第 2 章 電子回路の解析

[はじめに]
電子回路は、いうまでもなく現在の文明になくてはならないものである．電子回路は、これを構成する素子の特性が比較的単純で、きちんと解析することができ、また逆に所望の動作をするように机上で設計することができる．とはいうものの人手では、小規模で単純な回路であっても、回路に応じて解析のための工夫が必要であったり、計算そのものが厄介だったりする．そこで行列やベクトルを用いて、特定の回路によらず回路(回路網)を一般的に表現し、これを解析する手法を学ぶことにする．ここでは、基礎を身につけることを主眼として、抵抗と電圧源のみからなる直流回路から始めて、発展としてトランジスタなどの能動素子にも言及し、これを使った増幅回路の動作を見ていく．ここで学ぶ手法によって、電子回路に限らず、より広く複雑な事象の解析、そのツールとして、コンピュータ・シミュレーションの手法を獲得する糸口となることを期待する．

2.1 簡単な電子回路

乾電池に豆電球をつないだことはないだろうか？ このとき乾電池 1 つで豆電球が光って、だいたい 100mA～200mA くらいの電流が流れる．ここでは簡単に電池を 1.5V とし、豆電球を抵抗に置き換えて 10Ω としよう．そうすると回路は図 2.1.1 のようになる．

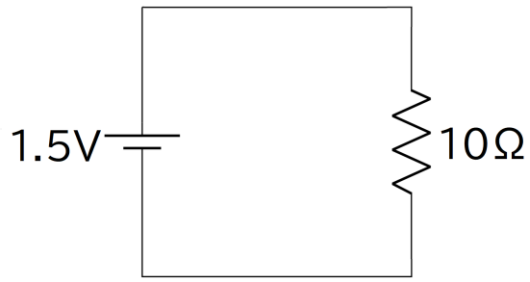

図 2.1.1 電池と抵抗

ここで、電池の両端＝抵抗の両端の電圧 V と抵抗 R と流れる電流 I の関係は、電圧の単位を V(ボルト)、抵抗は Ω（オーム)、電流は A(アンペア)として、

$$V = RI \tag{2.1.1}$$

となっている．これをオームの法則といい、中学校までに習うはずだ．図 2.1.1 に当てはめると、

$$1.5 = 10 \times 0.15 \tag{2.1.2}$$

という関係になっている. もし今この回路に流れる電流がわからなかったとしよう. 電圧と抵抗がわかっているならば、オームの法則の式を変形して、

$$I = \frac{V}{R} = \frac{1.5}{10} = 0.15\mathrm{A} \tag{2.1.3}$$

簡単に電流をもとめることができる. 回路が決まれば抵抗が決まり、このような簡単な回路であれば電圧も決まるから、回路の解析というのは、しばしば回路を流れる電流を求めるだけの作業となる.

ところで電位という言葉は覚えているだろうか？ 電荷に係る位置エネルギー、静電ポテンシャルという定義が与えられるが、具体例で考えたほうがわかりやすい. 例えば図 2.1.1 の場合に乾電池のプラスとマイナスの間の電圧は 1.5V だが、もしここで電池のマイナス極を回路の基準としてその電位を 0V としたとき、電池のプラス極の電位は 1.5V である. 逆に電位の差、電位差が電圧である. そしてその電位差を生み出しているものが起電力である.

さてそれでは 10Ω の抵抗の部分が図 2.1.2 に示すように、2 本の抵抗が直列につながれたものだったら、どうなるだろうか？それは簡単に、両方の抵抗の値の和を新たな抵抗として計算すればよい. すなわち、

