lab1ans

Calculus A, Lab 1 Solutions

Tim McLarnan

I'll do all these problems with *Sage* just so I end up with a single document in the end. It would be perfectly fine to use whatever tools you like for the project, and my expectation is that most of you probably used the Apple *Grapher*.

Problem 1

Two equivalent but grammatically different ways to get 20 digits of π are

```
pi.n(digits=20)
```

3.1415926535897932385

```
n(pi,digits=20)
```

3.1415926535897932385

If we want to compute values if the trig functions, the first thing we have to work out is whether *Sage* uses radians or degrees. One way to find out might be to compute $\cos 1$. If *Sage* uses degrees, then this is the x-coordinate of a point 1° up from the x-axis; so it should be very close to 1. If *Sage* uses radians, then it's the x-coordinate of a point 1 radian up from the x-axis, which should be somewhere around $1/2$.

```
cos(1).n()
```

0.540302305868140

OK, so *Sage* uses radians. In that case, to get 20 digits of $\cos(37^\circ)$ we would say

```
cos(37*pi/180).n(digits=20)
```

0.79863551004729284628

Getting 20 digits of $\sin(1.3 \text{ radians})$ turns out to be trickier than I thought. I thought we would just do this:

```
sin(1.3).n(digits=20)
```

0.96355818541719295833

The problem is that the last 4 digits of this answer are wrong. How can that be? Well, it turns out that unless we do something a bit fancy, *Sage* interprets 1.3 as a double-precision floating point number, an approximate quantity with about 15 digits precision. Thus, even before we call **n**, some approximation has taken place. To keep that from happening, the easy thing to do is to replace the 1.3 with $13/10$, which is an exact rational number in *Sage*. Here are the right first 20 digits:

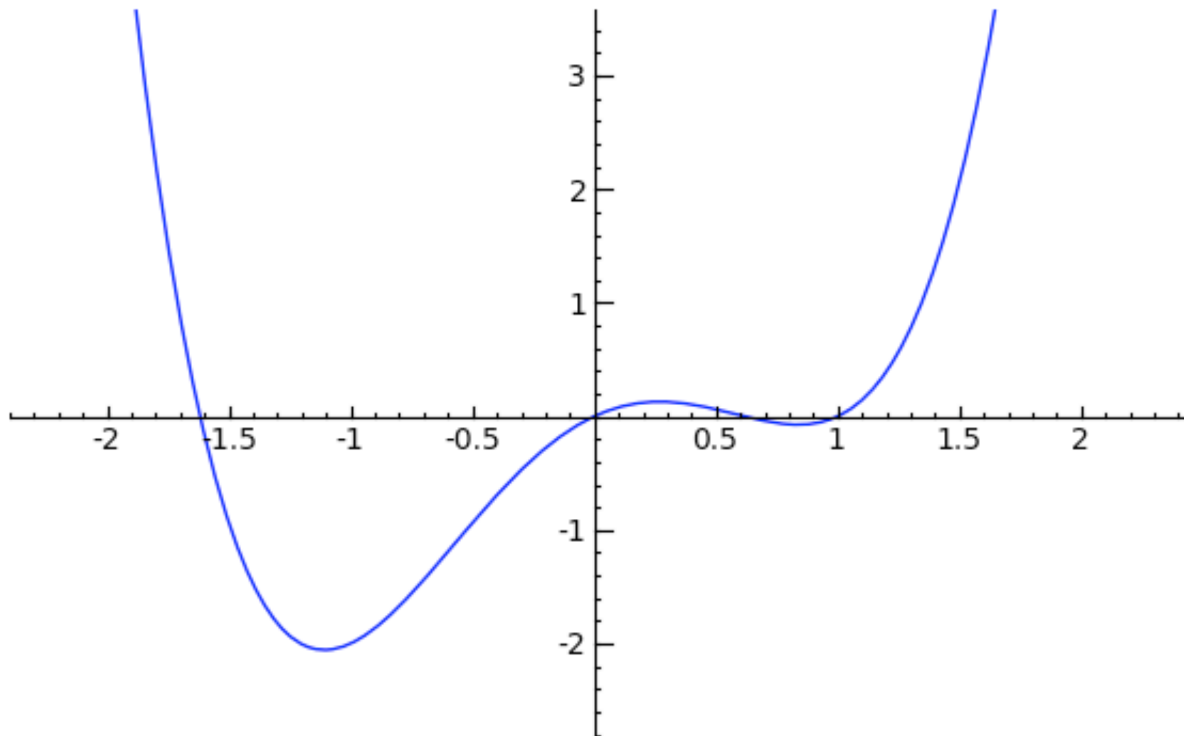
```
sin(13/10).n(digits=20)
```

0.96355818541719296470

Problem 2

Ideally, a plot of $y = x^4 - 2x^2 + x$ should show 2 local minima and a local maximum, and should suggest that the function grows rapidly as $x \to \pm\infty$.

```
show(plot(x^4-2*x^2+x, (x, -2, 2)), ymin=-2.3, ymax=3)
```



