グラフィックスプログラミング入門

小波秀雄

April, 2008

目次

はじめに

このテキストはグラフィックスを通じてプログラミングを学ぶという意図で書かれています。コンピュータグラフィックスを学ぶための本ではありません。そもそも著者はその専門家ではなく，論文や教科書に掲載するための図版を作成するのが，グラフィックスを利用する主要な目的であるにすぎません。どうしてそれでこんなテキストを書くのかというのは，いくぶんいいわけが必要なようです。

プログラミングを学ぶための標準的なテキストというものはある言語の文法を基本から学びながら，条件判断や繰り返し処理などといった手法を身につけていくというスタイルで書かれています。それが王道であることはたぶん間違いありません。

しかしながら，標準的なコースで何かを学ぶというのは，時として退屈です。特に目的意識の希薄な学習者にとって。率直なところ，大学生というのは，その目的意識をあまり持たない学習者に属する人々が多く，せいぜい単位をとればいいという浅薄な意識で勉強しているケースも多いようです。

が，しかし，プログラミングというのは創造的な作業であると同時に，役に立つ技術です。それもあらゆる目的に対して役に立つのです。逆に言うと，何に使っていいかわからない。これが問題。

そこで，このテキストでは，その万能性をあえてグラフィックスという狭い分野に限定することにしたのです。画像を作成するという分かりやすい目的のために，初歩的なプログラミングの技術を使ってみる。これが基本的な目的です。同時に，コンピュータの中で情報がどのように扱われているのかを画像情報の取り扱いから知ることができるはずだというわけです。

最後の方には，通常のプログラミングではあまり扱うことがない Postscript について触れました。通常の手続き型言語ではない文法というものに触ってみることで，頭をちょっとやわらかくできそうだということ，ついでにちょっとした作画ができるツールを手に入れられるということをねらいとしました。

第 1 章

色の情報

1.1　色と視覚

光は電磁波の一種（可視光の波長 370 nm 〜 700 nm[*1]）で波長によって決まった色をもつ（☞ 虹の色＝太陽光のスペクトル）。波長が長い光は赤，短い方は紫色に見える。

一方，ヒトの目の網膜には，**Red, Green, Blue** それぞれの色を感じる 3 種の**錐体**(すいたい)，明るさだけを感じる**桿体**(かんたい)という**視細胞**がある。3 種の錐体の視細胞中には，それぞれの色を吸収して刺激を視神経に伝える色素物質，すなわち**視物質**がある。図 1.1 はこれらの視物質がどの光に反応するかを表している。

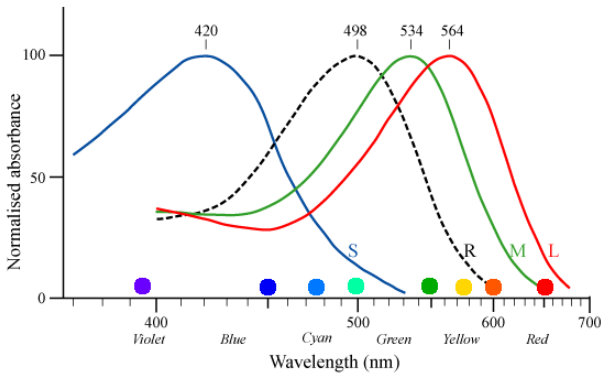


図 1.1　網膜上の 3 種の錐体の感度と光の波長の関係：破線は桿体のスペクトル

ヒトはこのように 3 種類の色を別々に感じる細胞をもつことから，3 色がどう混ぜ合わせられているかによって異なった感覚を受けることになる。これが**色覚**で，基本となる 3 つの色を **3 原色**という[*2]。

[*1] 1 nm = 100 万分の 1 mm

[*2] ヒトを含む霊長類は 3 原色の世界に生きているが，それ以外の哺乳類は錐体を 2 種類しかもたず，貧し

なお，ヒトが暗いところで色を感じにくくなって，明暗の感覚が主になるのは，錐体では光を捉えにくくなり，色の識別はできないが弱い光にも反応できる桿体が主に働くようになるためだ。

色覚異常とユニバーサルデザイン

色覚に関する科学的な知識を持っておくことは，色覚異常（色盲）の人に対する配慮を忘れないためにも非常に重要だ。一番多いのは赤緑性の異常で R と G の区別がむつかしい。

赤緑性色覚異常の発生率は男女によって異なり，それぞれ 20 人に 1 人，500 人に 1 人程度である。すなわち，赤と緑を知覚する視物質の遺伝情報は性染色体の X に乗っていること，また男性は X 染色体を 1 つしか持たないことから，その X 染色体に異常があるだけで色覚異常が発生する。一方，女性の場合には，約 10 人に 1 人の割合で片方の Y 染色体のみに色覚異常の遺伝子を持っているが，本人には異常が発現しない保因者になる。そして 2 つの相同な Y 染色体の両方に異常がある場合にのみ色覚異常が発生する。

つまり赤緑性の色覚異常は男性の場合には比較的ありふれていて，たいていの学校のクラスには何人かいると考えて，その対応をしないといけない。

色覚異常によって生じる日常生活での問題を回避するためには，まぎらわしい配色をしないようにするという配慮が必要である。基本的には，色相が違っていても明度が等しい色の組み合わせを避けるべきで，たとえば緑色の黒板に同じぐらいの明度の赤いチョークで字を書くのは好ましくない。ウェブのデザインにおいても，同様のまぎらわしい配色は避ける。このように，視覚の障害があっても工夫によって見やすくデザインすることを**ユニバーサルデザイン**と言っている。

い色の世界を生きている。一方，鳥類，魚類などは 4 種の錐体をもつものが多く，人間が識別できない色を見分けることができる。

1.2　3 原色と色の表現

3 原色をどう決めるかには任意性があるが，コンピュータグラフィックスで最もよく使われるのは，視細胞の種類にあわせて **Red(赤), Green(緑), Blue(青)** の 3 つであり，まとめて **RGB** と称される。これは**光の三原色**とも呼ばれている。

一方，印刷のインクなどに用いられるのは **Cyan(シアン)**，**Magenta(マゼンタ)**，**Yellow(黄)** の 3 つの色 (**CMY**) で，絵の具の三原色と呼ばれる。ただし，実際の印刷においてはこれらに**黒 (K)** を加えて **CMYK** の 4 色のインクがよく用いられる。自宅などにインクジェットプリンタがあったら，よく見てみよう。

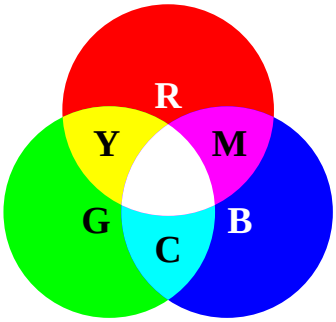


図 1.2　R, G, B と C. M, Y の関係：

1.3　3 原色と色の混ぜ合わせ

R, G, B は 3 種類の 光 の色なので，これらを単純に重ねていくと光が増えることになり，より明るい色が出てくる。C, M, Y は，そのようにして得られる色である。つまり次のような関係になっている。

$$\mathbf{Y} = \mathbf{R} + \mathbf{G}$$
$$\mathbf{C} = \mathbf{G} + \mathbf{B}$$
$$\mathbf{M} = \mathbf{B} + \mathbf{R}$$

R, G, B を混ぜ合わせるとどうなるだろうか？これは当然白になる。

一方，C, M. Y は色素（絵の具）の 3 原色である。たとえば Cyan の絵の具を塗った紙に白 (W) の光を当てると，色素によって R に相当する波長の光が吸収されて，残りの光が目に見えることになる。同様に Magenta は G の光を吸収する絵の具の色，Yellow は B の光を吸収する絵の具の色だ。

それでは Cyan と Magenta を混ぜるとどんな色になるだろうか？その場合，R と G の色が吸収されてしまうので， B だけが残る。他の色の混合についても同様に考えればよい。

C, M, Y のインクを全部混ぜるとどうなるだろう？この場合，すべての色が吸収されることになるはずだから，原理的には黒になる。しかし実際には汚い暗い色になることが多いので，黒のインクを別に用意しておくのである。

白と黒

白と黒も重要な色である。白は White から **W**, 黒は Black から **K** と通常省略される[*3]。

W, K はどちらも無彩色で，白はすべての色の光の足し合わせでできる。(図 1.2)。

$$\mathbf{W} = \mathbf{R} + \mathbf{G} + \mathbf{B}$$

一方，黒は光のない状態に相当する。つまりゼロで表される色ということになる。

$$\mathbf{K} = \mathbf{0}$$

1.4　色空間

色は 3 種の独立な要素（原色）の重ね合わせで表現できるので，さまざまの色を 3 次元空間，あるいは立方体の中の位置に対応づけることができる。いま，R, G, B の色の強さをそれぞれ 0 から 1 の範囲で表したとすると，すべての色は図 1.3 の立方体のどこかに位置づけられることになる。このようにして定義される空間のことを**色空間**または**カラースペース** (Color Space) という。

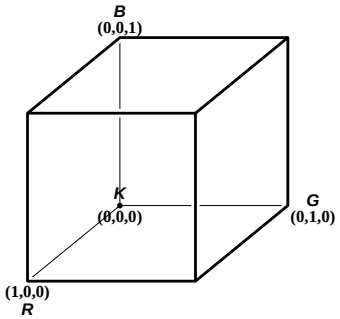


図 1.3　RGB の色空間を表す立方体

問題 1–1　1.3 図に記入されていない C, M, Y, W の各色は，この立方体のどこに位置づけられることになるのだろうか。

[*3] Blue がすでにあるので，黒を B と略記するわけにはいかない。

1.5 HSV/HSB による色の表現

RGB や CMY(K) といった 3 原色以外に，色を**色相 (Hue)**，**彩度 (Saturation)**，**明度 (Value, Brightness)** で区別するやりかたも一般によく行われる。これはそれぞれの英語の頭文字をとって **HSV** あるいは **HSB** と呼ばれる。英語の意味からすると HSB のほうが自然にみえるのだが，B は RGB の B と重なるので，以下では HSV を主に使うことにする。

HSV は色彩感覚にもとづいた表現法なので，一般によく用いられる。また美術の教科書にも載っている[*4]。図 1.4 の **12 色環**は似た色相をもつ色をぐるりと環状に並べたもので，体系的に色を理解したいときにはこれを覚えておくとよい。

なお，12 色環では色相が環状につながっているのに，光のスペクトルでは左端の紫と右端の赤はつながらない。これはとてもふしぎな事実であるが，そのことを解明しようとすると，人間の神経における色情報処理の問題が関わってくる。とても深い問題である。

1.5.1 補色

12 色環において，180 度の反対の位置で向かい合っている色の関係を補色という (反対色という人もいる)。これは色がたがいに打ち消し合う関係になっていて，補色関係にある 2 つの色をまぜるとだいたい無彩色になる。

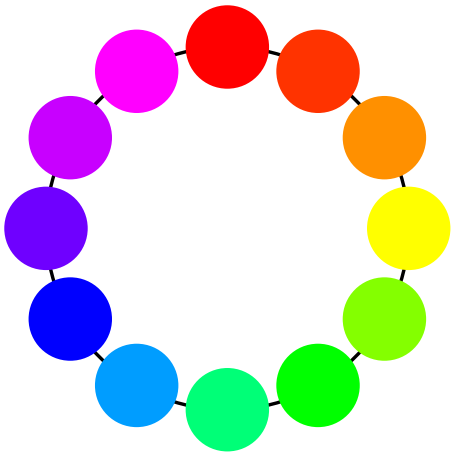


図 1.4 **12 色環:真上から時計回りにあか，だいだい，きだいだい，きいろ，きみどり，みどり，あおみどり，みどりあお，あお，あおむらさき，むらさき，あかむらさき**

[*4] このプリントの著者は中学 1 年のときに美術の時間に教わって，以来ずっとそれを覚えている。しかし今はそんなむつかしいことは中学では教えなくなってしまったらしい。

1.6　色の情報を数値で表す

コンピュータの世界では RGB による色の表現が最も普通に用いられ，印刷においては　CMYK による表現が主に用いられる。以下では RGB で表現する場合について述べる。

■連続な数値で表す　色を数値で表すには R,G,B ごとに 3 つの数値をあてて，それらの組み合わせを用いればよい。数値の割り当て方には任意性があるが，**連続な実数**を用いるときには 0〜1 の範囲で表現し，0 はその色の光が無い状態，1 は最も強い状態とする。

■離散的 (デジタル) な数値で表す　しかし，コンピュータの中の情報は本来**デジタル**，つまり 2 進法や 16 進法で表される離散的なものである。言葉をかえれば，光の強さを 0,1,2,. . . , のようなとびとびの段階で表さざるを得ない。この場合，何段階にとるかというのは画像の用途やハードウェアの性能によって決めることになる。現在では 0 〜 255 の 256 段階の値をとるように設定するのが一般的である。

ここで $256 = 2^8$ であり，16 進法 2 桁で 255 までの数値が表されることを思い出そう。これ位の粗さでは不十分に思えるかもしれないが，これで区別できる色彩の種類は 256^3 に上る。全部で何通りになるかは自分で計算してみてほしい。

$$\texttt{0}(0), \texttt{1}(1), \texttt{2}(2), \ldots, \texttt{A}(10), \texttt{B}(11), \texttt{C}(12), \texttt{D}(13), \texttt{E}(14), \texttt{F}(15), \texttt{10}(16), \texttt{11}(17),$$

$$\ldots, \texttt{F9}(249), \texttt{FA}(250)\texttt{FB}(251), \texttt{FC}(252), \texttt{FD}(253), \texttt{FE}(254), \texttt{FF}(255)$$