$$R = R_1 + R_2 \tag{2.1.4}$$

例えば、10Ω と 20Ω ならば、$10 + 20 = 30\Omega$である. それを R として、

$$I = \frac{V}{R} = \frac{1.5}{30} = 0.05\mathrm{A} \tag{2.1.5}$$

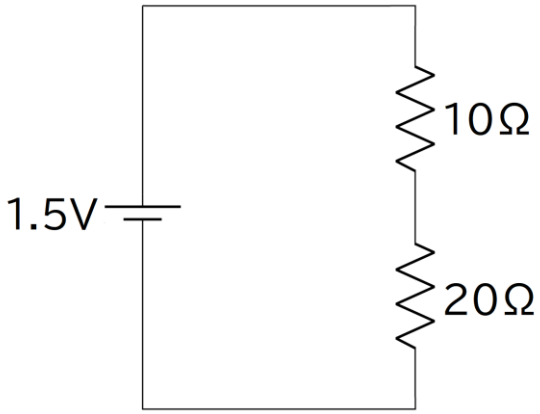

図 2.1.2 電池と抵抗 2 本直列

いっぽう同じく 10Ω の抵抗の部分が図 2.1.3 に示すように、2 本の抵抗が並列につながれたものだったら、どうなるだろうか？

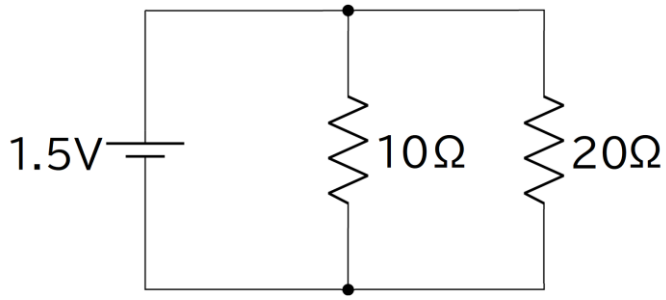

図 2.1.3 電池と抵抗 2 本並列

抵抗が並列に接続された場合に合成抵抗を求める公式があるが、あえてそれを使わずに求めてみよう．図 2.1.3 の場合だが、10Ω と 20Ω の 2 本の抵抗のいずれも両端の電圧は 1.5V である．そこを流れる電流を考えるとオームの法則から、10Ω の抵抗に流れる電流は、

$$\frac{1.5}{10} = 0.15\mathrm{A} \tag{2.1.6}$$

いっぽう、20Ω の抵抗に流れる電流は、

$$\frac{1.5}{20} = 0.075\mathrm{A} \tag{2.1.7}$$

だから、電池に流れる電流は、

$$0.15 + 0.075 = 0.225\mathrm{A} \tag{2.1.8}$$

合成抵抗は、まさにこの 2 本分と等価なものであって、両方の電流を合計した電流＝電池に流れる電流を流す 1 本の抵抗だから、

$$R = \frac{V}{I} = \frac{1.5}{0.225} \fallingdotseq 6.667\Omega \tag{2.1.9}$$

さてここまでを振り返るならば、合成抵抗というのは、その 2 本の抵抗を合わせて 1 本に見立てた抵抗のことであり、その 2 本の抵抗 R_1 と R_2 には、それぞれ電流 I_1 と I_2 が流れるならば、その電流の和 $I = I_1 + I_2$ が 2 本の抵抗を 1 本に見立て抵抗 R に流れる電流である. 電圧を V として、これをそのまま式にすると、

$$I = I_1 + I_2 = \frac{V}{R_1} + \frac{V}{R_2} = V\left(\frac{1}{R_1} + \frac{1}{R_2}\right) = \frac{V}{R} \tag{2.1.10}$$

だから、

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} \tag{2.1.11}$$

$$\therefore\ R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}} \tag{2.1.12}$$

まさに並列に接続した抵抗の合成抵抗を求める公式が得られた. 当然、合成抵抗をこの公式で求めて、電流を計算しても 2 本の抵抗を流れる電流の和は同じ値が得られる.

さてここで、電池に流れる電流は 2 本の抵抗を合わせた $I = I_1 + I_2$ だが、ではその電池から流れ出す電流と抵抗に流れ込む電流の全部を合わせるとどうなるだろうか？流れ出すのをプラス(＋)、流れ込むのをマイナス(－)として考えると、全部合わせれば 0 になる、というのも当然だろう. なにしろ、電池一つに抵抗 2 本で、これ以外に外とつながってはおらず、電流のやり取りはないのだから. さて、この関係は「キルヒホッフの法則」と呼ばれ、先のオームの法則と並んで、今回のシミュレーションに必要な法則である. 蛇足ながら、図 2.1.1 の場合も、電池と抵抗の接続点で、電池から流れ出る電流と抵抗に流れ込む電流は、流れ出すのを＋、流

れ込むのを－とすれば、大きさは同じなのだから合計は 0 になる.

