Sage

Kovács Kristóf, Magyar András, Simon András

November 22, 2023

Sage is a Computer Algebra System meaning that it can do both numeric and symbolic computation. It's built on the Python programming language, so it is programmable.

Some resources:

- Paul Zimmermann & al., Computational Mathematics with SageMath `https://www.sagemath.org/sagebook/english.html`
- Sage documentation `https://doc.sagemath.org/`
- Miscellaneous resources `http://www.gregory-bard.com/Sage.html`

How to try it?

- `https://sagecell.sagemath.org/` (you can't save your work here)
- `https://cocalc.com/` (you need to register, but you can save your work, a bit like with Overleaf)
- (doesn't work yet) Log in to leibniz, open a terminal, do

  ```
  leibniz:~$ ssh tarski
  ...@tarski's password: XXXXX
  tarski:~$ sage
  ```

  where `XXXXX` is your password. (You can also start sage on leibniz, but you get an old version.)

1 A taste of what is possible

```
sage: 1+1
2

sage: factor(123456)
2^6 * 3 * 643
```

```
sage: factorial(42)
1405006117752879898543142606244511569936384000000000

sage: factor(x^2-1)
(x + 1)*(x - 1)
```
This was a typical symbolic calculation.
```
sage: solve(x^2-1==0,x)
[x == -1, x == 1]

sage: m = matrix([[1,1,1],[1,2,3],[3,2,1]])

sage: m


[1 1 1]
[1 2 3]
[3 2 1]

sage: m+m


[2 2 2]
[2 4 6]
[6 4 2]

sage: m^2


[ 5  5  5]
[12 11 10]
[ 8  9 10]

sage: diff(sin(x),x)
cos(x)

sage: integral(sin(x),x)
-cos(x)
```

What is programmability, and why is it useful?

Here's a simple example. Fermat conjectured in the 17th century that all numbers of the form $F_n = 2^{2^n} + 1$ (where $n \in \mathbb{N}$) are primes. If he had Sage, he could've easily found a counterexample:

```
sage: for n in range(1,60):
          if not(is_prime(2^(2^n)+1)):
               print(f"n={n} is a counterexample: 2^(2^{n})+1 = {factor(2^(2^n)+1)}")
               break
n=5 is a counterexample: 2^(2^5)+1 = 641 * 6700417
```

So F_5 is a composite number. (Actually, it's still not known whether there is an $n > 5$ such that F_n *is* prime.)

Here's another:

```
sage: [[Subsets(n, m).cardinality() for m in range(n+1)] for n in range(10)]


[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

This is Pascal's triangle, because `Subsets(n,m)` returns all subsets of size m of the set $\{1, 2, ..., n\}$. We could've written

```
sage: [[binomial(n,m) for m in range(0,n+1)] for n in range(10)]
```

to get the same result.

2 Basics

`=` is for assignment

```
sage: 3 = 3
ValueError

sage: a = 3

sage: a
3

sage: type(a)
<type 'sage.rings.integer.Integer'>
```

This, by the way, is not the type of `a` but of the value of `a`:

```
sage: a = "Hello World!"

sage: type(a)
<type 'str'>
```

Let's do

```
sage: a = 3
```

again.

For comparison, use `==` (and `<=`, `>=`, `<`, `>`, `!=`):

```
sage: a == 3
True

sage: a > 3
False
```

Some mathematical operations:

```
sage: 2^16
65536

sage: 2**16
65536

sage: 13/4
13/4

sage: 13//4 #integer quotient
3

sage: 13%4 #remainder (13 modulo 4)
1
```

(The next three works only in the command line version.)

```
sage: _ #the last result
1

sage: ___ #the one before the last but one result
3

sage: __ #the last but one result
1
```

We can get multiple results in one go:

```
sage: a == 3, a > 3
(True, False)

sage: type(_)
<type 'tuple'>
```

And we can give more commands (to be executed in sequence) in one go:

```
sage: a = 3; a
3
```

Operators	**Description**
`or`	boolean or
`and`	boolean and
`not`	boolean not
`in`, `not in`	membership
`is`, `is not`	identity test
`>`, `<=`, `>`, `>=`, `==`, `!=`	comparison
`+`, `-`	addition, subtraction
`*`, `/`, `%`	multiplication, division, remainder
`**`, `^`	exponentiation

Here the later an operator comes, the higher its precedence. So for example

```
sage: 2^3*4 == (2^3)*4
True
```

but

```
sage: 2^3*4 == 2^(3*4)
False

sage: sqrt(2) #Sage is exact
sqrt(2)

sage: sqrt(2.0) #but only when it can be
1.41421356237310

sage: n(sqrt(2)) #and even then, we can force it to be nonexact
1.41421356237310
```

The function `n()` returns a numerical approximation.

```
sage: sqrt(2)^2 #Sage is exact
2
```

There's more to say about approximation, but before going any further, we note that the functions we used so far have method "counterparts", too, so we can write

```
sage: 2.sqrt(), 2.sqrt().n(), sqrt(2).n()
(sqrt(2), 1.41421356237310, 1.41421356237310)
```

Some more examples of numerical approximations (and methods):

```
sage: pi #exact
pi

sage: n(pi) #numerical approximation
3.14159265358979

sage: n(pi,digits=50) #...to 50 digits
3.1415926535897932384626433832795028841971693993751
```

We could have written `pi.n(digits=50)` instead.

```
sage: sin(pi/3) #exact
1/2*sqrt(3)

sage: n(sin(pi/3)) #approximation
0.866025403784439

sage: sin(n(pi/3)) #approximation
0.866025403784439
```

To get general help, type `?`. To get help on any sage function, type its name followed by `?`, e.g. `sin?` or `diff?`.

When you type something like `sin?` in the command line version of Sage, it may start a paging program to display the requested help. Type a space to scroll to the next page, type `h` to get help on the paging program, and type `q` to quit it and return to the `sage:` prompt. The `up/down` arrow keys and the `PgUp/PgDn` keys will likely work.

In the command line version, use `TAB`-completion!

2.1 Basic Algebra and Calculus

When doing symbolic computations, we need symbolic variables, as opposed to ordinary variables. One difference is that ordinary variables are always evaluated, while symbolic ones are not. And the latter is what we usually want in mathematics, because when we write $x^2 + 1$, we want the polynomial, not the number that one gets by evaluating this expression using the value of `x`. Similarly, in $\sin' x$ (`diff(sin(x),x)`) we don't care what the value of x is, we want to differentiate the *function* $\sin x$.[1]

`x` is a symbolic variable by default:

```
sage: type(x)
<type 'sage.symbolic.expression.Expression'>
```

as opposed to

```
sage: type(y)
NameError:  name 'y' is not defined
```

But we can turn `y` into a symbolic variable like this:

```
sage: var('y')
y
```

and now we have

```
sage: type(y)
<type 'sage.symbolic.expression.Expression'>
```

In fact, we can do this for more than one variable simultaneously like this:

[1]In fact, what are called "symbolic variables" here can be found in ordinary programming languages, such as Python, too. When defining a function, its variables are also not evaluated, they're just "placeholders".

