2014-05-19.sagews

May 19, 2014

Contents

1 Math 480b Sage Course

1.1 Linear Algebra, part 2

1.2 May 19, 2014

Screencast: `http://youtu.be/B40SL4JtqPo`

Plan

- Questions
- Homework:
  - hw7, etc., collected this morning, and re-destributed for grading
  - hw8 assigned (should find it in your project). This is the last homework assignment.
- Topic: Exact linear algebra, part 2
  - vector spaces
  - linear algebra over finite fields and coding theory
  - remarks about asymptotically fast algorithms
- Wednesday and Friday: Graph theory, Group Theory

1.3 Vector spaces

```
RR^5
```

```
Vector space of dimension 5 over Real Field with 53 bits of precision
```

```
# 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
```

```
span(QQ, [[1,2,3], [4,5,6]])
```

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

```
# 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]
```

kernel = nullspace

```
# 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]
```

```
m.kernel?
```

```
   File: /usr/local/sage/sage-6.2.rc0/src/sage/matrix/matrix2.pyx
   Docstring:
      Returns the left kernel of this matrix, as a vector space or free
   module. This is the set of vectors "x" such that "x*self = 0".

   Note: For the right kernel, use "right_kernel()".  The method
     "kernel()" is exactly equal to "left_kernel()".

   INPUT:

   * "algorithm" - default: 'default' - a keyword that selects the
     algorithm employed.  Allowable values are:

     * 'default' - allows the algorithm to be chosen automatically

     * 'generic' - naive algorithm usable for matrices over any field

     * 'pari' - PARI library code for matrices over number fields or
       the integers
```

  * 'padic' - padic algorithm from IML library for matrices over
    the rationals and integers

  * 'pluq' - PLUQ matrix factorization for matrices mod 2

* "basis" - default: 'echelon' - a keyword that describes the format of the basis used to construct the left kernel. Allowable values are:

  * 'echelon': the basis matrix is in echelon form

  * 'pivot' : each basis vector is computed from the reduced row-echelon form of "self" by placing a single one in a non-pivot column and zeros in the remaining non-pivot columns. Only available for matrices over fields.

  * 'LLL': an LLL-reduced basis. Only available for matrices over the integers.

OUTPUT:

A vector space or free module whose degree equals the number of rows in "self" and contains all the vectors "x" such that "x*self = 0".

If "self" has 0 rows, the kernel has dimension 0, while if "self" has 0 columns the kernel is the entire ambient vector space.

The result is cached. Requesting the left kernel a second time, but with a different basis format will return the cached result with the format from the first computation.

Note: For much more detailed documentation of the various options see "right_kernel()", since this method just computes the right kernel of the transpose of "self".

EXAMPLES:

Over the rationals with a basis matrix in echelon form.

```
sage: A = matrix(QQ, [[1, 2, 4, -7, 4],
...                   [1, 1, 0, 2, -1],
...                   [1, 0, 3, -3, 1],
...                   [0, -1, -1, 3, -2],
...                   [0, 0, -1, 2, -1]])
sage: A.left_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1  2 -1]
[ 0  1 -1  1 -4]
```

Over a finite field, with a basis matrix in "pivot" format.

```
sage: A = matrix(FiniteField(7), [[5, 0, 5, 2, 4],
...                               [1, 3, 2, 3, 6],
...                               [1, 1, 6, 5, 3],
...                               [2, 5, 6, 0, 0]])
sage: A.kernel(basis='pivot')
Vector space of degree 4 and dimension 2 over Finite Field of size 7
User basis matrix:
[5 2 1 0]
[6 3 0 1]
```

The left kernel of a zero matrix is the entire ambient vector space whose degree equals the number of rows of "self" (i.e. everything).

```
sage: A = MatrixSpace(QQ, 3, 4)(0)
sage: A.kernel()
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

We test matrices with no rows or columns.

