Додаток А

(обов'язковий)

Текст програми hermitian_code.py

```python
import unittest
import logging

logger = logging.getLogger('hermitian_code')
logger.setLevel(logging.CRITICAL)
while len(logger.parent.handlers) > 1:
    logger.parent.removeHandler(logger.parent.handlers[-1])



class DecodingError(Exception):
    pass

class CodeConstructionError(Exception):
    pass

class HermitianCode():
    def __init__(self, m, a=None):
        '''One-point AG code on Hermitian curve H_m with D=a*Q'''
        if not is_prime_power(m):
            raise Exception('m must be a prime power')
        self.m = m
        self.g = int(m*(m-1)/2)
        field_size = m**2
        self.F = GF(field_size,'w')
        x, y, z = PolynomialRing(self.F, 3, 'xyz').gens()
        self.x, self.y, self.z = x, y, z
        f = x**(m+1) + y**m * z + y * z**m
        self.C = Curve(f)
        if a is None:
            self.a = m**3 - m**2 + m + 1
        else:
            self.a = a
        # t - number of errors that can be corrected with SV algorithm
        self.t = int((self.a-3*self.g+1)/2)
```

```
        self.L_D = self.L(self.a)
        self.L_A = self.L(self.t + self.g)
        self.L_C = self.L(self.a - self.t - self.g)
        self.S = Matrix(self.L_A).transpose() * Matrix(self.L_C)
        logger.info('Syndrom matrix:\n%s' % self.S)

        # init points: P set and Q
        points = self.C.rational_points()
        self.Q = points[1]
        self.P = [points[0]] + points[2:]
        logger.info('Points in P-set\n%s' % self.P)

        self.H()
        logger.info('Parity-check matrix:\n%s' % self.H())
        self.G()
        logger.info('Generator matrix:\n%s' % self.G())
        self.n = len(self.P)
        self.k = self.G().nrows()
        self.d = self.a - 2*self.g + 2
        #cache for applying function to the P set
        self.f_point_cache = {}
        logger.info('Finished constructing code %s' % self)

    def __repr__(self):
        return 'AG [%d, %d, %d] code on Hermitian curve H_%d <%s> with D=%d*Q' % (
                self.n,
                self.k,
                self.d,
                self.m,
                self.C,
                self.a
                )

    def L(self, a):
        '''L(a*Q)'''
        if a <= 2*self.g - 2:
            raise CodeConstructionError('''The degree of Q is not greater than 2*g - 2
                %d <= 2*%d - 2''' % (a, self.g))
        L_D_basis = []
```

```
        m = self.m
        x, y, z = self.x, self.y, self.z
        for i in range(0, m + 1):
            for j in range(0, a/(m+1) + 1):
                if i*m + j*(m+1) <= a:
                    #print 'x^%d y^%d / z^%d' % (i, j, i+j)
                    L_D_basis.append((x**i * y**j)/z**(i+j))
        if len(L_D_basis) != int(a + 1 - self.g):
            raise CodeConstructionError('''The number of functions found does not
            satisfy the Riemann-Roch theorem
            Found: %d, needed %d''' % (len(L_D_basis), a + 1 - self.g))
        return L_D_basis

    def _map_functions_points(self, functions, points):
        return Matrix(
                self.F,
                [[self._apply(f, point) for point in points] for f in functions])

    def H(self):
        try:
            return self._H
        except AttributeError:
            #a_dual = len(self.P) - 2 + 2*self.g - self.a
            #if a_dual < self.a and a_dual > 2*self.g - 2:
            # print 'using dual'
            # self._G = self._map_functions_points(self.L(a_dual), self.P)
            # self._H = Matrix(self.F, self._G.transpose().kernel().basis())
            #else:
            self._H = self._map_functions_points(self.L_D, self.P)
            return self._H

    def G(self):
        try:
            return self._G
        except AttributeError:
            self._G = Matrix(self.F, self.H().transpose().kernel().basis())
            return self._G

    def encode(self, w):
```

```
        return vector(self.C.base_ring(), w) * self.G()

    def _apply(self, f, p):
        return f(*list(p))

    def multiply(self, v, f):
        '''vector-function multiplication'''
        try:
            f_vector = self.f_point_cache[f]
        except KeyError:
            f_vector = vector(self.C.base_ring(), [self._apply(f, p) for p in self.P])
