Sage 3: Lineare Algebra

Wiederholung: Sequenzielle Datentypen

Listen und Tupel

Wiederholung zu Listen, einem wichtigen Hilfs-Datentyp für Vektoren und Matrizen; Erstellen einfacher Listen und Auslesen von einem Eintrag (Indizierung beginnt bei 0):

```
l = [1, 2, 3]
print l
print l[0]
```

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

Listen können auch gemischte Datentypen enthalten:

```
[1, 2, "drei", 4.0]
```

```
[1, 2, 'drei', 4.00000000000000]
```

Auslesen von Teillisten ("Slices"), zwischen einem Start- (inklusive) und einem End-Punkt (exklusive):

```
l[0:2]
```

```
[1, 2]
```

Einzelne Einträge überschreiben:

```
l[1] = 9; l
```

```
[1, 9, 3]
```

Tupel sind ein verwandter Datentyp mit ähnlichen Zugriffs-Operatoren, allerdings "immutable" (nicht mehr änderbar). Angabe wie Listen, aber mit runden statt eckigen Klammern, die in manchen Situationen auch weggelassen werden können:

```
t1 = (1,2,3)
t2 = 1, 2, 3
print t1
print t2[0]
print t2[0:2]
```

```
(1, 2, 3)
1
(1, 2)
```

Tupel können auch auf der linken Seite von Zuweisungen stehen, z.B. um mehrere Variablen gleichzeitig zuzuweisen:

```
x, y = 100, 200
print x
print y
```

```
100
200
```

Bereits erstellte Tupel können so auch wieder "entpackt" werden in einzelne Variablen:

```
a, b, c = t1
print a
print b
print c
```

```
1
2
3
```

Eingebaute Funktionen für einfache Zähl-Listen (die unteren beiden Varianten funktionieren nur in Sage, nicht in reinem Python) durch Angabe des höchsten Wertes (exklusive bei der range-Funktion, inklusive bei Klammernnotation) sowie ggf. des Startwerts und einer Schrittweite:

```
print range(10)
print range(1,10)
print range(10,1,-1)
print [0..10]
print [10..1, step=-1]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 9, 8, 7, 6, 5, 4, 3, 2]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

List Comprehensions

Listen mit etwas komplexeren Erstellungsregeln kann man stückweise in einer Schleife zusammenbauen:

```
l = []
for x in range(10):
    l.append(x^2)
print l
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Eine einfachere und effizientere Notation für solche Konstruktionen bieten "List Comprehensions", bei denen die Schleife (ggf. auch verschachtelt und/oder mit ifs gefiltert) direkt in die Listendefinition verpackt wird:

```
l = [x^2 for x in range(10)]
print l
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[10*x + y for x in [1..5] for y in [0,5]]
```

```
[10, 15, 20, 25, 30, 35, 40, 45, 50, 55]
```

```
[10*x + y for x in [1..5] for y in [0..5] if y > x]
```

```
[12, 13, 14, 15, 23, 24, 25, 34, 35, 45]
```

Generatoren

Listen sind ein sehr wichtiges Werkzeug in Sage/Python, aber in vielen Situationen unnötig ressourcenintensiv. Beispielsweise ginge es zwar relativ flott, über eine große Anzahl von Werten in einer Schleife zu iterieren; aber wenn alle Werte zuerst in einer riesigen Liste im Speicher abgelegt werden müssen, kann das zu ineffizient werden. NICHT AUSFÜHREN:

```
for x in range(1000000000):
    if is_prime(x) and is_prime(x+2):
        print x, x+2
```

Ein Ersatz für die range-Funktion ist xrange, womit über Zählwerte iteriert werden kann, ohne vorher die gesamte Liste im Speicher abzulegen; die Zählwerte werden "on the fly" nach jedem Schleifendurchlauf hochgezählt, wie in C üblich:

(nur bei Langeweile ausführen; ggf. mit "Action" - "Interrupt" wieder abbrechen!)

```
for x in xrange(1000000000):
    if is_prime(x) and is_prime(x+2):
        print x, x+2
```

Allgemein kann dieses Verhalten ("nicht die ganze Liste erzeugen und ablegen, sondern nach einer Regel schrittweise durchlaufen") mit Hilfe von Generators erreicht werden. Analog zu List Comprehensions gibt es auch Comprehensions, die solche Generatoren erzeugen, die wie Listen (aber speichereffizienter) in Schleifen o.ä. verwendet werden können (mit runden statt eckigen Klammern - nicht verwechseln mit Tupeln, für die es keine solchen Comprehensions gibt!); auch ohne Schleife können mit der "next"-Methode die Werte des Generators händisch durchlaufen werden:

```
gen = (10*x + y for x in [1..5] for y in [0..9] if y > x)
print gen.next()
print gen.next()
print gen.next()
print gen.next()
```

```
12
13
14
15
```