表 1.1 に，主要な色とその数値表現をまとめておいた。HTML における色指定で `#FF0000` のように 3 つの 16 進数をまとめて表すことは，すでに知っているはずだ。

表 1.1　主な色の数値表現：

色名	英語名	実数表現	16 進数表現
あか	Red	$(1, 0, 0)$	`FF,00,00`
みどり	Green	$(0, 1, 0)$	`00,FF,00`
あお	Blue	$(0, 0, 1)$	`00,00,FF`
シアン	Cyan	$(0, 1, 1)$	`00,FF,FF`
マゼンタ	Magenta	$(1, 0, 1)$	`FF,00,FF`
きいろ	Yellow	$(1, 1, 0)$	`FF,FF,00`
しろ	White	$(1, 1, 1)$	`FF,FF,FF`
くろ	Black	$(0, 0, 0)$	`00,00,00`

1.7　GIMP のカラーパレットの色表現

GIMP はグラフィクス処理のためのフリーソフトで，UNIX, Widows, Mac などのプラットフォームで走り，高い機能をもっている。たいていのペイント系のアプリよりもできがよい[*5]。

GIMP は色を扱うためのすぐれたパレットを持っていて，図 1.5 のようにさまざまの色表現が同時にパレットに表示される。これを見ると，RGB と HSV の関係が明瞭にわかる。また，左上のタブを切り替えることで，印刷用の CMYK の表現も知ることができる。

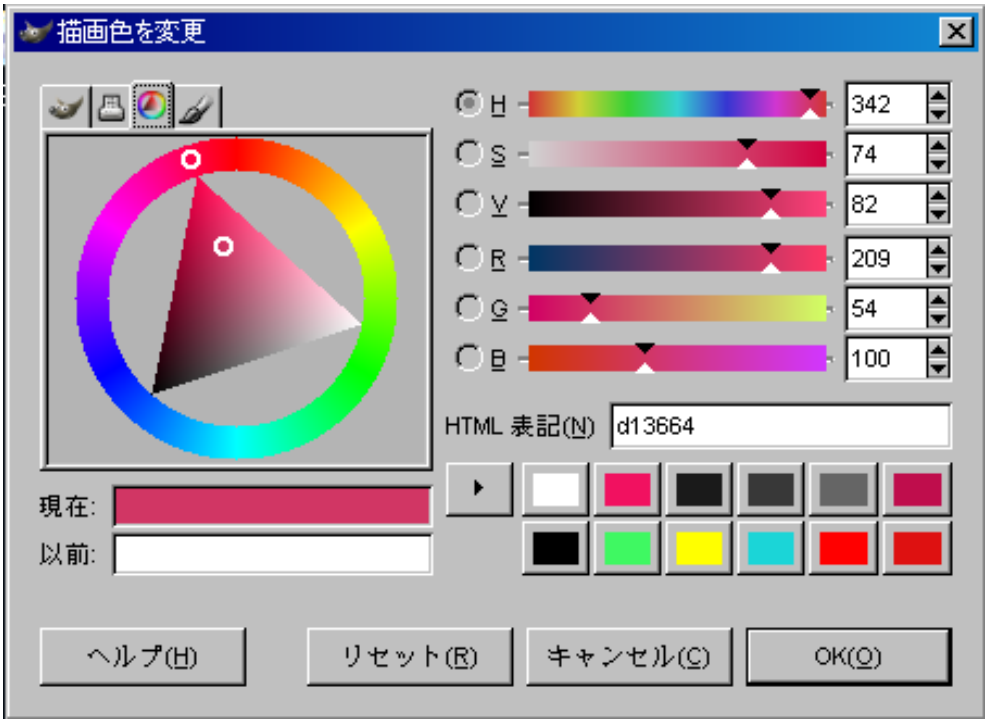


図 1.5　GIMP のカラーパレット:数値の範囲の取り方に注意。色相は一回り 360 度になっている。

[*5] Adobe の有名な（かつ高価な）アプリケーションで Photoshop というのがあるが，ほぼそれに匹敵する性能を持っている。

第 2 章

コンピュータグラフィックスの概要

2.1 ラスター画像によるグラフィックス

2.1.1 ラスター＝走査線

テレビやコンピュータの画面では，左から右へ水平に輝点が走って，線が描かれている。この線のことを**ラスター (raster)** あるいは走査線という。

従来のテレビの走査線は縦に 525 本並んで 1 枚の画像をつくり，毎秒 24 枚塗り替えられている。コンピュータの画面のラスター描画は，もっと高速かつ高精度である。

2.1.2 コンピュータ画面はピクセルの集合

コンピュータのディスプレイは，図 2.1 のように**ピクセル (pixel)** [*1] と呼ばれる点が縦横に多数並んでいて，それぞれのピクセルは R, G, B の各色で発光する 3 つの面から構成されている。

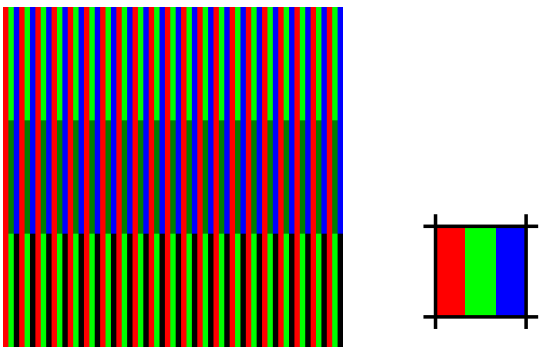


図 2.1 ピクセルで構成されているコンピュータ画面の一部 (左) と 1 個のピクセルの形 (右)

[*1] ピクセルはまた画素とも呼ばれる。

2.1.3　画面の解像度はピクセル数で決まる

一般的なコンピュータの画面では横と縦のピクセル数は 4 : 3 で，小さめのノート PC では 1024×768 が標準的。デスクトップではもっと高解像度の 1280×1024, 1600×1200 などが使われている。

2.1.4　ラスター画像の情報量

画像を表現するためには，どれだけのデータ量が必要だろうか。ひとつのピクセルの色の濃さが 256 段階とれるものとしよう。つまり RGB のそれぞれが 0 から 255 までの段階で表現できるものとする。ここで 256 という数字は，とうぜん 2^8 からきている。

2^8 というのは 1 B の情報量[*2]に相当するので，ひとつのピクセルあたり必要な情報量は，この場合 3 B ということになる。コンピュータのメモリを 3 個分使うと考えてよい。

つぎに画像のサイズを仮に 1024 × 768 としよう。計算してみると 786432 で，これだけのピクセルの数があるわけだ。したがって，この画像 1 枚に必要なのは，786432 × 3 = 2359296 B ということになる。桁数が多いので 1024 で割ると，2304 KB ということになる。 2.3 MB 弱というところだ。

このように，画像データには莫大なメモリが使われる。グラフィクスの処理がコンピュータの処理の中で最も重い[*3]のは，扱う情報量がきわめて多いからだ。

2.1.5　データを圧縮してファイルにする

上述のようにラスター画像のデータはきわめて大きい。ディスクやその他の記憶メディアにデータをしまったり，あるいはネットワーク上で画像データを流すためには，これは不便である。

そこで，一般に画像は「圧縮」してファイル化される。圧縮の原理を簡単にいえば，同じデータの繰り返しがあったらそれをまとめて表現するような方法を採用するということである。たとえば次のようなデータがあったとしよう。

```
ABC8ABC8ABC8ABC8ABC8ABC8ABC8ABC8ABC8ABC8
```

ここでは `"ABC8"` という 4 文字の文字列が 10 回繰り返されている。これを次のように書いたとしよう。

---

*2 1 B は 1 Byte(バイト) の商略。2 進法 1 桁の bit(ビット) は b で表す。

*3 「重い」という表現は，コンピュータを使うときによく使われる。処理する情報量が多く，かつ必要な処理の数も多いと，コンピュータの動作には時間がかかるようになり，つまり「重く」なる。それを回避するには，より少ない計算回数で処理がすむような効率のよいアルゴリズムを採用する必要がある。一方で，コンピュータのハードウェアにも高性能のものが要求される。そんなわけで，グラフィックスを支える技術は，ハードウェア，ソフトウェアいずれも急速に進展してる。

```
ABC8*0A
```

ここで `*0A` というのは，`0A` 回 (10 進法で 10 回) 繰り返すという意味であると決めてしまえば，上の 40 文字のデータが 7 文字に短くなったことになる。このように，なんらかの規則を決めてデータ量を減らすのが，圧縮という操作だ。また，画像は目で見てわかればよいので，細かな濃淡をならすことで情報を減らす処理も行われる[*4]。

現在，一般に用いられている代表的な圧縮画像形式としては，PNG, JPEG, GIF の 3 つがある。表 2.1 にこれらの特徴を示した。写真には JPEG, そうでなければ PNG というのが一般的な使い分けになっている。GIF は以前よく使われていたが，歴史的な経緯で使用が制限されたこともあって，最近ではアニメ画像を利用するために使われる程度になっている。

表 2.1　3 つの圧縮画像形式

形式	読み方	拡張子	特徴
PNG	ぴんぐ	`.png`	可逆圧縮
JPEG	じぇいぺぐ	`.jpg`，`jpeg`	写真に適している。非可逆圧縮
GIF	じふ	`.gif`	256 色しかない。アニメ可

[*4] デジカメ画像では JPEG のフォーマットが標準的に使われている。JPEG ではいったんならしてしまった濃淡をもとに戻すことはできない。このような圧縮の方式を非可逆圧縮という。

2.1.6　ラスター画像の泣きどころ

ラスター画像（ビットマップ画像）は，ピクセルの集合として一枚の画像を作るので，拡大するとピクセルが見えてきてしまう。図 2.2 にはギリシャ文字 $\alpha\xi$[*5]のラスター画像を拡大したときの効果を示した。

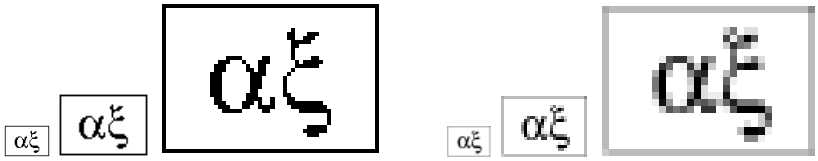


図 2.2　ラスター画像を拡大していったところ。左はアンチエイリアシングなし，右はあり。

本来，このような文字の輪郭は，内側が黒で外側が白というふうに単純に塗り分けられるもののはずだが，画像がピクセルで作られている場合には，左図のように**ジャギー** (ぎざぎざ) が目立って汚い印象を与えてしまう。

そこで右図のように，境界付近のピクセルの色の濃さを中間色にして，ぎざぎざをなるべく消すように処理することが多い。このような処理のことを**アンチエイリアシング**という。

しかし，このような処理を施しても，ラスター画像を拡大すると図のようにきわめて醜くなってしまうという，避けがたい欠点がある。

2.2　ベクターグラフィックス

2.2.1　ベクターフォント

たとえば α という文字を筆で描くことを考えよう。描く人の頭にある文字情報は，この文字の線が右上から左下へ斜めに走ったあとで，大きくカーブを描き，右下の端へ向かっていくということである。そのため，大きな文字でも小さな文字でも，どれも滑らかな境界線をもつように描くことができるわけだ。このように，デザインのための情報として，線の向き（正確には輪郭線の向き）の情報を持たせた文字フォントのことを**ベクターフォント**，という。ベクターというのは，ベクトルと同じ意味だ。

図 2.3 には，ベクターフォントを使って汚くならない文字の例を示した。**アウトラインフォント**と呼ばれることもある。

[*5] ξ は「グザイ」と読む。ラテン系文字の x に相当する。ここでは曲がりくねったデザインをもつ文字の例として使っている。

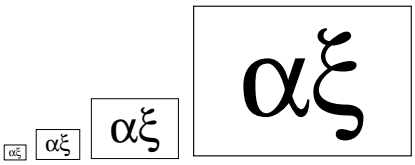


図 2.3 ベクターフォントで印刷された文字

2.2.2 ベクターグラフィックス

文字と同じように，図形についても，線の向き，輪郭などの情報を論理的に持たせることで画像を作ることが考えられる。このような方式のグラフィックスを**ベクターグラフィックス**という。図 2.4 は，ベクターグラフィックスで描かれた画像とラスター画像を比較したものである。

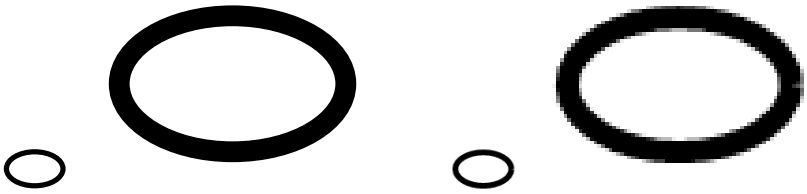


図 2.4 ベクター画像とラスター画像で描かれた楕円の比較

2.2.3 Postscript

代表的なベクターグラフィックスとしては Adobe 社[*6]が開発した Postscript がある。これは一種のプログラミング言語であって，線や面で構成された画像を記述することができる[*7]。また，豊富な文字フォントを利用でき，代表的な**ページ記述言語**として，**DTP**[*8]のアプリケーションの世界で広く用いられている。