ではここで、もう少し複雑な例を見てみよう.

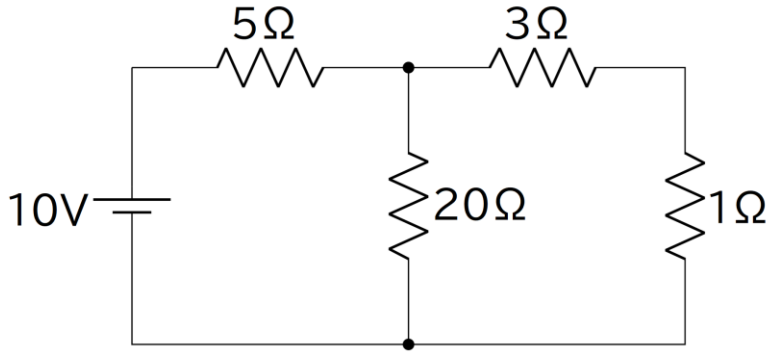

図 2.1.4 簡単な回路

図 2.1.1 に比べて抵抗が 3 本増えただけで電子回路としてはごく簡単なものだ. しかし、この回路を流れる電流を求めようとすると、結構面倒な計算をしなければならない. とくに 5Ω、3Ω、20Ω の3本の抵抗がつながった場所の電位がすぐにはわからない. このような場合には、回路の合成抵抗を求めて、オームの法則が使えるようにして、回路に流れる電流を求め、電流がわかったら、今度は逆に各抵抗の両端の電圧すなわち電圧降下を、またオームの法則を使って求める、ということを繰り返していくことになる. ここで実際にやってみよう.

① 3Ωの抵抗と1Ωの抵抗が直列となっているから、この 2 つで $3+1=4\Omega$
② この 2 つを合成した 4Ω と 20Ω の抵抗が並列になっているから、この合成抵抗は

