

画像ファイル生成による ANCI-Cでグラフを描くプログラム

神戸大学大学院 数学専攻
高山ゼミ 山口晃広

目 次

1	要旨	2
2	予備知識	3
3	BMP ファイル構造	3
3.1	イメージデータの仕組み	5
4	GIF ファイル構造	6
4.1	LZW 圧縮法	9
4.2	LZW 辞書	9
4.3	辞書テーブルからイメージデータの作成	12
5	描画アルゴリズム	13
5.1	線分の描画	13
5.2	ブレセンハムの線分描画アルゴリズム	13
5.3	線分のクリッピング	15
5.4	真円の描画	16
5.5	楕円の描画	17
5.6	楕円の弧の描画範囲	19
6	簡単な使い方	21
6.1	cbmp.c(cbmp.h) の使い方	21
6.2	to_cbmp.c(to_cbmp.h) の使い方	24
6.3	cgif.c(cgif.h) の使い方	25

1 要旨

研究者や技術者の方々は、C言語でプログラムを組んだ結果を数値で眺めることが多いと思います。そのような時、簡単にグラフを表示することができれば、視覚的に結果を捉えることができ、作業がはかどることと思います。自分自身も、このプログラムを使って画像処理の研究をしています。

このようなニーズから、私はC言語で簡単にグラフを描くプログラムを作成しました。

C言語のグラフィックス・ライブラリには、OpenGLやGWinなどがあります。しかしこのようなライブラリでは、ライブラリの中でグラフィックを表示するところまで作成している為に、OSが変わると使えなくなってしまったり、コンパイラの種類に依存してしまいます。

そこで私が採った方法は、画像を表示する部分はブラウザやペイントなどといった他の画像を表示できるソフトに任せてしまい、代わりにBMPやGIFといった画像ファイルを生成するというものです。

そうすることで、ANCI-Cのみで書かれたシンプルなプログラムとなり、OSやコンパイラに依存しないライブラリとなりました。サイズが小さく、どのような環境でも気軽に使えるということが、このライブラリの良いところです。

BMPやGIFといった画像ファイルを生成するプログラム自体は、MS-PaintやGIMPなどといったグラフィックエディタがあります。これらのソフトとの違いは、ソースの中で直接画像ファイルの大きさを決める関数、点や線を描く関数を呼び出して画像ファイルを生成するところです。そして、この部分が私の新しく開発したところになります。

現在、BMPとGIFファイルを生成するプログラムを作成しています。
以下、

- BMPフォーマット
- GIFフォーマット
- LZW圧縮の仕組み
- 簡単な使い方
- 描画アルゴリズム

を説明していきたいと思います。

2 予備知識

- コンピュータに表示させる画像は、拡大すると格子状に正方形が並んでいる。その正方形一つ一つを画素と言い、その画素の色に対応する数値が格納されている。それは、行列で表現できる。
- 任意の色は、赤青緑を重ね合わせることで表現できる。(光の3原色) BMPフォーマットでもGIFフォーマットでも、この方法でイメージを格納しています。(但し、色のテーブルを参照する形にしたり、GIF画像は、圧縮して格納されています。(後述))
- どのようなファイルも2進数のデータ列であるが、そのファイルに書かれた内容を記述する規定がフォーマット(ファイル構造)である。通常の英文のテキストファイルなら、1byte単位で一文字を表すというきまりです。BMPファイルなら、例えば最初の2byteでそれが、BMPファイルであることを'B'M'(0x42 0x4d)と表し、以下画像の大きさ、イメージのデータ列などが続きます。
以下、Texソースをダンプした例です。

```
$ dump draw_graph.tex | head
draw_graph.tex:
```

Addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	2	4	6	8	A	C	E
00000000	5c	64	6f	63	75	6d	65	6e	74	63	6c	61	73	73	7b	6a	\documentclass{j							
00000010	61	72	74	69	63	6c	65	7d	0d	0a	0d	0a	5c	74	69	74	article}	...	\tit					
00000020	6c	65	7b	89	e6	91	9c	83	74	83	40	83	43	83	8b	82	le{.f...t.@.C...							
00000030	f0	90	b6	90	ac	82	b7	82	e9	82	b1	82	c6	82	c9	82	p.6.,.7.i.1.F.I.							
00000040	e6	82	e8	41	4e	43	49	2d	43	82	cc	82	dd	82	c5	83	f.hANCI-C.L.].E.							
00000050	4f	83	89	83	74	82	f0	95	60	82	ad	83	76	83	8d	83	0...t.p.'.-.v...							

3 BMP ファイル構造

思い通りの画像ファイルを生成するプログラムを作成するには、まずそのファイル構造を知る必要があります。BMP フォーマットに関する文献としては[1, 2]などを参考にしました。

BMP ファイルフォーマットには、

- Windows Bitmap

- OS/2 Bitmap

の 2 種類があり以下のようになっています。

- ファイルヘッダ (BITMAPFILEHEADER)
 - ファイルタイプ 'B' 'M' (2byte 使用)
 - ファイルサイズ (4byte 使用)
 - 4byte 無視 (常に 0)
 - ファイル先頭から画像データまでのバイト数 (4byte 使用)
- 情報ヘッダ (OS/2 Bitmap の場合: BITMAPCOREHEADER)
 - 情報ヘッダのサイズ (byte 単位)12 が入る (2byte 使用) (12 なら OS/2 Bitmap、40 なら Windows Bitmap)
 - 画像の幅 (ピクセル単位) (2byte 使用)
 - 画像の高さ (ピクセル単位) (2byte 使用)

