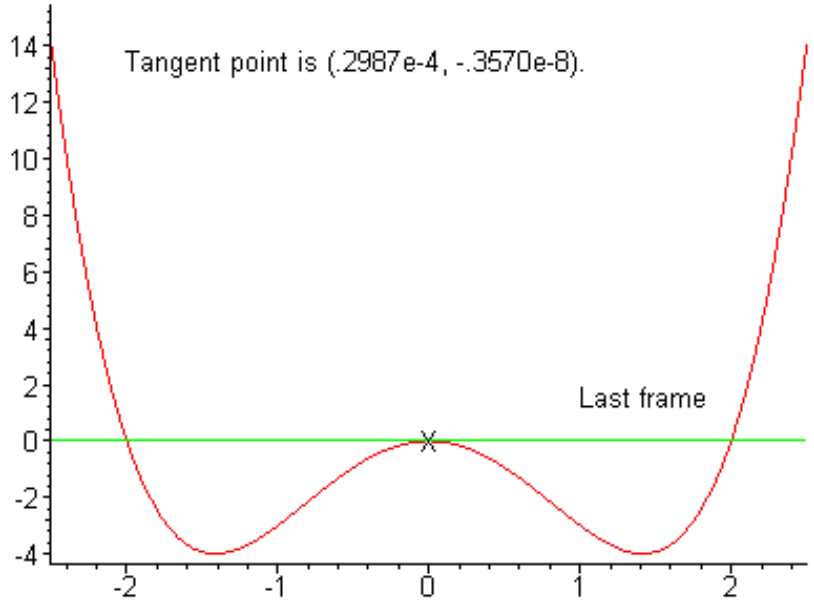


Let's try one where the curve is tangent to the x-axis. For the function below, it is obvious that the roots are -2, 0, and 2, and that the root at zero in tangential.

> **Digits:= 10:**

> **Newt_slides(x^2*(x^2-4), 1, -2.5..2.5, 20);**

(Approximate) root $x_{15} = .29873519e\text{-}4$ *found. For greater accuracy, increase Digits.*

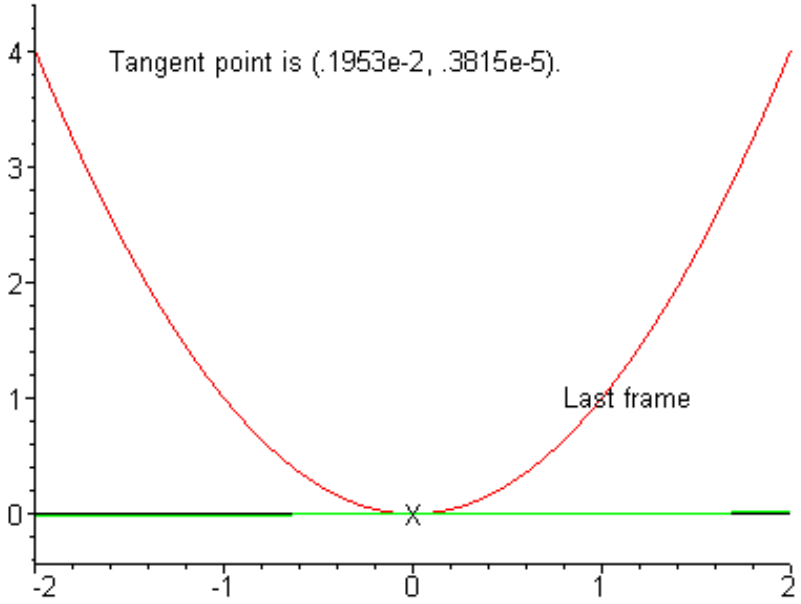


Note that it takes many more steps to find the root. Let's try another like that.

> **Newt_slides(x^2, 1, -2..2, 10);**

The function does not appear to cross the x-axis in this range. I will check for a tangential root anyway.

$x_{10} = .97656250e\text{-}3, f(x_{10}) = .3814697266e\text{-}5$. *No root (precisely) found. Try increasing the number of iterations. Play the slide show anyway.*

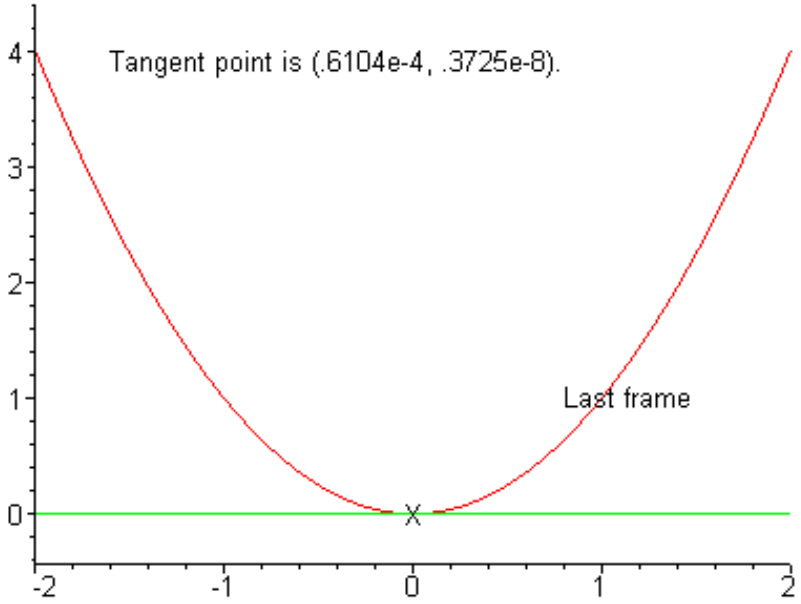


Let's take the avice.

