Python Examples: Linear Block Codes

Henry D. Pfister
Duke University

1 The Galois Field $\mathbb{F}_p$ for Prime p

This document provides Python (with NumPy) code snippets as examples of linear block coding and related computations over finite fields of prime order. Most examples assume basic familiarity with Python and NumPy. If you need help in Python, you can use the built-in `help()` function or refer to the online NumPy documentation.

1.1 A Few Commands

The following Python commands illustrate the rough equivalents of some common MATLAB commands:

```
>>> import numpy as np
>>> help(np.eye)
Help on function eye in module numpy:

eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)
    Return a 2-D array with ones on the diagonal and zeros elsewhere.
    ...

>>> help(np.mod)
Help on ufunc:

remainder = <ufunc 'remainder'>
    remainder(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K',
    dtype=None, subok=True[, signature, extobj])
    Return element-wise remainder of division.
    ...

>>> import math
>>> help(math.comb)
Help on built-in function comb in module math:

comb(n, k, /)
    Number of ways to choose k items from n items without repetition
    and without order.
    ...
```

Python does not have a built-in `de2bi` or or `bi2de`. However, one can write simple helper functions to convert integers to binary arrays and back. For instance, can you paste the following command into the python command line:

```
import numpy as np
def de2bi(n, k):
  # Convert integer list n into a list of k-bit arrays. Returns array of shape (len(n), k)
  arr = []
  n = [n] if isinstance(n,int) else n
  for i in n:
    bits = [int(x) for x in bin(i)[2:].zfill(k)]
    arr.append(bits)
  return np.array(arr, dtype=int)


def bi2de(arr):
  # Interpret each row of arr as a binary integer
  return np.array([int("".join(str(x) for x in row), 2) for row in arr])


bi = de2bi(2, 3)
bi

from itertools import combinations
list(combinations([1, 2, 3, 4, 5, 6], 2))
```

You will get the following output and then you can continue with the exercise.

```
>>> def de2bi(n, k):
...   # Convert integer list n into a list of k-bit arrays. Returns array of shape (len(n), k)
...   arr = []
...   n = [n] if isinstance(n,int) else n
...   for i in n:
...     bits = [int(x) for x in bin(i)[2:].zfill(k)]
...     arr.append(bits)
...  return np.array(arr, dtype=int)

>>> def bi2de(arr):
...   # Interpret each row of arr as a binary integer
...   return np.array([int("".join(str(x) for x in row), 2) for row in arr])

>>> bi = de2bi(2, 3)
>>> bi
array([[0, 1, 0]])
```

For combinations akin to `nchoosek`, one can use `itertools.combinations` or `math.comb` for binomial coefficients:

```
>>> from itertools import combinations
>>> list(combinations([1, 2, 3, 4, 5, 6], 2))
[(1, 2), (1, 3), (1, 4), ..., (5, 6)]
```

2 Now For Some Coding

In this section, we construct generator and parity-check matrices G and H for a simple linear block code over $\mathbb{F}_p$ (with $p = 2$ in this example). Past the following

```
n = 6
k = 3
p = 2
In = np.eye(n, dtype=int)
Ik = np.eye(k, dtype=int)
Ink = np.eye(n-k, dtype=int)
P = np.array([[1, 1, 0],[0, 1, 1],[1, 0, 1]], dtype=int)
P
G = np.concatenate((Ik, P), axis=1)
H = np.mod(np.concatenate((-P.T, Ink), axis=1), p)
# Check that G and H produce the zero matrix when computing G * H^T
np.mod(G @ H.T, p)
```

to get

```
>>> n = 6
>>> k = 3
>>> p = 2

>>> In = np.eye(n, dtype=int)
>>> Ik = np.eye(k, dtype=int)
>>> Ink = np.eye(n-k, dtype=int)

>>> P = np.array([[1, 1, 0],[0, 1, 1],[1, 0, 1]], dtype=int)
>>> P
array([[1, 1, 0],
       [0, 1, 1],
       [1, 0, 1]])

>>> G = np.concatenate((Ik, P), axis=1)
>>> H = np.mod(np.concatenate((-P.T, Ink), axis=1), p)

>>> # Check that G and H produce the zero matrix when computing G * H^T
>>> np.mod(G @ H.T, p)
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

3 Encoding and Listing Codewords

Below, we use the previous `de2bi` function to list all possible 2^k input messages and compute their codewords by multiplying by G (in modulo p arithmetic). Paste the following