画像は高さが正数ならボトムアップ (左下から右上へ) 高さが負数ならトップダウン (左上から右下へ)
 - 2byte 省略 (常に 1)
 - 1 画素あたりのデータサイズ (bit 単位) (2byte 使用)
 - 1 2 色ビットマップ
 - 4 16 色ビットマップ
 - 8 256 色ビットマップ
 - 24 1677 万色 (true color) ビットマップ
 - 32 1677 万色 (true color) ビットマップ
- 情報ヘッダ (Windows Bitmap の場合 BITMAPINFOHEADER) (私の BMP 出力プログラムでは Windows Bitmap を採用してます。)
 - 情報ヘッダのサイズ Windows Bitmap なので 40 が入る (byte 単位) (4byte 使用)
 - 画像の幅 (ピクセル)(4byte 使用)
 - 画像の高さ (ピクセル) (4byte 使用) 値が正数なら画像データは下から上へ。値が負数なら画像データは上から下へ。

- 常に 1 (2byte 使用)
 - 1 画素あたりのデータサイズ (bit 単位) (2byte 使用) cbmp.c での BMP 出力は 24bit 色ビットマップで行っています。
 - 1 2 色ビットマップ
 - 4 16 色ビットマップ
 - 8 256 色ビットマップ
 - 24 1677 万色 (true color) ビットマップ
 - 32 1677 万色 (true color) ビットマップ
 - 圧縮形式 (0:無圧縮を指定)(4byte 使用)

圧縮された BMP ファイルは、私のプログラムでは読み込めません。ですが圧縮 BMP ファイルに出会ったことはありません。
 - 画像データ部のサイズ 4byte 使用
 - 1m あたりの横方向の画素数 4byte 使用 (私の bmp 出力プログラムでは 0 にしています。)
 - 1m あたりの縦方向の画素数 4byte 使用 (私の bmp 出力プログラムでは 0 にしています。)
 - 格納されているパレット数 4byte 使用

2, 4, 8bit 色の場合は、カラーテーブルの数が記録されます。0 の場合は、全部使用で、上で指定した 1 画素あたりのデータサイズから値を求めます。(2 乗する) (私の BMP 出力プログラムでは 0 にしています。)
 - 重要なパレットのインデックス 4byte 使用 (私の BMP 出力プログラムでは 0 にしています。)
- 1, 2, 4, 8Bit の場合のみカラーテーブルが作られる。
 - イメージデータ

3.1 イメージデータの仕組み

まず、1, 4, 8, 24, 32Bit 色のビットマップに共通して言えることは、幅を 4byte 境界に合わせる必要があるということです。例えば 8bit 色, 99×99

ピクセルのデータは、幅が 4byte で割り切れません。このようなときは、余分に 1byte 加えて 100×99 のピクセルデータとして格納しています。余分なデータには 0 を入れます。

但し、32Bit 色の BMP ファイルは必ずピクセルデータの幅が 4 の倍数になるのでこの問題は起きません。

24Bit 色のビットマップは一つの画素に対して、青緑赤の順に 1byte ずつデータを割り当てています。つまり、1 ピクセルあたり 3byte 使って、画像を表現しています。32Bit 色のビットマップは青緑赤の次に 0 が 1byte 分入ります。

それに対し 1, 4, 8Bit 色は、画素の色を指定するのにカラーテーブルを用い、そのテーブルのインデックスをイメージデータに格納しています。

4 GIF ファイル構造

GIF フォーマットに関する文献は、[3, 4, 5]などを参考にしました。

- G I F H e a d e r

- 最初に'G' 'I' 'F' '8' '9' 'a' と入っています。
- G I F ファイルの横幅と高さの順に、それぞれ 2 バイトずつ入っています。
 - * 0x1234 なら、0x34,0x12 の順に格納されています。
- 1 バイト分省略
- グローバルカラーテーブルにおける背景色のインデックスが入っています。
- 1 バイト省略
- グローバルカラーテーブルが最後に続きます。
私の作成したプログラムでは 2 5 6 色扱うので、2 5 6 × 3 バイトあります。
(× 3 は、光の三原色で、R , G , B それぞれ 1 バイト使う為)
私の作成したプログラムでは、

- * 最初に基本 1 6 色を格納します。(1 6 × 3 バイト)
- * 次に未使用部分が、2 4 × 3 バイトあります。

* 最後に 2 1 6 色 (2 1 6 × 3 バイト) を

```
0x 00 00 00 -> 0x 00 00 33 -> 0x 00 00 66 -> -> 0x 00 00 ff ->  
0x 00 33 00 -> 0x 00 33 33 -> 0x 00 33 66 -> -> 0x 00 33 ff ->  
0x ff ff 00 -> ... -> 0x ff ff 99 -> 0x ff ff cc -> 0x ff ff ff
```

と格納します。

(色の指定を 0x*****で行ったときは、上の 2 1 6 色の中で最も近い色に近似します。)

● アプリケーション拡張ブロック

cgifNumberOfIterations が 1 の場合 (動画にしない場合) はこのブロックは飛ばします。

- 最初は 0x21,0x0b で、アプリケーション拡張ブロックであることを示す固定値です。
- 次に、'N"E"T"S"C"A"P"E"2"."0' と入っています。
- 2 バイト省略
- 2 バイト分使って、動画を繰り返す回数を指定します。
 - * cgifNumberOfIterations = 0; で無限回繰り返します。
- 最後はブロックの並びの終わりを表す 0x00 で終了します。

