In [1]:
```
def gauss(M):
    A=copy(M)
    n=A.nrows()
    for j in range(n):
        # Recherche du pivot
        if A[j,j]==0:
            for i in range(j+1,n):
                if A[i,j]!=0:
                    A.swap_rows(i,j)
                    break
        if A[j,j]==0:
            raise ValueError("non invertible matrix")
        for i in range(j+1, n):
            A.add_multiple_of_row(i, j, -A[i,j]/A[j,j])
    return A
```

In [2]:
```
def matrice_inversible(n, corps=QQ):
    # Renvoie une matrice inversible
    while True:
        m = random_matrix(corps, n)
        if m.det()!=0:
            return m
```

In [3]:
```
M=matrice_inversible(5); show(M); show(gauss(M));
```

In [4]:
```
show(M.echelon_form());
```

In [5]:
```
def max_coeff(M):
    return max(max(abs(c.numerator()) for row in M.rows() for c in row),
               max(abs(c.denominator()) for row in M.rows() for c in row
))
```

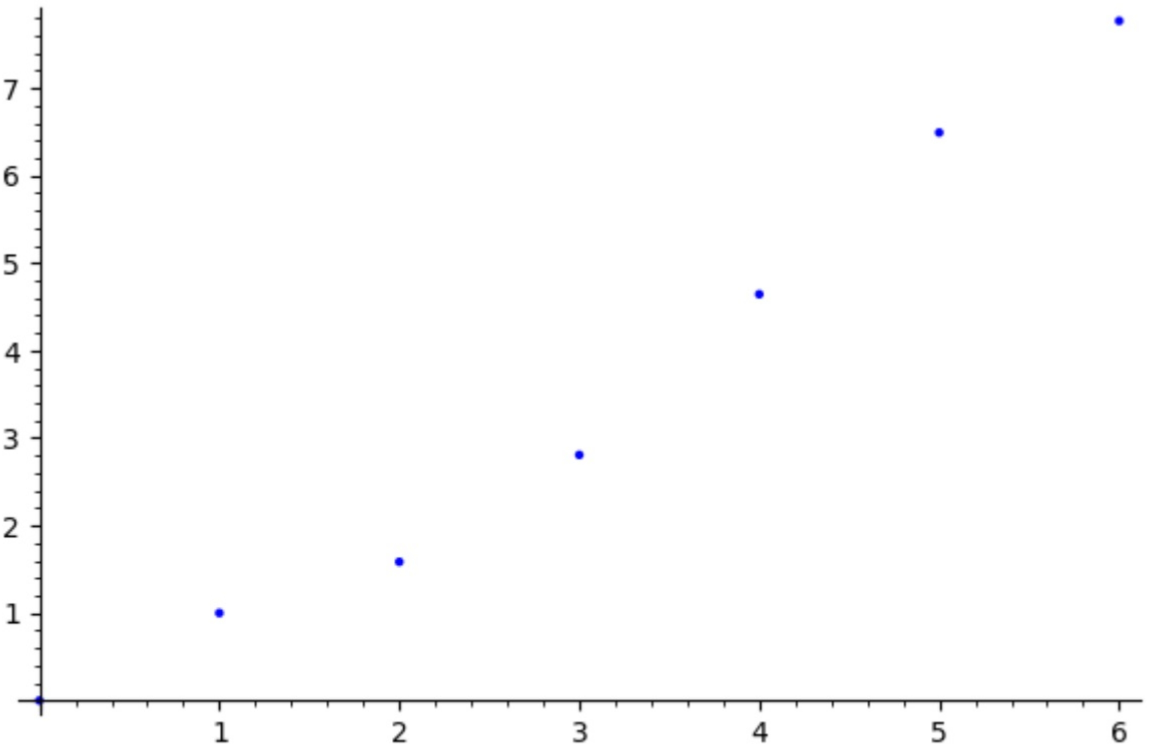
In [6]:
```
G=gauss(matrice_inversible(10)); show(G); show(max_coeff(G));
```

In [7]:
```
t=[max_coeff(gauss(matrice_inversible(2^k))) for k in range(7)]; show(t)
```

In [8]:
```
tt=[x.ndigits(2) for x in t]; show(tt);
```

In [9]:
```
lt=[float(log(tt[i],2)) for i in range(len(tt))]; show(lt);
```

```
In [10]: points(enumerate(lt))
Out[10]:
```



```
In [11]: # Cela corrobore l'hypothèse d'un nombre de chiffres des numérateurs et
          dénominateurs polynômial en la dimension de la matrice
```

```
In [12]: import time;
```

```
In [13]: def temps_gauss(n):
             M=matrice_inversible(n)
             debut=time.time()
             gauss(M)
             return(time.time() - debut)
```

```
In [14]: temps_gauss(5)
Out[14]: 0.0002124309539794922
```