$$\frac{1}{\frac{1}{4}+\frac{1}{20}}=\frac{10}{3}\Omega \tag{2.1.13}$$

③ 3Ω、1Ω、20Ω の 3 本の合成抵抗がわかったから、これとさらに、5Ω の抵抗が直列になっているとみて、これら合わせた抵抗 4 本の合成抵抗は、

$$\frac{10}{3}+5=\frac{25}{3}\Omega \tag{2.1.14}$$

④ 合成抵抗 $\frac{25}{3}\Omega$ が $10\mathrm{V}$ の電源につながっているとして、電流は

$$10 \div \left(\frac{25}{3}\right) = \frac{6}{5} = 1.2\mathrm{A} \tag{2.1.15}$$

⑤ 5Ω の抵抗に流れる電流は合成抵抗に流れる電流そのものだから $1.2\mathrm{A}$、したがって、この抵抗での電圧降下は

$$1.2 \times 5 = 6\mathrm{V} \tag{2.1.16}$$

よって、3本の抵抗の接続点の電位は、

$$10 - 6 = 4\ \mathrm{V} \tag{2.1.17}$$

⑥ 20Ω の抵抗に流れる電流は、

$$4 \div 20 = 0.2\ \mathrm{A} \tag{2.1.18}$$

⑦ 3Ω と 1Ω に流れる電流は、

$$4 \div 4 = 1\ \mathrm{A} \tag{2.1.19}$$

⑧ 1Ω の抵抗に $1\ \mathrm{A}$ の電流が流れると電圧降下は、

$$1 \times 1 = 1\mathrm{V} \tag{2.1.20}$$

よって、3Ω と 1Ω の接続点の電位は、$1\mathrm{V}$

これで各抵抗に流れる電流も、各抵抗の接続点の電位もわかったので整理すると、

1Ω ： $1\ \mathrm{A}$

3Ω ： $1\ \mathrm{A}$

20Ω ： $0.2\ \mathrm{A}$

5Ω ： $1.2\ \mathrm{A}$

3Ω と 1Ω の抵抗の接続点の電位：$1\mathrm{V}$

5Ω、3Ω、20Ω の3本の抵抗の接続点の電位：$4\mathrm{V}$

以上となる．電源１つに抵抗４本で、回路としては簡単なものだが、計算手順を考えて解いていくのは、結構骨の折れることである．

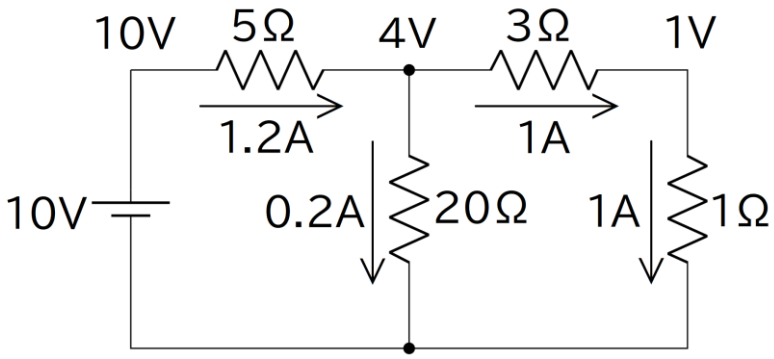

図 2.1.5 簡単な回路

2.2 ブランチとノード

図 2.1.4 の回路を解析する手順を振り返るならば、回路を構成する電源や抵抗に流れる電流や回路のあちらこちらの電位を求めるのに、パズルを解くように、解けるところを探しては解いていき、すべてわかるまで繰り返した．解くにあたって使った法則は、オームの法則とキルヒホッフの法則の 2 つの法則だけだった．ならば逆に、オームの法則とキルヒホッフの法則がどこにどう当てはまるのかを明確にしつつ解析の手順を組み替えることで、パズル的な要素を排して、回路構成によらない数式化ができるのではないだろうか？

2.2.1 オームの法則

オームの法則は、式 2.1.1 に示すように電圧と電流と抵抗の関係を示すが、この対象は言うまでもなく抵抗である．つまり回路がどんなに複雑であっても、その中の一つの抵抗に着目すると、その両端の電圧とその抵抗に流れる電流がその抵抗の値との関係で表される．すなわち、回路を抵抗を基本とした単位に分割するならば、そのおのおのにおいて、オームの法則が成り立つということである．たとえば図 2.1.5 で、これを構成する各抵抗について、両端の電圧とこれを流れる電流は、

1Ω ：1 V、1 A
3Ω ：3 V、1 A
20Ω ：4 V、0.2 A
5Ω ：6 V、1.2 A

いずれも、式 2.1.1 の $V = RI$ の関係が成り立っている．

2.2.2 キルヒホッフの法則

キルヒホッフの法則は、すでに並列の合成抵抗を求める際にも使ったが、これは回路を構成する部品(ここでは抵抗と電池)が、どう接続されているかとの関係であらわされ、その接続点で各部品の電流の出入りの合計が 0 ということである．逆に言うならば、これは回路がどう接続されているかそのものを、符号をつけて表現したものという見方をすることができる．たとえば図 2.1.5 に示す 5Ω、3Ω、20Ω の 3 本の抵抗について、3 本の抵抗の接続点から流れだす電流をプラス(＋)、流れ込む電流をマイナス(－)として、各抵抗に流れる電流 i の値は、

$$\begin{cases} i_{5\Omega} = -1.2 \\ i_{3\Omega} = 1.0 \\ i_{20\Omega} = 0.2 \end{cases} \tag{2.2.1}$$

ここで、キルヒホッフの法則は、

$$i_{5\Omega} + i_{3\Omega} + i_{20\Omega} = -1.2 + 1 + 0.2 = 0 \tag{2.2.2}$$

となって、各接続点で接続されている部品を流れる電流の和が 0 という関係、すなわち、

$$\sum i = 0 \tag{2.2.3}$$

である．これは流れだす電流をプラス(＋)、流れ込む電流をマイナス(－)とすれば 3Ω と 1Ω の接続点でも、また、電池と 5Ω の接続点でも成り立つ．

2.2.3 ブランチとノードの定義

ここまで見てきたように、回路を構成する部品においてオームの法則を、また、その接続点においてキルヒホッフの法則を適用すればよい、ということが明らかである．そこで、回路を構成する部品とその接続点を、それぞれ、ブランチとノードと呼ぶことにする．そして電池も別に扱うのではなく、ブランチに含めて抵抗と一括して扱えるようにするのが良いだろう．
すなわち、回路を基本構成要素から成る最小単位に分割し、この分割した最小単位をブランチとし、そのブランチどうしの接続点をノードとする．そして、この組み合わせで回路全体を表現するのである．ここでブランチは、抵抗となったり、電池となったりできるように、その両方の要素を含む形で定義する．