Problem 3

An ideal plot of

$$y = x^3 + \frac{1}{(x-1)^2} - \frac{150}{1+(x+3)^4}$$

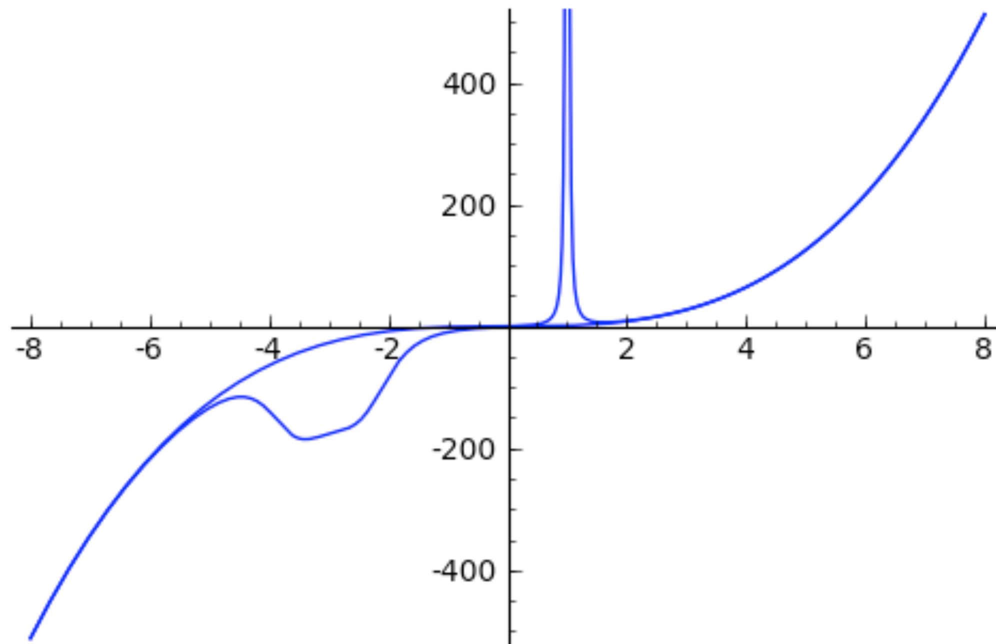
ought to show that when you zoom way out, the function looks basically like $y = x^3$, since the other two terms are small for large $|x|$. It should also show the upward-pointing vertical asymptote at $x = 1$ produced by the second term. Finally, it should show a large downward bulge centered at $x = -3$ produced by the third term. Here's one choice of axes that does all that:

```
x^3+1/(x-1)^2-150/(1+(x+3)^4)
```

$$\frac{1}{(x-1)^2} - 150\,\frac{1}{((x+3)^4+1)} + x^3$$

```
show(plot([x^3, x^3+1/(x-1)^2-150/(1+(x+3)^4)], (x,-8,8)), ymin=-500,
```

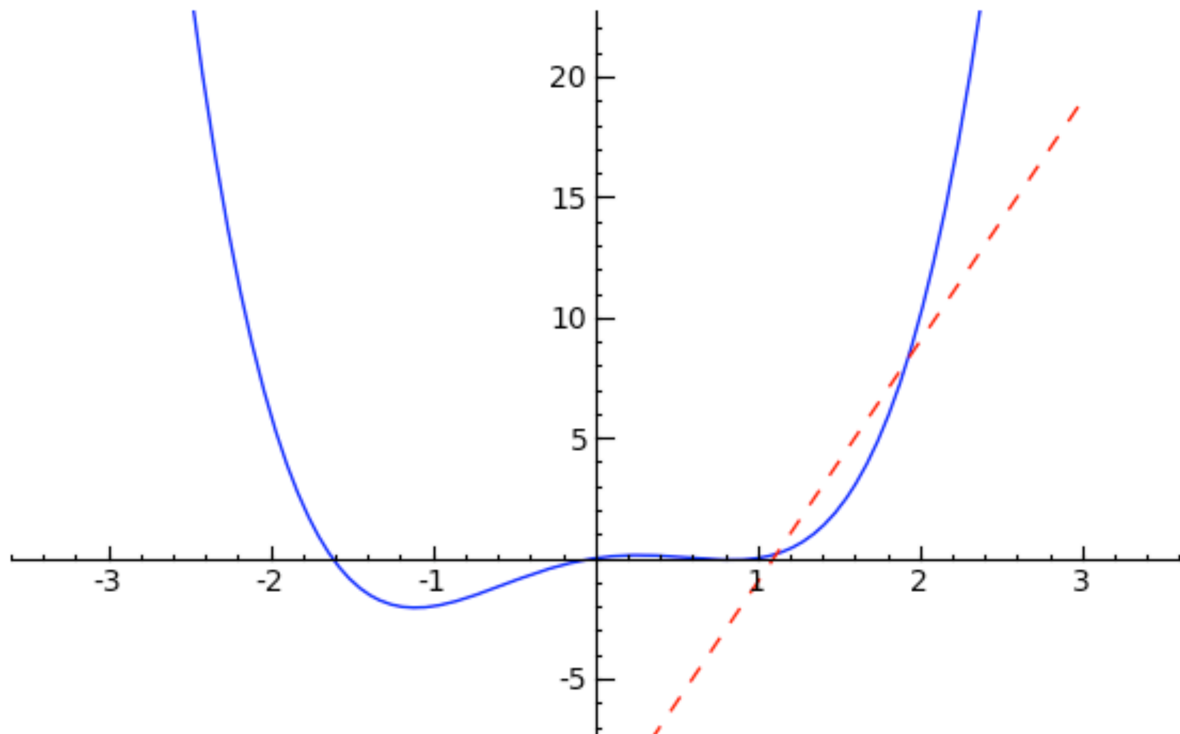
```
ymax=500)
```



Problem 4

The two curves $y = x^4 - 2x^2 + x$ and $y = 10x - 11$ certainly intersect, as shown by this plot:

```
p1 = plot(x^4-2*x^2+x, (x, -3, 3), color='blue', linestyle='-')
p2 = plot(10*x-11, (x, -3, 3), color='red', linestyle='--')
show(p1+p2, ymin=-5, ymax=20)
```



An easy way to get the x-coordinates of the two intersection points would be to use **find_root**

An easy way to get the x-coordinates of the two intersection points would be to use **find_root**.

```
root1 = find_root(x^4-2*x^2+x == 10*x-11, 1, 1.5)
root1
```

1.11805861517

```
root2 = find_root(x^4-2*x^2+x == 10*x-11, 1.5, 2.2)
root2
```

1.92538791456

Then get the y-coordinates by plugging these numbers into either of the two equations.

```
y1 = (10*x-11).substitute(x=root1)
y1
```

0.180586151675005

```
y2 = (10*x-11).substitute(x=root2)
y2
```

8.25387914558771

Here's another clever solution found by Carolina Quezada, who just solved the two equations for y together. I wouldn't have thought to do this, and was surprised when *Sage* came back with numerical solutions in this case.

```
var('x y')
print(solve([y == x^4-2*x^2+x, y == 10*x-11], x, y))
```

```
[
[x == 1.11805859204, y == 0.180586142639],
[x == (-1.52172326486 + 1.67159460967*I), y == (-26.2172326486 +
16.7159460967*I)],
[x == (-1.52172326486 - 1.67159460967*I), y == (-26.2172326486 -
16.7159460967*I)],
[x == 1.9253879168, y == 8.2538787024]
]
```

This solves the problem, but the natural question arises, what if we wanted, say, 30 digit accuracy for these points (x, y)? How would we get that? (Skip this discussion unless you want to read some slightly abstruse mathematics.)

