2014-05-14-linear-algebra.sagews

May 14, 2014

Contents

1 Math 480b Sage Course

1.1 Linear Algebra

1.2 May 14, 2014

Screencast: `http://youtu.be/r0kxxxZABjk`

Plan

- Questions
- Homework:
  - hw7, etc., due Monday morning at 6am
  - talk to Simon about any grading related issues
- Topic: Exact linear algebra
  - solving a system of equations: simple small naive approach
  - creating matrices and vectors
  - solving a matrix equations
  - computing invariants of matrices
  - vector spaces
  - linear algebra over finite fields (very important for coding theory)
  - remarks about asymptotically fast algorithms

1.3 Exact Linear Algebra

Our topic for today is exact linear algebra, by which I mean algebra with matrices whose entries are exact numbers (integers, rational numbers, elements of a finite field, etc.), rather than approximate numbers (floating point numbers).

These are very important in coding theory, combinatorics, much of pure math research, etc. They are not so important in applied math, where one usually works with matrices having floating point (or complex) entries, and the algorithms and issues (e.g., numerical analysis to deal with rounding errors) are much different.

1.4 Solving a system of linear equations

Here is an example to illustrate the naive (painful) direct approach, which is fine for small example and some educational applications.

```
# create 3 symbolic variables:
%var x,y,z

# make a list of 3 linear equations
v = [2*x + 3*y + 5*z == 10,
     7*x - 4*y == 5,
     2*x - 5*y + z == 2]

# solve the equations for x,y,z
s = solve(v, [x,y,z], solution_dict=True)
s
```

```
[{z: 62/41, x: 35/41, y: 10/41}]
```

```
# the only solution:
t = s[0]; t
```

```
{z: 62/41, x: 35/41, y: 10/41}
```

```
t[z]
```

```
62/41
```

NOTE: The solve command mostly uses some very generic/general machinery, so you can put some weird nonlinear terms in.

```
v = [2*x^2 + 3*y + 5*z == 10,
     7*x - 4*y == 5,
     2*x - 5*y + z^2 == 2]

solve(v,[x,y,z])
```

```
[[x == (-4.59043682722 - 2.3161925967*I), y == (-9.28326444764 - 4.05333704422*I), z ==
(1.28701382069 - 6.07386640933*I)], [x == (-4.59043682722 + 2.3161925967*I), y ==
(-9.28326444764 + 4.05333704422*I), z == (1.28701382069 + 6.07386640933*I)], [x ==
0.930873621713, y == 0.379028882378, z == 1.42597239649], [x == 3, y == 4, z == -4]]
```

```
# solve a cubic algebraic equation
show(solve(x^3 + 5*x + 2, x))
```

$$\left[x = -\frac{1}{2}\left(\frac{2}{9}\sqrt{38}\sqrt{3} - 1\right)^{\frac{1}{3}}\left(i\sqrt{3} + 1\right) + \frac{-5i\sqrt{3} + 5}{6\left(\frac{2}{9}\sqrt{38}\sqrt{3} - 1\right)^{\frac{1}{3}}}, x = -\frac{1}{2}\left(\frac{2}{9}\sqrt{38}\sqrt{3} - 1\right)^{\frac{1}{3}}\left(-i\sqrt{3} + 1\right) + \frac{5i\sqrt{3} + 5}{6\left(\frac{2}{9}\sqrt{38}\sqrt{3} - 1\right)^{\frac{1}{3}}}, x\right.$$

1.5 Creating matrices and vectors

```
#2*x + 3*y + 5*z == 10,
#7*x - 4*y == 5,
#2*x - 5*y + z == 2

m = matrix(3,3, [2,3,5,  7,-4,0,  2,-5,1])
v = vector([10,5,2])
```

```
show(m)
show(v)
```

$$\begin{pmatrix} 2 & 3 & 5 \\ 7 & -4 & 0 \\ 2 & -5 & 1 \end{pmatrix}$$

$$(10,\ 5,\ 2)$$

```
matrix([[1,2], [3,4/7]])
```

```
[  1   2]
[  3 4/7]
```

```
# a matrix with entries in the symbolic ring
m = matrix([[pi,e,sqrt(2)], [1,2,sin(x)]])
show(m)
```

$$\begin{pmatrix} \pi & e & \sqrt{2} \\ 1 & 2 & \sin(x) \end{pmatrix}$$

```
# a biger matrix (all entries are zero by default)
m = matrix(25,50)
m
```

```
25 x 50 dense matrix over Integer Ring (type 'print m.str()' to see all of the entries)
```

```
print m.str()
```

