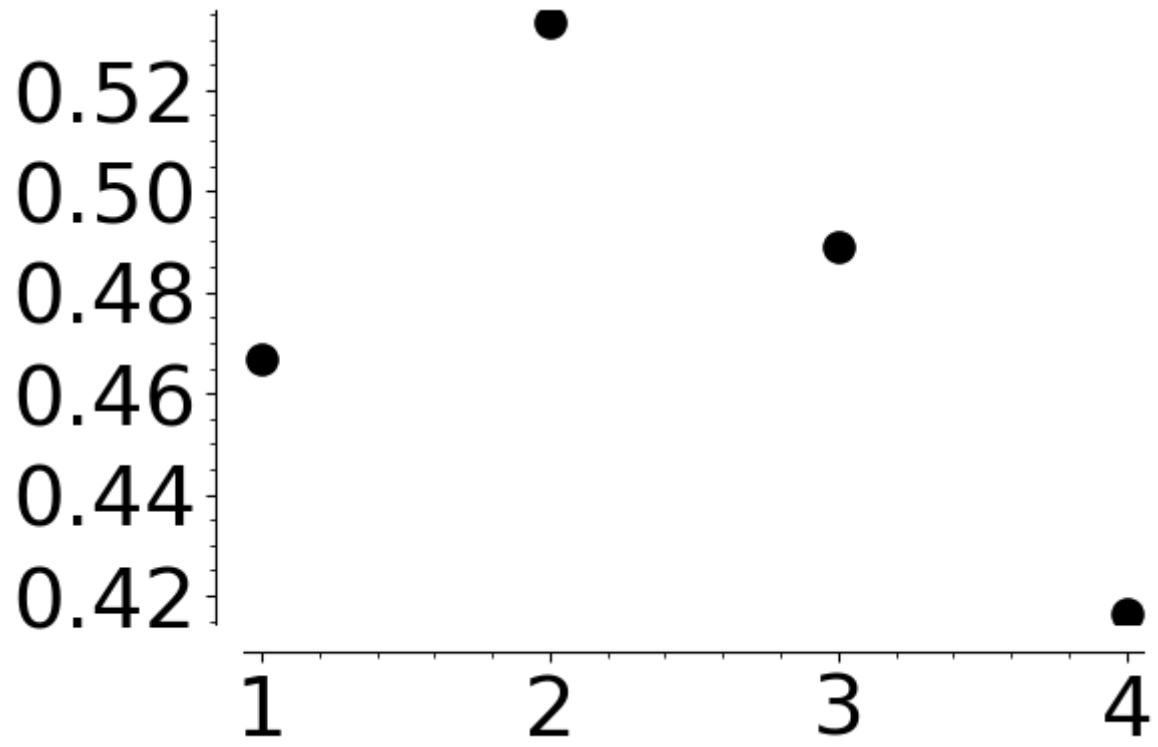
In [1]:
```
import itertools
from IPython.core.debugger import set_trace
```

In [2]:
```
# Hamming weight function
def weight(v):
    return sum(x!=0 for x in v)
```

In [3]:
```
# set code and Guruswami-Sudan parameters
n = 18
k = 4
Rp = (k-1)/n
ell = 4
r = var('r')
Theta = lambda r: 1 - (r+1)/(2*(ell+1)) - ell/(2*r)*Rp
pretty_print(Theta(r))
```

$$-\frac{1}{10}\,r - \frac{1}{3\,r} + \frac{9}{10}$$

In [4]:
```
# Theta as a function of r
rlist = [1,2,3,4]
tlist = [Theta(r) for r in rlist]
plist = [[r, Theta(r)] for r in rlist]
pl = scatter_plot(plist, markersize=120, facecolor='black', ticks=1)
pl.fontsize(30)
show(pl)
pl.save("GS-Theta-vs-r-ell4.pdf")
```