● 以下、グラフィック拡張ブロックとイメージブロックをセットで、アニメーションの数 (frame 数) だけ作成します。

● グラフィック拡張ブロック

- 最初は 0x21,0xf9 で、グラフィック拡張ブロックであることを示す固定値です。
- 1 バイト省略
- 次に表示処理パラメータ (PackedFields) が 1 バイト来ます。1 バイトを 3 ビット、3 ビット、1 ビット、1 ビットに分けます。
 - * 最下位の 1 ビットは、透過処理するかどうかを指定します。
 - * 次の 1 ビットは、処理を継続する前にユーザー入力が必要とするかを指定します。
(このあたりの機能は IE など今使用している環境では正しく動きませんでした。)

- * 次の3ビットは画像表示後の処理を指定します。
 - ・ 0：指定なし。
 - ・ 1：画像を残す。
 - ・ 2：背景色を回復。
 - ・ 3：以前のものを回復。
- * 最上位の3ビットは、予約済みで000で固定です。
- － 次は2バイトを使って次のイメージを表示するまでの待機時間(1/100秒)を指定します。
- － 透明色のカラーインデックスが来ます。
詳しくは、色の指定で説明します。
- － 最後にブロックの終了識別子0x00が来ます。

● イメージブロック

- － まず、イメージブロックであることを示す0x2cが来ます。
- － G I F画像全体に対するこのイメージの左端相対位置、上端相対位置の順にそれぞれ2バイトずつ記述します。
- － このイメージブロックの横幅、縦幅を順に2バイトずつ記述します。
- － 1バイト省略
- － このプログラムでは、ローカルカラーテーブルは使用しないので省かれ
次は、256色なら0x08が来ます。(Lzw Minimum Code Size)
- － 最後にイメージデータのバイト数とLZW圧縮されたイメージデータを格納します。
 - * LZW圧縮については、lzw圧縮法のセクションで説明します。
 - * イメージデータが256バイトを超えてしまった場合やLZW辞書がいっぱいになってしまった場合は、少しややこしいです。
私のプログラムでは、LZW辞書がDICT_MAXに達してしまったとき、
まず、最後のsuffixまで出力して、もう一度、イメージデー

タのバイト数、イメージデータ(クリアコード + prefix 達 + suffix) を繰り返し、最後に終了コードを入れます。

- 最後にブロックの並びの終わりを表す 0x00 を入れます。
- 最後に G I F データストリームの終わりを表す 0x3b を入れて終了です。

4.1 LZW 圧縮法

ここでは、cgif.c と cgif.h を作る上で特に大変だった、L Z W 圧縮法について説明したいと思います。

L Z W 圧縮法は、Unisys 社が特許を持っていましたが、2004 年 6 月 20 日、日本でも期限切れになり自由に使うことができるようになりました。

L Z W 圧縮をデータ列を登録する辞書を作成する部分と、実際にイメージデータをファイルに書き出す部分を分けて説明していきます。イメージの例として、図 1 を以下で考えます。

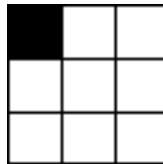


図 1: 3 × 3 の G I F 画像

4.2 LZW 辞書

G I F ファイルはカラーテーブルを持っていて、そのテーブルのインデックスを使って色を格納しています。

表 1: カラーテーブルの例

index	0	0	...	9	a	...	f	...
色	BLACK	MAROON		RED	LIME		WHITE	

例えば、表 1 のようなカラーテーブルを持っているとします。
ピクセルは左から右、上から下の順で

0 -> f -> f -> f -> f -> f -> f -> f -> f

のように配置されます。

a b c d 9 c a のようなデータ列がある時、

- 最後のデータを除いた部分 (この場合 a b c d 9 c) を prefix
- 最後のデータ (この場合 a) を suffix

と呼ぶことにします。

上で作成した「0 f f f f f f f f」というようなデータを頭から順に見てゆき、長さが 2 以上の新しい順列が現れるたびに、新しいコード (0x102 番から) を、その順列に割り当てていきます。(1 × 1 の GIF ファイルの時は、長さ 2 以上の順列を取り出すことができませんが、このときは prefix のない suffix だけの新しいコードを割り当てます。)

以下で具体的な手順を見ていきます。

1. まず 0 f が新しい順列なので、この順列にコード 0x102 を割り当てます。prefix が 0x0、suffix が 0xf となります。
2. 次に、前回取り出した順列の suffix から、新しい順列を調べていきます。
すると、f f が新しい順列として見つかりますので、この順列にコード 0x103 を割り当てます。prefix が 0xf、suffix が 0xf となります。
3. 次は f f という順列はすでに登場しているので、f f f が新しい順列となります。
4. この順列にコード 0x104 を割り当て、prefix を 0x103、suffix を 0xf とします。ここで、以前割り当てられた f f というデータ列のコードを prefix に使用することで、圧縮が起こりました。
5. 次は、f f f の順列のコードもすでに割り当てられているので、f f f f が新しい順列となります。この順列にコード 0x105 を割り当て、prefix が 0x104、suffix が 0xf となります。

6. 最後は、f f の順列にコード 0x106 を割り当て、prefix を 0xf、suffix を 0xf とします。

この様子を表に示すと、以下のようになります。

コードに割り当てられた内容	コード	prefix の中身	suffix の中身
カラーテーブルの index	0x0	無	無
	0x1	無	無
	0x2	無	無
	.	無	無
	.	無	無
	.	無	無
	0xff	無	無
クリアコード	0x100	無	無
終了コード	0x101	無	無
新しく作成されたコード	0x102	0x0	0xf
	0x103	0xf	0xf
	0x104	0x103	0xf
	0x105	0x104	0xf
	0x106	0xf	0xf