Statt mit Comprehensions können Generatoren auch mit ausgeschriebenen Schleifen erzeugt werden, und zwar mit Hilfe von Generator Functions. Diese sehen aus wie normale Funktionen (Keyword "def"), enthalten aber das Keyword "yield". Beim Aufruf von "next" wird die Ausführung der Generator Function vom letzten Status bis zum nächsten "yield" ausgeführt, dann unterbrochen. Der mit yield angegebene Wert wird als Ergebnis von "next" zurückgegeben. Beim nächsten Aufruf von "next" wird wieder an der aktuellen Stelle fortgesetzt. Um den Generator zu verwenden, einfach den Rückgabewert der Generator Function (der ein Generator ist) verwenden:

```
def mygen():
    for x in [1..5]:
        for y in [0..9]:
            if y > x:
                yield 10*x + y
gen2 = mygen()
print gen2.next()
print gen2.next()
print gen2.next()
print gen2.next()
```

```
12
13
14
15
```

Konstruktion von Matrizen

Die Erstellung einfacher Vektoren und Matrizen aus Listen ist aus der letzten Vorlesungseinheit

bekannt:

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

```
(1, 2, 3)
```

```
A = matrix([[1,2],[3,4]]); A
```

```
[1 2]
[3 4]
```

Anstatt die Matrix aus "Listen von Listen" (Listen von Zeilenvektoren) zu erstellen, können auch alle Einträge in eine einzige "flache" Liste zusammengefasst werden, unter Angabe der gewünschten Dimensionen (zuerst Anzahl der Zeilen, dann Anzahl der Spalten):

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

```
[1 2]
[3 4]
```

Sage ist dabei immer "zeilenorientiert", d.h. wenn möglich werden Vektoren und Listen als Zeilenvektoren interpretiert, und die lange Liste wird als Aneinanderreihung von Zeilen gesehen.

Nicht nur Matrizen aus Zahlen können erstellt werden, sondern auch z.B. mit symbolischen Variablen (oder mathematischen Funktionen, o.ä.):

```
var('x y z')
Asymb = matrix([[x, 2*x], [x+y, z]])
show(Asymb^2)
```

$$\begin{pmatrix} 2\,(x+y)x+x^2 & 2\,x^2+2\,xz \\ (x+y)x+(x+y)z & 2\,(x+y)x+z^2 \end{pmatrix}$$

Das Verhalten einiger Funktionen (z.B. Zeilenstufenform, Eigenvektoren) hängt davon ab, über welchem Datentyp ("welchem Ring") die Matrizen interpretiert werden, z.B. als ganzzahlig oder als reelle Zahlen. Die wichtigsten "Ringe" sind großteils aus den vorherigen Einheiten bekannt:

```
NN, ZZ, QQ, RR, RDF, CC, CDF, SR
```

```
(Non negative integer semiring,
 Integer Ring,
 Rational Field,
 Real Field with 53 bits of precision,
 Real Double Field,
 Complex Field with 53 bits of precision,
 Complex Double Field,
 Symbolic Ring)
```

Bei der Matrix aus symbolischen Einträgen kann etwa der "symbolic ring" SR als Basistyp angegeben werden:

```
matrix(SR, [[x, 2*x], [x+y, z]])
```

```
[    x   2*x]
[x + y     z]
```

Versucht man einen "zu eng gefassten" Typ anzugeben (z.B. reelle Zahlen, obwohl komplexe Einträge vorkommen, oder hier rationale Zahlen, obwohl symbolische Variablen vorkommen), wird man über eine Fehlermeldung informiert:

```
matrix(QQ, [[x, 2*x], [x+y, z]])
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: unable to convert x to a rational
```

Für typische Standard-Matrix-Konstruktionen gibt es fertige Funktionen zur Konstruktion, etwa für die Einheitsmatrix einer bestimmten Größe (bei einer quadratischen Größe reicht die Angabe einer Dimension):

```
identity_matrix(3)
```