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

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

```
# a random matrix with integer entries
random_matrix(ZZ, 5, 10)
```

```
[  0   1  -5   0   2  -1  -1   0   5  -1]
[  1  -2   0  -1  -3   1   0   1   0   0]
[  1  -2   0   1  -2  -1   1   0   2   0]
[ -4   0   0  -1   0   2   1  -1   2   1]
[ -1   8  -2  -1 -12   1   1  -1 -48 -14]
```

```
random_matrix(ZZ, 5, 10, x=-3, y=3)  # entries chosen uniformly from \
    [-3..3]
```

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

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

```
a = random_matrix(QQ, 3); a
b = random_matrix(QQ,3); b
```

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

```
a+b
```

```
[   0    1 -5/2]
[  -1 -1/2    0]
[  -1   -3  1/2]
```

```
# addition adds to the *diagonal*
a + 10
```

```
[  10    0 -1/2]
[   0 21/2    2]
[   1   -1 21/2]
```

```
# matrix multiplication is matrix multiplication
a * b
```

```
[   1    1    0]
[-9/2 -9/2   -1]
[   0    1    0]
```

```
a + a.transpose()
```

```
[  0   0 1/2]
[  0   1   1]
[1/2   1   1]
```

1.6 Solving a matrix equations

```
m = matrix(3,3, [2,3,5,  7,-4,0,  2,-5,1])
v = vector([10,5,2])
```

```
# solve m*x = v
x = m.solve_right(v); x
```

```
(35/41, 10/41, 62/41)
```

```
m*x
```

```
(10, 5, 2)
```

```
# solve x*m = v
```

```
x = m.solve_left(v); x
```

```
(129/164, 72/41, -317/164)
```

```
x*m
(10, 5, 2)
```

```
# use matlab notation
x = m \ v; x
(35/41, 10/41, 62/41)
```

Solving gives you back one solution, if there is one, even if there are infinitely many. To get all of them you would add elements of the nullspace (or kernel)

```
m = matrix(3,3, [2,3,5,  7,-4,0,  2,-5,1])
b = random_matrix(QQ, 3,2); b
[  0  -1]
[ -2 1/2]
[  2   2]
```

```
# solve m*x == b, where b is a *matrix*, so x is also a matrix
x = m.solve_right(b); x
[ -24/41  -15/82]
[ -43/82 -73/164]
[  45/82  23/164]
```

```
m*x == b
True
```

1.7 Computing invariants of matrices

```
m = matrix(3,3, [2,3,5,  7,-4,0,  2,-5,1])
```

```
m.determinant()
-164
```

```
m.rank()
3
```

```
m.nullity()
0
```

```
m.rref()  # watch out -- not the same as m.echelon_form(), in general...
[1 0 0]
[0 1 0]
[0 0 1]
```

```
m.echelon_form()  # this is the echelon form *over ZZ* -- no dividing \
    allowed
[  1   0  12]
[  0   1 103]
[  0   0 164]
```

```
m.characteristic_polynomial()
```

```
x^3 + x^2 - 41*x + 164
```

```
m.minimal_polynomial()
```

```
x^3 + x^2 - 41*x + 164
```

```
e = m.eigenvalues(); e
```

```
[-8.30939752116266?, 3.654698760581329? - 2.525839783704967?*I, 3.654698760581329? +
2.525839783704967?*I]
```

```
lamb = e[0]; lamb
```

```
-8.30939752116266?
```

```
# what's up with the "?"?
type(lamb)
```

```
<class 'sage.rings.qqbar.AlgebraicNumber'>
```

```
lamb.minpoly()
```

```
x^3 + x^2 - 41*x + 164
```

```
# really lamb is an *infinite* precision eigenvalue -- you can ask for \
   more (correct) digits...
lamb.numerical_approx(digits=50)
```

```
-8.3093975211626568404213346750855507682720825670697
```

```
# compute the eigespace decomposition
m.eigenspaces_right()
```

```
[
(-8.30939752116266?, Vector space of degree 3 and dimension 1 over Algebraic Field
User basis matrix:
[                  1 -1.624356993204802? -1.087265308309651?]),
(3.654698760581329? - 2.525839783704967?*I, Vector space of degree 3 and dimension 1 over
Algebraic Field
User basis matrix:
[                                          1   0.824678496602401? + 0.2721211925688051?*I
-0.1638673458451749? - 0.6684406722822763?*I]),
(3.654698760581329? + 2.525839783704967?*I, Vector space of degree 3 and dimension 1 over
Algebraic Field
User basis matrix:
[                                          1   0.824678496602401? - 0.2721211925688051?*I
-0.1638673458451749? + 0.6684406722822763?*I])
]
```

```
# Jordan Canonical Form
m.jordan_form(QQbar)
```

```
[                       -8.30939752116266?|                                          0|
0]
[-----------------------------------------+-------------------------------------------+-----
------------------------------------]
[                                        0|3.654698760581329? - 2.525839783704967?*I|
0]
[-----------------------------------------+-------------------------------------------+-----
------------------------------------]
```

```
[                                        0|
0|3.654698760581329? + 2.525839783704967?*I]
```