表 2: lzw 辞書テーブル

0 -> f -> f -> f -> f -> f -> f -> f -> f

だったものを prefix を登録した順に並べ、最後に登録した suffix をその後部に付け足したものが、

0 -> f -> 1 0 3 -> 1 0 4 -> f -> f

のように圧縮できているのが分かります。この圧縮は可逆圧縮です。

次は、prefix、suffix に登録されたコードを G I F ファイルの中でどのように格納しているかについて説明したいと思います。

4.3 辞書テーブルからイメージデータの作成

前回、図 1 の 3 × 3 の G I F 画像が、新しいデータ列が現れるたびにコードに割り当てることで、圧縮されていることを見ました。その結果、表 2 の prefix と suffix を持つ辞書テーブルが作成されました。

prefix を登録した順に並べ、最後に登録した suffix をその後部に付け足したデータ列

0 -> f -> 1 0 3 -> 1 0 4 -> f -> f

の最初にクリアコードを付け足し、最後に終了コードを付け足し、9bit で揃えたと以下ようになります。

```
clear code : 1 0000 0000
             0x0 : 0 0000 0000
             0xf : 0 0000 1111
0x103 : 1 0000 0011
0x104 : 1 0000 0100
             0xf : 0 0000 1111
             0xf : 0 0000 1111
end code : 1 0000 0001
```

それを

```
1 0000 0001 <- 0 0000 1111 <- 0 0000 1111 <-
1 0000 0100 <- 1 0000 0011 <-
0 0000 1111 <- 0 0000 0000 <- 1 0000 0000
```

と一列に並べて下の位から 8 ビット (1 バイト) ずつ格納していきます。

```
80 <- 83 <- C1 <- F0 <- 48 <- 18 <- 3C <- 01 <- 00
```

のようになります。

実際イメージは、このような L Z W 圧縮法でファイルに格納されています。今回の例では以下のように 0x334 ~ 0x33c に格納されています。

Addr	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	2	4	6	8	A	C	E
00000300	33ff	ff66	ffff	99ff	ffcc	ffff	ff21	ff0b	3..f.....L...!...															
00000310	4e45	5453	4341	5045	322e	3003	0100	0000	NETSCAPE2.0.....															
00000320	21f9	0401	1e00	1000	2c00	0000	0003	0003	!y.....,.....															
00000330	0000	0809	0001	3c18	48f0	c183	8000	3b<.HpA...;															

5 描画アルゴリズム

現在、cgif, cbmp とともに線分、円弧 (楕円も含む) の描画を実装しています。ここではそれを高速に描画するアルゴリズムを紹介します。

この節の直線、真円、楕円のブレセンハムのアルゴリズムについては、文献 [6] に書かれていることを書き直したものです。プログラムの方もここに書かれているプログラムを殆どそのまま使用させていただきました。

線分のクリッピングについては、文献 [7] をそのまま参照し、プログラムもこの本を真似て作りました。

楕円の弧の描画はオリジナルです。

アルゴリズムの基本方針は、図形描画開始点を (x, y) とした時、 x (または y) をインクリメントさせ、真の点の位置との誤差が最小になる方向へ y を移動させて描画します。誤差の比較が整数演算でできるため、誤差の蓄積が起らず、これらのアルゴリズムは高速になります。

5.1 線分の描画

現在私のプログラムでは、線分の描画は高速化を目的に、画像の範囲外にある線分を画像の範囲内の線分に変換して (この操作をクリッピングという)、ブレセンハムの線分描画アルゴリズムを適応しています。(もちろんクリッピングを行わなくても、線の描画関数は点の描画関数を呼び出し、範囲外の点は打たないだけのことです。)

以下で、これらのアルゴリズムを見ていきたいと思います。

5.2 ブレセンハムの線分描画アルゴリズム

(x_0, y_0) から (x_1, y_1) へ線を引くことを考えます。直線の式として、

- $|y_1 - y_0| \leq |x_1 - x_0|$ なら、 $y = m(x - x_0) + y_0$ ($m = \frac{y_1 - y_0}{x_1 - x_0}$)

- $|y_1 - y_0| > |x_1 - x_0|$ なら、 $y = m(x - x_0) + y_0$ ($m = \frac{y_1 - y_0}{x_1 - x_0}$)

を使います。

まず、誤差の比較を整数演算に変更しない自然な場合を考えます。以下、 $x_0 \leq x_1, y_0 \leq y_1$ の場合を考えます。

その中で特に $0 \leq m \leq 1$ の場合で説明します。 $m \geq 1$ の時は x と y を逆にして同様の処理を行います。

1. $x = x_0, y = y_0$ で初期化します。この時 (x, y) は真の線分上に存在するので真の値との誤差を e とすると $e = 0$ に初期化できます。
2. $x \leq x_1$ である限り以下を繰り返し実行します。
 - (a) 点 (x, y) をプロットします。
 - (b) x に 1 を加えます。
 - (c) x が 1 増加すると y は m 増加するので、真の値との誤差 e に m を加えておきます。
 - (d) 真の値との誤差を最小にする画素に点を打つので、 $0 \leq m \leq 1$ の条件下では、 y はそのままか、1 増加するかのいずれかです。 e が 0.5 以上の場合、 y が 1 増加した画素を選択し、新たな y 座標との誤差を示すよう e から 1 引きます。

次にこのアルゴリズムが整数演算のみで実行できるよう、 e を変更します。

1. 誤差の比較を正負の符号で判定できるようにする為、 $e = e - 0.5$ とします。
2. 次に e の初期値と m が未だ実数のままであるのでそこを変更します。
現在 e の比較は e の符号により判定するので、 e に適当な値をかけてもアルゴリズムは破綻しません。よって、 e の初期値と m に $2(x_1 - x_0)$ をかけることで整数化できます。