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

Null-Matrix mit 3 Zeilen und 5 Spalten:

```
zero_matrix(3,5)
```

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

Zufällige Matrix einer bestimmten Größe, mit einem bestimmten Basistyp (die genaue Zufallsverteilung hängt vom Typ ab, z.B. gleichverteilte Werte im Intervall $[0,1]$ für reelle Zahlen):

```
random_matrix(RR,3,5)
```

```
[-0.00108129581007987    0.336734500152562   -0.523085276728320
0.715783741632802    0.290620520751879]
[   0.286767833870595    0.635393566379757    0.824769190428260
-0.506689714003272   -0.253077337569762]
[   0.543661711570265    0.233326792867555   -0.431378954623786
-0.684594045000022   -0.258626245082391]
```

Riesige Matrizen (mit vielen 0-Einträgen) werden oft "sparse" gespeichert und können auch so definiert werden:

```
matrix(ZZ, 20, 80, {(5,9):30, (15,77):-6})
```

```
20 x 80 sparse matrix over Integer Ring (use the '.str()' method to
see the entries)
```

Zugriff auf Matrizen

Zugriff auf Einträge von Vektoren analog zu Listen:

```
print v
print v[0]
print v[0:2]
```

```
(1, 2, 3)
1
(1, 2)
```

Bei Matrizen ist ähnlicher Lese- und Schreibzugriff auf Einträge möglich; Vorsicht beim Schreiben, hier muss eine spezielle Syntax für die Indizes verwendet werden:

```
print A
print A[0,1]
```

```
A[0,1] = 999
print A
```

```
[1 2]
[3 4]
2
[  1 999]
[  3   4]
```

Normaler verschachtelter Listen-Zugriff funktioniert nur zum Lesen, nicht zum Schreiben:

```
print A[0][1]
A[0][1] = 9999
```

```
999
Traceback (click to the left of this block for traceback)
...
ValueError: vector is immutable; please change a copy instead (use
copy())
```

Auslesen von Teilmatrizen mit "Slicing":

```
A[0:1,0:2]
```

```
[  1 999]
```

Ausgabe als Liste von Zeilen- oder Spaltenvektoren, oder als gesamte "flache" Liste:

```
print A.rows()
print A.columns()
print A.list()
```

```
[(1, 999), (3, 4)]
[(1, 3), (999, 4)]
[1, 999, 3, 4]
```

Operationen und Eigenschaften

```
A = matrix(RDF, [[1,2], [3,4]])
```

Einfacher Operatoren funktionieren wie üblich, sofern die Matrix-Dimensionen zusammenpassen:

```
A * A + A - 2*A
```

```
[ 6.0  8.0]
[12.0 18.0]
```

Beispiel mit unpassenden Dimensionen (dreidimensionaler Vektor):

```
A * v
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: unsupported operand parent(s) for '*': 'Full MatrixSpace
of 2 by 2 dense matrices over Real Double Field' and 'Ambient free
module of rank 3 over the principal ideal domain Integer Ring'
```

```
A * v[0:2]
```

```
(5.0, 11.0)
```

Elementare Zeilen- (und Spalten-)Umformungen:

```
A = matrix(RDF, [[1,2], [3,4]]); A
```
```
[1.0 2.0]
[3.0 4.0]
```

... Zeilenvertauschungen...

```
A.swap_rows(0,1); A
```
```
[3.0 4.0]
[1.0 2.0]
```

... Zeilen skalieren mit einem Faktor...

```
A.rescale_row(0,5); A
```
```
[15.0 20.0]
[ 1.0  2.0]
```

... Vielfaches einer Zeile zu einer anderen addieren (zuerst Ziel-Zeile, dann Quell-Zeile, dann Faktor angeben):

```
A.add_multiple_of_row(0, 1, 0.5); A
```
```
[15.5 21.0]
[ 1.0  2.0]
```

Potenzen und Inverses einer Matrix (sofern sie invertierbar ist):

```
print A^2
print A^-1
print A.inverse()
```
```
[261.25  367.5]
[  17.5   25.0]
[ 0.19999999999999998  -2.0999999999999996]
[-0.09999999999999999   1.5499999999999998]
[ 0.19999999999999998  -2.0999999999999996]
[-0.09999999999999999   1.5499999999999998]
```

Auch symbolische Matrizen können invertierbar sein:

```
show(Asymb^-1)
```
$$\begin{pmatrix} -\frac{2\,(x+y)}{(2\,x+2\,y-z)x}+\frac{1}{x} & \frac{2}{2\,x+2\,y-z} \\ \frac{x+y}{(2\,x+2\,y-z)x} & -\frac{1}{2\,x+2\,y-z} \end{pmatrix}$$