```
sage: var('m xx yy zz')
(m, xx, yy, zz)
```

Defining mathematical ("callable symbolic") functions

```
sage: f(y) = y^3 ; f
y |-> y^3

sage: f(3)
27
```

A side effect of defining `f` above is that `y` is turned into a symbolic variable. So

```
sage: f(2*y)
8*y^3
```

also works, as does

```
sage: f(2*x^2)
8*x^6
```

but not `f(2*z)` unless `z` has already been turned into a symbolic variable. Some other things to do with our function:

```
sage: diff(f)
y |-> 3*y^2

sage: diff(f,2) #differentiate it twice
y |-> 6*y
```

But `diff(f,y)` and `diff(f,y,2)`, or indeed `diff(f(y),y)` and `diff(f(y),y,2)` might be clearer[2]. And one needs to name the variable anyway for multivariable functions[3] , as in this example:

```
sage: g(x,y) = y*exp(x*y)

sage: diff(g,y)
(x, y) |-> x*y*e^(x*y) + e^(x*y)
```

or

```
sage: g.diff(y)
(x, y) |-> x*y*e^(x*y) + e^(x*y)
```

or even

```
sage: g(x,y).diff(y)
x*y*e^(x*y) + e^(x*y)
```

Sage can find a primitive function of a function:

```
sage: integral(f(y),y)
1/4*y^4
```

[2]On the other hand, `diff(f)` and `diff(f,y)` are themselves "callable symbolic" functions, while `diff(f(y),y)` is just an expression. This means that one can write `diff(f,y)(42)`, but not `diff(f(y),y)(42)`; we need to substitute `42` for the variable `y` in `diff(f(y),y)` like this: `diff(f(y),y).subs(y=42)`. We'll meet the method `.subs()` very soon.

[3]unless one wants its Jacobi matrix

or integrate it:
```
sage: integral(f(y), y, 1, 4)
255/4
```
(that is, $\int_1^4 f(y)\,dy = \frac{255}{4}$) or compute its limits:
```
sage: g(y) = (1 + pi/y)^y

sage: limit(g(y), y=infinity)
e^pi
```
or
```
sage: g(y).limit(y=infinity)
e^pi
```
It can also mislead us:
```
sage: limit(1/y, y = 0)
Infinity
```
even though it knows that
```
sage: limit(1/y, y = 0, dir = '-')
-Infinity
```
By the way, `infinity` can be written as `oo`, too.

If we want to do just one thing with a function, there's no need to define it. For example, instead of defining `f(y) = y^3` and then integrating it as above, we could just say
```
sage: integral(y^3,y,1,4)
255/4
```
And even if we want to do more with it, the command line client has history (we can easily recall earlier commands, using not just the arrow keys, but incremental backward search, started by `Ctrl-r`, just as on the command line), so we may not want to define it. On the other hand, `TAB`-completion seems to work better on defined names. So for example, pressing `TAB` after writing `f.in` shows

```
f.integral        f.inverse_laplace
f.integrate       f.inverse_mod
```

but does nothing after `y^3.in`. Try it! The situation is similar with help for methods.

While dealing with terms ("expressions"), sometimes we want to substitute a term for a variable (or even for another, more complex subterm). We can do that with the method `subs()`. Some examples:
```
sage: ((x+cos(y))^3+x*y).subs(y=z+1)
(x + cos(z + 1))^3 + x*(z + 1)
```
This was typical. Don't forget the parentheses around the expression you want to do the substitution in:
```
sage: (x+cos(y))^3+x*y.subs(y=z+1)
(x + cos(y))^3 + x*(z + 1)
```
Here the first `y` was left alone, because Sage didn't know it was part of the expression `subs()` was supposed to operate on.

Less typical is when we want to replace a more complex subterm. In this case we need to use either `==` in place of `=`:

```
sage: ((x+cos(y))^3+x*y).subs(x*y==z+1)
(x + cos(y))^3 + z + 1
```

or a *dictionary* (this is a Python data structure):

```
sage: ((x+cos(y))^3+x*y).subs({x*y:  z+1})
(x + cos(y))^3 + z + 1
```

Two other often used methods for expressions are `expand()` and `collect()`:

```
sage: ((x+cos(y))^3+x*y).expand()
x^3 + 3*x^2*cos(y) + 3*x*cos(y)^2 + cos(y)^3 + x*y

sage: ((x+cos(y))^3+x*y).expand().collect(x)
x^3 + 3*x^2*cos(y) + cos(y)^3 + (3*cos(y)^2 + y)*x
```

This grouped the coefficients of the powers of `x`.

Solving equations

We have already seen the `solve()` function at work.

```
sage: solve(x^2-1==0,x)
[x == -1, x == 1]
```

This works too:

```
sage: solve(x^2-1,x)
[x == -1, x == 1]
```

because if only a term is given, not an equation, it will assume that we want to find its roots. What `solve()` returned is the list[4] of solutions.

