Sageにおけるコーディング理論

1　導入

本稿は、コーディング理論に関する Sage の使い方を知りたい初心者のために書かれています。Sage のコーディング理論ライブラリの重要な機能をいくつか紹介するとともに、必要なクラスとメソッドを見つける方法を説明します。

本稿では、以下の内容について説明します。

- 一般的な線形コード及びそれに関するメソッドについて
- 構造化されたコードファミリで何ができるか
- エンコード及び誤り訂正について
- 符号語へ簡単にエラーを乗せる方法

本稿の目的は、ライブラリでできることや、主な機能の使い方を簡単に概観することです。よって、全てのメソッドや特定のクラスの包括的な説明をするわけではありません。もし、コーディング理論におけるクラス・メソッドの詳細な情報が必要な場合は、そのクラス・メソッドのドキュメントを読む必要があります。

2　一般的な線形コード及びそれに関するメソッド

まず、最も一般的なコード (*generic linear code*) の1 つを作成します。*generic linear code* は、特殊な構造を持たない一般的な線形コードです。そのようなコードを作成するためには、次のように *generator matrix* を記述します。

```
sage: G = matrix(GF(3), [[1, 0, 0, 0, 1, 2, 1],
....:                    [0, 1, 0, 0, 2, 1, 0],
....:                    [0, 0, 1, 2, 2, 2, 2]])
sage: C = LinearCode(G)
```

以上で、線形コードが作成できました。Congratulations!

ここで、次に例示されるように、フルランクでない行列を渡すと、コードをビルドする前に Sage によってフルランク行列に変換されることに注意してください。

```
sage: G = matrix(GF(3), [[1, 0, 0, 0, 1, 2, 1],
....:                    [0, 1, 0, 0, 2, 1, 0],
....:                    [0, 0, 1, 2, 2, 2, 2],
....:                    [1, 0, 1, 2, 0, 1, 0]]) #r3 = r0 + r2
sage: C = LinearCode(G)
```

```
sage: C.generator_matrix()
[1 0 0 0 1 2 1]
[0 1 0 0 2 1 0]
[0 0 1 2 2 2 2]
```

ここまでの手順で、線形コードを使うことができるようになりました。あなたはそれで何ができると思いますか? ―多くのことを行うことができます。まずは、基本的な機能から始めてみましょう。

上の例では、既にコードの *generator matrix(*基本パラメータ*)*を求めています。次のように入力することで、それらの基本パラメータ (*length* と *dimension*) を問い合わせることができます。

```
sage: C.length()
7
sage: C.dimension()
3
```

さらに、次のようにコードの最小距離 (*minimum distance*) を求めることもできます。

```
sage: C.minimum_distance()
3
```

もちろん、C は一般的な線形コードなので、最小距離を見つけるために網羅的な探索アルゴリズムが用いられます。よって、コードが大きくなるにつれて探索スピードは遅くなります。

次のように、コードの名前を入力するだけで、そのコードの簡単な説明とパラメータを与える文を得ることができます。

```
sage: C
[7, 3] linear code over GF(3)
```

本稿の目的は、メソッドの包括的な解説をすることではないので、他のメソッドについての説明は割愛します。線形コードで実行できる全てのメソッドを取得したい場合は、次の 2 つの方法があります。

- 線形コードの一般構造ファイルのマニュアルページをチェックする
- Sage に次のように入力する

```
C.<tab>
```

C の利用可能な全てのメソッドのリストを取得します。その後、次のように入力します。

```
C.<method>?
```

メソッドのマニュアルページを見ることができます。

3 コードファミリーの構造とエンコード・デコードシステムの概観

3.1 Sageを用いて特殊なコードを作成する

前セクションで一般的な線形コードを作成する方法を説明したので、次はもう少し踏み込んで具体的なコードファミリを作成します。Sageでは全てのコードファミリに次のように入力することでアクセスできます。

```
codes.<tab>
```

上記により、Sageが構築できる全てのコードファミリの包括的なリストを得ることができます。

本パートの後半では、`sage.coding.grs.GeneralizedReedSolomonCode`クラスを操作し、これらのコードファミリの特殊な機能を説明します。まず、Generalized Reed-Solomon (GRS) コードを作成します。次のように入力します。

```
codes.GeneralizedReedSolomonCode?
```