In [5]:
```
# set r
r=2
tau = ceil(n*Theta(r))-1
print(tau)
```

```
9
```

In [6]:
```
F = GF(19)
```

In [7]:
```
# generator matrix for [18, 4, 15] RS code over GF(19)
alpha = vector(F, [i for i in range(1, 19)])
G = Matrix(F, [ [alpha[j]^i for j in range(18)] for i in range(k)])
pretty_print(G)
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 1 & 4 & 9 & 16 & 6 & 17 & 11 & 7 & 5 & 5 & 7 & 11 & 17 & 6 & 16 & 9 & 4 & 1 \\ 1 & 8 & 8 & 7 & 11 & 7 & 1 & 18 & 7 & 12 & 1 & 18 & 12 & 8 & 12 & 11 & 11 & 18 \end{pmatrix}$$

In [8]:
```
# bivariate polynomial ring over F
P.<x,z> = PolynomialRing(F)
```

In [9]:
```
# message vector
uv = vector(F, [18, 14, 3, 1] )
# message polynomial
ux = vector(P,[x^i for i in range(k)]).dot_product(uv)
pretty_print(ux)
```

$$x^3 + 3x^2 + 14x + 18$$

In [10]:
```
# codeword
c = vector(F, uv*G)
print(c)
```

```
(17, 9, 0, 15, 3, 8, 17, 17, 14, 14, 4, 9, 16, 12, 3, 14, 13, 6)
```

In [11]:
```
# error vector
ev = vector(F, [15, 9, 0, 0, 9, 17, 0, 8, 4, 0, 0, 0, 0, 4, 0, 7, 0, 12])
weight(ev)
```

Out[11]:
```
9
```

In [12]:
```
# received word
y = c + ev
print(y)
```

```
(13, 18, 0, 15, 12, 6, 17, 6, 18, 14, 4, 9, 16, 16, 3, 2, 13, 18)
```

In [13]:

```
# degrees in z and x
zdeg = ell
xdeg = lambda s: r*(n - tau) - 1 - (k-1)*s
# monomials vector
ml = [[x^j*z^i for j in range(xdeg(i)+1)] for i in range(zdeg+1)]
ml = list(itertools.chain(*ml))
mv = vector(P, ml)
pretty_print(mv)
print(f"length={len(mv)}")
```

$$\left(1,\ x,\ x^2,\ x^3,\ x^4,\ x^5,\ x^6,\ x^7,\ x^8,\ x^9,\ x^{10},\ x^{11},\ x^{12},\ x^{13},\ x^{14},\ x^{15},\ x^{16},\ x^{17},\ z,\ xz,\ x^2z,\ x^3z,\ x^4z,\ x^5z,\ x^6z,\ x^7z,\ x^8z,\ x^9z,\ x\right.$$
$$x^{13}z,\ x^{14}z,\ z^2,\ xz^2,\ x^2z^2,\ x^3z^2,\ x^4z^2,\ x^5z^2,\ x^6z^2,\ x^7z^2,\ x^8z^2,\ x^9z^2,\ x^{10}z^2,\ x^{11}z^2,\ z^3,\ xz^3,\ x^2z^3,\ x^3z^3,\ x^4z^3,\ x^5z^3,\ x^6z^3$$
$$\left.xz^4,\ x^2z^4,\ x^3z^4,\ x^4z^4,\ x^5z^4\right)$$

```
length=60
```

In [14]:

```
# Functions related to Hasse derivatives, interpolation matrix

# build the interpolation matrix
def build_interpolation(n, k, ell, r, tau):
    zdeg = ell
    xdeg = lambda i: r*(n - tau) - 1 - (k-1)*i
    ml = []
    for s in range(r):
        for t in range(r-s):
            mll = [[binomial(j, s)*x^(j-s)*binomial(i,t)*z^(i-t) for j in range(xdeg(i)+1)] for i in
range(zdeg+1)]
            mll = list(itertools.chain(*mll))
            mvv = vector(P, mll)
            # set_trace()
            #print(s,t,mvv)
            for h in range(n):
                row = mvv.subs(x=alpha[h], z=y[h])
                ml.append(row)
    return Matrix(F, ml)