```
u = de2bi(range(2**k), k)   # All binary input vectors of length k
u
C = np.mod(u @ G, p)  # Encode all of them
C
```

to get

```
>>> u = de2bi(range(2**k), k)   # All binary input vectors of length k
>>> u
array([[0, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 1, 1],
       [1, 0, 0],
       [1, 0, 1],
       [1, 1, 0],
       [1, 1, 1]])

>>> C = np.mod(u @ G, p)  # Encode all of them
>>> C
array([[0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 0, 1],
       [0, 1, 0, 0, 1, 1],
       [0, 1, 1, 1, 1, 0],
       [1, 0, 0, 1, 1, 0],
       [1, 0, 1, 0, 1, 1],
       [1, 1, 0, 1, 0, 1],
       [1, 1, 1, 0, 0, 0]])
```

4 Syndromes

Below, we list all weight-2 error patterns using Python's `itertools.combinations` and check the corresponding syndromes with respect to H.

```
>>> from itertools import combinations

>>> idx_2 = list(combinations(range(n), 2))
>>> E2 = np.zeros((len(idx_2), n), dtype=int)
>>> for i, (pos1, pos2) in enumerate(idx_2):
...     E2[i, pos1] = 1
...     E2[i, pos2] = 1

>>> E2
array([[1, 1, 0, 0, 0, 0],
       [1, 0, 1, 0, 0, 0],
       [1, 0, 0, 1, 0, 0],
       ...,
       [0, 0, 0, 1, 0, 1],
       [0, 0, 0, 0, 1, 1]])

>>> S2 = np.mod(E2 @ H.T, p)
>>> S2int = bi2de(S2)  # Each syndrome mapped to an integer
>>> S2int
array([5, 3, 2, 4, 7, 6, 7, 1, 2, 1, 7, 4, 6, 5, 3])
```

5 Simulation

We illustrate a small block of random messages, encoded codewords, transmitted over a BSC(0.1) channel, and received. This snippet assumes you already defined `G`, `k`, `n`, and `p`. Paste the following

```
M = 5  # Number of messages to transmit at once
msg = np.random.randint(0, 2**k, size=M)  # Uniform random [0, 2^k-1]
msg
# Convert message numbers to bit vectors of length k
u = np.array([ [int(x) for x in bin(m)[2:].zfill(k)] for m in msg ], dtype=int)
u
# Encode
c = np.mod(u @ G, p)
# Generate BSC noise with prob 0.1
noise = (np.random.rand(M, n) < 0.1).astype(int)
noise
# Received codewords
recv = np.mod(c + noise, p)
```

to get

```
>>> M = 5  # Number of messages to transmit at once
>>> msg = np.random.randint(0, 2**k, size=M)  # Uniform random [0, 2^k-1]
>>> msg
array([4, 6, 7, 5, 1])  # Example output

>>> # Convert message numbers to bit vectors of length k
>>> u = np.array([ [int(x) for x in bin(m)[2:].zfill(k)] for m in msg ], dtype=int)
>>> u
array([[1, 0, 0],
       [1, 1, 0],
       [1, 1, 1],
       [1, 0, 1],
       [0, 0, 1]])  # Example

>>> # Encode
>>> c = np.mod(u @ G, p)

>>> # Generate BSC noise with prob 0.1
>>> noise = (np.random.rand(M, n) < 0.1).astype(int)
>>> noise
array([...])  # Some random 0/1 pattern

>>> # Received codewords
>>> recv = np.mod(c + noise, p)
```

6 Matrix Tricks

Below are some Python "tricks" for performing matrix inverses modulo p. Note that Python's usual matrix inverse (`np.linalg.inv`) computes floating-point results. We must carefully convert back to integer arithmetic and then take the modulo. For binary fields ($p = 2$), one trick is to multiply the floating-point inverse by `np.round(det(A))`, then do modulo-2 arithmetic, provided the determinant is odd and not too large. Paste the following