GRSコードのドキュメントページに遷移します。そのページから定義を探し、Sageでビルドするために必要なコードを学ぶことができます。

ここでは、F_{13} 以上の [12,6] GRSコードを作成します。そのためには、最大で以下の3つの要素が必要となります。

- 評価点のリスト
- コードの次元
- 列の乗数のリスト (オプション)

以下のようにコードを構築します。

```
sage: F = GF(13)
sage: length, dimension = 12, 6
sage: evaluation_pts = F.list()[:length]
sage: column_mults = F.list()[1:length+1]
sage: C = codes.GeneralizedReedSolomonCode(evaluation_pts, dimension, column_mults)
```

GRSコードが作成されました。前セクションと同様に、パラメータを問い合わせることができます。

```
sage: C.length()
12
sage: C.dimension()
6
sage: C.base_ring()
Finite Field of size 13
```

C.evaluation_points() と C.column_multipliers() を呼び出すことで、評価点と列乗数を求めることもできます。

GRS コードの理論を知っていれば、最小距離を計算することは簡単です。次の式で求めることができます。

$$d = n - k + 1,$$

ここで、n はコードの長さ、k はコードの次元です。

Sage は C が GRS コードだと認識しています。よって、C の最小距離を探索するために前述した網羅的な探索アルゴリズムを実行せず、上記の処理を行います。そのため、すぐに結果を得ることができます。

```
sage: C.minimum_distance()
7
```

これらのパラメータはすべて、先と同様にコードの文字列表現にまとめられています。

```
sage: C
[12, 6, 7] Generalized Reed-Solomon Code over GF(13)
```

3.2 Sage によるエンコード・デコード

前パートでは、Sage で特定のコードファミリを検索し、これらファミリのインスタンスを作成する方法を学習しました。本パートでは、エンコードとデコードの方法を学びます。

まず、コードワードを生成し、それを再生します。そのためには 2 つの方法があります。

- 次のように、コードのランダム要素を生成します。

```
sage: c = C.random_element()
sage: c in C
True
```

- メッセージを作成し、これをコードワードにエンコードします。

```
sage: msg = random_vector(C.base_field(), C.dimension())
sage: c = C.encode(msg)
sage: c in C
True
```

いずれの方法にしても、コードワードを得ることができました。

次に、コードワードにいくつかのエラーを入れ、その後でそれらのエラーを修正してみましょう。コードワードに対して、ランダムにエラーを加え変化させることが可能なのは明らかですが、ここではより一般的なエラー (*communication channels*) を加えましょう。sage.coding.channel_constructions.Channel のオブジェクトは、通信チャンネル及びデータ表現の操作のためのオブジェクトであることを意味しています。今回は、多くはないエラーが乗ってしまった通信路をエミュレートしたいと思います。次のように入力します。

```
sage: err = 3
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), err)
sage: Chan
Static error rate channel creating 3 errors, of input and output space
Vector space of dimension 12 over Finite Field of size 13
sage: r = Chan.transmit(c)
sage: len((c-r).nonzero_positions())
3
```

ここで紹介した *Channels* についてより多くのことを学びたい場合は、本稿のセクション 4 を参照してください。

先のチャンネルにより、r という受信した単語が、エラーを含むコードワードとして得られました。次のようにして、エラーを修正し、元のコードワードを復元することができます。

```
sage: c_dec = C.decode_to_code(r)
sage: c_dec == c
True
```

コードワードではなく元のメッセージを戻したい時は、次のようにメッセージスペースに戻してエンコードを解除するだけです。

```
sage: m_unenc = C.unencode(c_dec)
sage: m_unenc == msg
True
```

次に示すように、前述した 2 つの処理(エラーを修正して元のメッセージを復元する) を 1 行で実行することもできます。

```
sage: m_unenc2 = C.decode_to_message(r)
sage: m_unenc2 == msg
True
```