> **Newt_slides(x^2, 1, -2..2, 20);**

The function does not appear to cross the x-axis in this range. I will check for a tangential root anyway.

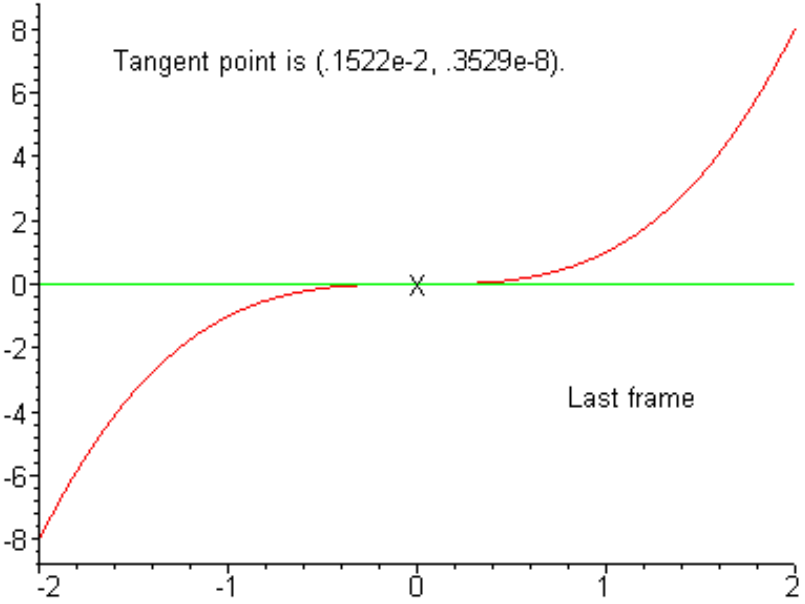
(Approximate) root $x_{15} = .61035156e\text{-}4$ *found. For greater accuracy, increase Digits.*



Suppose that function does cross the x-axis, but crosses it tangentially?

> **Newt_slides(x^3, 1, -2..2, 20);**

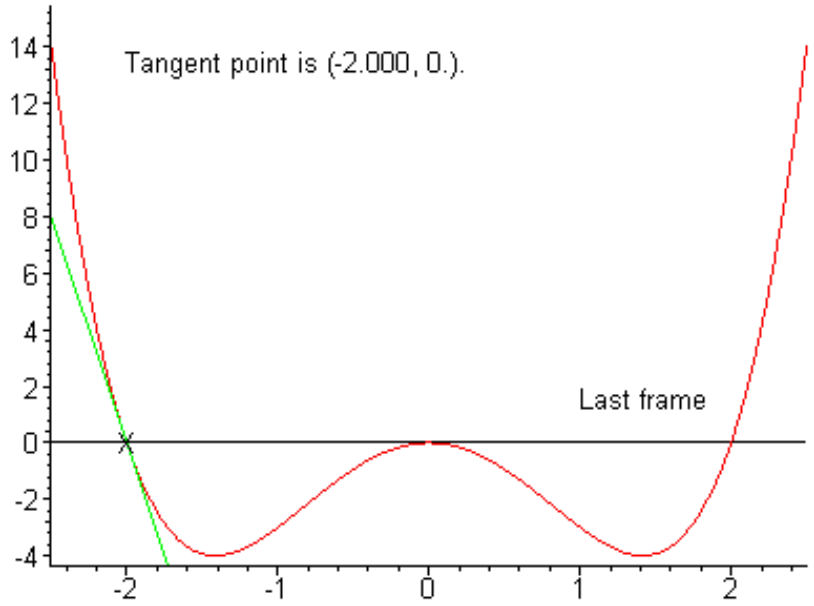
(Approximate) root $x_{17} = .15224388e\text{-}2$ *found. For greater accuracy, increase Digits.*



In the following example, we have convergence to a root other than the expected one.