```
sage: A = matrix(QQ, 2, 0)
sage: A.left_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: A = matrix(QQ, 0, 2)
sage: A.left_kernel()
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
```

The results are cached. Note that requesting a new format for the basis is ignored and the cached copy is returned. Work with a copy if you need a new left kernel, or perhaps investigate the "right_kernel_matrix()" method on the transpose, which does not cache its results and is more flexible.

```
sage: A = matrix(QQ, [[1,1],[2,2]])
sage: K1 = A.left_kernel()
sage: K1
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: K2 = A.left_kernel()
```

```
sage: K1 is K2
True
sage: K3 = A.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: B = copy(A)
sage: K3 = B.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
User basis matrix:
[-2  1]
sage: K3 is K1
False
sage: K3 == K1
True
```

```
type(V)
```

```
<class 'sage.modules.free_module.FreeModule_submodule_field_with_category'>
```

```
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]
```

1.4 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]
```

```
a = F(2); a
```

```
2
```

```
parent(a)
```

```
Finite Field of size 7
```

```
F(2) / F(4)
```

```
4
```

```
1/F(4)
```

```
2
```

```
k = GF(9,'a')
```

```
k
```

```
Finite Field in a of size 3^2
```

```
list(k)
```

```
[0, a, a + 1, 2*a + 1, 2, 2*a, 2*a + 2, a + 2, 1]
```

```
# 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)
```

```
random_matrix(F,10)^10
```

```
[2 3 6 5 0 0 5 2 3 0]
[3 3 5 0 2 5 6 5 2 6]
[2 6 4 1 3 4 6 3 3 6]
[4 5 0 3 1 2 6 2 5 0]
[3 2 1 5 0 6 5 4 2 5]
[3 6 2 3 4 3 1 4 1 1]
[2 1 4 1 4 5 4 4 2 2]
[0 0 2 1 0 5 2 1 1 0]
[1 0 4 0 0 5 4 1 1 6]
[2 2 3 5 0 0 3 6 3 3]
```

```
random_matrix(ZZ,10)^10
```

```
[  -605161146634   2456513699588 -52246013770460 -10091601259800  12973421399331
19736861221697 138060495509084  29585601997262   -668397487583   2139291053030]
[ -1520858893783   2057111692869  -8812279501938   -885185221038   1916708657898
1513671501362   6533081571264  -4498315915416  -2424858594341    487474019239]
[ -1396017369048  -3714889261973    728352017995   1123735909851  -1694611297223
-3586115805981  28013232476975   1307228946129   4835411660398    476924714336]
[  1391378038256    502433228198   6301602661032  -2047894000389    291668149567
5393545151046  13637149619489  -2209126278188  -5749348867114  -1508406581700]
[  1222040503189    728715159250   6515090381548  -2608468784840    640923026976
7636787261897    556449069276  -2030358154988  -5550054117171  -1857838412226]
[ -5731125783528  -1263719924450 -19112354155374   1399566430385   1223204628979
-3925906481106  -4988676049549  -1726553157428   9465802255512   2384475335074]
[  -108908388353   2166614137445  -3272265937592    947730079891    361607323206
-3418760946501   -193902683535  -4551532012396  -2588304610198    757590907305]
[  -332284099700    715344035861  -6386957212058   3660450539330   -686586065250
-9933464685587 -25636410186597   3322175782880   5975122163476   2275969813947]
[ -1793065271475  -1293119804780  -4041786720494    378274105343   -158480996782
-1705221909098  18407122389313  -1803532888340   2052443228482    581642666650]
[  -390174372366    245583392270  -6602928423304   5274867078283  -1593719237520
-14644117625936 -23721711387145   3631840570305   7882541584633   3043356709760]
```

In fact, Sage has extensive coding theory functionality. (See `http://www.sagemath.org/doc/reference/coding/sage/coding/code_constructions.html` and `http://www.sagemath.org/doc/reference/coding/index.html`)