Postscript のプログラムの例を次に示す。このように座標の数値や命令（オペレータ）を組み合わせることで，図 2.5 のような絵を描くことができる。詳しくは 4 節を参照のこと。

[*6] 「あどびぃ」と読む。電子出版技術では世界のトップ企業である。こういうことも常識として覚えておこう。

[*7] Postscript は，ふつうは追伸の意味で，よく PS と略される言葉だが，ここでは post(後ろ) に (script) 書くという意味を持たせている。これはこの後に示されたプログラムの例をみればわかるように，まず数値があって，その後に命令がくるという言語の仕様から来た命名だ。

[*8] Desk-Top Publishing. この用語は印刷出版における常識用語なので覚えておくこと。

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox:  0 0 200 150
1 0.5 0 setrgbcolor
100 75 moveto 100 75 40 45 180 arc fill
0 0 0.8 setrgbcolor
10 10 moveto
190 10 lineto
190 140 lineto stroke
/Times-Bold findfont 20 scalefont setfont
100 20 moveto
(ABC) show
```

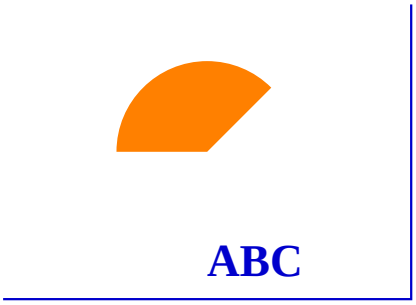


図 2.5　Postscript で描いた簡単な絵

ここでは示されていないが，ラスター画像を取り込むこともできる。つまり，文字，図形，写真などの画像を同時に表現できる。しかもベクターグラフィックスなので，スケール（縮尺）に関わらず滑らかで美しい画像が得られる[*9]ことから，そのまま出版に使える品質をもつ完全な DTP 機能を持っている。

Postscript は 1985 年に登場した古典的な言語で，使い方も比較的簡単だが，その後登場した Java や SVG といった言語のグラフィックスに大きな影響を与えている。また，Adobe 社の DTP アプリケーションとして有名な Adobe Illustrator や Photoshop に実装されている他，GNU によるフリーソフトとして開発された Ghostscript を用いれば，だれでも自由に Postscript 言語で書かれたプログラムをベクター画像として利用することができる[*10]。

2.2.4　PDF, Portable Document Format

PDF ファイルは，WWW 上で文書を提供するために広く用いられている。PDF も Adobe 社が開発したページ記述のためのファイル形式（フォーマット）で， Postscript の機能を簡略化したものになっている。現在のところ，電子媒体として人に配布するためのフォーマットとしては， PDF がもっとも便利だ。つまりコンピュータの OS やインストールされているアプリケーションを気にしないで人に渡すことができる。このように，異なったプラットフォーム[*11]の間で共有できるという意味で，現在の事実上の標準

*9 ただし，当然のことながらラスター画像の部分は拡大すれば粗くなる。

*10 この資料も，また 1 年生の情報リテラシー科目の教科書の『きわめる情報リテラシー』も Ghostscript をフルに活用して作られている。なお，Ghostscript という名称にこめられたユーモアもよく味わってほしい

*11 コンピュータには OS その他の違いによってさまざまのものがある。プラットフォームという言葉は，そのような異なったコンピュータのことを指す。Windows は数あるプラットフォームのひとつであって，それだけで世界が動いていると思ってはいけない。

フォーマットとされているわけだ。

2.2.5　SVG

SVG は Scalable Vector Graphics の略で Scalable は scale 可能，つまり伸縮可能という意味だ。SVG は HTML の拡張にあたる XML の一部に位置づけられ，コンソーシアム W3C(`http://www.w3c.org`) によって規格が定められている。XML の規格には他に数式を記述するための MathML などもあるので，これらが完全にブラウザに実装されれば，ウェブのコンテンツは現在よりも格段に表現能力を高めることになる[*12]。

下に簡単な SVG のソースと，それをレンダリング[*13]した画像を示した。スタイルシートが使われるなど，HTML っぽい書き方であることがわかる。

SVG は徐々に使われるようになって来ており，英語版の Wikipedia の画像にも採用されるようになってきたので，ブラウザへの実装は加速されつつある。また携帯にも採用されてきているので，徐々に普及してきている段階といっていいだろう。

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
              "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
   <style type="text/css">
        circle:hover {fill-opacity:0.9;}
   </style>
   <g style="fill-opacity:0.7;">
      <circle cx="6.5cm" cy="3.2cm" r="100" style="fill:red;   stroke:black" />
      <circle cx="4.6cm" cy="6.0cm" r="100" style="fill:blue;  stroke:black" />
      <circle cx="8.2cm" cy="6.0cm"r="100"  style="fill:green; stroke:black" />
   </g>
</svg>
```

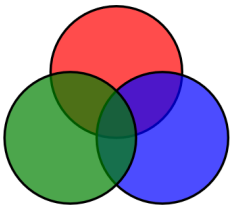


図 2.6　SVG を使って描いた像:透明色が使えることに注意

[*12] SVG は仕様書がほぼ完成した状態にあるものの，その仕様どおりに作られたファイルで実際に画像が表示されるようにするためには，その通りにコンピュータが働くようにプログラムしなければならない。このように，プログラムを書いて，ある機能を実現することを**実装**するという。

[*13] rendering というのは，ソースを解釈して人間が見れる状態にすることをいう。

第 3 章

ラスター画像をプログラムする

この節で扱う一連の画像ファイルは，**Sourceforge netpbm project** が開発したオープンソースプログラム **Netpbm** の仕様[*1]にもとづいている。

以下ではアスキー形式と呼ばれるタイプのものをまず取り上げて，画像ファイルとはどのようなものなのかを調べてみよう。

3.0.6 黒白の画像を作る

エディタ (Meadow/Emacs) で `feet.pbm` というファイルを新規に開き，次の内容を書き込んでセーブしよう。

```
P1
# feep.pbm
24 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 1 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0
0 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 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

このようなテキストを編集するときには，全部の行をいちいち手で打っていたら時間がかかるし，まちがいも起こしやすい。こんなふうに数字が並んだ行を編集するときには，次のようにするとよい。

[*1] `http://netpbm.sourceforge.net/doc/` 参照。

1. 24 個の 0 をスペースを空けて打つ
   これは，次のように入力すると速く正確に打てる。
   (a) 0 0 0 0 0 と 4 個打つ。最後にもスペースを置く。
   (b) C-a で行の頭に戻る
   (c) C-k で切り取る
   (d) C-y を 6 回繰り返す
2. 上で作った行を 7 行に複製する
   これは次のようにすると速い
   (a) C-k C-k で切り取る
   (b) C-y を 7 回繰り返す
3. **INS** キーを押す (Ovwrt という表示が現れる)
4. 書き換えるべき 0 の位置にカーソルを動かして 1 をタイプする
5. **INS** キーを押す (Ovwrt の表示が消える)
6. 最後の行の行末に改行が入っているかどうか確認する

その後，GIMP でこのファイルを開いてみるとどうなるだろうか。

得られる画像はとても小さいので，かなり拡大しないとわからないが，図 3.1 のような画像ができているはずだ。これを `feep.pbm` のソースとよく見比べると，どういう仕組みでこのような画像ができているかを理解できる。

図 3.1　PBM 形式のビットマップ画像

3.0.7 グレイスケールやカラーの画像を作る

こんどは 2 つのファイル feep.pgm, feep.ppm を上の要領で作成して，GIMP で表示させてみよう。図 3.2,3.3 のような画像が得られるはずだ。

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
P3
# feep.ppm
4 4
15
 0  0  0  0  0  0  0  0  0 15  0 15
 0  0  0  0 15  7  0  0  0  0  0  0
 0  0  0  0  0  0  0 15  7  0  0  0
15  0 15  0  0  0  0  0  0  0  0  0
```

図 3.2　PGM 形式のビットマップ画像

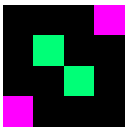


図 3.3　PPM 形式のビットマップ画像

3.1　画像ファイルを解剖する

ここまで見てきた画像ファイルは**テキストファイル**なので，どういう仕組みで画像が作られているのかを簡単に知ることができる。最初のファイルから順に，そのフォーマットがどうなっているかを調べてみよう。

3.1.1　PBM — 黒白画像を表現する

PBM は Portable Bit Map を意味している[*2]。18 ページの `feep.pbm` の最初の 3 行は，次のように書かれている。他のファイルとも見比べながら考えよう。

```
P1
# feep.pbm
24 7
```

`P1` は，「このファイルは黒白のビットマップファイルだよ」ということを宣言するために置かれている。最初の行に特別の意味を持たせることで，このファイルを読み込んだ画像処理プログラムは，冒頭のファイルを読み込んだ時点で，これから後に行うべき処理を決めることになるわけだ。このように全体に大して重要な意味を持つ数字のことを**マジックナンバー**と呼ぶことがある。

`# feep.pbm` は，`#` の後にファイル名が置かれている。ファイル名はわざわざファイルの中に置かれる必要はないが，このファイルのソースを見る人のためにはあったほうがよいかもしれない。つまりデータとしては特に意味を持たない。このような部分を**コメント**という。コメントはなくてもかまわない。

`24 7` の意味は，その下にある数値データとでき上がった画像の情報を見比べると，この画像の横と縦のサイズを表していることがすぐにわかる。

このように，ファイルの最初の 3 行は画像データの前にあって，画像の全般的な情報を受け持っている。画像ファイルだけではなく，多くのデータファイルには，情報をまとめた部分が冒頭に置かれることがで多い。このような部分をデータファイルの**ヘッダ (header)** と呼ぶ。

4 行目以降のデータ本体の構造をみてみよう。ここには 24 列× 7 行＝ 168 個の `0` と `1` がスペースを置いて並べてある。画像と比較すると，`0` は白，`1` は黒のピクセルにそれぞれ対応していることがわかる。なお，このファイルではデータの数字を画像に合わせて行

[*2] bit map というのは，本来 1 bit で黒白を表すような方式について名づけられたもので，この PBM のケースはそのぴったりの例になっている。しかし，その後ピクセルで構成された画像，つまりラスター画像をビットマップと呼ぶケースも増えてきていて，本来の意味がややあいまいになっている。

を分けて並べているが，168 個のデータが置かれていさえすれば，途中の改行はあっても なくても構わない。

■文字を作ってみよう PBM 形式のファイルの構造が分かったところで，これを使って文字を作成してみよう。横縦 16 x 20 の PBM ファイルで適当な日本語の文字を作成してみなさい。図 3.4 はこのようにして作ったひらがなのフォントである。ビットマップ形式のフォントはさまざまなところで用いられているが，それらはこのようにしてひとつひとつ人間の手で作られたものなのだ。

図 3.4 ビットマップで作った「う」

PBM 形式の画像の場合，黒白 2 色しかないので，用途はかなり限られる。ただし使われる情報量は少なく，ヘッダを除いた部分はデータ数を N としたとき， N bit だけですむことから，ここでためしたように，文字を表現するには好適である。

3.1.2　PGM — グレイスケールで描く

PGM は，Portable Gray Map の省略。PBM の場合，画像に使えるのは黒と白の 2 色だけだったが，途中段階の灰色も使えるようにしたのが，このデータ形式だ。19 ページの `feep.pgm` をみてみよう。

このファイルのヘッダ部分は次のようになっている。

```
P2
# feep.pgm
24 7
15
```

PGM 形式の場合，マジックナンバーは `P2` となっている。画像処理ソフトはまずこれを見て，処理の段取りを組み立てるわけだ。次に，コメント行と画像サイズの記述は PBM ファイルと同じだが，4 行目の `15` にあたる数字は今回初登場である。これは何を意味するのだろうか。

それを知るために，ヘッダの後につづくデータ部分，それとレンダリングされた画像を見くらべてみよう。すると，データ部分の数字はほとんどが 0 で，他に 7, 11, 15 が使われている。そして 0 が黒に， 15 が白に対応していることがわかる。

最も暗い色　0
最も明るい色　15

というわけで，この 15 というのは，最も明るい色 (白) を表す値になっている。

それにしてもどうして 15 なのだろうか。それには 0 から 15 まで 16 段階の明るさ（階調）を表せるということに気付けばいい。情報の世界では $16 = 2^4$ であって，16 進法の 1 桁で表されるのが 0 から 15（16 進数では 0 から F）というわけだ[*3]。

データの部分については，0 が黒，15 が白で，中間の灰色がその間に段階をなすようになっている。なお，PBM の場合には 0 が白，1 が黒となっていたので，混乱しないようにしよう。

[*3] たとえば 20 とか 30 とかをここにおいてもかまわないが，情報量を無駄にしないためには，16 進数で 1 桁とか 2 桁といった値を基準にしたほうがよい。したがって，ふつうは 15 または 255 という数字が使われることになる。

3.1.3 PPM — カラー画像

PPM は Portable Pixel Map の意味。カラー画像を表現できる形式だ。まず，19 ページの `feep.ppm` のヘッダ部分を見ることにしよう。

```
P3
# feep.ppm
4 4
15
```

`P3` がこの場合のマジックナンバーだ。3 行目の `4 4` がピクセル数，4 行目の `15` は PGM ファイルと同様に明るさの階調ということになる。

次にデータ部分をみてみよう。

```
 0  0  0  0  0  0  0  0  0 15  0 15
 0  0  0  0 15  7  0  0  0  0  0  0
 0  0  0  0  0  0  0 15  7  0  0  0
15  0 15  0  0  0  0  0  0  0  0  0
```