$x_0 > x_1$ または $y_0 > y_1$ の場合は、+1 する代わりに 1 ずつ減らすことで同様の処理となります。

この処理は、cgif.c の `cgif_line()`、cbmp.c の `cbmp_line()` で実装されています。

5.3 線分のクリッピング

線分の端点が画像の範囲外にある時、そのまま線分描画の関数を呼ぶと、画像の範囲外に対しても点を求める為、その分だけ遅くなります。

画像上と交わる線分を改めて求めてから、そこにだけ線を引くことで高速化を実現しています。

この画像と重なった部分を求める操作をクリッピングと呼びます。以下にその手順を示します。

まず、線分の始点、終点が画像に対してどの領域にあるかをそれぞれ 4bit のコードで表 3 で表します。

第 0bit	画面の左端を外れているかどうか	0101	0100	0110
第 1bit	画面の右端を外れているかどうか	0001	0000	0010
第 2bit	画面の上端を外れているかどうか	1001	1000	1010
第 3bit	画面の下端を外れているかどうか			

表 3: ビットコード対応表

このコードを使い、以下の手順でクリッピング処理します。

1. 始点、終点とも 4bit コードが 0 なら、両方とも画面の内側なのでそのまま線を引く関数に渡します。
2. 始点と終点の AND をとり、0 でない時は画面と重ならないので、線分を引かずに終了します。これは始点と終点が両方とも 1 であるビットがあるということですが、例えば、第 2 ビットが両方とも 1 なら線分全てが画像より上端に来ているということになり、画像とは交わらないからです。
3. 終点に注目し、終了の 4 ビットコードが 0 (終点が画像の内側) なら始点と終点を交換します。最初のチェックにはかからなかったのに、新しい終点の 4 ビットコードは 0 でないことになります。
4. 終点の 4 ビットコードのうち、1 になっているビットの中から 1 つ選びます。それを第 i ビットとして、以下の処理を行います。
 - (a) まず、2 番目のチェックにかからなかったのに、始点の第 i ビットは 0 のはずです。よって、第 i ビットがチェックの対応をし

ている画像の端をこの線分が横切ることになります。そこで、その画像の端との交点を求め、それを新しい終点とします。

(b) 次に、ビットコードを更新します。

5. 以上の処理を 1 番か 2 番のチェックにかかるまで繰り返します。はみ出していた線分も画像の端を延長する直線で少しずつ切られ、多くとも 4 回のループを回る間にクリッピングは完了します。

この処理は、cgif.c では cgif_looker(), cgif_line()、cbmp.c では cbmp_looker(), cbmp_line()、で実装されています。

5.4 真円の描画

真円の描画方法は以下のようになります。

- まず原点中心で、 $(r, 0)$ から反時計回りに半径 r の 8 分円を描くことを考える¹。
- $y++$ した時、 $x--$ するか否か (垂直方向に移動するか斜め方向に移動するか) のみ考えればよく、それぞれ移動した結果の誤差 $e := x^2 + y^2 - r^2$ の大小で決める。

$x--$ しない場合 (垂直方向 (vertical) に移動) の誤差

$$e_v := x^2 + (y+1)^2 - r^2$$

$x--$ する場合 (斜め方向 (diagonal) に移動) の誤差

$$e_d := (x-1)^2 + (y+1)^2 - r^2$$

- $e_v^2 - e_d^2 = (e+2y+1)^2 - (e-2x+2y+2)^2 = (2e-2x+4y+3)(2x-1)$ となり、上式の正負で移動する方向 ($x--$ するか否か) を決めることができる。ここで半径 $r > 0$ なら必ず $x \geq 1$ であるから²、 $(2x-1)$ で割って、 $e_r := 2e - 2x + 4y + 3$ の正負で判定できる。

$x--$ すると、 $e_r(x+1, y) - e_r(x, y) = 4x - 4$

¹理由: 真円は、 x 軸、 y 軸、直線 $y = x$ 、 $y = -x$ で対称より

²理由: x は必ず整数値であり、今描こうとしている 8 分円は y 軸と交わらない

– ++ y すると、 $e_r(x, y) - e_r(x, y - 1) = 4y + 2$

と e_r が変化することに注意して、 $e_r(r, 0) = -2r + 3$ から順次 e_r を求め、 $x \geq y$ である限り、 e_r 最小の方向へ描画がし続ける。

cbmp.c(またはcgif.c)では、cbmp_create_window(またはcgif_create_window)で作成した座標系に描いたグラフを伸縮して実際の画像に描き込みます。その為、真円を描画するには楕円を描画するアルゴリズムが必要になります。

それで、このアルゴリズムは私のプログラムの中では不要ですが、そのサンプルプログラムを表4に掲載します。

表 4: circle_algorithm.c

```
1 void circle(int x, int y, int r)
2 {
3     int er;
4
5     /* 開始点 (x=r,y=0) の各々の値を求める */
6     x = r;
7     y = 0;
8     er = -2 * r + 3; /* (x=r,y=0) 時の er */
9
10    /* 8 分円を描く */
11    while ( x >= y ) {
12        putpixel( x0 + x, y0 + y); /* 8 分円 */
13        putpixel( x0 - x, y0 + y); /* y 軸対称 */
14        putpixel( x0 + x, y0 - y); /* x 軸対称 */
15        putpixel( x0 - x, y0 - y); /* 原点对称 */
16        putpixel( x0 + y, y0 + x); /* 直線 y=x 対称 */
17        putpixel( x0 - y, y0 + x); /* y=x,x 軸対称 */
18        putpixel( x0 + y, y0 - x); /* y=x,y 軸対称 */
19        putpixel( x0 - y, y0 - x); /* 直線 y=-x 対称 */
20        /* 真の円弧により近くなるように、x をデクリメントするか決める */
21        if ( er >= 0 ) {
22            x--;
23            er -= 4 * x;
24        }
25        /* 必ず y はインクリメント */
26        y++;
27        er += 4 * y + 2;
28    }
29 }
```