- A code is a subspace of a finite dimensional vector space.
- One encodes messages as elements of this subspace.
- When a message is corrupted (say one bit flipped) it becomes something not in the subspace.
- Decoding involves finding the closest vector in the subspace to what you get.

```
codes.
```

```
C = codes.HammingCode(3,GF(2)); C
```

```
Linear code of length 7, dimension 4 over Finite Field of size 2
```

```
C.basis()
```

```
[(1, 0, 0, 0, 0, 1, 1), (0, 1, 0, 0, 1, 0, 1), (0, 0, 1, 0, 1, 1, 0), (0, 0, 0, 1, 1, 1,
1)]
```

```
span(C.basis())
```

```
Vector space of degree 7 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

```
len(C)
```

```
16
```

```
for v in C:
    print v
```

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

```
# a corrupted message
corrupted_message = [0, 1, 1, 0, 0, 0, 0]
# check this out:
C.decode(corrupted_message)
```

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

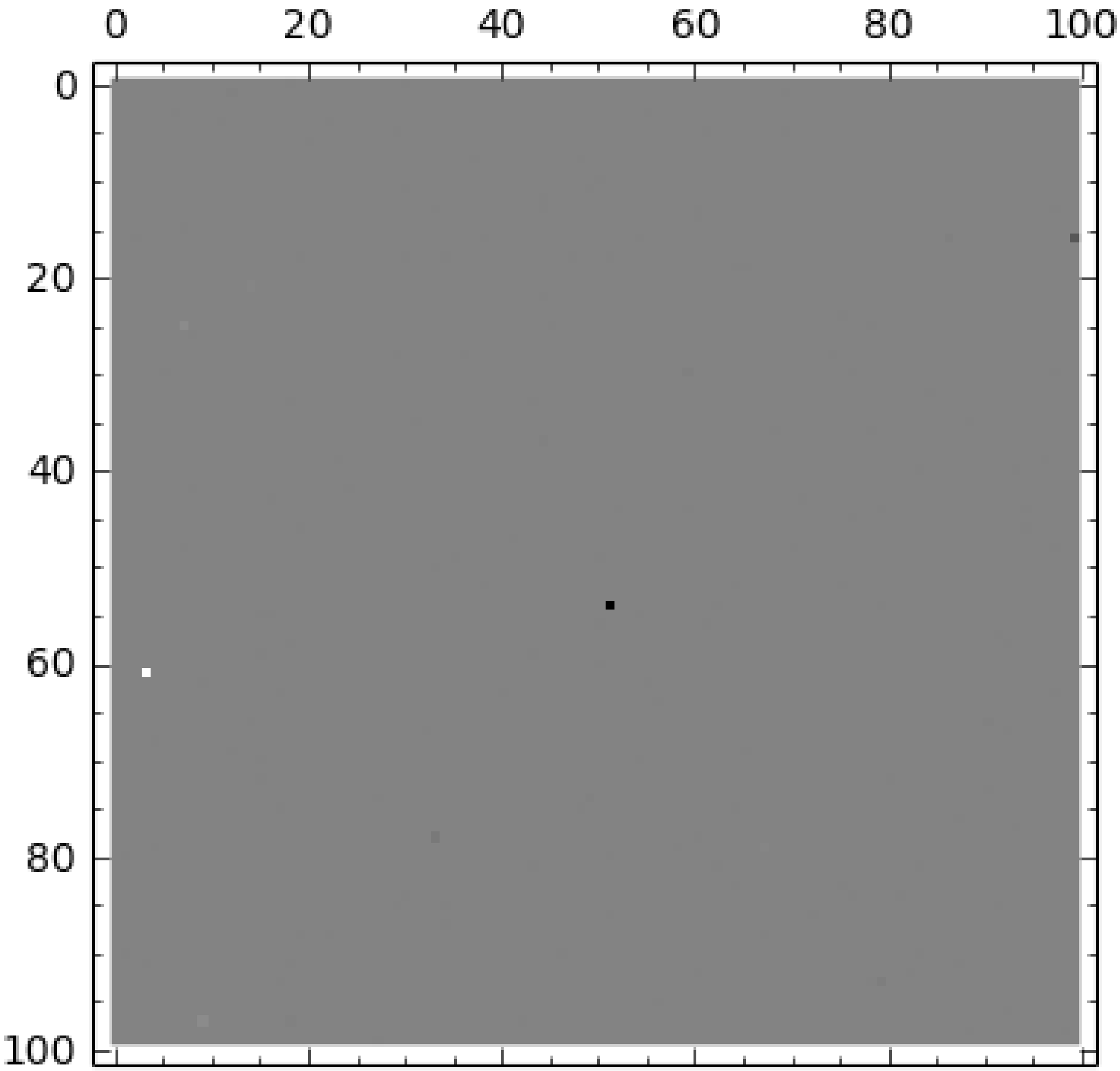
1.5 Remarks about asymptotically fast algorithms

- 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
```

```
(-2, -1, 0, 0, -2, -4, 2, -2, -1, -1, -3, 1, -1, 0, 0, 3, 0, -1, -98, 6, -1, -1, 1, 0, -4,
2, -8, 6, 2, -1, -86, -4, 8, 0, -3, -2, 1, 4, 1, -1, -1, 0, 0, -2, 1, 1, 1, -1, -1, -2, 1,
-7, 1, 1, -4, -6, 0, 1, 0, 2, -9, -4, 1, -2, -1, -1, 0, -2, -1, -6, 1, -5, 1, 1, 0, 0, 0,
-1, -5, 0, 3, 0, -2, -1, -14, -2, 0, -1, 1, -1, 4, -1, 0, -2, 0, 1, -1, -2, 2, -1)
```

```
matrix_plot(m)
```




```
# 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.