```
sage: type(_)
<class 'sage.structure.sequence.Sequence_generic'>
```

This is a special kind of list but a list nonetheless:

```
sage: isinstance(solve(x^2-1,x), list)
True
```

For this reason, the individual solutions can be accessed by indexing it with `[i]`; so for example, after

```
sage: sols = solve(x^2-1,x)
```

we can write

```
sage: sols[1]
x == 1
```

Note that the indexing starts with 0. And if we want the actual value (we probably do), for example, to test that it really is a solution (see `subs()` above), then we take the right hand side of this equation:

```
sage: sols[1].rhs()
1
```

[4]`list` is a Python data structure which contains a collection of objects that can be accessed by indexing with natural numbers.

solve() will always attempt to find exact solutions (at least with single equations). If it can't, it returns the original equation, sometimes in slightly disguised form:

```
sage: solve(x^7-3*x^2+1,x)
[0 == x^7 - 3*x^2 + 1]
```

If this happens, we can use find_root[5] to find *real* roots :

```
sage: find_root(x^7-3*x^2+1,-100,100)
-0.5715684166370613
```

to get a numerical approximation to one of its solutions. The last two arguments specify the interval on which to look for a root. So to find a different root we may try this:

```
sage: find_root(x^7-3*x^2+1,0,100)
1.1792979467976112
```

Inequalities can also be solved by solve():

```
sage: solve(x^2>8,x)
[[x < -2*sqrt(2)], [x > 2*sqrt(2)]]
```

We can solve equations for one variable in terms of others:

```
sage: var('x a b c')

sage: solve(a*x^2 + b*x + c,x)
[x == -1/2*(b + sqrt(b^2 - 4*a*c))/a, x == -1/2*(b - sqrt(b^2 - 4*a*c))/a]
```

We can also solve the same equation for the other variables, even though this is not very useful in this case:

```
sage: solve(a*x^2 + b*x + c==0,b)
[b == -(a*x^2 + c)/x]
```

Solving multiple equations (systems of equations) and inequalities simultaneously is also possible:

```
sage: solve([x+y==6, x*y==4], x, y)
[[x == -sqrt(5) + 3, y == -4/(sqrt(5) - 3)], [x == sqrt(5) + 3, y ==
4/(sqrt(5) + 3)]]
```

The first argument in this case is the list of equations (and inequalities) we want to solve, and the remaining arguments are the "unknowns".

Here's an example with an inequality:

```
sage: solve([x^2 -1==0, x<0],x)
[[x == -1]]
```

Sometimes solve() doesn't return all solutions:

```
sage: solve(sin(x),x)
[x == 0]
```

But we can force it to work harder:

```
sage: solve(sin(x),x, to_poly_solve='force')
[x == pi*z35]
```

z35 is freshly generated symbol and stands for any integer. Another example:

```
sage: solve(abs(x)==1,x)
[abs(x) == 1]
```

[5]or the keyword argument to_poly_solve of solve() – see below!

but

```
sage: solve(abs(x)==1,x,to_poly_solve='force')
[x == -1, x == 1]
```

Much better. See the help or the documentation for other possibilities.

Assumptions

```
sage: solve(x^2==1,x)
[x == -1, x == 1]

sage: assume(x>0); solve(x^2==1,x)
[x == 1]

sage: assumptions()
[x > 0]

sage: forget(x>0); solve(x^2==1,x)
[x == -1, x == 1]
```

There are other kinds of assumptions. For example:

```
sage: assume(y,'integer')
```

or even

```
sage: assume(y,'odd')
```

Linear algebra

We have already seen how to solve (systems of) equations: of course, these equations can be linear.

```
sage: solve([2*x + 4*y == 14,3*x + 2*y == 13],x,y)
[[x == 3, y == 2]]
```

Here is one with infinitely many solutions (not surprisingly):

```
sage: solve([4*x - 2*y + 3*z == 1, 8*x - 4*y + 6*z == 2, 12*x - 6*y
+ 9*z == 3],x,y,z)
[[x == -3/4*r3 + 1/2*r4 + 1/4, y == r4, z == r3]]
```

`r3` and `r4` are freshly generated symbols and stand for any real numbers.

If there are no solutions, `solve()` returns the empty list:

```
sage: solve([x + y == 0, x + y == 1],x,y)
[]
```

We can solve the system of equations above (the one with infinitely many solutions) in matrix form, too. The simplest way to define a matrix is to call the function `matrix()` with the list of rows (each of which is a list of its entries):

```
sage: A = matrix([[4,-2,3],[8,-4,6],[12,-6,9]]) ; A


[ 4 -2  3]
[ 8 -4  6]
```

```
[12 -6  9]
```

But it's safer to write

```
sage: A = matrix(QQ,[[4,-2,3],[8,-4,6],[12,-6,9]])
```

to let Sage know that it's OK to do computations in $\mathbb{Q}$ and not just in $\mathbb{Z}$ (where the coefficients come from). (Writing `RR` in place of `QQ` would work, too, but then we'd lose exactness.)

```
sage: v = vector([1,2,3]).column() ; v


[1]
[2]
[3]

sage: A.solve_right(v)
```

(or `A \ v`)

```
[1/4]
[0]
[0]
```

This is a particular solution, and is correct, because

```
sage: A*(A \ v)


[1]
[2]
[3]
```

If we want all solutions, we need to add to this all linear combinations of a basis of the null space (or kernel) of `A`[6]. These are the rows of

```
sage: k = A.right_kernel() ; k


Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 -4/3]
[0 1  2/3]
```

This looks like a matrix but it is not. (Check with `type(k)`!) To get a matrix, do

```
sage: m=k.matrix() ; m, type(m)


[1 0 -4/3]
[0 1  2/3]
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
```

[6]If $Ax_0 = b$, then the set of all solutions of the equation $Ax = b$ is $x_0 + \operatorname{Ker} A (= \{ x_0 + x : Ax = 0 \})$.

and then we can access its rows:

```
sage:   m[0] , m[1]
((1, 0, -4/3), (0, 1, 2/3))
```

whose linear combinations should be added to the particular solutions above to get all solutions.

Some other things to do with matrices: `A` is still the matrix we used before:

```
sage: A


[ 4 -2  3]
[ 8 -4  6]
[12 -6  9]
```

Asking for a specific row or column of a matrix:

```
sage: A.row(1)
(8, -4, 6)
```

or

```
sage: A[1]
(8, -4, 6)
```

and

```
sage: A.column(1)
(-2, -4, -6)
```

We can augment a matrix with extra columns

```
sage: A.augment(v)


[ 4 -2  3  1]
[ 8 -4  6  2]
[12 -6  9  3]
```

and reduce it to echelon form:

```
sage: A.augment(v).echelon_form()


[1 -1/2  3/4  1/4]
[0    0    0    0]
[0    0    0    0]
```

From this we know already that

```
sage: A.rank()
1
```

If `B` is an invertible matrix

```
sage: B = matrix(QQ,[[2,4,0],[0,-1,1],[1,1,8]]); rank(B)
3
```

then inverting it is just a matter of taking its -1st power:

```
sage: B, B^(-1), B^(-1)*B
```

```
(
[2  4  0]  [ 9/14  16/7  -2/7]  [1 0 0]
[0 -1  1]  [-1/14  -8/7   1/7]  [0 1 0]
[1  1  8], [-1/14  -1/7   1/7], [0 0 1]
)
```

`A.eigenvalues()` and `A.eigenvectors_right()` and `A.charpoly()` return the eigenvalues, the eigenvectors, and the characteristic polynomial, respectively, of the matrix `A`.[7]