5.5 楕円の描画

楕円の描画法は以下のようになります。

- まず原点中心で、 $(a, 0)$ から反時計回りに 4 分円を描くことを考える³。
- 真上か左上か真左のいずれに進むかをそれぞれ移動した結果の誤差 $e := (x/a)^2 + (y/b)^2 - 1$ が最小となるように決める。次のように各変数を

真上 (vertical) に移動時の誤差

$$e_v := (x/a)^2 + ((y+1)/b)^2 - 1$$

左上 (diagonal) に移動時の誤差

$$e_d := ((x-1)/a)^2 + ((y+1)/b)^2 - 1$$

真左 (left) に移動時の誤差

$$e_l := ((x-1)/a)^2 + (y/b)^2 - 1$$

- $Err_1 := 2e - 2(1/a^2)x + 4(1/b^2)y + 1/a^2 + 2(1/b^2)$
- $Err_2 := 2e - 4(1/a^2)x + 2(1/b^2)y + 2(1/a^2) + 1/b^2$
- $Err_3 := 2e - 2(1/a^2)x + 2(1/b^2)y + 1/a^2 + 1/b^2$
- $e_1 := ev^2 - ed^2 = (1/a^2)(2x-1)Err_1$
- $e_2 := ev^2 - el^2 = ((1/a^2)(2x-1) + (1/b^2)(2y+1))Err_3$
- $e_3 := ed^2 - el^2 = (1/b^2)(2y+1)Err_2$

とみると、

- $e_1 < 0$ and $e_2 < 0$ なら e_v が最小より真上に移動
- $e_3 < 0$ and $e_1 \geq 0$ なら e_d が最小より左上に移動
- $e_2 \geq 0$ and $e_3 \geq 0$ なら e_l が最小より真左に移動
- $x \geq 1, y \geq 0$ であることに注意すると、 e_1, e_2, e_3 の代わりに Err_1, Err_3, Err_2 の正負で判定できる。また、 $e_3 = e_1 - (1/b^2)(2y+1) < e_1$, $e_2 = e_3 - (1/a^2)(2x-1) < e_3$ より、 $e_2 < e_3 < e_1$ が必ず成り立つことから、最終的な判定は、整数化した式 $Er_1 := (a^2b^2)e_1, Er_2 := (a^2b^2)e_2$ を使い、

³理由: 真円は、 x 軸、 y 軸で対称より

- $Er_1 < 0$ なら真上に移動
- $Er_1 \geq 0, Er_2 < 0$ なら左上に移動
- $Er_2 \geq 0$ なら真左に移動

となる。

- $++y$ すると、

$$Er_1(x, y) - Er_1(x, y - 1) = 4a^2y + 2a^2$$

$$Er_2(x, y) - Er_2(x, y - 1) = 4a^2y$$

- $--x$ すると、

$$Er_1(x, y) - Er_1(x + 1, y) = -4b^2x$$

$$Er_2(x, y) - Er_2(x + 1, y) = -4b^2x + 2b^2$$

と Er_1, Er_2 が変化することに注意して、

$$Er_1(a, 0) = -2b^2a + b^2 + 2a^2$$

$$Er_2(a, 0) = -4b^2a + 2b^2 + a^2$$

から順次 Er_1, Er_2 を求める。

- 上記手順を $x \geq 0$ である限り続ける。

このプログラムは、cgif.c の `cgif_circle()`、cbmp.c の `cbmp_circle()` で実装されています

最後に弧の描画範囲を調べる方法を説明して、楕円描画アルゴリズムの話を終了します。

5.6 楕円の弧の描画範囲

ここでは、弧の描画範囲を現在どう決めているのか説明します。文献を調べて作ったわけではないので、もっと良い方法があるかもしれません。

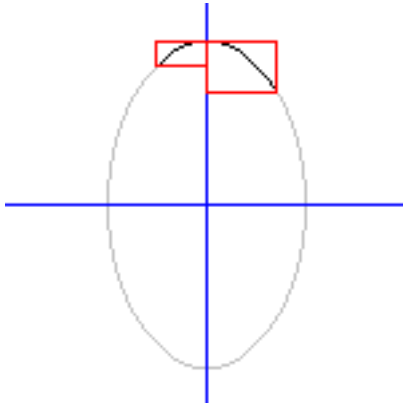
4分円ずつ描画しているので、それぞれの4分円で描画範囲を調べる必要があります。

まず場合分けを簡単にする為、入力された円弧描画の開始角度 $radian_s$ と終了角度 $radian_e$ を $0 \leq radian_s < 2\pi, 0 \leq radian_e - radian_s \leq 2\pi$ の範囲に直します。

まず以下の範囲は必ず描画します。

- 開始角度が4分円の最小角度より小さければ、その4分円の描画範囲は、4分円の最小角度から反時計回りの方向に存在。
- 開始角度が4分円の最小角度と最大角度の間にあれば、その4分円の描画範囲は、開始角度から反時計回りの方向に存在。
- 開始角度が4分円の最大角度より大きければ、その4分円は描画しません。
- 同様に、終了角度が4分円の最大角度より大きければ、その4分円の描画範囲は、4分円の最大角度から時計回りの方向に存在。
- 終了角度が4分円の最小角度と最大角度の間にあれば、その4分円の描画範囲は、終了角度から時計回りの方向に存在。
- 終了角度が4分円の最小角度より小さければ、その4分円は描画しません。