```
m._clear_cache()
```

```
# very fast
t = walltime()
s = cputime()
%time m.det()
print walltime(t)   # == walltime() - t
print cputime(s)    # == cputime() - s
```

```
19273378847246561136798939841246743706113269253095433516380247143875116254260627819776125922605943209074084684455334990841414838714196200782807182967459629589393838322724588770200935009855338887861294091463
CPU time: 0.04 s, Wall time: 0.04 s
0.0398569107056
0.039854
```

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

```
25 loops, best of 3: 38.3 ms per loop
```

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

```
3440945270705876453228189508139272203860783109504987559008420062561686327693325818992006320635865243878670442425196135263320888394104965326052953229676521723820685066287406261778384503863106916173289504013322281844484555963647059010626429641356976712001049269333298064625691681916705326977571863594036071467370490799288425594423901466184230742280594369987266294132166091765267259412133969103826215498451843248442026761962866425100045012182086424217282790441

CPU time: 0.46 s, Wall time: 0.46 s
```

```
m = random_matrix(ZZ, 400)
%time m.det()
```

```
339780396119534618027700911725863144185882752839337178067878084146999915129357626232887343206890788992948208201621497762372366428669941166703245185719660250668791903610356720665523683528897746977550902330889251053516702580861098412204357824298722846206863573241355926733671968116246557917327592987383233212027049247385386563929918846584113554733423439252223027256976342582032836088815521978060135878804818678134553677572929011610888949130668568322189130712485891124889629895200460377049839769717031268458120784028206403292522916338778042107819092701962556732644118031743401543281577459094642994366199883206762727986036616445603329444608064447440844249009686683761429799575191234764972182998015118150409965551349234506976184008050384905919714243922070834869489246121198068365580595609573356233483574281225069450933083105034410818071519395427294173353606919621593014154505239224205831062005710713913901016351426504932742341024129136470932138606799444137247546394770410039259810969768121793972744195680720011310186723336234477150880242997268661114442

CPU time: 2.27 s, Wall time: 2.26 s
```

```
m = random_matrix(ZZ, 800)
%time m.det()
```

```
-4822801924860893342113540293598058645144777146420138754552300296290888160481251691860360
```