```
A = np.random.randint(0, 2, (5, 5))
A
# Compute determinant in integer domain
detA = round(np.linalg.det(A))
detA
# Provided detA is odd, we do:
invA_fp = np.linalg.inv(A)
invA_candidate = np.mod(np.round(invA_fp * detA), 2)
# Check
np.mod(invA_candidate @ A, 2)
```

to get

```
>>> A = np.random.randint(0, 2, (5, 5))
>>> A
array([[1, 0, 1, 0, 1],
       [0, 0, 1, 0, 1],
       [0, 1, 0, 0, 1],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 1, 1]])

>>> # Compute determinant in integer domain
>>> detA = round(np.linalg.det(A))
>>> detA
-1  # Example output

>>> # Provided detA is odd, we do:
>>> invA_fp = np.linalg.inv(A)
>>> invA_candidate = np.mod(np.round(invA_fp * detA), 2)
>>> # Check
>>> np.mod(invA_candidate @ A, 2)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

For prime $p > 2$, the idea is similar, but conversions from float back to int must be done carefully to avoid precision issues. An alternative in Python is to use libraries dedicated to finite field arithmetic (e.g. the `galois` package) which compute exact results in GF(p) or GF(p^m).

7 Extension Fields $\mathbb{F}_{2^m}$

7.1 A Few Commands

Unlike MATLAB, Python does not have a built-in type for extension fields of characteristic 2. However, one may install a package like `galois` (available on PyPI) or implement the required operations (addition, multiplication) in Python directly. For instance, the `galois` package offers:

```
>>> import galois
>>> GF4 = galois.GF(4,irreducible_poly="x^2 + x + 1")
>>> a = GF4([0, 1, 2, 3])            # All elements in GF(4)
>>> b = GF4([1, 1, 1, 1])
>>> a * b
GF([0, 1, 2, 3], order=2^2)
```

You can consult the `galois` documentation for more details on how to construct and manipulate finite fields over 2^m or other fields.

7.2 Standard Codes

Here we illustrate a small example to mimic a `(5,3)` Hamming code over GF(4). With `galois`, we can write the following:

```
>>> import galois

>>> n, k, m = 5, 3, 2
>>> Ik = GF4.Identity(k)
>>> Ink = GF4.Identity(n - k)

>>> # Matrix P in GF(4)
>>> P = GF4([[1, 1],
...         [1, 2],
...         [1, 3]])  # Example

>>> G = np.hstack([Ik, P])
>>> H = np.hstack([P.T, Ink])
>>> # Test that G*H^T = 0
>>> G @ H.T
GF([[0, 0],
    [0, 0],
    [0, 0]], order=2^2)
```

7.3 Reed-Solomon Codes via FFTs

Again, one could implement these via custom polynomials or rely on libraries that provide FFT-based methods in finite fields. For example, in `galois`:

```
>>> # Example: Reed-Solomon code over GF(256) => p=2, m=8
>>> import galois
>>> GF256 = galois.GF(2**8,irreducible_poly="x^8 + x^4 + x^3 + x + 1")
>>> q, r = 2**8, 6
>>> n, k = q - 1, n-r

>>> # Example of "FFT" approach depends on the library
>>> # Generate random message of length k
>>> msg = GF256(np.random.randint(0, q, size=k))
>>> # Zero-pad to length n
>>> u = np.concatenate([msg, GF256.Zeros(n-k)])
>>> # Evaluate polynomial with an FFT approach (library-dependent)
>>> x = np.fft.fft(u)  # Hypothetical function call

>>> # Add a random error of weight ne
>>> ne = 4
>>> e = GF256.Zeros(n)
>>> loc = np.random.permutation(n)
>>> mag = GF256(np.random.randint(1, q, size=ne))
>>> for i in range(ne):
...     e[loc[i]] = mag[i]

>>> y = x + e
>>> syn = np.fft.ifft(y)[k:]  # GF ifft
```