Unfortunately, **find_roots()** doesn't let us specify a number of digits precision for our estimates, and any way we could proceed involves knowing something more about the depths of *Sage*. One approach would be to take advantage of the fact that for polynomials of degree 4 or less, *Sage* can find exact solutions. We therefore get the exact solutions (which take several pages to print out) and approximate them to 3 digit accuracy.

```
solns = solve(x^4-2*x^2+x == 10*x-11,x)
```

```
for r in solns:
    print(r.rhs().n(digits=30))
```

```
-1.52172326486313613620207224088 - 1.67159460966627464203522541779*I
-1.52172326486313613620207224088 + 1.67159460966627464203522541779*I
1.11805861516750037331177730462
1.92538791455877189909236717714
```

What if we were dealing with higher degree polynomials, though, where *Sage* doesn't know an exact solution? In that case, we could construct the ring of polynomials whose coefficients are reals with 100 bits (roughly 30 digits) of precision, and whose variable is called t. We would then form the polynomial $(t^4 - 2t^2$ in this ring and look for its real roots. To get the y-values at these roots, we would then apply the function that takes x to $10x - 11$ to each of the roots. Here's how to say that in *Sage*.

```
R = RealField(100)
print(R)
```

```
Real Field with 100 bits of precision
```

```
PR.<t> = PolynomialRing(R)
print(PR)
```

```
Univariate Polynomial Ring in t over Real Field with 100 bits of
precision
```

```
poly = (t^4 - 2*t^2 + t) - (10*t - 11)
print(poly)
```

```
1.0000000000000000000000000000*t^4 -
2.0000000000000000000000000000*t^2 -
9.0000000000000000000000000000*t + 11.000000000000000000000000000
```

```
root_list = poly.real_roots()
root_list
```

$[1.1180586151675003733117773046, 1.9253879145587718990923671771]$

```
map(lambda x: 10*x-11, root_list)
```

$[0.18058615167500373311777304621, 8.2538791455877189909236717713]$

This approach still relies on the fact that we are working with polynomial equations. What if we wanted to solve equations that involved something like trig functions as well? It seems to me that there ought to be a way to do this within *Sage* itself, but I haven't been able to find it. Some of the other packages like gp and Maxima included in *Sage* do have this functionality, though. Here's how we would ask gp to set its default precision to 30 digits and then to find the root between 1 and 1.5:

```
%gp
\p 30
solve(x=1, 1.5, (x^4 - 2*x^2 + x) - (10*x -11))
```

```
1.11805861516750037331177730462
```

```
(10*x-11).substitute(x=1.11805861516750037331177730462)
```