Determinante:

```
det(A)
```
```
9.999999999999996
```

Norm eines Vektors (Euklidische Norm sowie 1-Norm und Maximumsnorm):

```
print v.norm()
print v.norm(2)
print v.norm(1)
print v.norm(Infinity)
```

```
sqrt(14)
sqrt(14)
6
3
```

Lösen von Gleichungssystemen

Einige der wichtigsten Anwendungen von Matrizen drehen sich um das Lösen von (linearen) Gleichungssystemen, typischerweise von der Form $A \cdot x = b$ mit Matrix A, Vektor b und unbekanntem Vektor x:

```
A = matrix([[1,2],[3,4]])
b = vector([1,2])
```

```
(1, 2)
```

Die Form $A \cdot x = b$ entspricht einer "spaltenorientierten" Modellierung (mit Spaltenvektoren x und b, und einer spaltenweise gewichteten Matrix A). Durch die zeilenorientierte Definition von Sage verwenden aber viele Sage-Funktionen standardmäßig eher die zeilenorientierte Variante $x \cdot A = b$, mit Zeilenvektoren x und b. Die mit "vector" deklarierten Vektoren können sowohl die Rolle von Zeilen- als auch von Spaltenvektoren übernehmen, aber die Multiplikationsreihenfolge ist natürlich relevant. Daher gibt es von vielen Sage-Funktionen zwei Varianten, eine mit dem Namenspostfix "_right" und eine mit "_left". Typischerweise entspricht "_right" der am Papier gewohnten spaltenorientierten Version (Multiplikation des unbekannten Vektors rechts, also Gleichungssystem $A \cdot x = b$), und "_left" der zeilenorientierten Version (oft Default).

$A \cdot x = b$ löst man also folglich so:

```
A.solve_right(b)
```

```
(0, 1/2)
```

Und $x \cdot A = b$ so:

```
A.solve_left(b)
```

```
(1, 0)
```

Für $A \cdot x = b$ gibt es auch einen eigenen Operator:

```
A \ b
```

```
(0, 1/2)
```

Die solve-Funktion liefert immer entweder eine einzelne Lösung oder eine Fehlermeldung (=keine Lösung) zurück. Aber lineare Gleichungssysteme können natürlich auch mehr Lösungen haben - insbesondere oft unendlich viele ("Lösungsraum")! Wie bekommt man die anderen Lösungen?

Schauen wir zuerst "homogene" Gleichungen $A \cdot x = 0$ an:

```
A = matrix([[1,2,3], [2,4,5], [0,0,1]]); A
```

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

In diesem Beispiel hat $A \cdot x = 0$ eindeutig mehrere Lösungen (denn die Spalten bzw. Zeilen sind linear abhängig). Dennoch bekommen wir mit solve nur die "Triviallösung":

```
A \ vector([0,0,0])
```

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

Was wir eigentlich gerne hätten ist der ganze Unterraum von Lösungen, der "Kern" der Matrix (bzw. der dazugehörigen linearen Abbildung). Sage kann diesen Kern in Form einer Basis des Lösungsraums angeben und auch damit rechnen:

```
V = A.right_kernel()
print V
```

```
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 2 -1  0]
```

Der Kern enthält z.B. auch den Lösungsvektor $(4, -2, 0)$:

```
vector([4,-2,0]) in V
```

```
True
```

```
vector([4,-2,1]) in V
```

```
False
```

Analog bekommt man auch den Lösungsraum von $x \cdot A = 0$:

```
A.left_kernel()
```

```
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 2 -1 -1]
```

Bei inhomogenen Systemen ($b \neq 0$) setzen sich bekanntlich alle Lösungen zusammen als Addition von irgendeiner Partikulärlösung (Ergebnis von solve) plus einem Vektor aus dem Kern. Man kann also den Lösungsraum durch Angabe der Kern-Basis plus einer Partikulärlösung allgemein angeben, und beliebig viele Lösungen eines (unterbestimmten) Systems ausgeben, indem man zufällige Vektoren aus dem Kern mit der Partikulärlösung kombiniert:

```
b = vector([1,2,0]); b
```

```
(1, 2, 0)
```

```
c = A \ b
V = A.right_kernel()
print "Partikulärlösung:", c
print "Kern:\n", V
print "Basis des Kerns:", V.basis()

print "Probe mit zufälligen kombinierten Lösungen:"
for i in range(10):
    x = V.random_element() + c
    print A * x == b
```

```
Partikulärlösung: (1, 0, 0)
Kern:
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 2 -1  0]
Basis des Kerns: [
(2, -1, 0)
]
Probe mit zufälligen kombinierten Lösungen:
True
True
True
True
True
True
```

```
True
True
True
True
```

Auf der rechten Seite des Gleichungssystems muss nicht unbedingt ein Vektor stehen, es kann auch eine Matrix von geeigneter Dimension sein (dann ist auch die Unbekannte eine Matrix): $A \cdot X = B$ bzw. $X \cdot A = B$.