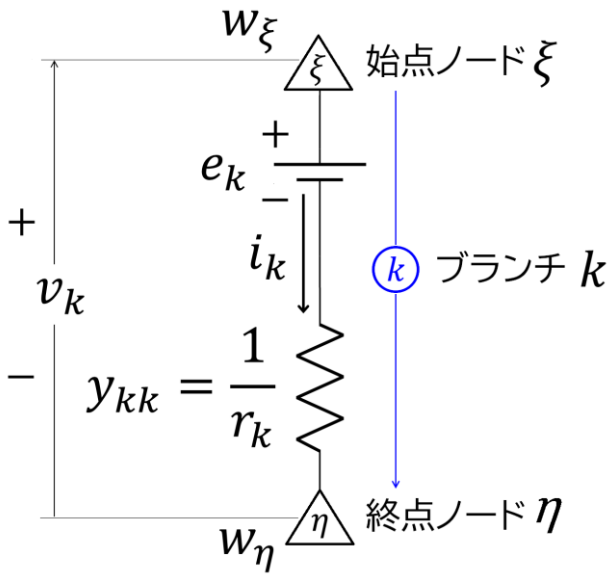

図 2.2.1　ブランチとノード

回路の中で個々のブランチ、ノードを区別するために番号を付けることにするが、回路全体ではN個のノードとB個のブランチが含まれるものとし、各ノードにはそれぞれ0,1～Nの番号をつけ、各ブランチにはそれぞれ1～Bの番号をつける．ただし回路を構成するノードのうち基準となるもの、通常はグランドないし接地を 0 番とし、回路のどこか 1 点がノード 0 であるものとする．図2.2.1はブランチの番号を k として、その始点ノード ξ、終点ノード η としている．

ブランチは電圧源と抵抗を含むが、両者は図2.2.1に示すように直列につながっているものとする．電圧源は流れる電流によらず一定の電圧 e_k を発生する理想的なものとし、抵抗は抵抗値の逆数であるアドミタンスないしはコンダクタンス $y_{kk} = \frac{1}{r_k}$ つまり電流の流れやすさで、その大きさを表現することにする．ブランチを流れる電流は i_k、ブランチの両端の電圧は V_k、始点ノード ξ、終点ノード η の電位をそれぞれ W_ξ、W_ηとする．また電圧と電流の向きは回路解析の際に統一しておく必要があるので次のように決める．すなわち図 2.2.1 に示すように、電流は始点ノード ξ から終点ノード η へと流れる向きをプラス(+)、逆方向をマイナス(−)とし、電圧は始点ノード ξ が終点ノード η よりも高いときにプラス(+)、逆をマイナス(−)とする．ブランチの構成要素の電圧源についても始点ノード側が高いときにプラス(+)、逆をマイナス(−)とする．

[補足 2.2.1]
図 2.2.1 ではブランチを構成する要素として電圧源と抵抗だけとした．しかしトランジスタなどのいわゆる能動素子が回路に含まれる場合には、この図で示した要素のほかに、電圧源と抵抗の直列の経路と並列に電圧制御電流源を加えることが必要となる．関連してアドミタンスの添字は Y_{kk} とブランチ番号 k を 2 つ重ねているが、この理由については補足 2.4.1 を参照のこと．