0.180586151675003733117773046199

```
%gp
solve(x=0, 1, sin(x)-cos(x))
```

0.785398163397448309615660845820

```
(pi/4).n(digits=30)
```

0.785398163397448309615660845820

Problem 5

Primes that are one less than a power of 2 are called *Mersenne primes*. At any given moment in time, the largest known prime is a Mersenne prime. As of this writing, the largest known Mersenne prime is $2^{43,112,609}$. For more information on the quest for these primes, visit http://mersenne.org, the home page of GIMPS, the Great Internet Mersenne Prime Search. There are currently 47 Mersenne primes known. While nobody knows a simple necessary and sufficient condition for $2^n - 1$ to be prime, it's not hard to get a necessary condition: $2^n - 1$ is prime only if n is prime. I hoped you'd be able to guess this condition by looking at the first few Mersenne primes. This could be done rather easily with *Sage*. Here's a list of the exponents up to 100 and whether or not $2^n - 1$ is prime.

```
for n in [2..100]:
    print(n, factor(2^n-1))
```

```
(2, 3)
(3, 7)
(4, 3 * 5)
(5, 31)
(6, 3^2 * 7)
(7, 127)
(8, 3 * 5 * 17)
(9, 7 * 73)
(10, 3 * 11 * 31)
(11, 23 * 89)
(12, 3^2 * 5 * 7 * 13)
(13, 8191)
(14, 3 * 43 * 127)
(15, 7 * 31 * 151)
(16, 3 * 5 * 17 * 257)
(17, 131071)
(18, 3^3 * 7 * 19 * 73)
(19, 524287)
(20, 3 * 5^2 * 11 * 31 * 41)
(21, 7^2 * 127 * 337)
(22, 3 * 23 * 89 * 683)
(23, 47 * 178481)
(24, 3^2 * 5 * 7 * 13 * 17 * 241)
(25, 31 * 601 * 1801)
(26, 3 * 2731 * 8191)
(27, 7 * 73 * 262657)
(28, 3 * 5 * 29 * 43 * 113 * 127)
```

```
(28, 3 * 5 * 29 * 43 * 113 * 127)
(29, 233 * 1103 * 2089)
(30, 3^2 * 7 * 11 * 31 * 151 * 331)
(31, 2147483647)
(32, 3 * 5 * 17 * 257 * 65537)
(33, 7 * 23 * 89 * 599479)
(34, 3 * 43691 * 131071)
(35, 31 * 71 * 127 * 122921)
(36, 3^3 * 5 * 7 * 13 * 19 * 37 * 73 * 109)
(37, 223 * 616318177)
(38, 3 * 174763 * 524287)
(39, 7 * 79 * 8191 * 121369)
(40, 3 * 5^2 * 11 * 17 * 31 * 41 * 61681)
(41, 13367 * 164511353)
(42, 3^2 * 7^2 * 43 * 127 * 337 * 5419)
(43, 431 * 9719 * 2099863)
(44, 3 * 5 * 23 * 89 * 397 * 683 * 2113)
(45, 7 * 31 * 73 * 151 * 631 * 23311)
(46, 3 * 47 * 178481 * 2796203)
(47, 2351 * 4513 * 13264529)
(48, 3^2 * 5 * 7 * 13 * 17 * 97 * 241 * 257 * 673)
(49, 127 * 4432676798593)
(50, 3 * 11 * 31 * 251 * 601 * 1801 * 4051)
(51, 7 * 103 * 2143 * 11119 * 131071)
(52, 3 * 5 * 53 * 157 * 1613 * 2731 * 8191)
(53, 6361 * 69431 * 20394401)
(54, 3^4 * 7 * 19 * 73 * 87211 * 262657)
(55, 23 * 31 * 89 * 881 * 3191 * 201961)
(56, 3 * 5 * 17 * 29 * 43 * 113 * 127 * 15790321)
(57, 7 * 32377 * 524287 * 1212847)
(58, 3 * 59 * 233 * 1103 * 2089 * 3033169)
(59, 179951 * 3203431780337)
(60, 3^2 * 5^2 * 7 * 11 * 13 * 31 * 41 * 61 * 151 * 331 * 1321)
(61, 2305843009213693951)
(62, 3 * 715827883 * 2147483647)
(63, 7^2 * 73 * 127 * 337 * 92737 * 649657)
(64, 3 * 5 * 17 * 257 * 641 * 65537 * 6700417)
(65, 31 * 8191 * 145295143558111)
(66, 3^2 * 7 * 23 * 67 * 89 * 683 * 20857 * 599479)
(67, 193707721 * 761838257287)
(68, 3 * 5 * 137 * 953 * 26317 * 43691 * 131071)
(69, 7 * 47 * 178481 * 10052678938039)
(70, 3 * 11 * 31 * 43 * 71 * 127 * 281 * 86171 * 122921)
(71, 228479 * 48544121 * 212885833)
(72, 3^3 * 5 * 7 * 13 * 17 * 19 * 37 * 73 * 109 * 241 * 433 * 38737)
(73, 439 * 2298041 * 9361973132609)
(74, 3 * 223 * 1777 * 25781083 * 616318177)
(75, 7 * 31 * 151 * 601 * 1801 * 100801 * 10567201)
(76, 3 * 5 * 229 * 457 * 174763 * 524287 * 525313)
(77, 23 * 89 * 127 * 581283643249112959)
(78, 3^2 * 7 * 79 * 2731 * 8191 * 121369 * 22366891)
(79, 2687 * 202029703 * 1113491139767)
```

```
(79, 2687 * 202029703 * 1113491139767)
(80, 3 * 5^2 * 11 * 17 * 31 * 41 * 257 * 61681 * 4278255361)
(81, 7 * 73 * 2593 * 71119 * 262657 * 97685839)
(82, 3 * 83 * 13367 * 164511353 * 8831418697)
(83, 167 * 57912614113275649087721)
(84, 3^2 * 5 * 7^2 * 13 * 29 * 43 * 113 * 127 * 337 * 1429 * 5419 *
14449)
(85, 31 * 131071 * 9520972806333758431)
(86, 3 * 431 * 9719 * 2099863 * 2932031007403)
(87, 7 * 233 * 1103 * 2089 * 4177 * 9857737155463)
(88, 3 * 5 * 17 * 23 * 89 * 353 * 397 * 683 * 2113 * 2931542417)
(89, 618970019642690137449562111)
(90, 3^3 * 7 * 11 * 19 * 31 * 73 * 151 * 331 * 631 * 23311 *
18837001)
(91, 127 * 911 * 8191 * 112901153 * 23140471537)
(92, 3 * 5 * 47 * 277 * 1013 * 1657 * 30269 * 178481 * 2796203)
(93, 7 * 2147483647 * 658812288653553079)
(94, 3 * 283 * 2351 * 4513 * 13264529 * 165768537521)
(95, 31 * 191 * 524287 * 420778751 * 30327152671)
(96, 3^2 * 5 * 7 * 13 * 17 * 97 * 193 * 241 * 257 * 673 * 65537 *
22253377)
(97, 11447 * 13842607235828485645766393)
(98, 3 * 43 * 127 * 4363953127297 * 4432676798593)
(99, 7 * 23 * 73 * 89 * 199 * 153649 * 599479 * 33057806959)
(100, 3 * 5^3 * 11 * 31 * 41 * 101 * 251 * 601 * 1801 * 4051 * 8101
* 268501)
```

And here are all the exponents $n \leq 5000$ for which $2^n - 1$ is a Mersenne prime:

```
for n in [2..5000]:
    if is_prime(2^n-1):
        print(n)
```

```
2
3
5
7
13
17
19
31
61
89
107
127
521
607
1279
2203
2281
3217
4253
4423
```

It's easy to see that the first few of these exponents are primes, and factoring them or using **is_prime()** makes quick work of the rest.

So why does $2^n - 1$ being prime force $n$to be prime; or equivalently, why does n being composite force $2^n -$ to be composite? Well, remember the polynomial factorization

$$x^b - 1 = (x-1)(x^{b-1} + x^{b-2} + b^{b-3} + \cdots + x^2 + x + 1)?$$

(If you don't remember this factorization, it's easy to check by just multiplying it out.) If $n = ab$ is composite, $1 < a \leq b < n$, then this factorization with x replaced by 2^a gives

$$2^{ab} - 1 = (2^a)^b - 1 = (2^a - 1)(2^{ab-a} + 2^{ab-2a} + 2^{ab-3a} + \cdots + 2^{2a} + 2^a + 1).$$

Obviously, neither factor in ths product is 1; so $2^{ab} - 1$ is composite.