```
1820404157816577745712188486783726591764210837033504127598909194805390899694358686889275142
2314935632546189728503086312282373309228640327694949554530079795798828176313895655094070082
4646122396364452113094933468581547567314667047365717399527531649026253436884804968057385482
1896414889703757693068812708082208532493498082570510785203446253361698569128140172975027202
0161734570364178078251989338750718286411707126256701615424230597167781435855949380515878782
7579600555888433886352535883270736701164527962369531090720544769228402470317571213523954182
4013596794483629579391949823393445834078956750656084785884763286992822906335454295046762272
8924775536501053845595148649426151500875901792782667386417943979817985693177802477064146692
2744049260931220634763987636457051240122339943982633332036839286990489746036882213181085662
3535030037255798300474827829356822667830493821620167021481740596252868892703330893610879352
0458911025795580982054141799365053760917448033555117717002846345344119680902324745451017862
6110770833707646569197541761371701192973912950481470416023217696975739393474697110056231342
4575098429572440532641909663225259383683817492204572547671294245168342674414888439965845972
7142019325674410154482544474955732297598638828058080217224406349019608950695844215689289792
6975175984600247530682611134671982298452072463886535576655858316385459816592191334487866962
2897802509442011285651506423146245962787534929850374649200757244678309829442817801260877232
2921884170291428983997629087962939046286083144464608902536283715335729028658597264556062622
6515666989545963457489685162969922311917645095564927381767097396535070431811524774200154332
9322883305103363054207453619272348437611407665113793068312956802153630497554762185241878422
5962153625487701810835458340808379554192389498973065601343830001603856974826627367503380282
5712690190958497047412598375944848243060572587047185270527017875399847685028883043112815042
5281204891639894994090182720893453607997350105183870978215834714133207642020243849654935352
4283481568038892051299741352163017655550310104197271711419528556141459502167427406771797732
4276238625442681983852887870242428032493420426623185383795219843025141455458502442673989662
45329184395566071630844551041802030157160375157649550029058213099953759678616

CPU time: 14.23 s, Wall time: 14.19 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()
```

```
1357530467855922726709496391137311380106285712921476404836307055849207310366155945166554632
6489205942666528416132448125060258601567536595933206389102356831496940558209458472865791642
0885835334008763428391976538629203400298403143090777476405202412471551875554063642153254752
2782878314112368151570679985904413807054172485226416618885286102646606927790316841955688882
7985475485206359074580613990832424721952806508087455473552339101813854427408897685905551522
3414618216712

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

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

```
-3665671559062730925927355648188178946290193992353136079386728161038510720168122048288022
3937618880248207968643115626045584171847248409613908423171567215624117648076497904473530282
7620366597494224976030715231799294648584783369102020532525626386781285268965001577851650542
2937941812612363408779293240498328958811629226449416973192228382242745285818647748167600982
6338464265688937615132616530401737795799740722077263658220705225667529603866539850328509572
```

```
2853645437772781226045738555247653513277873417933246996481493662404681758885741639546017870679354141086987213391783907060130608677420543177746680495779483805844769186736034561175217893700511469733489532371165812991395704481599648955720919464448101181801457087199470611612013694378788052328507250588121937210000
```

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

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

```
12442360668329469660078158320118667036408109067962987155074726391159369956330460011397510600663743892624530066631727299141612723274352535821486024866163234053469830711025696128415033460237135908267845146693074023405349790769021484784194671422887102445370291335805190655367275184519999450128960165572934471412446626832697894033740387369678755045928642175227580300043086879436471316224778819929671380467381606152998211136833097690730349914937184414186148324955122834801874791589888307054800007460737982155600669834631210491028908141827656523708467911088546922883721041524674853051531816381907932449960970203449362577404845466991894817579358240011326836487715208730581135230860384488233354492775464641828333164273639255547108440997296590993717148580810165823597455623706977942098232034960048210261946050762652935291961058431100420188836200886861196728371371076428203974517706621600923880080219548988610530310795317198211641713457844735210238211026036643526821095710920500370726105311655027482970805116737435660276343075818368528876465054179845903876241370576302891445329820788156943540094440331060702629060678834604653121924216155375468394446501965578450667745873889785376613139249757817777872934395865632613644395944358911428119460648332258313353247678056548319242662104437155000132427330537310690673197041754064805633256756997746908460825564263356736291833236832957627046
```

```
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.