Eigenvektoren

Auch bei Eigenvektoren gibt es die Unterscheidung von "Rechtseigenvektoren" ($A \cdot x = \lambda \cdot x$, die üblichere Variante) und "Linkseigenvektoren" ($x \cdot A = \lambda \cdot x$) - die Eigenwerte sind gleich!

```
A.eigenvalues()
```
```
[5, 1, 0]
```

Das Ergebnis von "eigenvectors" ist eine etwas unübersichtliche Liste von Tupeln. Jedes Tupel besteht aus einem Skalar (dem Eigenwert), einer Liste von Tupeln (die Liste der Basisvektoren des Eigenraums, d.h. der Eigenvektoren zu diesem Wert), sowie noch einem Skalar (der algebraischen Vielfachheit des Eigenwerts). Die folgende Liste enthält also drei Tupel (eines pro Eigenwert), jedes mit dem Eigenwert, einer Liste mit nur einem Eigenvektor, sowie jeweils der algebraischen Vielfachheit 1:

```
A.eigenvectors_right()
```
```
[(5, [
  (1, 2, 0)
  ], 1), (1, [
  (1, 6, -4)
  ], 1), (0, [
  (1, -1/2, 0)
  ], 1)]
```

Für die Überprüfung händischer Rechnungen ist es oft nützlich, sich das charakteristische Polynom auszugeben ($\det(A - \lambda I)$):

```
A.charpoly()
```
```
x^3 - 6*x^2 + 5*x
```

Verfügbar auch in bereits faktorisierter Form (Vorsicht, es hängt vom Basisring ab, welche Nullstellen hier herausfaktorisiert werden - z.B. je nach Typ nur reelle Nullstellen):

```
A.fcp()
```
```
(x - 5) * (x - 1) * x
```

Vektorräume

(kein Klausurstoff)

Aus RDF $\approx \mathbb{R}$, QQ $= \mathbb{Q}$ usw. kann man auch Vektorräume erstellen:

```
QQ^3
```
```
Vector space of dimension 3 over Rational Field
```
```
[sqrt(2), 2/3, 4] in QQ^3
```
```
False
```

```
V = QQ^3
V.basis()
```

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

Unterräume durch Angeben einer Basis definieren (intern wird nicht die vom User angegebene Basis gespeichert, sondern eine äquivalente):

```
v1 = vector([1,2,0])
v2 = vector([0,0,3])
v3 = vector([2,6,0])
```

```
W = V.subspace([v1, v2])
W.basis()
```

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

```
U = V.subspace([v3]); U
```

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

Schnittmenge von Vektorräumen (in diesem Fall nulldimensional, leere Basis):

```
W.intersection(U)
```

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

Testen, ob W Unterraum von U ist (bzw. W von V):

```
W <= U
```

```
False
```

```
W <= V
```

```
True
```

Summe von Vektorräumen bilden - der Ergebnis-Raum wird wieder durch eine Basis angegeben:

```
W + V
```

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

Die Koordinaten eines Vektors in einem Vektorraum in Bezug auf seine Basis angeben (Achtung: in Bezug auf die gespeicherte, nicht die vom User angegebene Basis):

```
W.coordinates([2,4,3])
```

```
[2, 3]
```

Zufälliges Element aus einem Vektorraum picken:

```
W.random_element()
```

```
(1/25, 2/25, 1/19)
```

Lineare Abhängigkeit von Vektoren testen - das Ergebnis ist entweder eine Liste von Linearkombinationen, die 0 ergeben (falls abhängig) oder eine leere Liste (unabhängig):

```
V.linear_dependence([v1, v2, v3])
```

```
[

]
```

```
V.linear_dependence([v1, v2, v1+2*v2])
```

```
[
(1, 2, -1)
]
```