```
sage: A.charpoly()
x^3 - 9*x^2

sage: A.eigenvalues()
[9, 0, 0]

sage: A.eigenvectors_right()


[(9,
  [
  (1, 2, 3)
  ],
  1),
 (0,
  [
  (1, 0, -4/3),
  (0, 1, 2/3)
  ],
  2)]
```

And indeed:

```
sage: A*x*vector([1,2,3]), A*(x*vector([1, 0, -4/3]) + y*vector([0,
1, 2/3]))


((9*x, 18*x, 27*x), (0, 0, 0))
```

But even Sage didn't have such functions, we could compute eigenvalues/eigenvectors:

```
sage: I3 = identity_matrix(QQ,3) ; I3


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

[7]Recall that the eigenvalues of A are the roots of $\det(A - \lambda I) = 0$, the *characteristic equation* ($\det(A - \lambda I)$ is the *characteristic polynomial*) of A. The eigenspace (the null vector and the eigenvectors) corresponding to the eigenvalue λ are the solutions of the equation $A - \lambda I$, that is, $\mathrm{Ker}(A - \lambda I)$.

```
sage: x * I3

[x 0 0]
[0 x 0]
[0 0 x]
```

So we can get the characteristic polynomial of `A` by evaluating

```
sage: det(A - x*I3)
-((x + 4)*(x - 9) + 36)*(x - 4) + 20*x
```

which, after some simplification, will look familiar:

```
sage: _.simplify_full()
-x^3 + 9*x^2
```

Then we get the eigenvalues of `A` with

```
sage: solve(det(A - x*I3),x)
[x == 0, x == 9]
```

In this case we may be interested in the multiplicities of the roots, so we evaluate

```
sage: solve(det(A - x*I3),x,multiplicities=True)
([x == 0, x == 9], [2, 1])
```

meaning that 0 is a double root. And now we can get a basis for the eigenvectors belonging to the various eigenvalues like this:

```
sage: (A - x*I3).subs(x=9).right_kernel()


Vector space of degree 3 and dimension 1 over Symbolic Ring
Basis matrix:
[1 2 3]
```

```
sage: (A - x*I3).subs(x=0).right_kernel()


Vector space of degree 3 and dimension 2 over Symbolic Ring
Basis matrix:
[1 0 -4/3]
[0 1  2/3]
```

Some other things to do with vectors:

```
sage: v = vector([1,2,3]) ; w = vector([3,2,1]); v, w
((1, 2, 3), (3, 2, 1))

sage: v.norm()
sqrt(14)

sage: v.dot_product(w)
10

sage: v.cross_product(w)
(-4, 8, -4)
```

```
sage: v.cross_product(w).dot_product(v),v.cross_product(w).dot_product(w)
(0,0)
```
as it should be.

Number theory

We've seen already that numbers (as well as polynomials) can be factorized using `factor()`:
```
sage: factor(123456)
2^6 * 3 * 643
```
`divisors()` returns the list of divisors of a number
```
sage: divisors(12)
[1, 2, 3, 4, 6, 12]
```

If we just want to take the remainder of a number on division by another, Python's `%` is fine:
```
sage: 13%11
2
```
But there's also Sage's own `mod()` which returns a representative of the appropriate equivalence class:
```
sage: a = mod(13,11) ; a
2
```
But there's more here than meets the eye:
```
sage: a.parent()
Ring of integers modulo 11
```
and
```
sage: a^4
5
```
because $2^4 \pmod{11} = 5$. So for example, to get $5^{-1} \mod 26$, we evaluate
```
sage: m = mod(5,26)^(-1); m
21
```
and indeed,
```
sage: m*mod(5,26)
1
```

`gcd()` computes the greatest common divisor of its two arguments, or of the elements of the list which is given as its only argument:
```
sage: gcd(123,240)
3

sage: gcd([123,240,500])
1
```
`lcm()` (least common multiple) works similarly.

Euler's φ function (the number of natural numbers below a number relatively prime to it) is computed by `euler_phi()`:

```
sage: euler_phi(8)
4
```
the number of odd numbers below 8.

`solve_mod()` returns all solutions to an equation or list of equations modulo the given integer modulus. Each equation must involve only polynomials in one or more variables. For example, to solve the linear congruence $12x \equiv 15 \pmod{21}$ we write
```
sage: solve_mod(12*x==15,21)
[(3,), (10,), (17,)]
```
We could've computed $5^{-1} \mod 26$ above this way, too:
```
sage: solve_mod(5*x == 1, 26)
[(21,)]
```
But we're not stuck with linear congruences. We can solve a congruence like $12x^5 \equiv 15 \pmod{21}$, too:
```
sage: solve_mod(12*x^5==15,21)
[(12,), (19,), (5,)]
```
and indeed, for example
```
sage: mod(12*5^5,21)
15
```

For simultaneous solution of linear congruences with different moduli we can use `crt()` (which is acronym for "Chinese Remainder Theorem"). For example, to solve

$$x \equiv 3 \pmod 8 \qquad x \equiv 4 \pmod 9 \qquad x \equiv 5 \pmod{25}$$

we write
```
sage: crt([3,4,5],[8,9,25])
355
```
and this result means that the set of all solutions is $\{n : n \equiv 355 \pmod{8\cdot 9\cdot 25}\}$.