上の条件だけで描画できる例として、



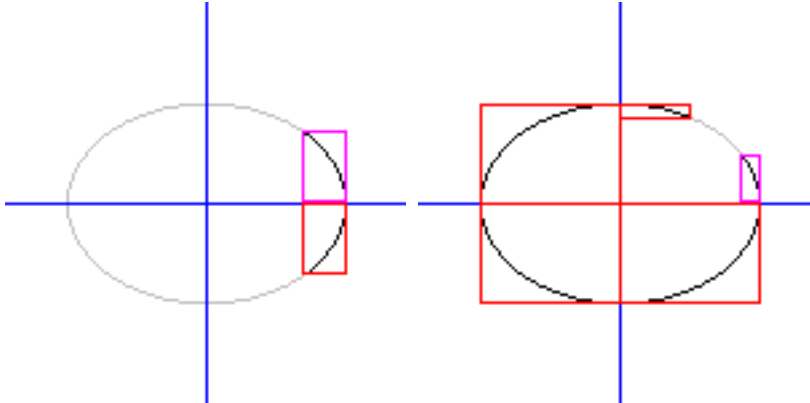
等があります。

加えて終了角度が 2π より大きい場合は、以下の範囲を満たす円弧も描画する必要があります。開始角度は、 2π より小さいはずなので、それぞれの4分円の描画範囲は、必ず4分円の最小角度から反時計回りの方向に存在します。

- 終了角度 -2π が4分円の最大角度より大きければ、その4分円の描画範囲は、4分円全範囲。
- 終了角度 -2π が4分円の最小角度と最大角度の間にあれば、その4分円の描画範囲は、4分円の最小角度から終了角度 -2π まで。

- 終了角度 -2π が4分円の最小角度より小さければ、その4分円は描画しません。

この条件で描画する例として、



等の紫色の範囲がこれに該当します。(赤色の範囲は、最初の条件で描画することになります。)

後は、反時計回り(または時計回り)に描画する時、 x についても y についても単調増加(または単調減少)することに注意して、上の図で描いた各々の4分円の描画する矩形領域を求め、円弧を描いています。

6 簡単な使い方

ここでは、実際に `cgif.c` と `cbmp.c` で何ができるのか、どうやって使うのかを説明していきたいと思います。A N C I - C 対応の全てのCコンパイラで、使う事ができると思います(確かめていません)が、ここでは `gcc` を使って説明します。

6.1 `cbmp.c(cbmp.h)` の使い方

まず、サンプルプログラムとして図5を用意します。以下このプログラムを使って説明します。

コンパイルの方法は、`cbmp.c`, `cbmp.h` `sample1_cbmp.c` を同一のディレクトリに置き、

```
gcc cbmp.c sample1_cbmp.c -lm
```

表 5: sample_cbmp.c

```

1  #include "cbmp.h" /* cbmp.c を使う時に必要 */
2  static double PI = 3.14159265358979323846;
3
4  int main()
5  {
6      type_cbmp *obj; /* BMP データのオブジェクト */
7      double x;
8
9      obj = cbmp_init(70,70); /* 大きさ 70*70 のビットマップを obj に作成 */
10     cbmp_create_window( -5, 10, 5, 0); /* 左下 (-5,10), 右上 (5,0) に座標変換 */
11
12     for ( x = -5; x < 5; x += 0.05){
13         cbmp_pset( x, x*x, 0xff0000, obj); /* (x,x*x) に赤色で点を打つ */
14     }
15     cbmp_line(0,2, 0,10, 0x0000ff, obj);/* (0,2) から (0,10) へ青色の線を引く*/
16     /* 中心 (1,4)、 x 軸方向の半径 2、 y 軸方向の半径 3 の楕円を
17        角度 (radian)PI/6.0 から 7.5*PI/4.0 まで、黒色で描く */
18     cbmp_circle(1,4, 2,3, PI/6.0,7.5*PI/4.0, 0x000000, obj);
19     cbmp_output("sample_cbmp.bmp", obj); /* obj の内容の sample1.bmp を作成 */
20     cbmp_free(obj); /* いらなくなったオブジェクト obj をフリー */
21     return 0;
22 }

```

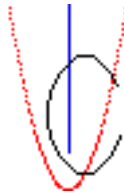


図 2: BMP 出力結果

などとします。

後は、作られた実行ファイルを実行すれば図2のようなBMPファイルが作られます。

`cbmp_init(width, height)` 最初に呼び出す関数です。width がBMP 画像の幅、height がBMP 画像の高さ（単位：ピクセル）となる `type_cbmp *`型のオブジェクトを作成し（そのポインタを）返します。

先に、`type_cbmp *`型のオブジェクトを定義しておく必要があります。

`cbmp_create_window(x1, y1, x2, y2)` BMP 画像に対応するウインドウを決定します。

x1 がウインドウの左上の点、x2 がウインドウの右下の点になります。

表5のプログラムでは、左上(-5, 10)、右下(5, 0)となる大きさ70×70のBMP画像が作られることになります。

`cbmp_pset(x, y, color, obj)` `cbmp_create_window` で指定したウインドウ (`cbmp_window`) に color の色の点を打ちます。

この場合 `cbmp_window` 上の座標 (x, x*x) に赤い点を打ちます。

color の指定方法は光の3原色を使い赤青緑の順にそれぞれの強さを 0x00 ~ 0xff で並べて指定します。0xff0000 なら赤の強さが 0xff、青と緑の強さは0になります。

obj は点を打つBMP オブジェクトを指します。

`cbmp_line(x0, y0, x1, y1, color, obj)` 端点を (x0, y0),(x1, y1) とする線分をBMP オブジェクト obj に color の色で描きます。