4 エンコーダとデコーダの構造の詳細

前のセクションでは、コードオブジェクトに含まれているメソッドを使用することで、ベクトルのエンコード、デコード、およびエンコード解除を簡単に行えることを確認しました。これらのメソッドは、実際にはショートカットであり、エンコードとデコードがどのようなメカニズムで行われているかを気にかけない現代におけるユーザビリティのために追加されたものです。しかし、場合によっては、より細かな制御が必要な場合があります。そこで、このセクションでは、エンコーダとデコーダのメカニズムについて詳しく説明します。

根本的には、前述の 3 つの操作(エンコード、デコード及びエンコード解除) は sage.coding.encoder.Encoder と sage.coding.decoder.Decoder によって処理されます。これらのオブジェクトは、自身のメソッドを扱うための独自のワードを持っています。次のように呼び出した時,

```
C.encode(msg)
```

実際には、C のデフォルト のエンコーダで sage.coding.encoder.Encoder.encode() が呼び出されています。どのような方法で他のエンコーダやデコーダを探すことができるのか見てみましょう。

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12, GF(59).list()[1:41])
sage: C.encoders_available()
['EvaluationPolynomial', 'EvaluationVector', 'Systematic']
sage: C.decoders_available()
['BerlekampWelch',
 'ErrorErasure',
 'Gao',
 'GuruswamiSudan',
 'InformationSet',
 'KeyEquationSyndrome',
 'NearestNeighbor',
 'Syndrome']
```

上記より、GRS コード のために利用可能なエンコーダとデコーダのリ ストを得ました。これまでのようにデフォルト のものを使用するのではなく、特定のエンコーダとデコーダへ尋ねることができます。

```
sage: Evect = C.encoder("EvaluationVector")
sage: Evect
Evaluation vector-style encoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
sage: type(Evect)
<class 'sage.coding.grs.GRSEvaluationVectorEncoder'>
sage: msg = random_vector(GF(59), C.dimension()) #random
sage: c = Evect.encode(msg)
sage: NN = C.decoder("NearestNeighbor")
sage: NN
Nearest neighbor decoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
```

つまり、呼び出すときは次のように入力します。

```
C.encoder(<encoder_name>)
```

実際には、sage.coding.grs.EncoderGRSEvaluationVector のコンストラクタを先述のように自分で呼び出し、手動でエンコーダを構築することは簡単です。<encoder_name>を sage.coding.linear_code.AbstractLinearCode.encoder() に指定しない場合、コード のデフォルト エンコーダーが取得されます。

sage.coding.linear_code.AbstractLinearCode.encoder() には重要な余禄があります。それは、 再使用する前に、**構築されたエンコーダをキャッシュすること**です。これは、C の同じ EvaluationVector エンコーダに、毎回アクセスすることを意味します。それによって構築時間が節約されます。

上記の全ては、デコーダの場合でも同様です。エンコーダとデコーダはほとんど構築されず、何度も同じものが使用されるため、将来的な使用のために構築や初回使用時に高負荷な事前計算を行うことができます。

このことは、要素が内部的にどのように機能するかについて良い考えを与えます。具体的なポイントについてもう少し詳しく説明しましょう。

4.1 メッセージスペース

エンコーダのポイントは、メッセージをコードにエンコードすることです。これらのメッセージの多くは、コードの基底フィールド上のベクトルであり、その長さはコードの次元と一致します。しかし、それは他のフィールド上のベクトルや多項式、それらとはかなり異なるものなど、何でも構いません。したがって、各エンコーダは `sage.coding.encoder.Encoder.message_space()` を有します。例えば、GRS コードには 2 つの使用可能なエンコーダがあることを前パートで見ました。前パートで使用したものを調べてみましょう。