Polynomials

If we want `x` to not simply be a mathematical variable but an indeterminate in a polinomial ring, we should write
```
sage: x = polygen(QQ, 'x')
```
or
```
sage: x = polygen(CC, 'x')
```
or
```
sage: x = polygen(Integers(8), 'x')
```
or
```
sage: x = polygen(GF(5), 'x')
```
etc. depending on which ring ($\mathbb{Q}[x]$, $\mathbb{C}[x]$, $\mathbb{Z}_8[x]$ or $\mathbb{Z}_5[x]$) we want, instead of simply `var('x')` (which, in interactive use, is equivalent to writing `x = var('x')`). Here's how we can check this:

```
sage: y = var('y'); q = (2*y+1)*(y+2)*(y^4-1); y.parent(),q.parent(),q
(Symbolic Ring, Symbolic Ring, (y^4 - 1)*(2*y + 1)*(y + 2))
```
but
```
sage: y = polygen(QQ,'y'); q = (2*y+1)*(y+2)*(y^4-1); y.parent(),q.parent(),q
(Univariate Polynomial Ring in y over Rational Field, Univariate Polynomial
Ring in y over Rational Field, 2*y^6 + 5*y^5 + 2*y^4 - 2*y^2 - 5*y
- 2)
```
or
```
sage: y = polygen(GF(5),'y'); q = (2*y+1)*(y+2)*(y^4-1); y.parent(),q.parent(),q
(Univariate Polynomial Ring in y over Finite Field of size 5, Univariate
Polynomial Ring in y over Finite Field of size 5, 2*y^6 + 2*y^4 + 3*y^2
+ 3)
```

Example. How many *different* roots does the polynomial $y^4 - 2$ have as a polynomial over $\mathbb{Q}$, $\mathbb{C}$, $\mathbb{R}$ and $\mathbb{Z}_8$?
```
sage: rings = [QQ, RR, CC, Integers(8)]
```

```
sage: for r in rings:
....:     y = polygen(r,'y')
....:     q = y^4 - 2
....:     print(r, len(q.roots(multiplicities=False)))
....:
(Rational Field, 0)
(Real Field with 53 bits of precision, 2)
(Complex Field with 53 bits of precision, 4)
(Ring of integers modulo 8, 0)
```

Why different roots? Because the method `.roots(multiplicities=False)` returns the list of different roots. Without this keyword argument (or with `.roots(multiplicities=True)`) it returns the list of pairs `(r, n)`, where `r` is the root and `n` is the multiplicity, so the length of the list is still the number of different roots.[8] For example
```
sage: x = polygen(QQ,'x'); p = (x-1)^2 * (x+3); p, p.roots()
(x^3 + x^2 - 5*x + 3, [(-3, 1), (1, 2)])
```

Of course, we can add, multiply, etc. polynomials. For example,
```
sage: y = polygen(Integers(3),'y'); p = 2*y+1; p+p, p*p, p^2, p/p
(y + 2, y^2 + y + 1, y^2 + y + 1, 1)
```

Division with remainder:
```
sage: x = polygen(QQ,'x'); p = x^3+1; q = 2*x-1; p//q, p%q
(1/2*x^2 + 1/4*x + 1/8, 9/8)
```
which says that $x^3 + 1 = (2x - 1)\left(\frac{1}{2}x^2 + \frac{1}{4}x + \frac{1}{8}\right) + \frac{9}{8}$.

[8] I used `multiplicities=True` here because Sage can't compute the multiplicities of roots of polynomials over $\mathbb{Z}_n$ for composite n. This is not a shortcoming of Sage but that of the universe.

Greatest common divisor:

```
sage: x = polygen(QQ,'x'); p = x^4-4*x^3-x^2+16*x-12; q = x^2-2*x-3;
p.gcd(q)
x - 3
```

Factorization:

```
sage: x = polygen(GF(2),'x'); p = x^5+3*x-1 ; p.factor()
(x^2 + x + 1) * (x^3 + x^2 + 1)
```

but

```
sage: x = polygen(ZZ,'x'); p = x^5+3*x-1 ; p.factor()
x^5+3*x-1
```

and this is how it should be, because

```
sage: p.is_irreducible()
True
```

Example. What are the rational roots of $2x^4 - 5x^3 - 8x^2 + 17x - 6$?

```
sage: x = polygen(QQ,'x'); p = 2*x^4 - 5*x^3 - 8*x^2 + 17*x - 6; p.factor()
(2) * (x - 3) * (x - 1) * (x - 1/2) * (x + 2)
```

or

```
sage: p.roots()
[(3, 1), (1, 1), (1/2, 1), (-2, 1)]
```

(the second members of the pairs are the multiplicities).

Plotting functions

The command `plot(f(x),(x,a,b))` plots the graph of f restricted to the interval $[a, b]$. How the graph will appear can be changed by various keyword arguments. (If we don't specify their values, `plot()` will work with their default values.) For example, the keyword argument `aspect_ratio` describes the height/width ratio of a unit square. So to make the vertical units be twice as big as the horizontal units, we need to specify an aspect ratio of 2.

On Figure 1, we see the effect of the default `aspect_ratio='automatic'`, and on Figure 2 the effect of `aspect_ratio=1`.

The following animation shows that affect of `aspect_ratio`:

```
sage: l = [plot(sin(2*x),(x,-5,5),aspect_ratio=2, color='red'), plot(sin(2*x),(x,-5,5),
aspect_ratio=1)]

sage: an = animate(l); an.show(iterations=3, delay=100)
```

The automatic `aspect_ratio` comes in handy sometimes, see for example Figure 3.

If the problem is only some extremely big or small values, we can use the keyword arguments `ymin` and `ymax`. For example, Figure 4 is a much more informative version of the graph of $1/x$ than the one shown in Figure 5.