Lineare Transformationen

(kein Klausurstoff)

```
var('x y z')
f(x, y, z) = [x + 2*y, y + z]
Tf = linear_transformation(QQ^3, QQ^2, f); Tf
```

```
Vector space morphism represented by the matrix:
[1 0]
[2 1]
[0 1]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
Mf = Tf.matrix(side='right'); Mf
```

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

```
T2 = linear_transformation(Mf, side='right'); T2
```

```
Vector space morphism represented by the matrix:
[1 0]
[2 1]
[0 1]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
T2.is_equal_function(Tf)
```

```
True
```

```
Tf.is_surjective()
```

```
True
```

```
Tf.is_injective()
```

```
False
```

```
Tf.kernel()
```

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

Zerlegungen usw.

(kein Klausurstoff)

Jordan-Form (entspricht Diagonalmatrix, falls A eine diagonalisierbare Matrix ist):

```
A = matrix([[1,2,3], [2,4,5], [0,0,1]]); A
A.jordan_form()
```

```
[5|0|0]
[-+-+-]
[0|1|0]
[-+-+-]
[0|0|0]
```

Auch Transformation für die Diagonalisierung angeben:

```
A.jordan_form(transformation=True)
```

```
(
[5|0|0]
[-+-+-]
[0|1|0]  [   1    1    1]
[-+-+-]  [   2    6 -1/2]
[0|0|0], [   0   -4    0]
)
```

QR-Zerlegung (approximiert, daher Rechnung in RDF $\approx \mathbb{R}$ statt QQ)::

```
A = matrix(RDF, [[1,2,3], [2,4,5], [0,0,1]])
Q, R = A.QR()
print Q
print " * "
print R
print " = "
print Q * R
```

```
[-0.44721359549995787  -0.8944271909999157                  0.0]
[ -0.8944271909999157   0.4472135954999581                  0.0]
[                -0.0                   0.0                  1.0]
 *
[    -2.23606797749979     -4.47213595499958    -5.813776741499453]
[                  0.0 4.440892098500626e-16  -0.44721359549995743]
[                  0.0                  -0.0                   1.0]
 =
[0.9999999999999999 1.9999999999999993 2.9999999999999987]
[1.9999999999999998 3.9999999999999996  4.999999999999998]
[               0.0                0.0                1.0]
```

Gram-Schmidth-Orthogonalisierung der Zeilenvektoren der Matrix:

```
G, M = A.gram_schmidt()
print "orthogonalisierte Zeilen:\n", G
print "\nTransformationsmatrix:\n", M
```

```
orthogonalisierte Zeilen:
[   -0.2672612419124243    -0.5345224838248488
-0.8017837257372732]
[  -0.35856858280031956    -0.7171371656006356
0.5976143046671967]
[   -0.8944271909999154      0.447213595499959
-9.992007221626409e-16]

Transformationsmatrix:
```

```
[    -3.7416573867739413                    0.0
0.0]
[      -6.68153104781061     -0.5976143046671974
-0.0]
[    -0.8017837257372732      0.5976143046671969
-1.0547118733938987e-15]
```

Zum Lösen von Gleichungssystemen bringt man Matrizen oft auch in Dreiecksform ("reduced row echelon form"). Auch die hängt vom Basisdatentyp ab (z.B. über reellen Zahlen müssen die Pivotelemente 1 sein, dafür dürfen reelle Skalare zur Gewichtung verwendet werden;, über ganzen Zahlen nicht). Hier gibt es zwei Varianten - "rref" ignoriert oft den Datentyp (und tut meist das erwünschte, z.B. auch bei einer ganzzahligen Matriz die reelle Dreiecksform), während sich "echelon_form" strikt an den Typ hält bei den Transformationen und daher oft zu unerwarteten Ergebnissen führt.

```
entries = [[2,0],[0,1]]
print "ganzzahlig, mit rref"
print matrix(ZZ, entries).rref()
print "ganzzahlig"
print matrix(ZZ, entries).echelon_form()
print "rational"
print matrix(QQ, entries).echelon_form()
print "reell"
print matrix(RDF, entries).echelon_form()
print "ohne explizite Angabe ganzzahlige Variante"
print matrix(entries).echelon_form()
```

```
ganzzahlig, mit rref
[1 0]
[0 1]
ganzzahlig
[2 0]
[0 1]
rational
[1 0]
[0 1]
reell
[1.0 0.0]
[0.0 1.0]
ohne explizite Angabe ganzzahlige Variante
[2 0]
[0 1]
```