```
sage: Epoly = C.encoder("EvaluationPolynomial")
sage: Epoly
Evaluation polynomial-style encoder for [40, 12, 29] Generalized Reed-Solomon Code over GF(59)
sage: Epoly.message_space()
Univariate Polynomial Ring in x over Finite Field of size 59
sage: msg_p = Epoly.message_space().random_element(degree=C.dimension()-1) #random
sage: msg_p
31*x^11 + 49*x^10 + 56*x^9 + 31*x^8 + 36*x^6 + 58*x^5 + 9*x^4 + 17*x^3 + 29*x^2 + 50*x + 46
```

`Epoly` は、多項式の評価として GRS コードが構築されることが多く、それらのメッセージを考慮する自然な方法は、最大 $k-1$ の次数の多項式であることを反映しています。ここで、k はコードの次元です。`Epoly` のメッセージスペースは、すべての単変量多項式であることに留意してください。`message_space` はメッセージの周囲空間であり、エンコーダはメッセージが実際にその部分空間から選択されることを要求することがあります。

コードのデフォルトのエンコーダは、常にメッセージ空間としてのベクトル空間を持ちます。したがって、最初の例に示すように、コード自体で `C.decode_to_message()` または `C.unencode()` を呼び出すと、コードの長さと同じ長さを持つベクトルが常に返されます。

4.2 正則行列

エンコーダのメッセージスペースがベクトル空間であり、そのエンコードにリニアマップが使用されるときはいつでも、そのエンコーダは生成行列（この概念は他の種類のエンコーダには意味がありません）を持ちます。そしてそれは、その線形マップを特定します。

コードには多くの生成行列があるため、生成行列はエンコーダ・オブジェクトに配置されていますが、これらはそれぞれメッセージを別々に符号化します。また、コードオブジェクトに対して `sage.coding.linear_code.AbstractLinearCode.generator_matrix()` を見つけることもできますが、これは単純に、クエリーのデフォルトのエンコーダーに転送する便利な方法です。

私たちが構築した最初のエンコーダを使用して、Sage でこれを見てみましょう：

```
sage: Evect.message_space()
Vector space of dimension 12 over Finite Field of size 59
sage: G = Evect.generator_matrix()
sage: G == C.generator_matrix()
True
```

4.3 デコーダとメッセージ

先でも見たように、どのコードにも decode_to_codeword と decode_to_message という 2 つの一般的なデコード方法があります。すべてのデコーダにもこれらの 2 つのメソッドがあり、コードのメソッドはこのコードのデフォルトのデコーダに呼び出しを転送するだけです。

これらの 2 つの方法を持つ理由は 2 つあります。利便性と速度です。利便性は明らかで、両方の方法を持つことによって、ユーザーのニーズに応じた便利なショートカットを提供することができます。速度に関しては、いくつかのデコーダは自然にコードワードにデコードし、他のデコーダはメッセージ空間に直接デコードします。したがって、両方の方法をサポートすることで、エンコードおよびエンコードの不必要な作業を回避することができます。

しかし、decode_to_message は、メッセージ空間とその空間が、コードのエンコードを行っているバックグラウンドに存在することを含意しています。デコーダーには 2 つのメソッド（ message_space と connected_encoder）があり、ユーザにこれを通知します。長い例を挙げてみましょう。

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[1:41], 3, GF(59).list()[1:41])
sage: c = C.random_element()
sage: c in C
True

#Create two decoders: Syndrome and Gao
sage: Syn = C.decoder("KeyEquationSyndrome")
sage: Gao = C.decoder("Gao")

#Check their message spaces
sage: Syn.message_space()
Vector space of dimension 3 over Finite Field of size 59
sage: Gao.message_space()
Univariate Polynomial Ring in x over Finite Field of size 59

#and now we unencode
sage: Syn.decode_to_message(c) #random
(55,9,43)

sage: Gao.decode_to_message(c) #random
43*x^2 + 9*x + 55
```

5 チャンネルについての詳細

セクション 3 で、文字中にエラーを付ける方法としてチャンネルオブジェクトを紹介しました。このセッションでは、その機能をもっと詳しく見ていくことと、セカンドチャンネルを紹

介します。

> **注意**:再びこのセクション中の実行例として特定のクラスを選びました。それは、全てのチャンネルを網羅した一覧を作りたいわけではないためです。次のように sage に打ち込むことで、このリストが得られます。

```
channels.<tab>
```

再びセクション 3 で使った、**ChannelStaticErrorRate()** について考えます。これは、送られてきたベクトルに、制限された境界内のエラーを乗せるチャンネルです。この境界を得るためには、二つの方法があります。