横には 4 個のピクセルが並ぶはずなのに，データとして書かれている数は 12 になっている。これはどういうことだろうか。

もちろん，3 つの数値で 1 個のピクセルを表そうというのは，RGB で表現するためであることがすぐにわかる[*4]。このファイルの場合，最大の明るさが 15 だから，白は `15 15 15` となる。その他の色については，表 1.1 の色の名前と比較して作ってみるとよい。実際にデータをいろいろと書き換えてみて，このフォーマットの意味するところを実感としてつかもう。

問題 3–1 12×12 の PGM ファイル，8×12 の PPM ファイルを作ってみなさい。最初にベースとなる色を決めて，繰り返し作業を簡単に行うようにする，あるいは縞模様のような単純なパターンを段取りよく設計するといったように，作業の段取りを合理的に考えて進める工夫をして，短時間できれいなものを作りなさい[*5]。エディタの置換機能をうまく使いこなすと，もっと大きなカラー画像でも割合簡単につくれる。

[*4] データの数字が 3 個で 1 組になるようにスペースを設けてあるのは，人に見やすくするためであって，データを区切るためのスペースは 1 個以上であれば構わない。コンピュータの最大の得意技は数えることで，人間と違って数えまちがえることはない。

[*5] このような合理的な作業の組み方というのは，あらゆる仕事で大切な心がけになる。大工の世界には「段取り八分」という有名な格言が昔から伝わっている。

3.2　プログラムでラスター画像を作成する

前節ではラスター画像をエディタで生書きするという，少々めんどうな作業に取り組んでみた。実はこんなことをやる人は（ごくごく少数のマニアックなテクニシャンを除いては）たぶんいない。実際には，この種の画像を作るには何らかのプログラムを使うものだ。

画像を構成する莫大な数のデータを処理するというのは，人間にとってはとてつもない忍耐と時間を必要とする作業だ。仮に 1024×1024 の画面のピクセルを，1 つあたり 3 秒で人が処理したとすると，不眠不休でがんばっても 1 月以上かかってしまう。こんな処理でも，コンピュータは一瞬にして実行してくれるわけだ。

ここでは，PPM 形式の単純な画像を生成するプログラムをいろいろと作ってみて，画像を生成するプログラムがどんなふうに働くのかを調べてみよう。

3.2.1　最も簡単なプログラム

最初のプログラムは，単色の画面を出力するだけのものにしよう。横縦 240×180 の赤い画面を作成してみる。色の階調は 0 〜 255 の 256 段階としよう。PPM ファイルのフォーマットから，次のような手順で画像を作成すればよいことがわかる。

1. PPM 形式のマジックナンバー `P3` を出力して改行する。
2. `#` で始まるコメント行を出力して改行する。
3. 画像サイズとして `240 180` を出力して改行する。
4. 赤を表すデータ `255 0 0` を 240×180 個出力する。このとき改行は必要でないが，間のスペースを忘れないようにする。

それでは上記の方針でソースを書いてみよう。もっとも素朴なソースは次のようになるはずだ。

```
# ppm001.rb
puts "P3"
puts "# PPM test"
puts "240 180"
puts "255"
for i in 0 .. 239
  for j in 0 .. 179
    print "255 0 0 "
  end
end
```

これを書き換えるだけで，サイズや色の異なる画像ファイルをいくらでも作ることができる。

3.2.2 puts と print

ppm001.rb では次のように puts というメソッドが登場している。

```
puts "P3"
```

これを print を使って書くと次のようになる。

```
print "P3\n"
```

print を使ったほうで文字列の最後に現れる \n は改行を意味する。改行がないとヘッダとして無効になってしまう。

print は引数として与えられた文字列を正直にそのまま出力するので，改行があるのならそれもきちんと入れておかないといけない。それがちょっと面倒なので，puts という勝手に改行を最後に入れてくれる道具も用意されているというわけだ。

3.2.3 二重の for ループの意味

このソースでは for ループの中にもうひとつの for ループが入っている。このような二重のループは 2 次元 (2D)[*6]のグラフィックスで常に用いられている。このプログラムで使われている二重ループはどのような働きをしているのだろうか。それを知るために，次のプログラムを走らせてみよう。

```
# doubleloop.rb
for i in 0 .. 7
  for j in 0 .. 9
    print j, ",", i," "
  end
  print "\n"
end
```

走らせてみると，10 列 8 行に並んだ数字の対が表示される。左から右に向かっては，対の左側の数字が 0, 1, 2,...,9 と変化していき，上から下へは右側の数字が 0, 1, 2,...,7 と変化していくことに注意しよう。

[*6] 2D とか 3D というのはグラフィックスでよく使われる言葉で，D は dimension つまり次元の意味だ。

この変化の仕方をプログラムを見ながら考えると，ループ変数 j で回る内側のループが水平方向の変化を作り出し，ループ変数 i で回る外側のループが垂直方向の変化を作り出すことがみてとれる。そのようすを図 3.5 に示した。

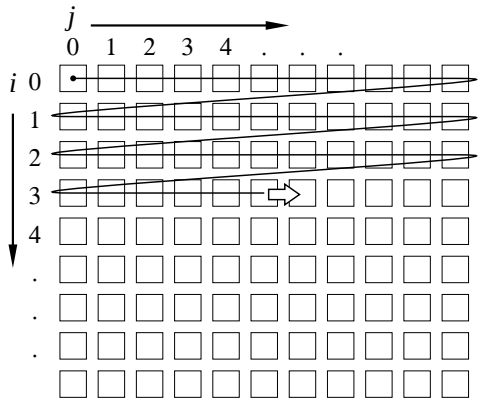


図 3.5　二重ループを使って画面上の点を動かすときの添字 i, j の変化

3.2.4　プログラムを洗練する

上で作成してみた ppm001.rb というプログラムには次のような問題がある。つまり，もしも横縦のサイズを変更しようとすると，次の複数の箇所でプログラムを直さなければならなくなる。仮に横を 120 に，縦を 100 に変更するならば，直す行は次のようになる。

```
puts "120 100"
for i in 0 .. 119
  for j in 0 .. 99
```

しかし，これだと「うっかり」片方だけを直して走らせるというへまをやる可能性がおきてしまう。上の例では 119, 99 は 120, 100 から計算された値なので，横と縦を決めたら自動的に決まることがのぞましい。

そこで，横と縦のピクセル数にそれぞれ nx, ny という変数を使うことにする。

```
# ppm002.rb
nx = 240; ny = 180
puts "P3"
puts "# PPM test"
puts "#{nx} #{ny}"
```

```
puts "255"
for i in 0 ... nx
  for j in 0 ... ny
    print "255 0 0 "
  end
end
```

こうしておけば，nx と ny に代入する値を書き換えるだけで，サイズが変更できる。適切な変数を導入することで，情報を一元管理できるというわけだ。

大事なパラメータは前の方で変数に代入しておいて利用しよう。

3.2.5 2 種類の範囲演算子

ppm002.rb では範囲演算子が .. ではなく，... になっていることに注意しよう。そのちがいは表 3.1 に説明してある。

表 3.1 2 種類の範囲演算子

a .. b	：	a から b まで
a ... b	：	a から b-1 まで

プログラムの中では，ある数の 1 だけ手前まで処理するといったケースがしばしば現れる。そこで「未満」を表すための範囲演算子として ... が用意されているというわけだ。

3.2.6 irb で簡単にチェック

ここで少し脱線して，irb について触れておこう。わざわざエディタでプログラムを書くまでもなく簡単に Ruby を使ってみるための道具として **irb** がある。まずはシェルのコマンドラインから次のように入力してみよう。

```
irb
```

すると irb が起動して，次のようなプロンプトが現われる。

```
irb(main):001:0>
```

この状態で，次のような入力を試してみよう（文字列を打ち込んではリターンキーを打つ）。

```
10 + 20
a = 120
a
a * 2
```

すると 30, 120 などと表示されるので，何が起きているかは分かるはずだ。このように irb ではプログラムを 1 行ずつ実行する感覚で対話的に Ruby を走らせることができ，ちょっとしたチェックには非常に便利な道具である[*7]。

3.2.7　文字列中の式展開

プログラム ppm002.rb には他にも新しく登場した記法がある。それは次のような行の書き方だ。

```
puts "#{nx} #{ny}"
```

このように，ダブルクォート (") で囲まれた文字列の中に **#{変数名}** という形が現れると，変数の内容でそれが置き換えられる。変数名ではなく式であってもよい。この式展開がどのように働くのか，irb を使ってチェックしてみることにしよう。

```
x = 125
"x = #{x}"
"#{x*3}"
'x = #{x}'
```

2 行目の入力に対して，#{x}が変数 x の値に展開され，また 3 行目の入力に対しては x*3 という式を展開した値が出力されることが分かる。またシングルクォート (') を使った文字列に対しては式展開は起きないことも分かる。

文字列の式展開はプログラムを簡単に書く上で非常に便利なものであるが，C, Java などには，この機能はない。

[*7] 著者は電卓代わりによく利用している。

3.2.8 変数名の付け方

上のプログラムでは，二重配列のサイズについて，横には nx, 縦には ny という変数を使っている。これらの先頭の n は Number のイニシャルで，「x 方向の要素の数」といった意味を持たせてある。これに限らず 'n' で始まる変数は，たいてい何かの数を表しているというのが慣習化された変数名の使い方だ。

ソースの中でどのような変数名を使っても，プログラムとしては正常に走る。しかし，自分にも他人にも分かりやすいソースを書くためには，変数名には分かりやすいものを使う必要がある。そのためには模範となるプログラムをたくさん見ることが大切だし，英語のセンスも身につけないといけない。

たかが名前の問題と考えがちだが，ちゃんとしたプログラムを書くためにはとても大事なことだということを，ここで覚えておこう。

3.3 バイナリファイルでサイズを節約する

3.3.1 ファイルサイズをチェックしよう

前節で作ったプログラムで PPM の画像を生成した場合，そのファイルサイズはどのようになっているだろうか。それはプログラムから簡単に知ることができる。

たとえば 240 × 180 のサイズで赤の単色の画像を作ったとしよう。赤の点ひとつを表すのには，’255 0 0 ’ と 8 文字を使っているから，データ部分の文字の数は次のようになる。

$$240 \times 180 \times 8 = 345600$$

これにヘッダ部分の文字数が 10 個程度加わって，全体の文字数になる。半角文字（正確には 1 バイト文字）は 1 字で 1 Byte の情報量をもつので，ざっと 340 KB 程度のサイズのファイルができることが分かる。

実際にファイルのサイズを知るには，`ls` コマンドを活用するとよい。画像ファイルを作成したら，シェルのコマンドラインから次のように入力してみよう。

```
ls -l
```

すると，ファイルの一覧の中にサイズを表す数字が表示される。計算値と比較して確認してほしい。

さて，この 340 KB というファイルサイズはいかにも大きい。たとえばデジカメの画像だったら，この画像サイズなら 10 分の 1 程度のはずだ。情報として処理するにも，保存するにも，ネットワーク上で転送するにも，あまり大きなファイルは望ましくない。

その問題を解決するには，コンピュータの中でどのように情報がしまわれているのかを知る必要がある。しばらく脱線して，そのことを簡単に見ておくことにしよう。

3.3.2 アスキーコード

いわゆる半角英数字やその他の記号類を**アスキーコード (ASCII code)** という。アスキーコードは，エディタで編集できる**テキスト**の文字でもある。これらの文字は，コンピュータの内部ではどのような形でしまわれているのだろうか。それを簡単に知るには，次のようにするとよい。

irb でまず次のように入力してみよう。

```
irb(main):001:0>10 + 10
```

すると，次のような返答が返ってくる。

```
=> 20
```

こんなふうに irb は入力された行を評価して，その結果を対話的に返してくれる。

■数と文字の対応を知る irb に `"a"` を入力すると，そのまま `"a"` が返される。

```
irb(main):002:0>"A"
=> "A"
```

これじゃ何も面白くない。それでは次に `65.chr` と打ち込んでほしい。

```
irb(main):003:0>65.chr
=> "A"
```

なんと，`"A"` が現われる。65 という数は文字 `"A"` に対応付けられているらしい。それでは，いろいろな数字でこの操作を行ってみて，どんな結果が得られるかを試してみよう。

アスキーコードの意味

前節の実験をていねいにやってみると，次のようなことが分かる。

- 文字に変換できる数字の範囲は決まっている。
- 32 から 126 までの数は，何らかの文字や記号に対応している。
- 0 から 31, 127 から 255 までの数字では，`"\024"` といった数のようなものが表示される。これはその数を 8 進法で表したものである。
- 256 以上の数字だと，`RangeError: 256 out of char range` というエラーメッセージが表示される。ここで “char” というのは character の略で，文字を意味している。

上のような結果は次のことを意味している。

- 256 未満の整数は文字に対応付けられていて，それを超えると，文字としての範囲を超えてしまう。つまり，文字というのは 1 バイトの範囲に収められている。
- 実際の文字に対応しているのは，32 から 126 までの 95 個の整数である。これらの整数に対応づけられた文字を**アスキーコード (ASCII code)** という。この対応関係をまとめた表は**アスキーコード表**という。表 3.2 に示した。
- 256 未満で，かつ文字に対応していない数については，irb は 8 進法でそれを出力するようになっている。