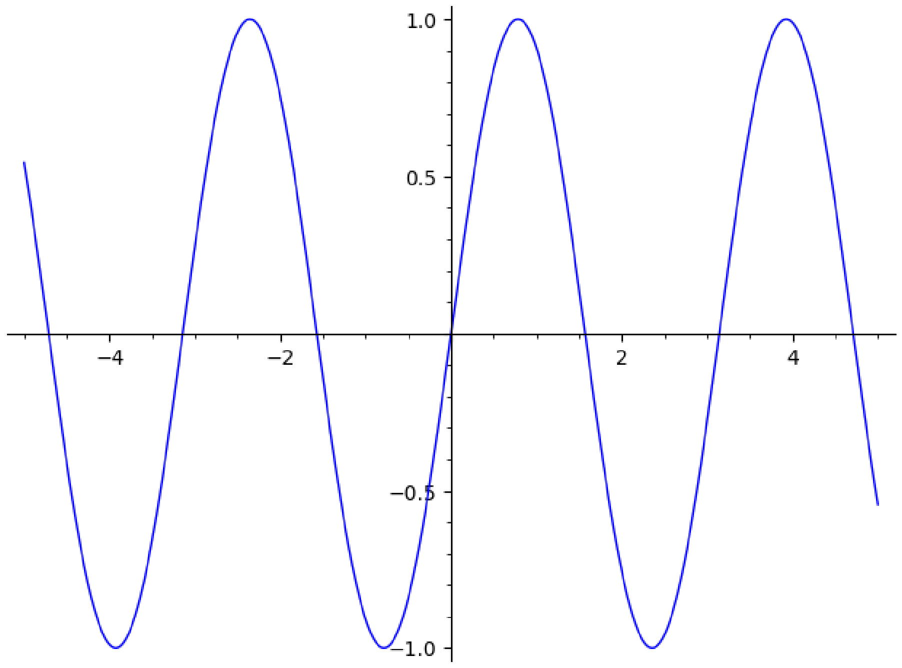

Figure 1: `plot(sin(2*x),(x,-5,5))`

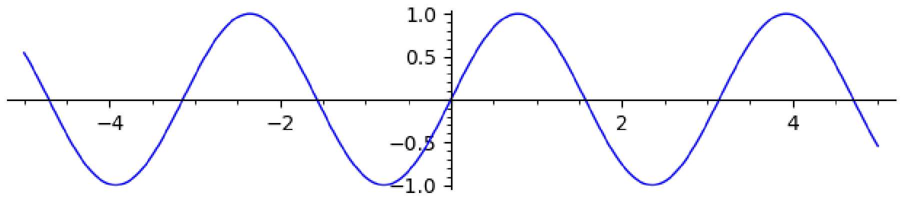

Figure 2: `plot(sin(2*x),(x,-5,5), aspect_ratio=1)`

The area between the x axis and the graph can be filled (see Figure 6) using the keyword argument `fill`. But one can chose any other horizontal line in place of the x axis (that is, the line $y = 0$): `fill=42` will fill the area between the line $y = 42$ and the graph, see Figure 7.

One can annotate the figure with the name of the function whose graph it contains using the keyword argument `legend_label` (see Figure 8). Its value is a LaTeX string, but if that contains a '\' character, we need to write `r'latexstring'` in place of `'latexstring'` (this is needed because otherwise Python would think that we want to escape the character following '\').

This is especially useful if a figure contains the graphs of two or more functions, as in Figure 9.

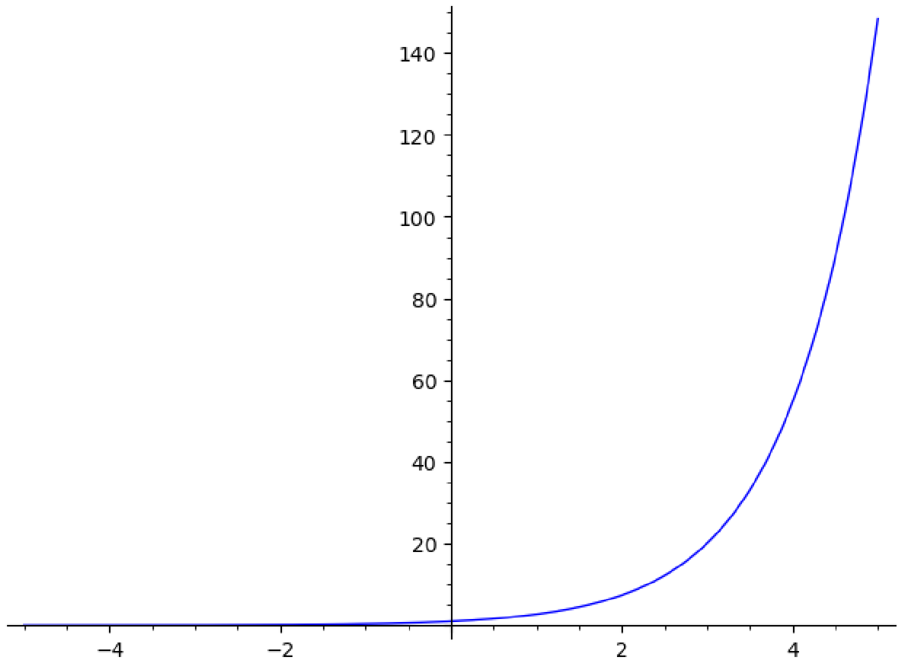

Figure 3: `plot(exp(x),(x,-5,5))`

There are two more interesting points here. The first is that one can specify the colour of the graph (`color='red'`). The other is a nice feature of Python: since we need the character `'` within the LaTeXstring (to denote derivation), we can use the character `"` instead of the character `'` to delimit the string `"$\sin(2x)'$"`.

We can graph functions in polar coordinate system, by using the keyword arguments `polar=True` and `aspect_ratio=1`. This is how Figure 10 was created. (This, by the way, is one of the exercises in `tikzlaben`.) But it could have been created with the command `polar_plot()` as follows:

```
sage: polar_plot(exp(x/8),(x,0, 4*pi))
```

Parametric plots

These are the curves that are given by their coordinate functions. We have encountered such in Ti*k*Z when we plotted functions, but there we may have used the identity function as the first coordinate function, as in Figure 11. But there is no need to "simulate" `plot()` this way, see Figures 12, 13 and 14.

One can also fill the inside of a closed curve, but here the value of `fill` can only be `True` (or `False`, which is the default). See Figure 15.