1.8 Vector spaces

```
# The vector space of all 3-tuples of rational numbers (i.e., vectors in \
    3-space with tail at the origin)
V = QQ^3
V
```

```
Vector space of dimension 3 over Rational Field
```

```
# These arise natural as spans, kernels (=nullspaces), etc.

m = matrix(QQ, 2,3, [2,3,5,  7,-4,0]); m
```

```
[ 2  3  5]
[ 7 -4  0]
```

```
# Compute the vector space of vector x such that m*x = 0
V = m.right_kernel(); V
```

```
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[     1    7/4 -29/20]
```

```
V.dimension()
```

```
1
```

```
V.basis()
```

```
[
(1, 7/4, -29/20)
]
```

```
# compute another 1-dimensional vector space
m = matrix(QQ, 2,3, [1,2,3,4,5,6]); m
W = m.right_kernel(); W
```

```
[1 2 3]
[4 5 6]
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

```
V.intersection(W)
```

```
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
V + W
```

```
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[     1      0 -23/75]
[     0      1 -49/75]
```

1.9 Linear algebra over finite fields (very important for coding theory)

```
# define a finite field
F = GF(7)
F
list(F)  # the elements of F
```

```
Finite Field of size 7
[0, 1, 2, 3, 4, 5, 6]
```

```
# define a matrix and vector over F
m = matrix(F, 3,3, [2,3,5,  7,-4,0,  2,-5,1]); m
v = vector(F, [10,5,2]); v

# notice how 7 == 0 below, since we are working in F.
```

```
[2 3 5]
[0 3 0]
[2 2 1]
(3, 5, 2)
```

```
# solve system
x = m.solve_right(v); x
```

```
(0, 4, 1)
```

```
m*x
```

```
(3, 5, 2)
```

1.10 Remarks about asymptotically fast algorithms

- All the problems I showed you above are trivial and you could do them by hand.
- One of the key things that distinguishes Sage from certain other famous (or not) programs is that it implements many asymptotically fast algorithms for exact linear algebra, i.e., these algorithms work even if the matrices are a bit bigger. (Tell Alan Steels store about him getting money from Knuth for provingwith Magma in the 90sthat asymptotically fast algorithms are practical, which Knuth said in his book they arent.)
- Some examples to get a sense of speed and capabilities.

```
m = random_matrix(ZZ, 100)
m[0]  # 0th row of our 100x100 matrix
```

```
(4, 0, -2, -3, -2, 0, -3, 0, -6, -1, -1, 0, -1, -1, 1, 7, 1, -1, 1, -1, -1, -1, 2, -2, 0,
1, 0, 0, -1, -1, 1, -1, -10, -1, -1, 0, 0, -1, 6, -1, 1, 1, -16, 0, 2, 0, 0, -1, -6, 3,
-6, 33, 0, -2, -1, -1, 10, 1, -1, -7, 1, -3, 5, -1, -1, 58, -1, -71, 1, -53, 1, 1, 3, -1,
-1, 0, 1, -2, 1, 2, -3, 7, -1, 1, 1, 0, 2, -2, 0, -3, 0, -2, 0, 0, 4, 26, -1, -1, 7, 3)
```

```
# LIE!!!
%timeit m.det()
```

```
625 loops, best of 3: 208 ns per loop
```

Note, the above 208ns is a very misleading. The reason is because m.det() caches the result of the computation.

And the timeit command takes the best of 3 the first time is long, and the others are short.

You can use m._clear_cache() to delete everything from this cache.