コンピュータの中の情報の格納のされ方，およびそれに関する知識を下にまとめて示しておこう。

表 3.2　ASCII コード表:括弧で書かれているのは制御コード。16 進 2 桁の 1 桁目は上端の横に，2 桁目は左端の縦に書かれている。たとえば，41H は 'A' にエンコードされている。20H の SP はスペースのこと。

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	(NUL)	(DLE)	SP	0	@	P	‘	p
1	(SOH)	(DC1)	!	1	A	Q	a	q
2	(STX)	(DC2)	"	2	B	R	b	r
3	(ETX)	(DC3)	#	3	C	S	c	s
4	(EOT)	(DC4)	$	4	D	T	d	t
5	(ENQ)	(NAK)	%	5	E	U	e	u
6	(ACK)	(SYN)	&	6	F	V	f	v
7	(BEL)	(ETB)	’	7	G	W	g	w
8	(BS)	(CAN)	(	8	H	X	h	x
9	(HT)	(EM)	)	9	I	Y	i	y
A	(NL)	(SUB)	*	:	J	Z	j	z
B	(VT)	(ESC)	+	;	K	[	k	{
C	(NP)	(FS)	,	<	L	\	l	|
D	(CR)	(GS)	-	=	M	]	m	}
E	(SO)	(RS)	.	>	N	^	n	~
F	(SI)	(US)	/	?	O	_	o	(DEL)

1. **コンピュータの中の情報は，すべて 1 Byte ごとにメモリに入っている。つまりコンピュータの情報はすべて「数」である。**
2. **いわゆる「半角英数字」は，何らかの数に対応付けられている。その対応表をアスキーコードという。ただし 1 Byte で表される 0 〜 255 (0x00 〜 0xFF) までのうちの一部だけが文字に対応付けられている。**
3. **つまりいわゆる「半角英数字」はコンピュータの基本的な情報としての文字である。これらは正しくは 1 バイト文字あるいは 1 バイトコードといい，「半角英数字」は非専門用語である。**
4. **アスキーコードに対応していない数もあり，それらを含むファイルをバイナリファイルという。**

3.3.3 バイナリのフォーマットを利用する

前に登場したプログラム ppm002.rb をバイナリファイルを吐き出すように変更してみよう。

```
# ppm002.rb
nx = 240; ny = 180
puts "P3"
puts "# PPM test"
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... nx
  for j in 0 ... ny
    print "255 0 0 "
  end
end
```

PPM フォーマットの仕様を参照すると，変更の要点は次の 2 点になる。

- データにバイナリのコードを使うときのマジックナンバーは P6
- 色データとして"255 0 0" という文字列の代わりに，255.chr，0.chr，0.chr を使う。

問題 3–2 ppm002.rb を上の方針で書き換えたプログラムを ppm003.rb という名前で作成しなさい。それを走らせてみてから，次のページにあるプログラム例を見て確認しなさい。

書き換えたプログラムは下の通りである。

```
# ppm003.rb
nx = 240; ny = 180
puts "P6"
puts "# PPM test"
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... nx
  for j in 0 ... ny
    print 255.chr, 0.chr, 0.chr
  end
end
```

このプログラムを走らせて得られる画像を GIMP で見てみなさい。また，`ls -l` コマンドを使って，アスキーフォーマットの PPM 画像とファイルサイズを比較しなさい。

3.4　パターンを生成する

ここまでのところで，コンピュータ内部に情報がどのように置かれているのか，画像ファイルというのはどういう構造をとっているのかについて学び，簡単なテストプログラムを試作した。しかし，単色の画像だけではつまらない。なんとかして，かっこいいパターンを作りたい。これが次の新しい問題だ。人間ならピクセルをひとつずつ考えて埋めていくというやり方をとれるのだが，そんなわけにはいかない。その代わり，プログラムでパターンの生成の手順を実現することになる。それにはどうしたらいいだろうか。いくつかのサンプルを示すことにしよう。

2 つに塗り分ける

図 3.6 の左と中央の画像を生成するプログラムを `bicolor001.rb`, `bicolor002.rb` として示そう。まずはこれらを自分で書いて走らせてみよう。

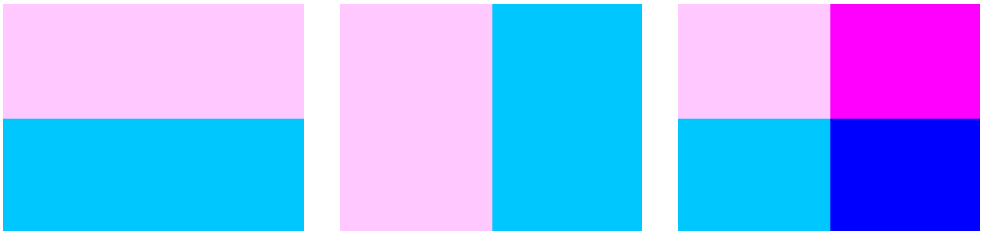


図 3.6　旗のようなパターン

```
# bicolor001.rb
nx = 240; ny = 180
puts "P6"
puts "# PPM test"
puts "#{nx} #{ny}"
puts "255"
r = 255.chr
g = 200.chr
b = 255.chr
for i in 0 ... ny
  if i > ny / 2 then
    r = 0.chr
  end
```

```
  for j in 0 ... nx
    print r, g, b
  end
end
```

```
# bicolor002.rb
nx = 240; ny = 180
puts "P6"
puts "# PPM test"
puts "#{nx} #{ny}"
puts "255"
g = 200.chr
b = 255.chr
for i in 0 ... ny
  for j in 0 ... nx
    r = 255.chr
    if j > nx / 2 then
      r = 0.chr
    end
    print r, g, b
  end
end
```

問題 3–3　上の 2 つのプログラムで使っているアイディアを組み合わせて，図 3.6 の右端のパターンを出力するプログラムを作りなさい。

色に名前を付けて使う

上のソースでは，色を決めるのに r, g, b の 3 つの原色を独立にあつかうというやり方をとった。しかしある決まった色をピクセルに与えるは，次のようなやり方が可能だ。

- RGB の数値を個別に設定せずに，あらかじめ次の例のように色をひとつの変数として定義しておく。なお，Ruby の場合，大文字で始まる変数名は定数として扱われ，変更が許されないという性質をもっている。

  ```
  Cyan = 0.chr + 255.chr + 255.chr
  Magenta = 255.chr + 0.chr + 255.chr
  ```

- ピクセルの出力では，これらを使う。

  ```
  print Cyan
  ```

のように。

この表現法を使ったソースの例を下に示す。

```
# bicolor004.rb
cyan = 0.chr + 255.chr + 255.chr
magenta = 255.chr + 0.chr + 255.chr
color = cyan
nx = 240; ny = 180
puts "P6"
puts "# PPM test"
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... ny
  for j in 0 ... nx
    if i > ny / 2 then
      print magenta
    else
      print cyan
    end
  end
end
```

16 進の色表示をそのまま使うには

たいていの色名のデータでは，ある色の名前と，その 16 進数による表現とが書かれている。いくつかの色のデータを下に挙げてみる。

```
wheat   #F5DEB3
orange    #FFA500
hotpink   #E179B4
tomato    #FF6347
lavender  #E6E6FA
turquoise #40E0D0
```

これらの 16 進表記は HTML で使うことを念頭においているわけだが，このテキストではプログラムの中で色を表現しようとしているので，色の表現を 16 進数ではなく，10 進数で書いてきた。でもこんなふうに 16 進数で書かれたものは，そのまま使ったほうが

楽だ。

Ruby, C, Java など多くのプログラミング言語では，16 進数は接頭語として 0x を付けて表すことになっている。したがって，小麦色の例は次のように直せばよい。

```
Wheat = 0xF5.chr + 0xDE.chr + 0xB3.chr
```

なお，0xFF でも 0xff でもどちらでもよい。

シアン，マゼンタも同様にして，

```
Cyan = 0.chr + 0xFF.chr + 0xFF.chr
magenta = 0xFF.chr + 0.chr + 0xFF.chr
```

としたほうが，色名のデータからプログラムを書きやすい。

問題 3–4　これまでに学んだ手法をまねして，四角を 4 色に塗り分けるソースを書きなさい。

3.5　グラデーションをかける

次のソースによって，図 3.7 のようにグラデーション[*8]のかかったパターンを生成することができる。

```
# grad001.rb
nx = 400; ny = 80
g = 250
puts "P6"
puts "# Draw Gradient "
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... ny
  for j in 0 ... nx
    r = j % 255
    b = j % 255
    print r.chr + g.chr + b.chr
  end
end
```

図 3.7　グラデーションの例

3.5.1　剰余計算は強力な道具

上のソースによって，どうしてきれいな傾斜が生まれるのかを説明しておこう。このソースの中で働いているのは，% による剰余（余り）の計算だ。増加し続ける整数の数列を，ある定数で割ってやると，繰り返しが発生する。

例として次のソースを走らせてみなさい。ファイル名は test.rb のように適当なものを使えばよい[*9]。

*8　コンピュータグラフィックス関連の記事を読んでいると，英語ではグラデーション (gradation) ではなく，グラジェント (gradient) ということが多いようだ。これは傾斜のことで，それはそれで意味が通っている。どうして訳語が違う外来語になってしまったのか，どうもよく分からない。いずれにせよ，英文でグラフィックスの説明を読むときには注意しておかないといけない。

*9　ある目的でプログラムを書くときに，その一部をテストするための小さなプログラムを書くのは，よく行われることである。

```
for i in 0 .. 20
   puts i % 9
end
```

走らせた結果として，次のような数列が出力されるはずだ。

```
0 1 2 3 4 5 6 7 8 0 1 2 3 4 5 6 7 8 0 1 2
```

これは for ループによって i が 0 から 20 まで 1 ずつ増加していくときに，それを割った余りの方は除数の 9 未満までの数を繰り返すことからきている。

グラデーションを描いたソースでは，水平方向の位置を決める j が増加するときに，RGB のうちの r と b の色が 0 から 255 まで増えては，また 0 に戻るという繰り返しが発生するので，グラデーションがかかるというわけだ。

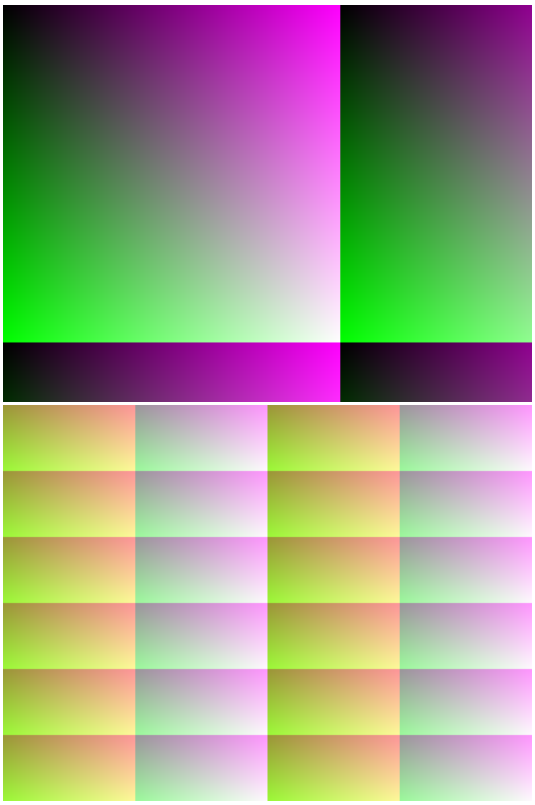


図 3.8　少し凝ったグラデーション

3.6 関数で塗り分ける

1 次関数は直線を作り，2 次関数は放物線，双曲線，円，楕円といった美しい曲線を作り出すことができる。これらをプログラムで作ってみよう。

3.6.1 円を描く

円を数学的に表す

円というのはある点から等しい距離にある点の集合のことで，次の式はそのことを意味している。

$$(x-a)^2+(y-b)^2=r^2 \tag{3.1}$$

ここでは，中心が (a,b) にあって，半径が r であるものとしている。これは三平方の定理 (図 3.9 右) によって，(a,b) から距離が r のところにあるすべての $(x.y)$ の集まりであるということを意味している。

ということは，次の不等式を満たす点と，そうでない点を塗り分ければ，円の内部を塗りつぶすことができるはずだ (図 3.9 左)。

$$(x-a)^2+(y-b)^2<r^2$$

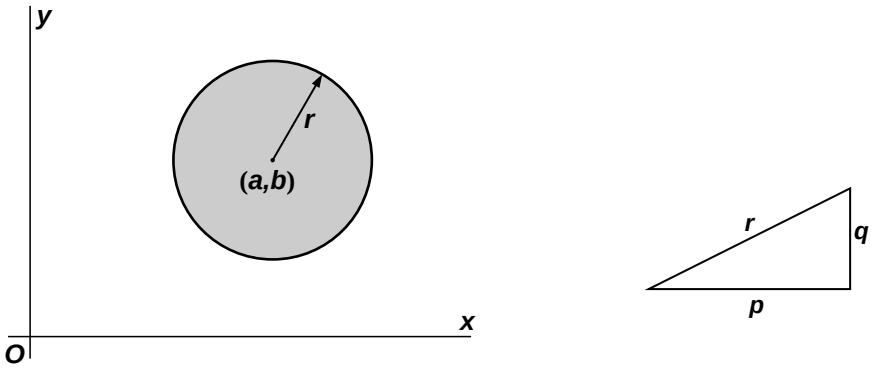


図 3.9 左図：中心から一定の距離 r 未満の点を塗りつぶせば円が描ける。右図：三平方 (ピタゴラス) の定理で $\boldsymbol{r^2=p^2+q^2}$，または $\boldsymbol{r=\sqrt{p^2+q^2}}$ となる。