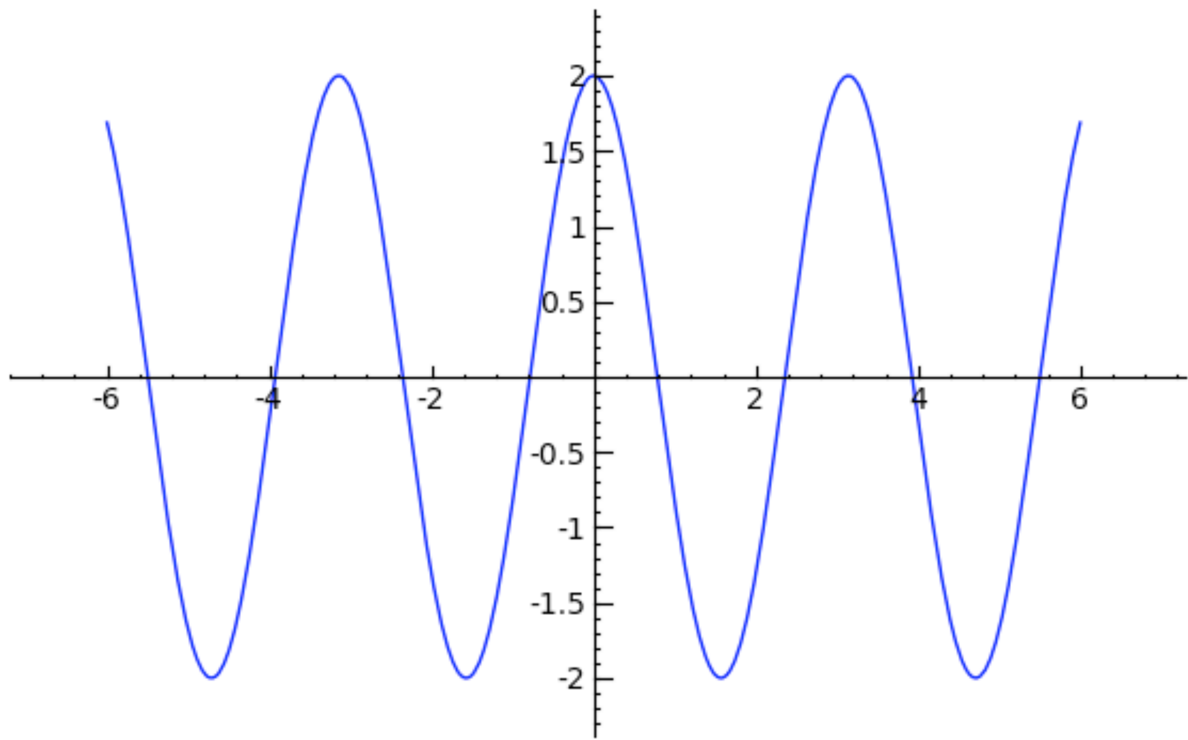
Number theorists would be thrilled if one could determine exactly what exponents yield Mersenne primes, but already this little *Sage* calculation finds 20 of the 47 known Mersenne primes and brings us up to the state of knowledge at the time of Hurwitz' work in 1961.

Problem 6

The game here was mostly to get a rough formula for the function and then to move it and stretch it until it worked out exactly.

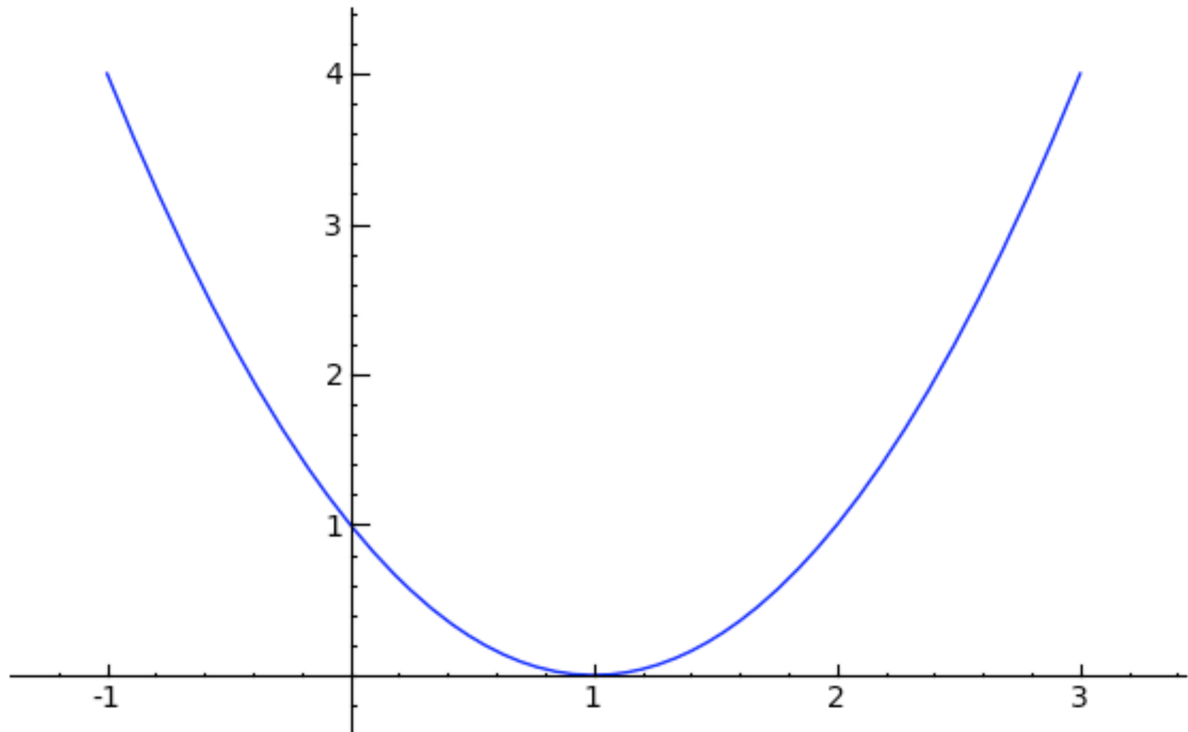
For part (a), the function looks a lot like $y = \cos x$ except that its amplitude is too great by a factor of 2 and it has been squeezed horizontally by a factor of 2. So why not try $y = 2\cos(2x)$:

```
plot(2*cos(2*x), (x, -6, 6))
```