`cbmp_output(ファイル名, obj)` 指定したファイル名でBMP オブジェクト obj の内容を Bitmap ファイルとして出力します。

この時初めてBMP ファイルが作られます。

`cbmp_free(obj)` 作成したビットマップのオブジェクトを解放します。

作ったオブジェクトが要らなくなったら使いましょう。

それから `cbmp.c` を使う時は必ず

```
#include "cbmp.h"
```

などと `cbmp.h` をインクルードする必要があります。

6.2 to_cbmp.c(to_cbmp.h) の使い方

ビットマップファイルを読み込むプログラムが、to_cbmp.c(to_cbmp.h) です。

このプログラムを使用する時は、必ず cbmp.c(cbmp.h) が要ることに注意してください。

表 6: sample_to_cbmp.c

```
1  #include <stdlib.h>
2  #include "cbmp.h" /* to_cbmp.h の前にインクルードすること */
3  #include "to_cbmp.h" /* to_cbmp.h を使う時に使用 */
4  static double PI = 3.14159265358979323846;
5
6  int main()
7  {
8      type_cbmp *obj; /* BMP データのオブジェクト */
9
10     obj = to_cbmp("sample_cbmp.bmp"); /* 指定した BMP を obj に取り込む */
11     if (obj == NULL) exit(EXIT_FAILURE);
12     cbmp_create_window(-10, 10, 10, -10); /* 座標変換 */
13
14     /* 中心 (0,0)、 x 軸方向の半径 2、 y 軸方向の半径 3 の楕円を
15        角度 (radian)PI/6.0 から 7.5*PI/4.0 まで、黒色で描く */
16     cbmp_circle(0,0, 5,5, 0, 2*PI, 0xff0000, obj);
17     cbmp_output("sample_to_cbmp.bmp", obj); /* obj の内容を BMP に出力 */
18     cbmp_free(obj); /* いらなくなったオブジェクト obj をフリー */
19     return 0;
20 }
```

ここでは、図 5 のプログラムで作成した図 2 を読み込んで、赤い円を描き加えるプログラム (図 6) を作成してみます。

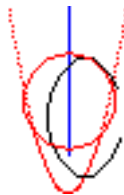


図 3: BMP 出力結果 (sample_to_cbmp.bmp)

コンパイルの方法は、

```
gcc cbmp.c to_cbmp.c sample_to_cbmp.c -lm
```


などとコンパイルします。後は実行ファイルを実行することで図 3 のような BMP ファイルが作られます。

以下、新たな関数を説明します。

`to_cbmp(filename)` ファイル名が `filename` であるビットマップファイルを `type_obj` 型のオブジェクトに読み込みます。

6.3 `cgif.c(cgif.h)` の使い方

`cgif_init(width, height, frame, iteration)` 最初に呼び出す関数です。

`width` が G I F 画像の幅、`height` が G I F 画像の高さとなります。
(単位：ピクセル)

`frame` はアニメーションの数です。今回はアニメーションはしないので、1 としています。

`iteration` は、アニメーションを何回繰り返すかを指定するところですが、今回は動画ではないので、とりあえず 0 にしてます。

`cgif_create_window(x1, y1, x2, y2)` G I F 画像に対応するウィンドウを決定します。

`x1` がウィンドウの左上の点、`x2` がウィンドウの右下の点になります。

上のプログラムでは、左上 $(-5, 10)$ 、右下 $(5, 0)$ となる大きさ 70×70 の G I F 画像が作られることになります。

`cgif_pset(x, y, color, frame)` `cgif_create_window` で指定したウィンドウ (`cgif_window`) に `color` の色の点を打ちます。

この場合 `cgif_window` 上の座標 $(x, x \cdot x)$ に赤い点を打ちます。

`color` の指定方法は

<http://www.math.kobe-u.ac.jp/HOME/yamaguti/color.html>
を御覧になってください。

`frame` はアニメーションのページ番号で、0 から数えます。

G I F のアニメーションはぱらぱらアニメのように複数のページを一定間隔で切替えることで実現しています

が、今点を打とうとしているページの番号 (0 から数える) を指定します。

今回は 0 番目のページに点を打つので、0 を指定しています。

`cgif_line(x0, y0, x1, y1, color, frame)` 端点を (x0, y0), (x1, y1) とする線分をページ番号が frame のページに color の色で描きます。

color の指定方法は

<http://www.math.kobe-u.ac.jp/HOME/yamaguti/color.html> を御覧になってください。

`cgif_output(ファイル名)` 指定したファイル名で G I F ファイルを出力します。この時初めて G I F ファイルが作られます。

それから `cgif.c` を使う時は必ず

```
#include "cgif.h"
```

などと `cgif.h` をインクルードする必要があります。

アニメーション GIF も簡単に作ることができます。

<http://www.math.kobe-u.ac.jp/HOME/yamaguti/use3.html> で説明していますので、興味があれば御覧になってください。

参考文献

BMP ファイルフォーマット関連

- [1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdi/bitmaps_62uq.asp, “MSDN の bitmap ページ”
- [2] <http://www.kk.ij4u.or.jp/kondo/bmp/index.html>, “BMP ファイルフォーマット”

GIF ファイルフォーマット関連

- [3] <http://www.geocities.co.jp/SiliconValley/1361/gif89a.txt> “GIF89a フォーマット原文”

[4] <http://www.tohoho-web.com/wwwgif.htm> “GIF フォーマットの詳細”

[5] <http://www02.so-net.ne.jp/koujin/gifformat/format.html> “Graphics Interchange Format”

描画アルゴリズム関連

[6] <http://www2.starcatt.ne.jp/fussy/algo/> “アルゴリズムの紹介”

[7] “C プログラムブック 2”, 東京：アスキー出版局, 1984-1986, 打越浩幸, 小西弘一, 清水剛