以上の事実を円を描くために応用しよう。図 3.10 を見てほしい。

横に nx, 縦に ny 並んだピクセルの並びの中に，中心の座標が cx1, cy1 で， 半径 rad1 の円を描くことを考える。 ただし横軸の座標は j で，縦軸の座標は i である。

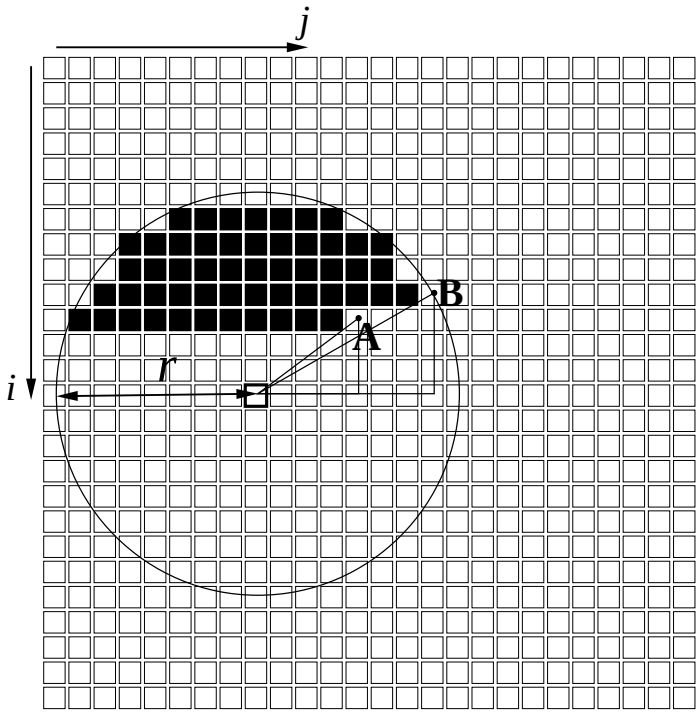


図 3.10　円周の内側のピクセルを塗りつぶす手順。**A** のピクセルは中心との距離が半径よりも小さいので円の内側，**B** は円の外側。

```
# circle001.rb
nx = 240; ny = 180
cx1 = 100; cy1 = 60;
rad1 = 50; srad1 = rad1 ** 2
indigo = 0x4b.chr + 0x00.chr + 0x82.chr
gold   = 0xff.chr +  0xd7.chr + 0x00.chr
puts "P6"
puts "# Draw Cirlce "
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... ny
  for j in 0 ... nx
    if (j - cx1) ** 2 + (i - cy1) ** 2 < srad1 then
```

```
      print gold
    else
      print indigo
    end
  end
end
```

このソースのアイディアをちょっとひねって，RGB それぞれについて異なった位置に円を描いてみるようにしてみることもできる (図 3.11 右)。

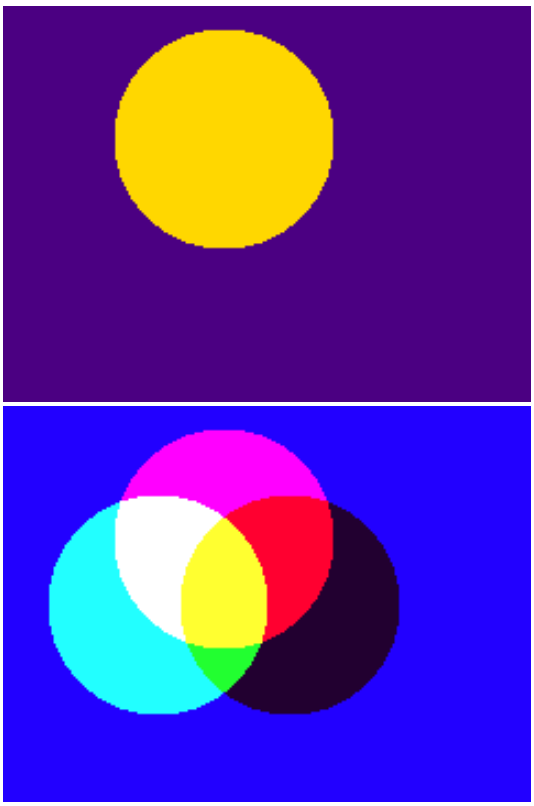


図 3.11　円を描く

3.6.2　楕円，双曲線

式 (3.1) の円の方程式を少し変えると，楕円の方程式になる。

$$p(x-a)^2 + q(y-b)^2 = 1 \tag{3.2}$$

式 (3.2) で p と q の値が異なるとさまざまな縦横の比率の楕円になる。
また，単に符号を変えるだけで双曲線が得られる。

$$p(x-a)^2 - q(y-b)^2 = 1 \tag{3.3}$$

図 3.12 にこれらの典型的な形を示した。

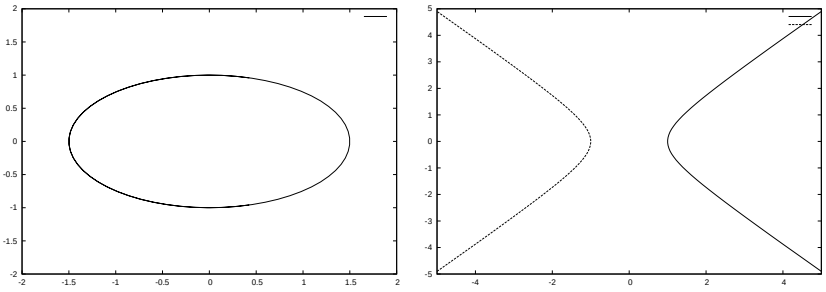


図 3.12　楕円 (左) と双曲線 (右)

問題 3–5　円を描くプログラムを楕円や双曲線の式を使ったものに書き換えてやることで，それらの図形による描画ができる。工夫してやってみよう。

3.7　立体座標上の関数で立体的に考える

前節では円などの図形を x, y を用いた二次関数で表して，それを描画に使ってみた。しかし，単に境界線を区切るためにではなく，グラデーションを施すために利用することもできる。

図 3.13 は，ピクセルをぐっと拡大したもので，円の中心の座標を中心として，中心からの座標のずれを括弧の中に書き込んであり，いくつかのピクセルには，ずれの 2 乗の和が灰色の大きい文字で書き込んである。たとえば，括弧で $(2,-1)$ と書いてあるピクセルを見てほしい。大きく '5' とあるのは，$2^2+(-1)^2=5$ として計算した値である。

ちょっと手間をかけて，2 乗の和がまだ記入されていないところも計算して埋めてほしい。そして図をよく眺めよう。

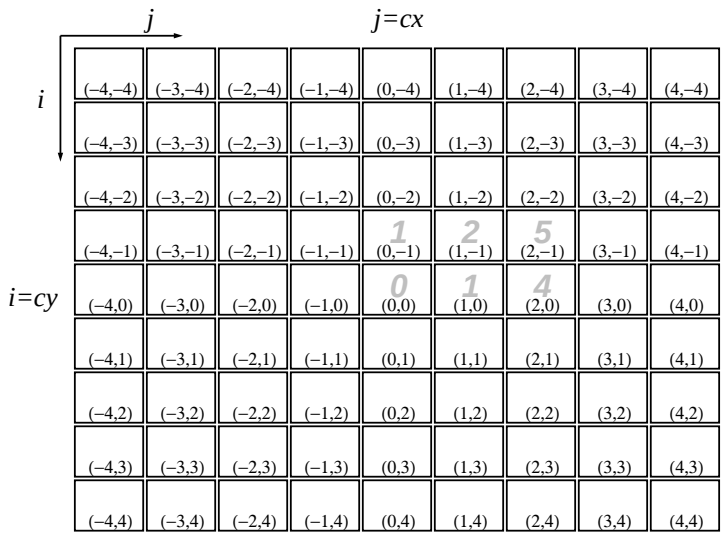


図 3.13　円の中心を (0,0) として，そこからの横と縦の「ずれ」を書き込んだもの。またいくつかのピクセルには，ずれの 2 乗の和が灰色の文字で書き込んである。

この図で，2 乗の和の値がたとえば 10 以下のところを塗りつぶせば，半径が 3 の円に近い形が得られる。これはすでにやってみたように，円を描くアルゴリズムになっている。

しかし，ここで視点を少し変えて，計算して得られた値だけ紙面の上の方に盛り上がっているという立体的な形を想像してみよう。中心がくぼんでいて，まわりはお椀の形のように盛り上がっている形が浮かび上がる。図 3.14 に大体の形を示した。

つまり中心から外側に向かって急激に大きくなっていく勾配があるわけで，これを使ってグラデーションをか

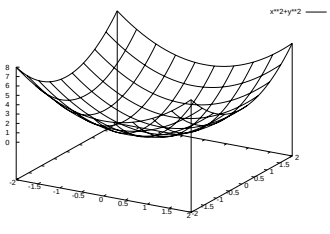


図 3.14

けたらどうなるだろうか？たとえば中心から右に進んでいくと，$0, 1, 4, 9, 16, \ldots$ と増えていくので，これを 255 で割った余りを計算してみよう。次の短いプログラムを走らせればいい。

```
for i in 0 .. 50
  puts i ** 2 % 255
end
```

本当は `(i ** 2 + j ** 2) % 255` を平面的にずっと計算していくのを，ここでは真横だけ計算していることに注意しておこう。すると次のような数が現れる。

0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 25, 56, 89, 124, 161, 0, 41, 84, 129, 176, 25, 76, 129, 184, 41, 100, 161, 24, 89, 156, 25, 96, 169, 44, 121, 0, 81, 164, 49, 136, 25, 116, 9, 104, 1, 100

これを見ると，数が大きくなっては小さい値に戻っていくのが分かるだろう。これで色を付けたら同心円状のグラデーションが得られないだろうか？以上の発想でプログラムを書いてみると次のようになる。話は長かったが，プログラムは短い。得られる画像は図 3.15 のようになる。

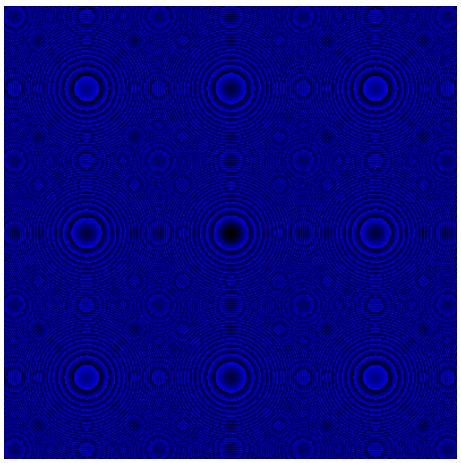


図 3.15

```
# circle_grad001.rb
nx = 400; ny = 400
r = 0; g = 0
cx1 = 200; cy1 = 200
puts "P6"
puts "# Draw Gradient "
puts "#{nx} #{ny}"
puts "255"
for i in 0 ... ny
  for j in 0 ... nx
    b = ((j-cx1) ** 2 + (i-cy1) ** 2) % 255
    print r.chr + g.chr + b.chr
  end
end
```

このプログラムでは青一色しかいじっていないし，単純な円の関数を使っている。3 色を使って楕円や双曲線，さらには 3 次関数を利用したり，あるいは複数の円を組み合わせたり，グラデーションの傾斜がゆるくなるようにいじったりすれば，もっともっと複雑で華麗なパターンができるかも知れない。

3.8 いろいろなアイディア

3.8.1 乱数を使ってみる

■rand 関数のふるまい コンピュータが持っている万能サイコロが**乱数**だ。その働きを知るために irb で試してみよう。まず irb で次の入力を何回も繰り返してみてほしい。

```
rand(6)
rand(6)
rand(6)
...
```

すると，0 から 5 までの数がランダムに出力されることがわかる。様子が見えたら，こんどは 6 の代わりに別の数を使ってみよう。こんなふうにいじっていると，rand という関数の使い方をハックできる。

rand(n) という関数は，0 から $n-1$ までの整数をランダムに出力する。

なお，rand(0) あるいは rand() とすると，0 以上 1 未満の一様乱数が得られる。そのことも irb で確認してほしい。

ピクセルの色を決定するのに，乱数を使ってみたらどうなるだろう？たとえば

```
r = rand(256)
```

とか

```
r = 120 + rand(136)
```

のように。

3.8.2 バッファ配列を使う

ここまでは，計算結果をファイルに直接書き出すというやり方をとって来た。しかしそれだと，すでに描いたパターンの上に他のパターンを描き加えるということができないのでちょっと不便である。そこで，ピクセルを配列データとして用意しておき，その配列のデータを加工することで，図形を作成するようにしてみよう。