ではここで、ブランチのデータを入力するプログラムを作っておこう．

リスト 2.2.1：ブランチの python のコードイメージ

```
# Number of Branches
b = int(input('Number of branches '))
print('branch# =', b)

def set_branch(b):
    Y = [[0.0 for j in range(b)] for i in range(b)]
    E = [0.0 for i in range(b)]
    print('Input spec of each branch.')
    for k in range(b):
        print('Branch#' + str(k+1))
        e = float(input('electro motive force(V)   '))
        r = float(input('electrical resistance(ohm) '))
        y = 1/r
        Y[k][k] = y
        E[k] = e
    return Y, E

Y, E = set_branch(b)

# display the spec of each branch
for i in range(b):
    print('branch#{:d}：electro motive force = {:6.3f}V admittance = {:6.3f}S' ¥
          .format(i+1, E[i], Y[i][i]))
```

このプログラムは回路を構成するブランチのデータを入力して、それを表示するだけで何もおもしろいところはない．しかしプログラム内でデータをリストととして持つことなどを確認しておきたい．そして後で使えるように関数として用意しておくことにする．プログラムでは、はじめにブランチ数 b を入力する．続いて関数 set_branch()を定義する．この関数ではまず、アドミタンス $\boldsymbol{Y}$ と電圧源 $\boldsymbol{E}$ の空のリストを用意するが、アドミタンス $\boldsymbol{Y}$ は b×b の大きさのすべて値 0.0 の行列として初期化する．この理由は補足 2.2.1 および補足 2.4.1 を参照のこと．続いて、ブランチ 1 つずつについて for 文で回しながら、その電圧源の値と抵抗値を入力させる．そして、これらをリストの所定の要素に書き込んでいく．このとき抵抗値は逆数

1/r をとってアドミタンスに変換する．それをすべてのブランチについて行ったら for ループをぬけて、アドミタンス $\boldsymbol{Y}$ と電圧源 E を返り値として関数を終わる．次の１行、

```
Y, E = set_branch(b)
```

が関数の実行である．これを実行すると、アドミタンス $\boldsymbol{Y}$ と電圧源 $\boldsymbol{E}$ が得られるから、あとは結果を for 文でループしながら各ブランチについて表示して終わる．この中で注意すべきこととして、本稿の説明の中ではブランチは 1～B 番だが、プログラムでは、それぞれ 0 ～ b-1 番となっている．これは、python のリストのインデクスが 0 から始まり、range(b)でも最後の番号は b - 1 となることに対応しているもので、番号が１つずれることに注意が必要．このつじつま合わせのために、ブランチの番号を表示する際には、ループ変数 i ではなく i+1 としている．

さてそれでは、図 2.1.1 に示した電池１つ、抵抗１本の回路の場合にはどうなるだろうか？この場合、図 2.2.2 に示すように、ブランチは電池①と抵抗②の 2 つである．そして電池のプラス・マイナスそれぞれに、抵抗の両端がつながれているから接続点であるノードは電池のマイナス極をノード 0 として、0 と 1 の 2 つ（0 番を数えなければ 1 つ）、ということになる．図 2.1.1 では乾電池の両端の電圧をその公称値の $1.5\mathrm{V}$ としていたが、起電力と内部抵抗に分けて表すことにする．実際の乾電池の特性は時間経過とともに変化し簡単ではないが、本題からそれるので、ここでは簡単に乾電池の起電力は $1.65\mathrm{V}$、内部抵抗は 1Ω ということにしておこう．整理すると、

回路を構成するノード： 2 つ
回路を構成するブランチ： 2 つ
ブランチ 1 は電池：起電力 $1.65\mathrm{V}$ 内部抵抗 1Ω
→ 起点ノード 1、終点ノード 0、$e_1 = 1.65$、$y_{11} = 1.0(r_1 = 1\Omega)$
ブランチ 2 は抵抗：10Ω
→ 起点ノード 1、終点ノード 0、$e_2 = 0.00$、$y_{22} = 0.1(r_2 = 10\Omega)$
電圧の単位は V(ボルト)、アドミタンスの単位は S(ジーメンス、昔は ℧ モーが使われた)

