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: REMIND ME. And watch to see if it crashes.

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

```
# 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]
```

```
type(V)
```

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

```
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]
```

```
# 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)
```

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.

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

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
```
```
(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()
```

```
74996333186140588881854742915191977323767239214383201903285454060453805335370455462205920330290664701404280019444746165755070966366903724775165742367645118916871457433207223665448874535011757119353
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()
```

```
-64669184758437239663183673979917197112110496549640403795057967741844317344472132082700872154668833160552494368323839338066085287883221575176676490342951367404623495306652944781582441536472093846800309570810483796344473084572291617332841990161013300998240186468559391490470919565641814961196573721530953737756400690804642916690448036268023345060055365765180543024624602123866399730144748758038656931789371038406200969185401458605233665580015630409753748368355891777932

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()
```

```
13575304678559227267094963911373113801062857129214764048363070558492073103661559451665546364892059426665284161324481250602586015675365959332063891023568314969405582094584728657916408858353340087634283919765386292034002984031430907774764052024124715518755540636421532547527828783141123681515706799859044138070541724852264166188852861026466069277903168419556888879854754852063590745806139908324247219528065080874554735523391018138544274088976859055515234146182167 12

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

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

```
-36656715590627309259273556481881789462901939923531360793867281610385107201681220482880222393761888024820796864311562604558417184724840961390842317156721562411764807649790447353028762036659749422497603071523179929464858478336910202053252562638678128526896500157785165054293794181261236340877929324049832895881162922644941697319222838224274528581864774816760098633846426568893761513261653040173779579974072207726365822070522566752960386653985032850957285364543777278122604573855524765351327787341793324699648149366240468175888574163954601787
```

```
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.