# compute a Hasse derivative of a bi-variate polynomial
def hassederive(Q, s, t):
    QD = 0
    for term in Q0:
        coef = term[0]
        mono = term[1]
        i = mono.degree(z)
        j = mono.degree(x)
        dterm = coef*binomial(j, s)*x^(j-s)*binomial(i, t)*z^(i-t)
        QD += dterm
    return QD

# verify a polynomial satisfies the interpolation constraints
def verify(Q, r, alpha, y):
    n = len(alpha)
    assert n==len(y), "Inconsistent lengths"
    ok = True
    for s in range(r):
        for t in range(r-s):
            QD = hassederive(Q, s, t)
            for h in range(n):
                q = QD.subs(x=alpha[h], z=y[h])
```

```
                if q != 0:
                    print(f"Failed interpolation s={s} t={t} h={h} q={q}")
                    ok = False
    print(f'Test {"PASSED" if ok else "FAILED"}')
```

In [15]:
```
# build the matrix
M = build_interpolation(n, k, ell, r, tau)
```

In [16]:
```
print(M.dimensions())
```

```
(54, 60)
```

In [17]:
```
# print(M)
```

In [18]:
```
# find space of solutions to the interpolation constraints
B = M.right_kernel().basis()
print(B)
```

```
[
(1, 0, 0, 0, 0, 0, 8, 11, 17, 12, 14, 8, 5, 10, 17, 14, 3, 4, 9, 4, 15, 11, 15, 2, 7, 3, 5, 12, 11,
8, 0, 8, 2, 0, 6, 7, 11, 6, 10, 6, 6, 14, 11, 13, 11, 18, 5, 15, 14, 5, 8, 17, 14, 1, 7, 7, 18, 4,
2, 1),
(0, 1, 0, 0, 0, 0, 11, 12, 13, 6, 18, 7, 13, 12, 8, 2, 9, 11, 15, 8, 14, 7, 8, 5, 13, 1, 18, 9, 14,
17, 3, 12, 9, 4, 8, 15, 4, 1, 15, 8, 14, 6, 8, 2, 0, 12, 3, 12, 5, 0, 8, 14, 13, 1, 4, 4, 17, 10, 9,
17),
(0, 0, 1, 0, 0, 0, 7, 8, 1, 8, 0, 18, 4, 16, 13, 12, 3, 17, 13, 2, 15, 3, 4, 14, 5, 18, 1, 4, 5, 7,
4, 8, 17, 3, 4, 16, 2, 16, 2, 18, 5, 3, 5, 0, 10, 5, 6, 10, 2, 18, 10, 10, 0, 16, 15, 4, 14, 12, 17,
16),
(0, 0, 0, 1, 0, 0, 11, 3, 1, 3, 17, 18, 11, 2, 4, 9, 18, 7, 13, 17, 16, 15, 1, 2, 15, 12, 15, 0, 3,
12, 6, 8, 15, 12, 8, 6, 3, 12, 0, 10, 12, 6, 11, 2, 0, 0, 15, 8, 18, 1, 12, 12, 7, 5, 1, 6, 8, 4, 9,
11),
(0, 0, 0, 0, 1, 0, 4, 4, 9, 10, 2, 1, 16, 1, 8, 11, 13, 3, 13, 16, 17, 3, 4, 13, 16, 9, 13, 9, 4, 1
3, 2, 8, 2, 12, 9, 0, 7, 11, 6, 6, 8, 10, 15, 3, 12, 14, 8, 13, 1, 10, 6, 14, 3, 10, 15, 3, 16, 16,
15, 11),
(0, 0, 0, 0, 0, 1, 6, 11, 16, 1, 1, 6, 3, 4, 7, 12, 3, 5, 16, 15, 3, 3, 4, 10, 14, 14, 16, 14, 8, 1
2, 8, 11, 2, 9, 18, 9, 13, 0, 14, 8, 13, 9, 18, 15, 18, 9, 14, 13, 9, 12, 17, 17, 13, 17, 16, 6, 2,
4, 5, 15)
]
```

In [19]:
```
# pick one solution
s0 = vector(F, B[0])
s0
```

Out[19]:
```
(1, 0, 0, 0, 0, 0, 8, 11, 17, 12, 14, 8, 5, 10, 17, 14, 3, 4, 9, 4, 15, 11, 15, 2, 7, 3, 5, 12, 11,
8, 0, 8, 2, 0, 6, 7, 11, 6, 10, 6, 6, 14, 11, 13, 11, 18, 5, 15, 14, 5, 8, 17, 14, 1, 7, 7, 18, 4,
2, 1)
```

In [20]:
```
# build polynomial
Q0 = mv.dot_product(s0)
```

In [21]:
```
# verify the interpolation constraints
verify(Q0, r, alpha, y)
```