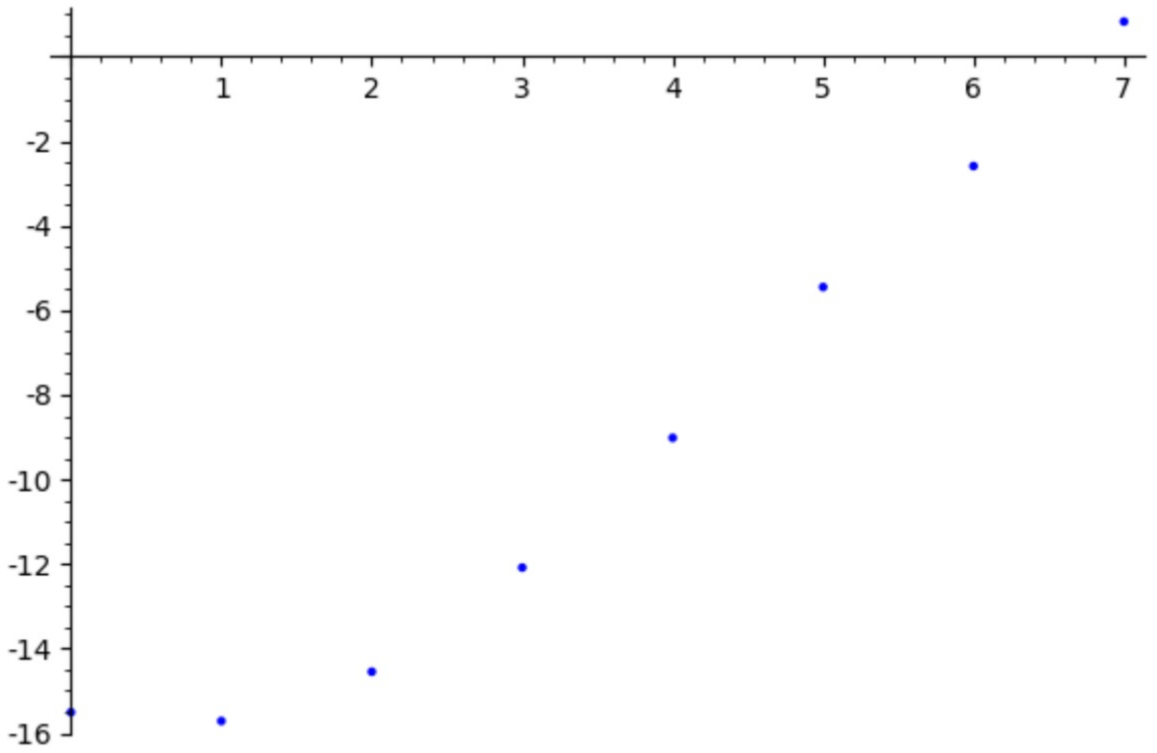
```
In [15]: timeit('gauss(M)')
Out[15]: 625 loops, best of 3: 117 µs per loop
```

```
In [16]: t=[temps_gauss(2^k) for k in range(8)]; show(t)
```

```
In [17]: tt=[float(log(t[k],2)) for k in range(8)]; show(tt)
```

In [18]:
```
points(enumerate(tt))
```

Out[18]:



In [19]:
```
#cela corrobore une complexité polynômiale en la dimension de la matrice
```

In [20]:
```
def gauss_ent(M):
    A=copy(M)
    n=A.nrows()
    for j in range(n):
        # Recherche du pivot
        if A[j,j]==0:
            for i in range(j+1,n):
                if A[i,j]!=0:
                    A.swap_rows(i,j)
                    break
        if A[j,j]==0:
            raise ValueError("non invertible matrix")
        for i in range(j+1, n):
            c=-A[i,j]; A.rescale_row(i,A[j,j]); A.add_multiple_of_row(i,
 j, c)
    return A
```

In [21]:
```
M=random_matrix(ZZ,5,x=10,y=99); show(M); show(gauss_ent(M));
```

In [22]:
```
#On voit bien que la taille des coefficients double en gros à chaque éta
pe
```

In [23]:
```
def gauss_b(M):
    A=copy(M).change_ring(QQ)
    n=A.nrows()
    for j in range(n):
        # Recherche du pivot
        if A[j,j]==0:
            for i in range(j,n):
                if A[i,j]!=0:
                    A.swap_rows(i,j)
                    break
        if A[j,j]==0:
            raise ValueError("non invertible matrix")
        for i in range(j+1, n):
            c=-A[i,j]; A.rescale_row(i,A[j,j]); A.add_multiple_of_row(i,
 j, c);
            if j>0:
                A.rescale_row(i,1/A[j-1,j-1]);
    return A
```