            self.f_point_cache[f] = f_vector
        return v*f_vector

    def decode(self, v):
        new_rows = []
        for row in self.S:
            new_row = []
            for f in row:
                new_row.append(self.multiply(v, f))
            new_rows.append(new_row)
        S = Matrix(self.C.base_ring(), new_rows)
        logger.info('Syndrom matrix for vector %s:\n%s' % (v, S))
        try:
            theta = vector(self.L_A)*vector(S.kernel().basis()[0])
            logger.info('Error locator: %s' % theta)
        except IndexError:
            raise DecodingError
        error_positions = []
        logger.info('Error locator equals to zero at following points')
        for i, p in enumerate(self.P):
            if self._apply(theta, p) == 0:
                error_positions.append(i)
                logger.info('P_%d' % i)
        error_value_system = []
        for f in self.L_D:
            row = []
            for i in error_positions:
                row.append(self._apply(f, self.P[i]))
```

```
            row.append(self.multiply(v, f))
            error_value_system.append(row)
        error_value_system = Matrix(
                self.F,
                error_value_system
                )
        logger.info('Error value system:\n%s' % error_value_system)
        error_value_system = error_value_system.echelon_form()
        logger.info('Error position and value:')
        for i, pos in enumerate(error_positions):
            logger.info('P_%d: %s' % (pos, error_value_system[i][-1]))
            v[pos] -= error_value_system[i][-1]
        logger.info('Recovered vector: %s' % v)
        decode_g = Matrix(
                self.C.base_ring(),
                self.G().rows() + [v]
                ).transpose().echelon_form()
        decoded_message = [decode_g[i][-1] for i in range(0, decode_g.ncols()-1)]
        return decoded_message

def test(AG):
    print AG
    print AG.H()
    print '===='
    print AG.G()
    print 'We may add up to %d errors for SV algorithm' % AG.t
    for j in range(0, 10):
        w = list(AG.C.base_ring())[1]
        message = [w**(i+j) for i in range(0, AG.k)]
        print 'Original message:\n%s' % message
        v=AG.encode(message)
        print 'Encoded message:\n%s' % v
        for i in range(0, AG.t):
            v[i*2] += w**(i+j)
        print 'Message with errors:\n%s' % v
        print 'Decoding...'
        decoded = AG.decode(v)
        print 'Decoded message:\n%s' % decoded
        if message == decoded:
```

```
                print 'Decoded correctly'
            else:
                raise DecodingError('Failed to decode correctly')
            #if raw_input() == 'q':
            # break
        print 'Cached %d results for applying function to P-set' % len(AG.f_point_cache)

class TestHermitianCode(unittest.TestCase):
    def setUp(self):
        self.ag_2_6 = HermitianCode(2, 6)
        self.ag_4_40 = HermitianCode(4, 40)

    def test_2_6_H_G(self):
        AG = self.ag_2_6
        w = list(AG.F)[1]
        self.assertEqual(AG.G().echelon_form(), Matrix([
            (1, 1, 0, 0, w + 1, w + 1, w, w),
            (0, 0, 1, 1, w, w, w + 1, w + 1)]
            ).echelon_form()
            )
        self.assertEqual(AG.H().echelon_form(), Matrix([
            (1, 1, 1, 1, 1, 1, 1, 1),
            (0, 1, w, w + 1, w, w + 1, w, w + 1),
            (0, 1, w + 1, w, w + 1, w, w + 1, w),
            (0, 0, 1, 1, w, w, w + 1, w + 1),
            (0, 0, w, w + 1, w + 1, 1, 1, w),
            (0, 0, 1, 1, w + 1, w + 1, w, w)]
            ).echelon_form()
            )

    def decoding_helper(self, AG):
        w = list(AG.F)[1]
        for j in range(0, 10):
            message = [w**(i+j) for i in range(0, AG.k)]
            v=AG.encode(message)
            for i in range(0, AG.t):
                v[i*2] += w**(i+j)
            decoded = AG.decode(v)
            self.assertEqual(message, decoded)
```

```

    def test_2_6_decoding(self):
        self.decoding_helper(self.ag_2_6)

    def test_4_40_decoding(self):
        self.decoding_helper(self.ag_4_40)

if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(TestHermitianCode)
    unittest.TextTestRunner(verbosity=2).run(suite)
    #test(HermitianCode(2, 6))
    #test(HermitianCode(4, 40))
```