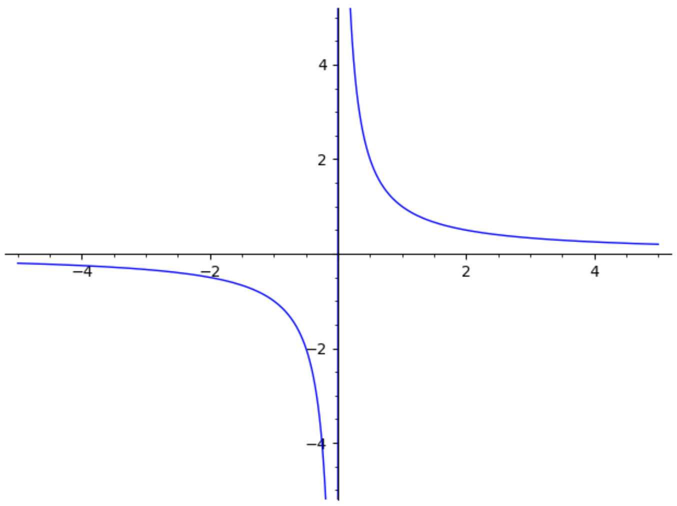

Figure 4: `plot(1/x,(x,-5,5), ymax=5, ymin=-5)`

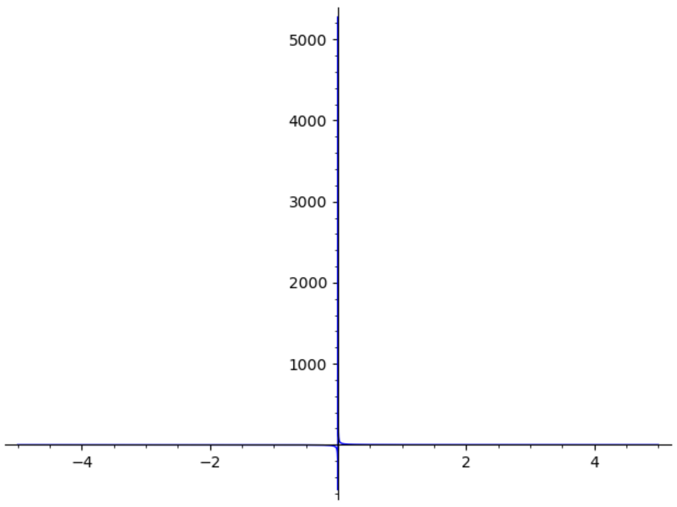

Figure 5: `plot(1/x,(x,-5,5))`

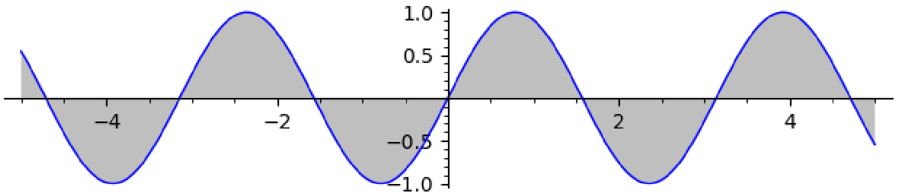

Figure 6: `plot(sin(2*x),(x,-5,5), fill='axis',aspect_ratio=1)`

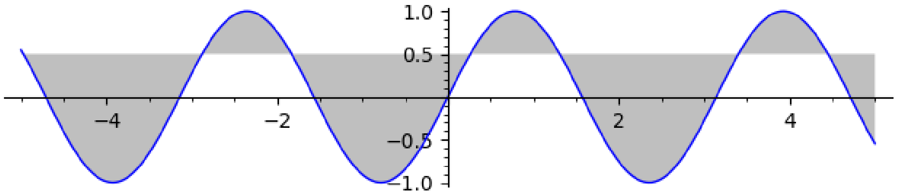

Figure 7: `plot(sin(2*x),(x,-5,5), fill=0.5,aspect_ratio=1)`

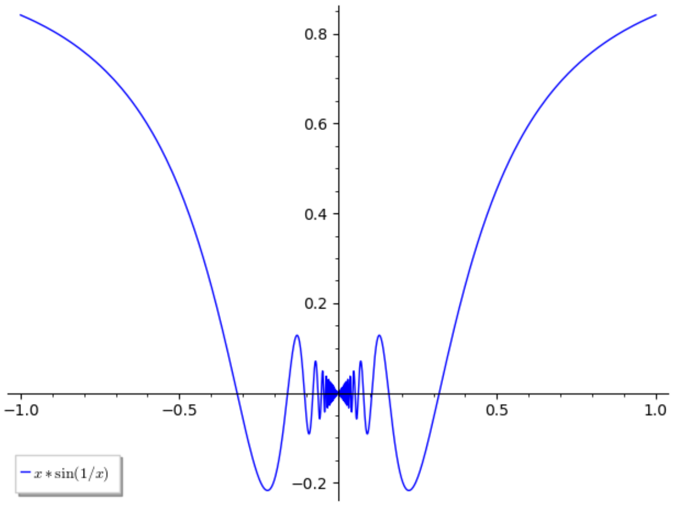

Figure 8: `plot(x*sin(1/x),(x,-1,1), legend_label=r'$x\sin(1/x)$')`

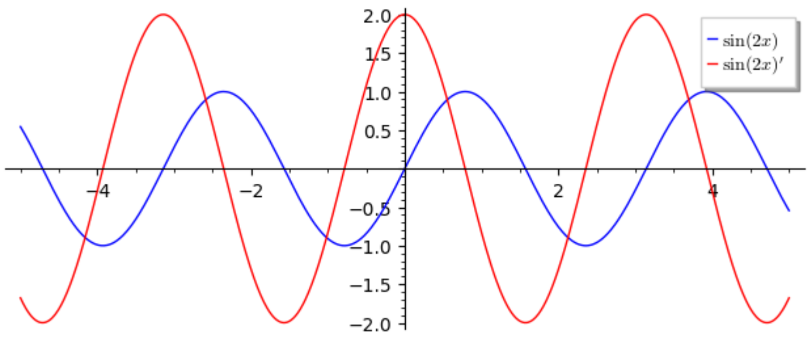

Figure 9: `f(x) = sin(2*x); P1 = plot(f(x),(x,-5,5), aspect_ratio=1, legend_label=r'$\sin(2x)$'); P2 = plot(diff(f(x),x),(x,-5,5), aspect_ratio=1, color='red', legend_label=r"$\sin(2x)'$"); P1 + P2`