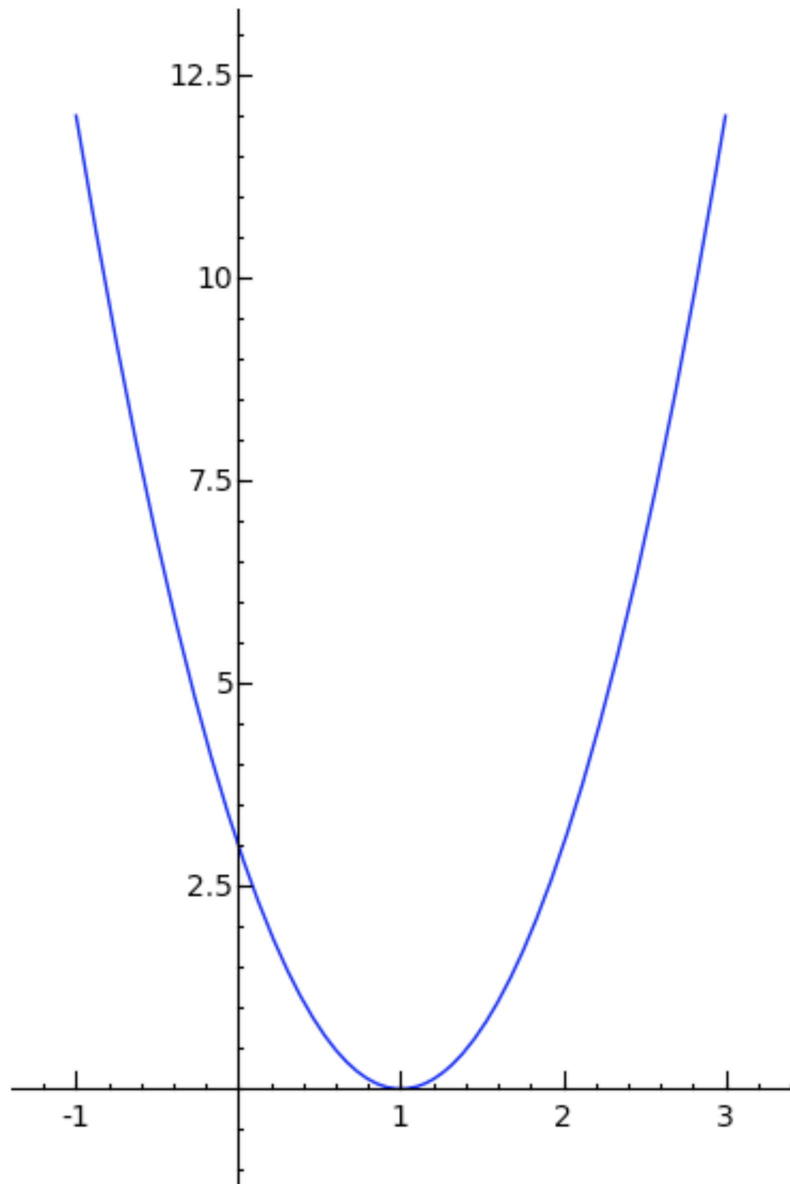
The function in part (b) looks like a parabola moved right by one unit

```
plot((x-1)^2, (x, -1,3))
```



ah, and then stretched vertically by a factor of 3.

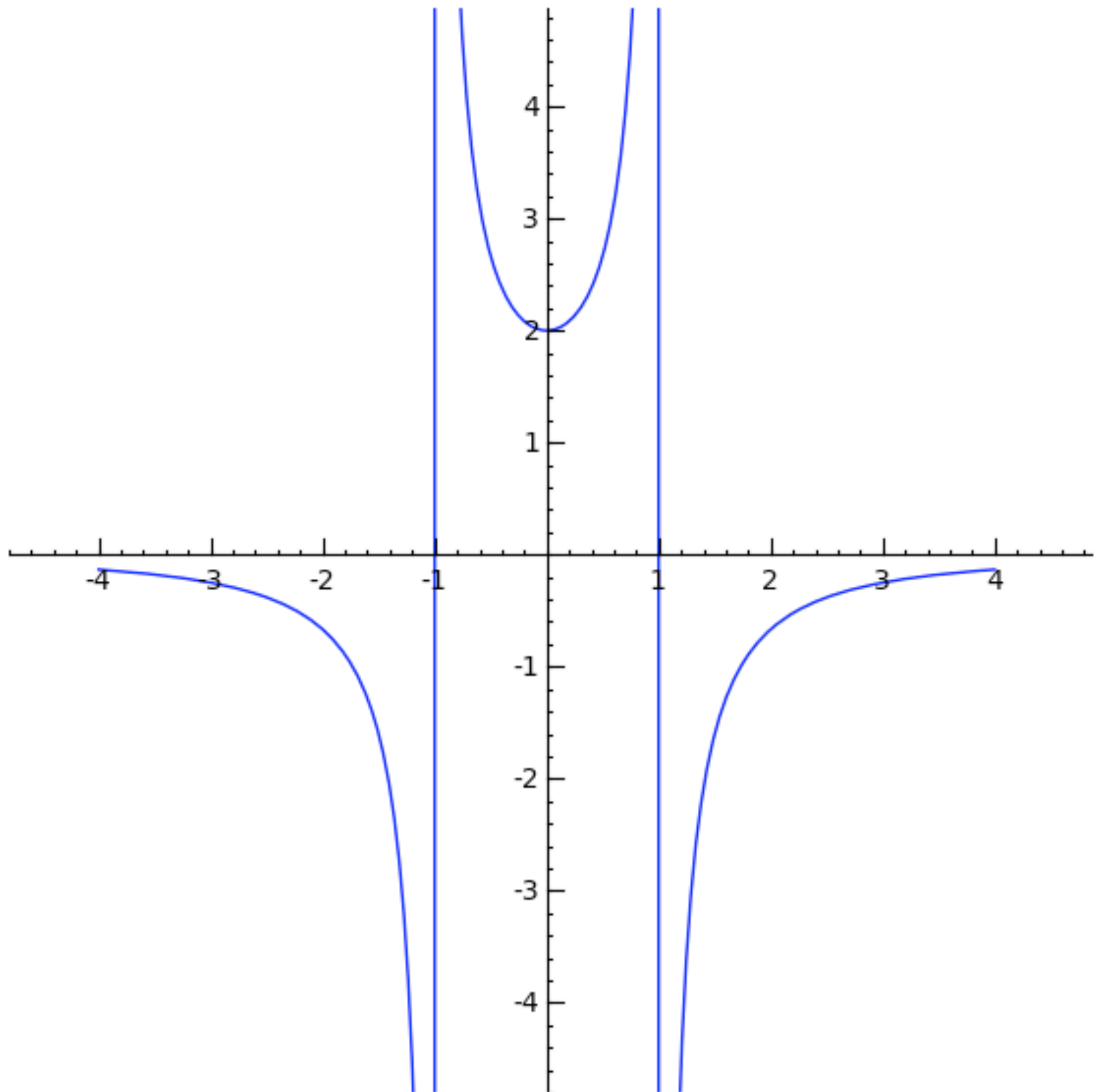
```
show(plot(3*(x-1)^2, (x, -1, 3)), aspect_ratio=2)
```