```
Test PASSED
```

In [22]:
```
# factor Q(x)
Qfactors = factor(Q0)
Qfactors
```

Out[22]:
```
(-x^3 - 3*x^2 + 5*x + z + 1) * (-4*x^14 + 9*x^13 - 4*x^12 - 6*x^11*z - 2*x^11 + 4*x^10 + 4*x^9*z + 2
*x^8*z^2 + 7*x^9 - 9*x^8*z + 8*x^8 + 2*x^7*z + 3*x^6*z^2 + x^5*z^3 + x^7 + 5*x^6*z + 8*x^5*z^2 + 2*x
^4*z^3 - 6*x^6 + 8*x^5*z + 6*x^4*z^2 + 4*x^3*z^3 - 5*x^5 + x^4*z + 5*x^3*z^2 - x^2*z^3 - 6*x^4 - 9*x
^3*z + 2*x^2*z^2 + 7*x*z^3 - 2*x^3 - 5*x^2*z + x*z^2 + 7*z^3 + 9*x^2 + 7*x*z - 8*z^2 - 5*x + 8*z +
1)
```

In [23]:
```
# get z-linear factor
Qfactors[0][0]
```

Out[23]:
```
-x^3 - 3*x^2 + 5*x + z + 1
```

In [24]:
```
# extract the z-root
uu = -Qfactors[0][0].subs(z=0)
```

In [25]:
```
# compare to original message
uu-ux
```

Out[25]:
```
0
```

In [26]:
```
# try with a different solution
s1 = vector(F, B[1])
Q1 = mv.dot_product(s1)
verify(Q1, r, alpha, y)
```

```
Test PASSED
```

In [27]:
```
factor(Q1)
```

Out[27]:
```
(-2) * (-x^3 - 3*x^2 + 5*x + z + 1) * (-4*x^14 + 7*x^13 - 2*x^12 - 9*x^11*z + 3*x^11 + 2*x^10*z - 6*
x^10 - 4*x^9*z - 9*x^8*z^2 + 9*x^9 - 4*x^8*z - 8*x^7*z^2 - 3*x^8 - 5*x^7*z - 2*x^6*z^2 + x^5*z^3 - 5
*x^6*z - 6*x^5*z^2 + 5*x^4*z^3 - 3*x^6 - 8*x^5*z + 2*x^4*z^2 - 5*x^3*z^3 + 3*x^5 + 4*x^4*z - x^3*z^2
+ x^2*z^3 + x^4 - 6*x^3*z - 3*x^2*z^2 - 2*x*z^3 + 5*x^3 + 7*x^2*z + x*z^2 - 2*z^3 - 7*x^2 - 4*x*z -
4*z^2 + 9*x + 2*z)
```

In [ ]:
```
```

BACK