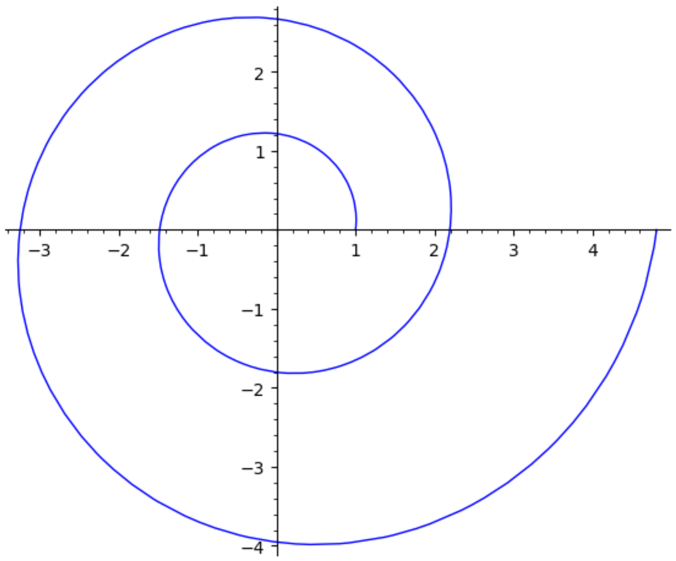

Figure 10: `plot(exp(x/8),(x,0, 4*pi), polar=True, aspect_ratio=1)`

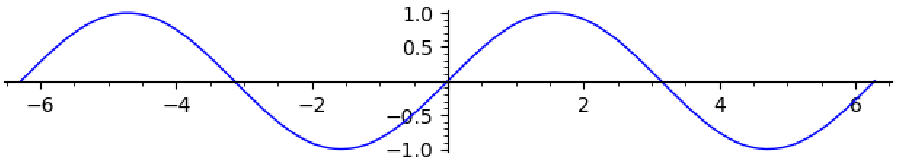

Figure 11: `parametric_plot((x,sin(x)),(x,-2*pi,2*pi),aspect_ratio=1)`

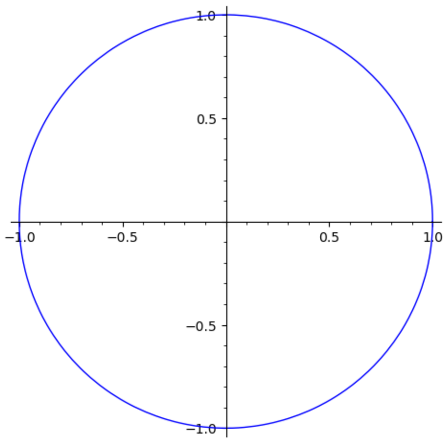

Figure 12: `parametric_plot((cos(x),sin(x)),(x,0,2*pi),aspect_ratio=1)`

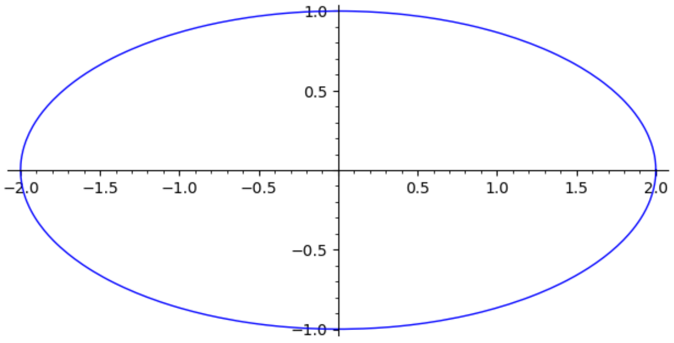

Figure 13: `parametric_plot((2*cos(x),sin(x)),(x,0,2*pi),aspect_ratio=1)`

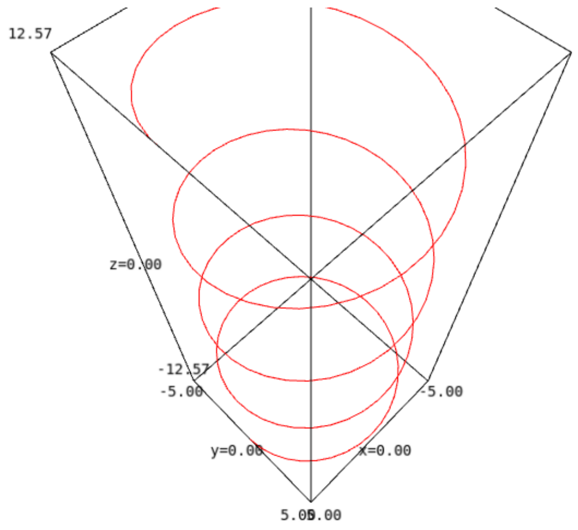

Figure 14: `parametric_plot((5*cos(x),5*sin(x),x),(x,-4*pi,4*pi),plot_points=150, color="red")`

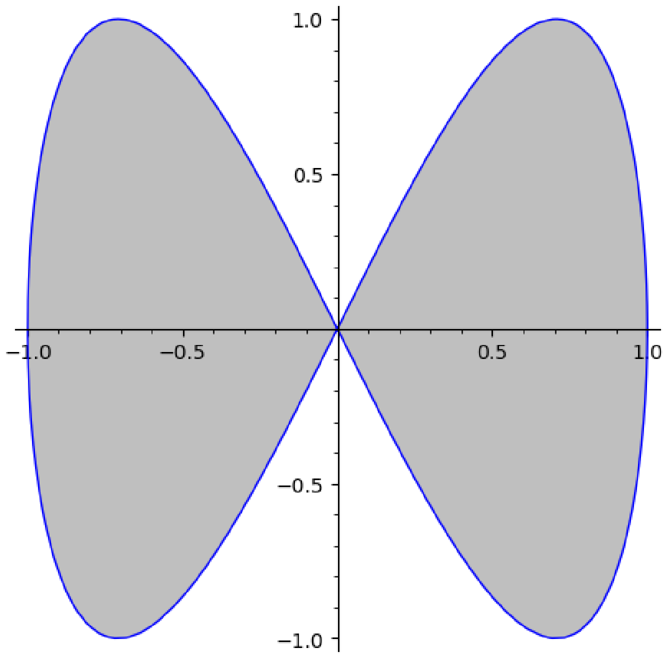

Figure 15: `parametric_plot((sin(x),sin(2*x)),(x,0,2*pi),fill=True)`

3 Appendix

Eigenvalues and eigenvectors of square matrices

Definition. λ is the eigenvalue of the square matrix[9] A if $\exists x \neq 0$ for which $Ax = \lambda x$; in this case x is an eigenvector of A with eigenvalue λ. (λ can be 0, but not x! If 0 were an eigenvector, it would always be an eigenvector with eigenvalue 0.) The eigenspace associated with the eigenvalue λ is 0 and the eigenvectors with eigenvalue λ.

Example. The identity matrix has 1 as its only eigenvalue, and the whole space is the associated eigenspace. The matrix 0 has 0 as its only eigenvalue and the whole space is the corresponding eigenspace. The matrix of rotation by the angle $\pi/2$ around the origin in the plane has no eigenvalues.

How can these be calculated?

$$\begin{aligned}\lambda \text{ is an eigenvalue of } A &\iff (\exists x \neq 0) Ax = \lambda x \\ &\iff (\exists x \neq 0)(A - \lambda I)x = 0 \iff \det(A - \lambda I) = 0.\end{aligned}$$

Hence the eigenvalues of A are the roots of $\det(A - \lambda I) = 0$, the *characteristic equation* of A. ($\det(A - \lambda I)$ is called the *characteristic polynomial* of A.) The eigenspace (the null vector and the eigenvectors) corresponding to the eigenvalue λ are the solutions of the equation $A - \lambda I$, that is, $\operatorname{Ker}(A - \lambda I)$.

[9]In reality, it's linear transformations that have eigenvalues and eigenvectors, but so do square matrices because they encode linear transformations