> **Newt_slides(x^2*(x^2-4), 1.33, -2.5..2.5, 20);**

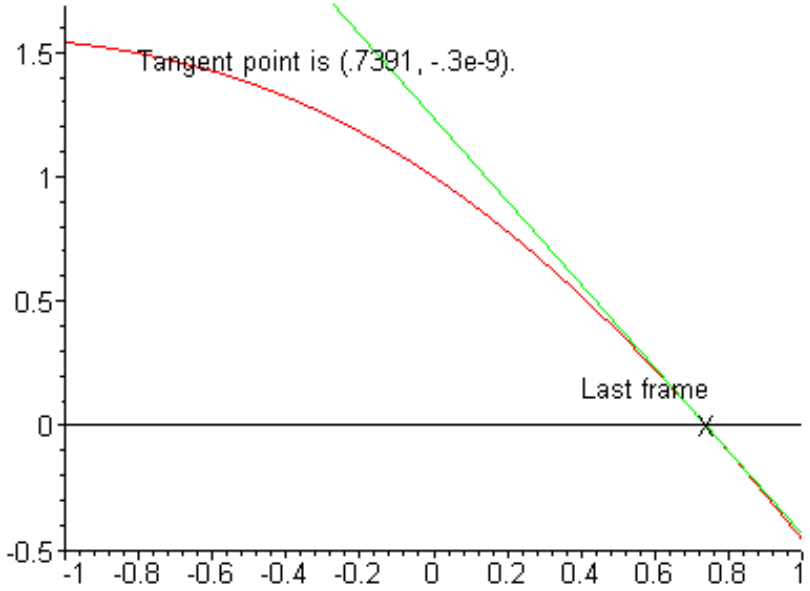
(Approximate) root $x_6 = -2.0000000$ *found. For greater accuracy, increase Digits.*



Let's try to solve an equation that can't be solved with algebra, cos(x) = x. This is equivalent to finding a root of $f(x) = \cos(x) - x$. Obviously the root, if it exists at all, is between -1 and 1.

> **Newt_slides(cos(x) - x, 0, -1..1, 10);**

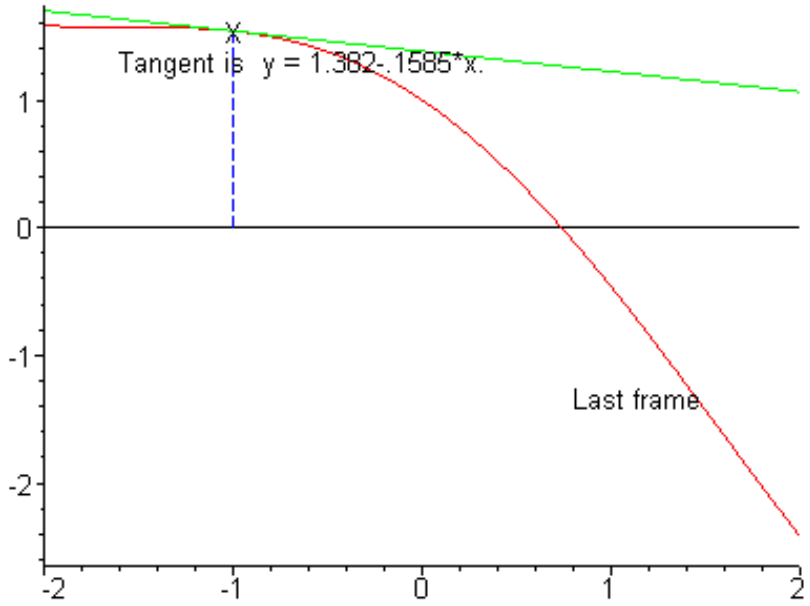
(Approximate) root $x_5 = .73908513$ *found. For greater accuracy, increase Digits.*



Let's try to make a bad guess.

> **Newt_slides(cos(x)-x, -1, -2..2, 10);**

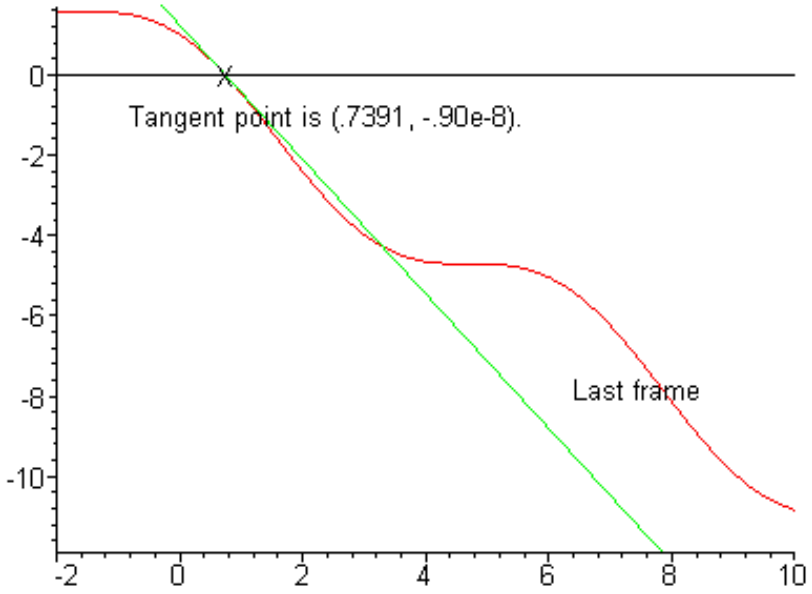
x_1 = 8.716 is outside the specified range. Try increasing the range. Play the slide show anyway.



Let's take the advice and increase the range.

> **Newt_slides(cos(x)-x, -1, -2..10, 10);**

(Approximate) root $x_8 = .73908514$ *found. For greater accuracy, increase Digits.*



>