このような目的で使われる配列データを**バッファ (Buffer)** という。データをためておく場所といった意味である。

下にサンプルのソースを示そう。ここでは最初に黒に相当するピクセル（[0,0,0]）を要素とする二重の配列を作成して，その配列に対して 2 通りの処理を行っている。これを参考にして，いろいろな工夫をやってみてほしい。

```
# buff001.rb
def print_buffer(buff,nx,ny) # バッファを書き出すメソッド
  for i in 0 ... ny
    for j in 0 ... nx
      r = buff[i][j][0]
      g = buff[i][j][1]
      b = buff[i][j][2]
      print r.chr +  g.chr + b.chr
    end
  end
end

def print_header(nx,ny) # ヘッダを書き出すメソッド
  puts "P6"
  puts "# Bufferd version"
  print nx, " ", ny ,"\n"
  puts 255
end
nx = 400; ny = 300
# サイズ nx の配列からなるサイズ ny の配列を用意し，それぞれに [0,0,0]
# を代入しておく。

buff = Array.new(ny){Array.new(nx){[0,0,0]}} # バッファを用意して初期化しておく

print_header(nx,ny) # ヘッダ書き出し

# 中央に黄色の矩形を描いてみる
for i in (ny / 2 - 20) ... (ny / 2 + 20)
  for j in (nx / 2 - 20) ... (nx / 2 + 20)
    buff[i][j] = [0xff,0xff,0]
  end
```

```
end

# 左下の三角を青くしてみよう。
for i in 0 ... ny
  for j in 0 ... i
    buff[i][j][2] = 0xa0
  end
end

print_buffer(buff,nx,ny) # 最後にバッファ書き出し
```

第 4 章

Postscript プログラミング

4.1 Postscript とは何か

4.1.1 Postscript と Ghostscript

Postscript(PS) は Adobe Systems 社[*1]が規格を定めたページ記述言語で，ラスター画像とは異なり，論理的な記述で図形やテキストを表現することができる。言い換えれば Postscript は一種のプログラミング言語でもある。また，後から述べるように， Postscript では**スタック**というデータ構造の中に，数値や文字列といったデータを格納するというやり方をとっている。

なお，Postscript で記述された情報は， その文法を解釈できる PS プリンタで印刷することができ，多くの電子出版 (DTP) の主力はこの方式によっている。DTP やデザイン業界では Mac が主に使われるが，それは Apple が Postscript プリンタを開発して，それまでにない高品質の出版を実現したからである。

Ghostscript(GS) は， Postscript で記述された描画プログラムを解釈してラスター画像を生成するフリーソフトで， GNU プロジェクトで開発されてきた。これは無料で利用できて，Unix, Windows, Mac など多くのプラットフォームで走るので，さまざまなところで利用されている[*2]。

Ghostscript そのものは， PS ファイルからラスター画像情報への変換を行うだけで，表示の処理は他のアプリケーションに任せるという方式をとっている。そのためによく用いられているのは GSview である。以下では， Ghostscript と GSview がインストールされているものとして進めていくことにする。

4.1.2 Postscript からの発展

Postscript は論理的な記述による描画をおこなうプログラミング言語なので，現在の多くのプログラミング言語のグラフィックスの手法と共通するところが多い。特に Java の Graphics2D クラスでは，パスの扱い方やフォントの処理などに PS の手法が引き継がれているので，あらかじめ PS を知っておくと理解しやすい。

---

*1 Adobe は「アドビ」と読む。

*2 このプリントの図版も， Ghostscript を利用して紙面にはめ込まれている。

4.1.3 Postscript を使いこなすための情報

Postscript を解説したドキュメントは Adobe 社からリファレンスマニュアルとクックブック[*3]の 2 冊が PDF 版で公開されていて，だれでも入手できる。ただし英語で書かれている。

Postscript Language Reference Manual(PLRM) 『Postscript 言語リファレンスマニュアル』として邦訳がある。基本概念と文法を詳解した解説書。

Postscript Language Tutorial and Cookbook 通称 Blue Book と呼ばれる青い表紙の本があるが邦訳はない（たぶん)。英語だが多数のソースと図があるので，使いやすい。

なお，著者が簡単にまとめたウェブコンテンツがある。上記の原文マニュアル以外に，Postscript プログラミング入門のための解説と，独自に開発したライブラリが置かれている。下を参照していただきたい。

`http://www.cs.kyoto-wu.ac.jp/~konami/documents/ps/`

[*3] 実例をもとにした解説書のことをクックブックという。

4.2 ちょっと使ってみる

4.2.1 Ghostscript を対話的に使ってみる

GS を使って描画してみよう。Cygwin のシェル画面から次のように入力して GS を立ち上げる。

```
gswin32
```

これで白い GS コンソールが出現するので，つぎのように入力していってみよう。

```
120
70
moveto
250 300
lineto
stroke
```

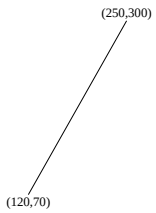


図 4.1 Ghostscript を使った簡単な描画

すると，別画面に斜めの直線が描かれる (図 4.1)。

上の入力によってどういうことが起きているのだろうか？ Postscript 言語では**スタック (stack)** というデータ構造が使われている。これは物を縦に詰め込める容器みたいなものだ。

図 4.2 を見てほしい。最初の状態 (1) でスタックは空になっている。そこに **120** というデータを入力すると，スタックのトップにデータが押し込まれる。この動作を push という。次に 70 を入力するとトップに push されて，前にあったデータは 1 段下に移動させられて (3) の状態になる。

次に (3) の状態に与えられている `"moveto"` はデータではなく，データを操作する**オペレータ**だ。`moveto` が来ると，スタックの上部にある 2 個のデータが取り出され (「**pop** される」という)，それらのデータが表す座標 $(120, 70)$ に描画のペン先が移動する。

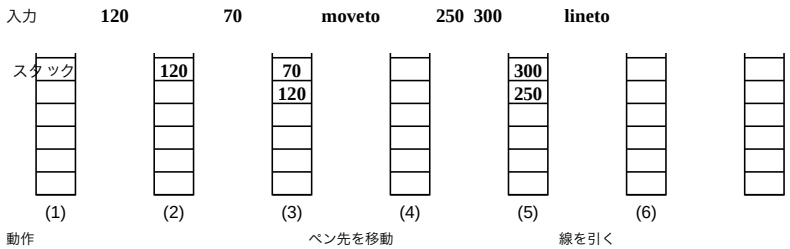


図 4.2 PS のスタックにデータとオペレータを入力していったときの状態の変化と描画の内容:一連の入力によって，(120,70) から (250,300) の座標へと直線が引かれる。

次は `250 300` とデータが続けて入力され，(5) の状態になる。これに対して今度は `"lineto"` というオペレータが入力される。これはその意味の通りに，「線をそこまで引け」ということで，スタックの上部 2 つのデータを取り出して，**パス** (ペンが動いた跡) が作られる。

ただし，この状態では図形のもとになっているパスは定義されているだけで実際には引かれない。線を引くためには，さらに `stroke` というオペレータを入力する。これではじめて，蓄えられていたパスが実際に描画される。

4.3 Postscript の基本

4.3.1 EPS ファイル

Postscript は DTP のためのシステムなので，複数ページの印刷画面をまとめて記述できるようになっている。しかし，手作業で文章に貼り込むための画像は，サイズを指定した 1 枚の図版として扱うことが主なので，そのために **EPS (Encapsulated Postscript)** という形式の PS ファイルが用意されている。

EPS では，BoundingBox という境界線の枠を定義している。その枠を設定するには次のような 2 行がヘッダとして必要である。

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 0 400 300
```

2 行目の `%%BoundingBox:` の後の 4 つの数字は描画する範囲をポイント単位で表す (4.3 節参照)。ちなみに A4 サイズであれば紙面は 210 mm × 297 mm なので，`0 0 595 842` となる。これを大体の目安としてサイズをきめるとよい。

ここをたとえば次のようにすると，負の領域に作図したものを表現できる。ただし，アプリケーションによってはこれを正しく解釈しないことがあるので，使うときには確かめた方がよい。

```
%%BoundingBox: -200 -150 200 150
```

ヘッダを先頭に置けば，あとは描画のためのデータとオペレータを Postscript の約束に従って記述していけばよい。以下では，上記のヘッダが常にあるものとして解説を進めることにする。

4.3.2 基本的な約束

■長さの単位 PS の長さの単位は**ポイント (pt)** で，1 pt = 1/72 インチ = 25.4/72 mm = 0.3528 mm である。PS はどんな大きさにでも拡大，縮小できるが，文字の大きさや線の太さがアンバランスにならないように，大体この基準を満たすように大きさを決めた方がよい。

■座標系 PS の画面の座標は，左下が原点 $(0, 0)$ としてある。ただし，実際に描画する範囲をすらすこともできる (EPS のヘッダの説明を参照)。

■絶対座標と相対座標 すでに出てきた `moveto` や `lineto` は画面の原点に対する絶対座標で位置を決める。しかし，今いる点を基準にした相対座標を使うこともできる。そのためのオペレータは先頭に `"r"` が付いた `rmoveto`, `rlineto` となっている。

■コメントは % 記号 行の中に `%` があると，その後の部分は読み飛ばされる。コメントを書いたり，ある行を一時的にコメントアウトして無効にするときに使う。また，EPS 形式のファイルでは，特定のコメント文がヘッダーに使われる (後述)。

■def — 変数や手続きを定義する `def` オペレータは変数や手続きの定義 (definition) のために用いられる。Ruby や C などでは変数に代入するのに `x = 24` のような文を用いるが，PS ではそれとかなりちがっていて，次のようにして変数 `x` の定義と代入が行われる。

```
/x 200 def
```

また，次のようにすると，数値の組がまとめて定義され，p1 だけで 100 150 が入力される。

```
/p1 {100 150} def
```

PS のオペレータ名は長いので，次のように短く定義してしまうこともできる。

```
/m {moveto} def
```

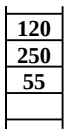
こうすると，単に m とするだけで moveto の代わりになる。

4.4 オペレータとスタック

4.4.1 スタックの表し方

Postscript ではスタックにデータが詰め込まれて利用されが，いちいち縦のスタックの図で表すのは煩雑なので，以後は単にデータを横に並べただけでスタックを表すことにする。

たとえば，空のスタックに 55，250，120 の順で数値を入力したとすると，下の左図のようにデータが置かれる。これを右のように**スタックのトップが右端に来るように**並べて表す。



⇒ 55 250 120

4.4.2 スタックのデータを調べる — pstack

データを入力したり処理したりすると，スタックの中のデータはその都度変化していく。現在，スタックにどのようなデータが入っているかを gs で知るためには pstack というオペレータが用意されている。gs の画面で次のように入力してみよう。

```
20
30
pstack
moveto
pstack
```

1 回目の pstack では 下から 20, 30 という数値が並んでいること，2 回目の pstack ではスタックが空になっていることがわかる。

GS を使って対話的にプログラムを書くときには，しばしば pstack を使うので，あらかじめ次のように入力しておくとよい。

```
/p {pstack} def
```

すると，これ以降は単に p だけでスタックの中身を知ることができる。

4.4.3 算術計算とスタック操作のオペレータ

スタックで計算する

Postscript ではさまざまな算術演算が用意されている。

```
10 20 add  % 加算 => 30
30 12 sub  % 減算 => 18
14 7  mul  % 乗算 => 98
```

```
72 30 div  % 除算 => 2.4
72 30 idiv % 整数の除算 => 2
40 6 mod   % 剰余（余り） => 4
24  neg    % 符号反転   => -24
```

ここで最初の行は，スタックに 10, 20 と入力した後で add を入力すると，10 と 20 は使われてしまって，和の 30 がスタックのトップに残ることを意味している。

この様子は pstack を使って次のようにして調べてみることができる。

```
10 20
pstack
add
pstack
```

■逆ポーランド算法 上記の計算では，まず数値が最初に入力されて，その後に算術オペレータが置かれる。一方，通常の二項計算では次のように演算子 (ここでは +) は数値の間に置かれる。

$$10 + 20$$

このように Postscript ではデータを並べてから加減乗除などの演算子がくる。このような計算の順序の表記法を**逆ポーランド記法**という。

問題 4–1 次の計算を実行して結果がスタックのトップに残るようにしなさい。

(例) $4 + 5 - 6$

解答例：`4 5 add 6 sub`

1. $(10 + 20)/2$
2. $20 + 40/4$

■スタックのデータを操作する スタックのデータを複製したり，入れ替えたりする操作が用意されている。

- トップの 2 つのスタックデータを交換 (exchange) する。

  `50 100  exch % => 100  50`
- トップのデータを複製 (duplicate) して push する。

  `60  dup % => 60 60`
- 指定した数のデータをコピーしてスタックに置く。

  `10 20 30 40 50`

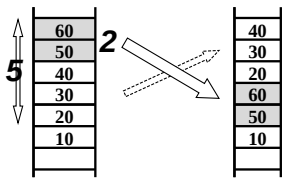
  `3 copy % => 10 20 30 40 50 30 40 50`

  上部の 3 個のデータ `30 40 50` が複製されている。
- 指定した数のデータの一部を回転 (roll) させる

  `10 20 30 40 50 60`

  `5 2 roll  % => 10 50 60 20 30 40`

  上部の 5 個のデータ `20 30 40 50 60` が下のように「回る」。



問題 4–2 下の矢印の左辺のようにスタックにデータが置かれているとき，どのような操作をすれば右辺のように変換できるか。スタック操作と算術演算のオペレータを使って実現しなさい。

(例) `120 150 => -120 150` 解答例 `exch neg exch`

1. `120 150 => -120 -150`
2. `66 77 88 120 250 => -120 -250 66 77 88 120 250`
3. `240 280 => 260` (中間の値を求める操作)

4.5　描画の基本操作

4.5.1　色を指定する

オペレータ setrgbcolor を使う。0〜1 の実数で RGB の値を指定する。また setgray でグレイスケールの指定もできる。色と数値の対応は表 1.1 を参照してほしい。

```
1 0 0 setrgbcolor % これ以降の描画は赤で行われる
0.8 setgray       % グレイスケールの指定もできる
```

4.5.2　線の種類を指定する