In [24]:
```
show(M) ;show(gauss_b(M));
```

In [25]:
```
M.determinant()
```

Out[25]:
```
732193080
```

In [26]:
```
def gauss_b_det(M):
    A=copy(M.change_ring(QQ))
    n=A.nrows(); perm=1;
    for j in range(n):
        # Recherche du pivot
        if A[j,j]==0:
            for i in range(j+1,n):
                if A[i,j]!=0:
                    A.swap_rows(i,j); perm=-perm;
                    break
        if A[j,j]==0:
             return(0)
        for i in range(j+1, n):
             c=-A[i,j]; A.rescale_row(i,A[j,j]); A.add_multiple_of_row(i
, j, c);
             if j>0:
                 A.rescale_row(i,1/A[j-1,j-1]);
    return perm*A[n-1,n-1];
```

In [27]:
```
show(gauss_b_det(M));
```

In [28]:
```
range(1,5);
```

In [29]:
```
M=matrice_inversible(5,corps=GF(3)); show(M); show(gauss(M));
```

In [30]:
```
def gauss_det(M):
    A=copy(M)
    n=A.nrows(); perm=1;
    for j in range(n):
        # Recherche du pivot
        if A[j,j]==0:
            for i in range(j+1,n):
                if A[i,j]!=0:
                    A.swap_rows(i,j); perm=-perm;
                    break
        if A[j,j]==0:
            return(0)
        for i in range(j+1, n):
            A.add_multiple_of_row(i, j, -A[i,j]/A[j,j])
    return(perm*prod([A[i,i] for i in range(n)]))
```

In [31]:
```
show(gauss_det(M)); show(M.determinant())
```

In [32]:
```
def det_mod(M):
    A=copy(M); n=A.nrows();
    m=max(abs(A[i,j]) for i in range(n) for j in range(n));
    Maj=n^(n/2)*m^n;
    N=floor(log(2*Maj+1,2));
    P=Primes(); L=[P.unrank(i) for i in range(N)]
    D=[ZZ(gauss_det(copy(A).change_ring(GF(L[i])))) for i in range(N)]
    p=prod(L[i] for i in range(N)); res=crt(D,L);
    if res>Maj:
        return(res-p)
    return(res)
```

In [33]:
```
M=random_matrix(ZZ,5,x=10,y=99); show(M.determinant()); show(det_mod(M
));
```

In [34]:
```
show(crt([0,1,2],[2,3,5]));
```

In [35]:
```
22%3; 22%5
```

Out[35]:
```
2
```

In [36]:
```
def det_mod1(M):
    A=copy(M); n=A.nrows();
    m=max(abs(A[i,j]) for i in range(n) for j in range(n));
    Maj=n^(n/2)*m^n;
    N=floor(2*Maj);
    P=Primes(); p=P.next(N);
    d=ZZ(gauss_det(A.change_ring(GF(p))));
    if d>Maj:
        return(d-p)
    return(d)
```

In [37]:
```
show(det_mod1(M)); show(M.determinant())
```

```
In [38]: timeit('gauss_det(copy(M).change_ring(QQ))')

Out[38]: 625 loops, best of 3: 129 µs per loop

In [39]: timeit('gauss_b_det(M)')

Out[39]: 625 loops, best of 3: 280 µs per loop

In [40]: timeit('det_mod(M)')

Out[40]: 125 loops, best of 3: 3.94 ms per loop

In [41]: timeit('det_mod1(M)')

Out[41]: 625 loops, best of 3: 1.13 ms per loop

In [42]: # On a du mal à voir l'apport de GB par rapport à Gauss classique sur ce
         t exemple ; les méthodes modulaires sont moins rapides avec cette implém
         entation et en particulier la méthode modulaire à plusieurs nombres prem
         iers est loin derrière

In [43]: M=random_matrix(ZZ,5,x=10^1000,y=10^1001)

In [44]: timeit('gauss_det(copy(M).change_ring(QQ))')

Out[44]: 125 loops, best of 3: 4.45 ms per loop

In [45]: timeit('gauss_b_det(M)')

Out[45]: 625 loops, best of 3: 816 µs per loop

In [46]: # GB est meilleur que Gauss classique, en accord avec le cours.

In [47]: M=random_matrix(ZZ,20,x=10^5,y=10^6)

In [48]: timeit('gauss_b_det(M)')

Out[48]: 25 loops, best of 3: 7.01 ms per loop

In [49]: timeit('det_mod(M)')

Out[49]: 5 loops, best of 3: 224 ms per loop

In [50]: timeit('det_mod1(M)')

Out[50]: 5 loops, best of 3: 786 ms per loop

In [51]: # La version à plusieurs nombres premiers est plus performante que celle
          à un seul, mais GB est bien meilleur ! Cela ne reflète pas la compléxit
         é théorique.
```