One way to attack part (c) is to observe that he function has 2 vertical asymptotes, one at $x = 1$ and one at $x =$. The one at $x = -1$ looks like what you'd get by shifting $y = 1/x$ one unit to the left. The one at $x = 1$ looks like what you'd get by first turning $y = 1/x$ upside down and then shifting it right by one unit. So a reasonable point to start would be with

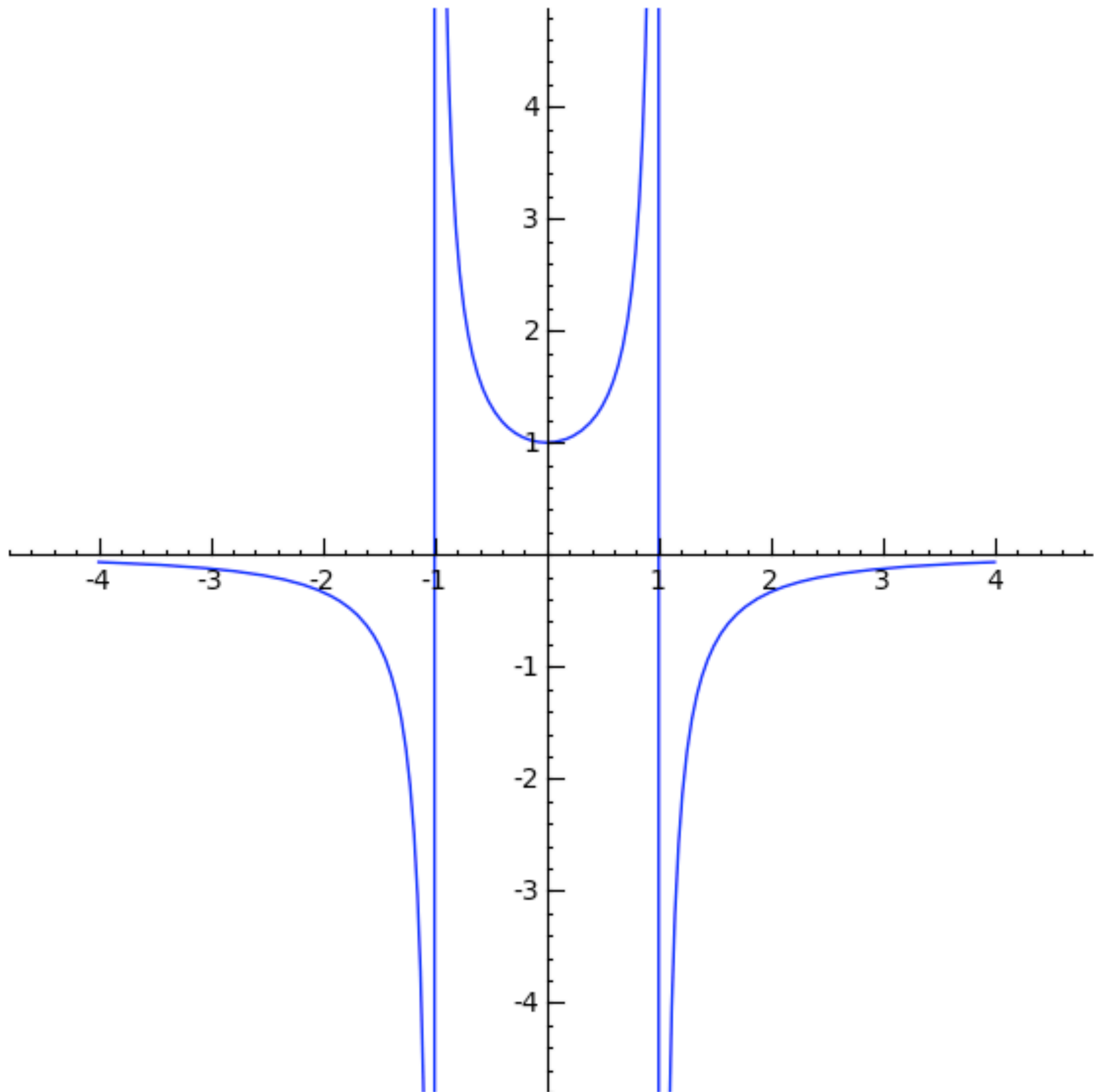
$$y = \frac{1}{x+1} - \frac{1}{x-1}.$$

```
show(plot(1/(x+1)-1/(x-1), (x, -4, 4)), ymin=-4, ymax=4,
aspect_ratio=1)
```



This looks pretty close except that the vertical heights are too large by a factor of 2. (Look at the center piece of the function to see this.) So if we cut it in half, we should be good to go.

```
show(plot((1/(x+1)-1/(x-1))*(1/2), (x, -4, 4)), ymin=-4, ymax=4,
aspect_ratio=1)
```



To see another way we might have arrived at this function, start out by simplifying it.

```
((1/(x+1)-1/(x-1))*(1/2)).simplify_rational()
```

$-\frac{1}{(x^2-1)}$

Ah, right. Another way to have gotten vertical asymptotes at $x = \pm 1$ would have been to take