- 一つ目は、セクション 3 で説明したもので、整数を渡すものです。次のようになります。

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
sage: t = 14
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), t)
sage: Chan
Static error rate channel creating 14 errors, of input and output space Vector space
of dimension 40 over Finite Field of size 59
```

- 次は、二つの整数の組を渡します。このとき、一つ目の整数は、二つ目の整数より小さいものにします。文字が送られた後に、この二つの整数間における乱数エラーが生成されます。

```
sage: t = (1, 14)
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), t)
sage: Chan
Static error rate channel creating between 1 and 14 errors, of input and output space
Vector space of dimension 40 over Finite Field of size 59
```

チャンネルには、チャンネルへの伝達を行う、 `transmit()` メソッドがあることを前セクションで学びました。このとき、リターンされる文字にはエラーが乗っていることがあります。また、このメソッドでは、送られた文字がチャンネルの入力スペースにあるかを常にチェックしています。一つの文字が入力スペースにあることが確実の場合、このチェックは冗長なものとなります。なぜなら、チェックには時間がかかるからです。それは特に、何百万回もシミュレートする場合に顕著です。そこで、`transmit_unsafe()` を使用します。これは `transmit()` と同じ働きをしますが、入力のチェックがないものとなっています。

```
sage: c = C.random_element()
sage: c in C
True
sage: c_trans = Chan.transmit_unsafe(c)
sage: c_trans in C
False
```

次は `transmit()` の便利なショートカットです。

```
sage: r = Chan(c)
sage: r in C
False
```

5.1 エラー及び削除に関するチャンネル

エラーと削除の追加をするチャンネルオブジェクトを紹介します。文字が送られてきたとき、エラーが追加されると、その分その場所が削除されます。

```
sage: Chan = channels.ErrorErasureChannel(C.ambient_space(), 3, 4)
sage: Chan
Error-and-erasure channel creating 3 errors and 4 erasures of input space Vector space of
dimension 40 over Finite Field of size 59 and output space The Cartesian product of
(Vector space of dimension 40 over Finite Field of size 59, Vector space of dimension 40
over Finite Field of size 2)
```

最初のパラメータはチャンネルの入力スペースを表しています。その次の二つは、それぞれエラーの数値と、削除箇所の数値を表しています。これらも StaticErrorRateChannel のときと同様に、二つの組で表示されます。このチャンネルとは対照的に ErrorErasureChannel の出力は入力スペースと同じように表されません。つまり、C 言語の空間と同じということになります。そこで、二つのベクトルを返します。一つ目のベクトルはエラーが追加され、消去位置が0 に設定され、送られてきた文字です。(一つめに送られてきたワードは、エラーが追加され、消去位置が0 に設定されています) 二つ目は消去された位置に1 を含む削除ベクトルです。これは sage.coding.channel_constructions.output_space() 内で繰り返されます。

```
sage: C = codes.random_linear_code(GF(7), 10, 5)
sage: Chan.output_space()
The Cartesian product of (Vector space of dimension 40 over Finite Field of size 59,
Vector space of dimension 40 over Finite Field of size 2)
sage: Chan(c) # random
((0, 3, 6, 4, 4, 0, 1, 0, 0, 1),
 (1, 0, 0, 0, 0, 1, 0, 0, 1, 0))
```

エラーと削除が重複することは、構造上ありえません。e のエラーと t の削除を入力すれば、e のエラーがのり、t が消去されたベクトルを得られるでしょう。

6 まとめ

このセクションで、符号化理論の説明を終わりにします。これを読んだ後なら、Sage で符号をつくり、うまく扱うだけの十分な知識がついたことでしょう！

今回の指導で、ライブラリーの内容すべてを説明したわけではありません。例えば、Sage によるコード限界の処理についての説明をしていません。符号化理論の構造体やメソッド (メンバ関数) などの、全オブジェクトは Sage の接頭符号に内包されています。よって、次のようにコードを打ち込んでも、ビルドすることができます。

```
codes.encoders.<tab>
```

したがって、コードに関連する特定のオブジェクトを探している場合は、次のように入力する必要があります。

```
codes.<tab>
```

もし、他に必要なサブカテゴリがあれば、自分で探してみてください!