```
2  setlinewidth  % 線の太さを 2 pt に
[3 3] 0 setdash  % 破線にする
[] 0 setdash     % 破線を実線に戻す
```

4.5.3　直線で描く

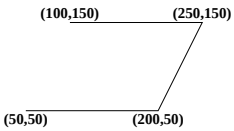
直線をつないで線画を描いたり，その内側を塗りつぶす。

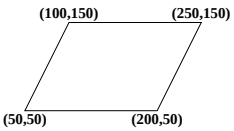
下の例では，同じパスに対して，それらを線でつなぐ，パスを閉じてからつなぐ，面として塗りつぶすという 3 通りの処理が行われている。

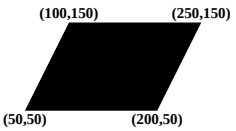
```
50 50 moveto
200 50 lineto
250 150 lineto
100 150 lineto
stroke
```

```
50 50 moveto
200 50 lineto
250 150 lineto
100 150 lineto
closepath
stroke
```

```
50 50 moveto
200 50 lineto
250 150 lineto
100 150 lineto
fill
```







4.5.4 円弧を描く

円弧 (円周の一部) を描くには，次のようにする。ここで x y は中心の座標である。角度は図 4.3 のように反時計方向に定める。

```
x  y  半径  開始角  終了角 arc stroke/fill
```

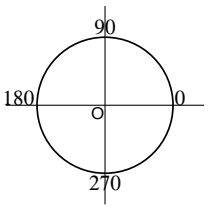


図 4.3

図 4.4 にこれらの利用例を示した。

なお，描画に際しては，下の例のように newpath を先に実行しておいたほうがよい。そうしないと，それ以前に設定されていた位置からパスを構成してしまい，余計な線が現れることがよくある。

```
newpath
90 60 50 30 270 arc
stroke
```

```
newpath
90 60 50 30 270 arc
fill
```

```
newpath
90 60 50 0 360 arc
stroke
```

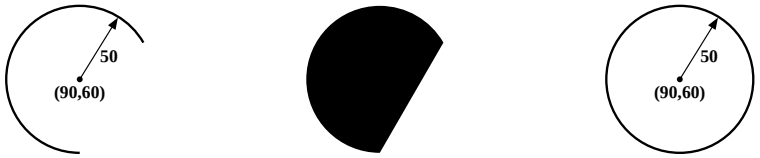


図 4.4

4.6 座標変換で移動，拡大／縮小，回転

グラフィックスで必要な操作として，図形を移動，拡大／縮小，回転するといった操作がある。これらは座標系そのものを変換してその後で描画するという処理で実現されている。

4.6.1 座標系の移動

座標系を移動させて，原点をずらして描画することができるといろいろと便利なことがある。実例をみてみよう。

図 4.5 を描くためのソースを下に示す。これを見ると，原点 $(0, 0)$ に中心をもつ半径 30 の円を描く行が 2 回現れている。何もなければ，これらはぴったり重なり合ってひとつの円にしかならないはずだ。ところがこれらの円の描画の間に，次の行が挿入されている。

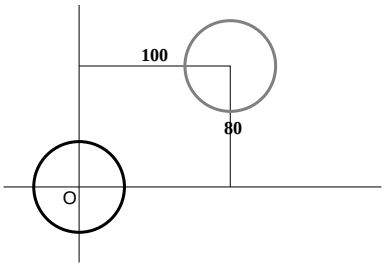


図 4.5

```
100 80 translate
```

座標を移動 (translate) させるオペレータがここで使われているのだ。2 度目の円は灰色で描くように指定してあるので，原点が $(100, 80)$ に移動していることがわかる。

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -50  -50 200 120
0 0 30 0 360 arc stroke
0.5 setgray
newpath
100 80 translate
0 0 30 0 360 arc stroke
```

上のソースを見る限りでは，$(100, 80)$ を中心とする円を描くためにわざわざ座標を変換するメリットはないように思えるかもしれない。つまり次のように最後の 2 行は 1 行で書ける。

```
100 80 30 0 360 arc stroke
```

簡単にできることをわざわざ難しくやることはない。しかし実は，この後に出てくるスケーリングや回転と組み合わせられたときに，座標変換による移動の処理は威力を発揮することになる。

4.6.2 スケーリング

拡大縮小のオペレータ

座標を伸縮することをスケーリングという。 Postscript では，`scale` というオペレータがそのために用意されている。たとえば横に 2 倍，縦に 1.5 倍に拡大したいときには，次のようにする。

```
2 1.5 scale
```

この行の後では，図形は指示された倍率に従って拡大して描画される。それでは実際にこの処理を行った例を見てみよう。

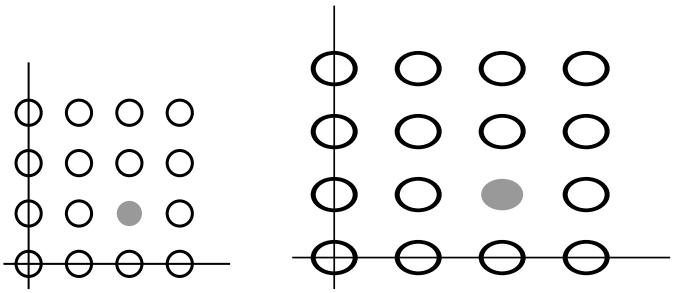


図 4.6 左図を `2 1.5 scale` で拡大して右図が得られる

原点を中心として拡大される

図 4.6 を見ると，ひとつ重要なことに気づく。それは，**原点を中心として画面が拡大される**ということだ。したがって，図の原点に中心が置かれていた円はその位置で拡大されて楕円に変形されているが，それ以外の円は，拡大と同時に原点から遠ざかるような移動が起きている。

任意の位置の図形を拡大するには

しかしながら，私たちが図形を拡大したり縮小したりしたいというときには，**図形の位置をそのままにして大きさだけを変えたい**というのが普通であろう。たとえば上の図の灰色の円だけを横縦それぞれ 2 倍と 1.5 倍に拡大したい。どうしたらよいだろうか？

原点を中心にしてしか大きさをスケーリングできないのであれば，**原点を移動させてから必要な図形を描けばよい。**これが簡単な解決法だ。いま, (x, y) の位置に中心をもつ図形に対して，横に a 倍，縦に b 倍のスケーリングを施したいとしよう。その手順は次のようにする。

1. `x y translate` で原点を (x, y) の位置に移動させる。
2. `a b scale` で，その後の描画がスケーリングして行われるようにする。
3. 原点 $(0, 0)$ を中心に，目的の図形を描く。

■楕円を描く 実際にやってみよう。次のソースは最初原点に中心を持つ半径 20 の円を描き，そのあと同じ情報を使って $(80, 40)$ の位置を中心とする長径 40, 短径 30 の楕円を描く。

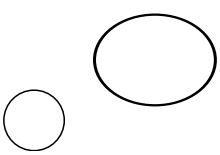


図 4.7

```
0   0  20 0 360 arc stroke
80 40 translate
2 1.5 scale
0   0  20 0 360 arc stroke
```

座標変換処理の後始末

上記の手法を使えば，思い通りの場所の図形をスケーリング
できることになる。しかし，この後の処理が必要なことが多い。つまり，**この後に描かれるものはすべて移動とスケーリングの影響を受けてしまう**のである。そこでその効果を打ち消してもとにもどすという処理が必要になることがある。それには次のようにすればよい。

1. **【後始末 1】** `1/a 1/b scale` で横に $1/a$ 倍，縦に $1/b$ 倍のスケーリングを施す。これでスケーリングの効果は横縦それぞれ 1 倍と 1 倍になり，消える。
2. **【後始末 2】** `-x -y translate` で `x y` でずらされていた原点の位置を $(0, 0)$ にもどす。

これらの後始末の部分も含めたソースをつぎに示す。2 と 1.5 の逆数を 0.5 や 0.666677 とせずに，`1 2 div`，`1 1.5 div` としていること，また 80, 40 から $-80, -40$ を得るのに `neg` を使っていることに注意しよう。

```
0   0  20 0 360 arc stroke
80 40 translate
2 1.5 scale
0   0  20 0 360 arc stroke
1 2 div 1 1.5 div scale   % 後始末 1
80 neg 40 neg translate   % 後始末 2
```

問題 4–3 上記のソースの最後の 2 行を省いて，その上の 3 行を繰り返していくと，どのような描画の結果が得られるか。結果を予測した後で実際にソースを書いてみて確かめなさい。

問題 4–4 $(20, 100)$, $(30, 50)$, $(100, 90)$ をそれぞれ中心とする楕円が得られるように Postscript のソースを書きなさい。楕円のサイズや形は自由とする。

4.6.3 座標の回転

回転のオペレータ

図形を回転させたいこともよくある。そのためには rotate オペレータが用意されており，使い方は次のようにする。

角度 rotate

角度は通常の度を使い，反時計回りに正としておく。その効果を見てみよう。

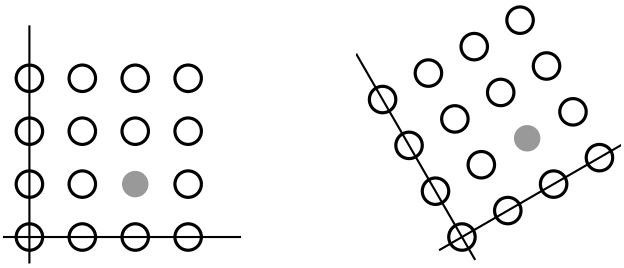


図 4.8 左図の図形を描く前に 30 rotate を入れると右図になる

図 4.8 の右側の図は，左側の図の座標軸と円を描く前に次の行を挿入して得られたものである。

```
30 rotate
```

このように，原点を中心として，**すべての図形**が 30 度回転した位置に描かれている。

任意の点を中心にして回転させるには

作画していて図形を回転させたい場合，上のように原点を中心に回ってもらっては困ることが多い。たとえば図 4.9 のような図形を書こうとしたら，楕円を任意の場所に置いて，その位置で回転させるテクニックが必要だ。この場合にも，translate オペレータで座標系をずらしてから原点を中心に描画すればよい。

下のソースは 1 つの楕円を 45 度斜めに描くためのものである。

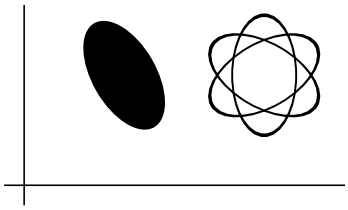


図 4.9 楕円を使った簡単なデザイン

```
50 50 translate              % (50,50) に原点をずらす
45 rotate                    % 座標系を 45 度回す
0.9 2.3 scale                % この 2 行で楕円を描く
0 0 20 0 360 arc stroke      %
1 0.9 div 1 2.3 div scale    % これ以降は後始末
45 neg rotate
50 neg 50 neg translate
```

4.6.4 正多角形を描く

座標の回転を利用すると，簡単に正多角形を描くことができる。正六角形を例に，図 4.10 を見ながら説明しよう。

1. `translate` で，描画の中心として原点を (x_0, y_0) に移動させる。
2. `moveto` でペン先を $(0, r)$ に置く。実際には `translate` が効いているので，図の六角形の右端になる。
3. `60 rotate` で座標系を 60 度回転させる。これによって x 軸は右上に回転して，x' の向きになる。
4. `lineto` で $(r, 0)$ まで線を引く。これによって辺が 1 つ引かれる。
5. 3，4 を 5 回繰り返す。6 回でないことに注意！
6. `closepath` でパスを閉じてから `stroke` で線を引く。

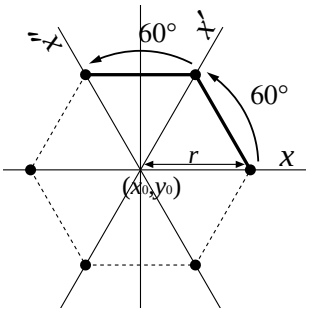


図 4.10 正六角形を描く手順の説明：辺を 2 本引いたところ

`repeat` で処理を繰り返す

上の説明には，同じ処理を 5 回繰り返すというところがある。このような単純な繰り返しを行わせるために Postscript では `repeat` というオペレータが用意されている。5 回繰り返すのであれば次のような形で使う。

```
5 {繰り返したい処理} repeat
```

処理の部分が長ければ，自由に改行を入れてよい。

こんな形になる

問題 4–5 次の処理

```
-14 -5 moveto 28 0 rlineto 0 10 rlineto -28 0 rlineto closepath stroke
```

でできる細長い長方形を，$(30, 30)$ を中心にして 45 度回した形を描くソースを書きなさい。

問題 4–6 `repeat` オペレータを利用して上の細長い長方形を 45 度ずつ回しては描く作業を 4 回繰り返して重ね合わせなさい。欄外の図のようになる。

問題 4–7 正五角形を描くソースを書きなさい。その後，回転角を 144 度にしてみなさい。どんな図形が現れるだろうか。

ひとまずおしまい

この辺でとりあえず時間切れになると思うので， Postscript の解説もここまでとしよう。Postscript はほとんどの印刷物の作成に使える強力なグラフィックス言語なので，もっともっと奥は深く，範囲も広い。まだ触れていないが大事な項目をいくつか挙げておくと，まず文字列の印刷の技法，また本格的なプログラミングへの発展といったところだろう。これらも参考資料や小波のウェブサイトに詳しく書かれているので，興味のある人は挑戦してみることをお勧めしたい。