```
# very fast
%time m.det()
```

```
7499633318614058888185474291519197732376723921438320190328545406045380533537045546220592033
0290664701404280019444746165755070966366903724775165742367645118916871457433207223665448874535011757119353
CPU time: 0.00 s, Wall time: 0.00 s
```

```
%timeit m._clear_cache(); m.det()
```

```
5 loops, best of 3: 98.8 ms per loop
```

```
m = random_matrix(ZZ, 200)
%time m.det()
```

```
-646691847584372396631836739799171971121104965496404037950579677418443173444721320827008721
5466883316055249436832383933806608528788322157517667649034295136740462349530665294478158244
1536472093846800309570810483796344473084572291617332841990161013300998240186468559391490
4709195656418149611965737215309537377564006908046429166904480362680233450600553657651805430
2462460212386639973014474875803865693178937103840620096918540145860523366558001563040975374
8368355891777932

CPU time: 1.21 s, Wall time: 0.00 s
```

```
# PARI -- an open source "competitor" -- which doesn't implement \
    asymptotically
# fast algorithms... takes 61 seconds on what takes Sage only 1.2 seconds\
    :

m = random_matrix(ZZ,200)
g = gp(m)
%time g.matdet()
```

```
1357530467855922726709496391137311380106285712921476404836307055849207310366155945166554636
4892059426665284161324481250602586015675365959332063891023568314969405582094584728657916408
8583533400876342839197653862920340029840314309077747640520241247155187555406364215325475
2782878314112368151570679985904413807054172485226416618885286102646606927790316841955688887
9854754852063590745806139908324247219528065080874554735523391018138544274088976859055515234
14618216712

CPU time: 61.81 s, Wall time: 0.00 s
```

```
m = random_matrix(ZZ, 300)
%time m.det()
```

```
-366567155906273092592735564818817894629019399235313607938672816103851072016812204828802223
9376188802482079686431156260455841718472484096139084231715672156241176480764979044735302876
2036659749422497603071523179929464858478336910202053252562638678128526896500157785165054
2937941812612363408779293240498328958811629226449416973192228382242745285818647748167600986
3384642656889376151326165304017377957997407220772636582207052256675296038665398503285095728
53645437772781226045738555247653513277873417933246996481493662404681758885741639546017877
```

```
067935414108698721339178390706013060867742054317774668049577948380584476918673603456117521
789370051146973348953237116581299139570448159964895572091946444810118180145708719947061161
201369437878805232850725058812193721000
```

```
CPU time: 3.52 s, Wall time: 1.92 s
```

```
m = random_matrix(ZZ, 500)
%time m.det()
```

```
124423606683294696600781583201186670364081090679629871550747263911593699563304600113975106
006637438926245300666317272991416127232743525358214860248661632340534698307110256961284150
334602371359082678451466930740234053497907690214847841946714228871024453702913358051906553
672751845199994501289601655729344714124466268326978940337403873696787550459286421752275803
000430868794364713162247788199296713804673816061529982111368330976907303499149371844141861
483249551228348018747915898883070548000074607379821556006698346312104910289081418276565237
084679110885469228837210415246748530515318163819079324499609702034493625774048454669918948
175793582400113268364877152087305811352308603844882333544927754646418283331642736392555471
084409972965909937171485808101658235974556237069779420982320349600482102619460507626529352
919610584311004201888362008868611967283713710764282039745177066216009238800802195489886105
303107953171982116417134578447352102382110260366435268210957109205003707261053116550274829
708051167374356602763430758183685288764650541798459038762413705763028914453298207881569435
400944403310607026290606788346046531219242161553754683944465019655784506677458738897853766
131392497578177778729343958656326136443959443589114281194606483322583133532476780565483192
426621044371550001324273305373106906731970417540648056332567569977469084608255642633567362
91833236832957627046
```

```
CPU time: 10.63 s, Wall time: 3.56 s
```

Depending on time, say something about how m.det is so frickin fast it uses a whole bunch of surprising tricks.

- Strassen: matrix multiplication done by decomposition matrix into blocks and doing 7 multiplies instead of 8
- Multimodular: working modulo prime powers, and using the Chinese Remainder theorem.
- Cramers Rule: Solving a random linear system and looking at the denominator of the resulting vector, then fixing it.