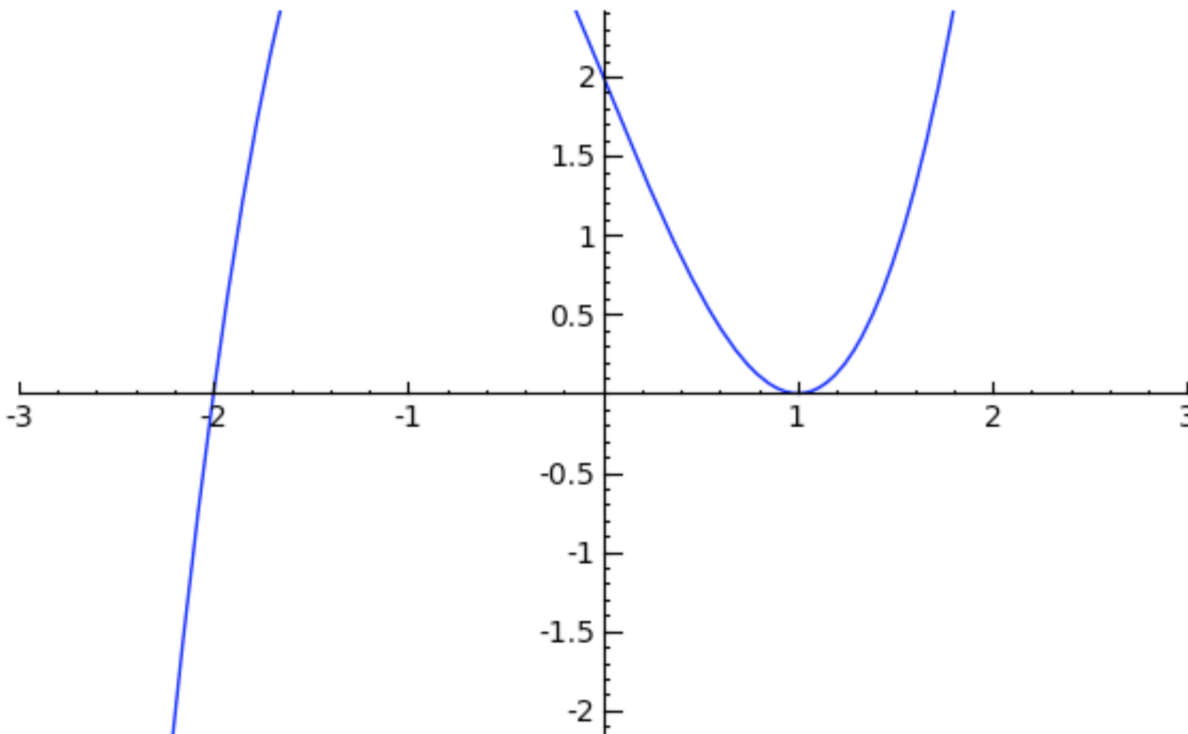
$$y = \frac{1}{(x+1)(x-1)} = \frac{1}{x^2-1}.$$

Plotting this would have given just the function we wanted, only turned upside down; so flipping it over with the minus sign would do the trick.

Finally, let's look at part (d). There are various ways to try to get this function by taking $y = x^3$ and

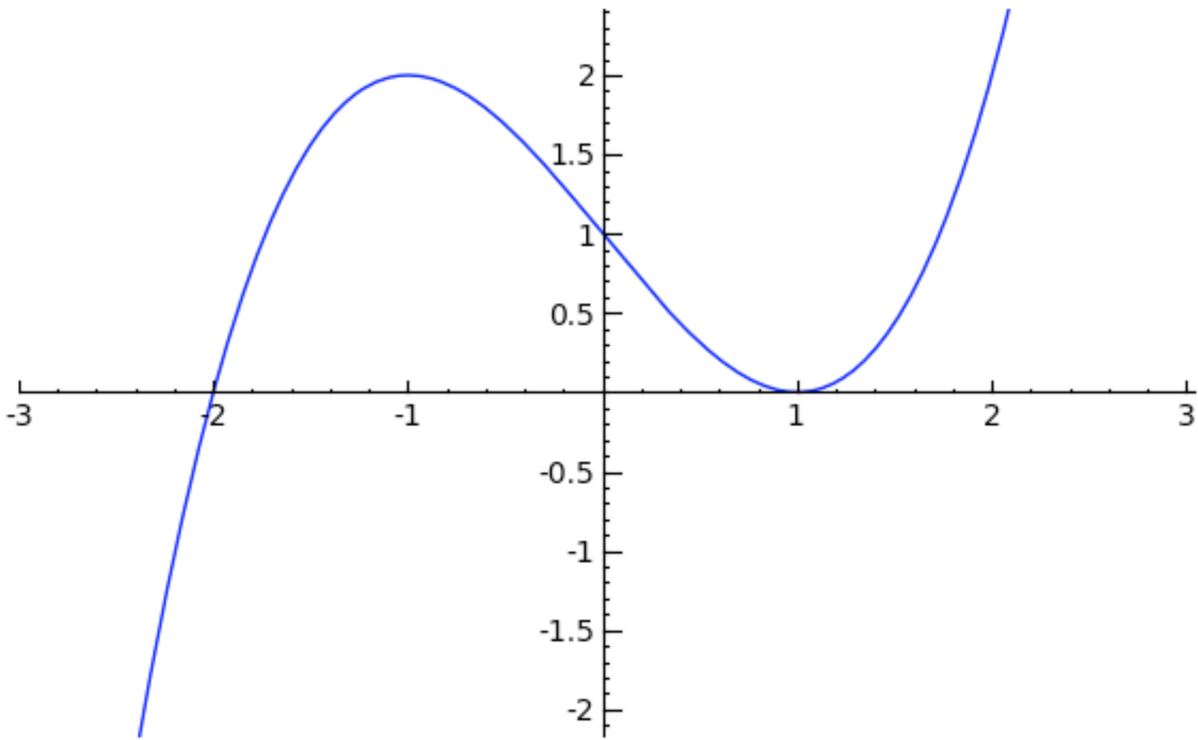
subtracting off something linear or quadratic and then shifting, but a better approach would be to look at the zeros of the function. There seems to be a root at $x = -2$ and a double root at $x = 1$; so a natural way to get a cubic function to start out with would be by looking at $y = (x + 2)(x - 1)^2$.

```
show(plot((x+2)*(x-1)^2, (x, -2.5, 2.5)), ymin=-1.8, ymax=2)
```



Not bad but too tall by a factor of 2, right? So let's fix it by dividing by 2.

```
show(plot((x+2)*(x-1)^2/2, (x, -2.5, 2.5)), ymin=-1.8, ymax=2)
```



Expanding out this function gives something just a little bit complicated that we might have had trouble getting by messing around if we hadn't focused on the roots.

```
((x+2)*(x-1)^2/2).expand()
```

$$\frac{1}{2}\,x^{3} - \frac{3}{2}\,x + 1$$