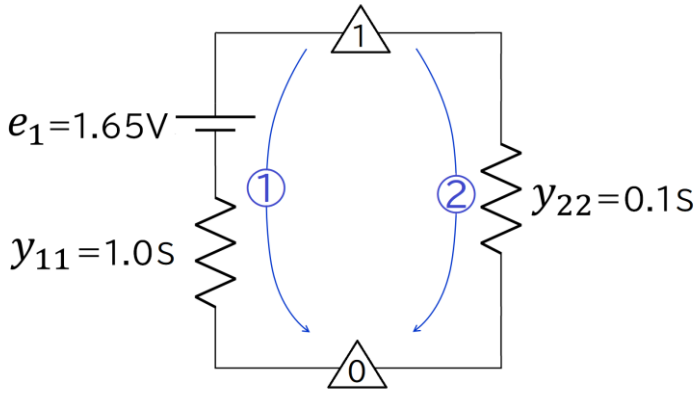

図 2.2.2　電池と抵抗

さて図 2.2.2 に示した回路で、ブランチ 1 に着目する．ブランチ 1 の両端の電圧を v_1、これを流れる電流を i_1 とすると、ブランチ 1 の内部抵抗においてオームの法則から、

$$r_1 i_1 = \frac{i_1}{y_{11}} \tag{2.2.4}$$

の電圧が発生する.

電圧源の電圧と内部抵抗の両端の電圧を加えたものがブランチ 1 の両端の電圧となるから、

$$v_1 = e_1 + \frac{i_1}{y_{11}} \tag{2.2.5}$$

$$y_{11} v_1 = y_{11} e_1 + i_1 \tag{2.2.6}$$

$$\therefore i_1 = y_{11} v_1 - y_{11} e_1 = 1.0 \times v_1 - 1.0 \times 1.65 = v_1 - 1.65 \tag{2.2.7}$$

次はブランチ 2 に着目する．ブランチ 2 の両端の電圧を v_2、これを流れる電流を i_2 とすれば、抵抗だけからなるブランチなのでオームの法則から、

$$v_2 = r_2 i_2 = \frac{i_2}{y_{22}} \tag{2.2.8}$$

$$\therefore i_2 = y_{22} v_2 = 0.1 \times v_2 = 0.1 v_2 \tag{2.2.9}$$

ここまで個々のブランチについてオームの法則から関係式を導き出した．式 2.2.7 と式 2.2.9 がそれである．
ここからは、各ノードに着目していこう．といっても着目すべきノードはただ一つだけだが．
ともかく、キルヒホッフの法則からノード 1 において、

$$i_1 + i_2 = 0 \tag{2.2.10}$$

あとは電圧と電位の定義の通りに、ノード 1 の電位を w_1 とおけば、

$$v_1 = v_2 = w_1 \tag{2.2.11}$$

ここからは得られた関係式を解いていく．
式 2.2.7、式 2.2.9 を式 2.2.10 に代入して、

$$(v_1 - 1.65) + 0.1v_2 = 0 \tag{2.2.12}$$

式 2.2.11 を式 2.2.12 に代入して、

$$(w_1 - 1.65) + 0.1w_1 = 0$$

$$1.1w_1 = 1.65$$

$$\therefore w_1 = \frac{1.65}{1.1} = 1.5 \tag{2.2.13}$$

式 2.2.13 で得られた結果を式 2.2.11、式 2.2.7、式 2.2.9 に順次代入して整理すると、

$$\begin{cases} v_1 = v_2 = w_1 = 1.5 \\ i_1 = -0.15 \\ i_2 = 0.15 \end{cases} \tag{2.2.14}$$

いささか回りくどい解き方をしたが結果が得られた．

注意：ブランチ 1 に流れる電流はその始点ノード 1 に向かって流れ出す方向なのでマイナス（－）の符号がつく．図 2.1.4 で定義した電流の向き、始点ノードから終点ノードへ流れるのとは逆向きなので注意がいる．このほか電圧源の向き、ブランチ両端の電圧の向きにも注意する必要がある．

図 2.1.1 に示した電池と抵抗 1 つずつの回路について、図 2.2.2 のようにブランチとノードをあてはめて電圧や電流がどうなっているのかを解いた．では図 2.1.4 ないし図 2.1.5 に示す電源 1 つと抵抗 4 本の回路ならばどうだろうか？ここまで見てきたようにややこしくなる一方だから、ここではブランチとノードを当てはめたらどうなるかを図 2.2.3 に示すにとどめることにする．

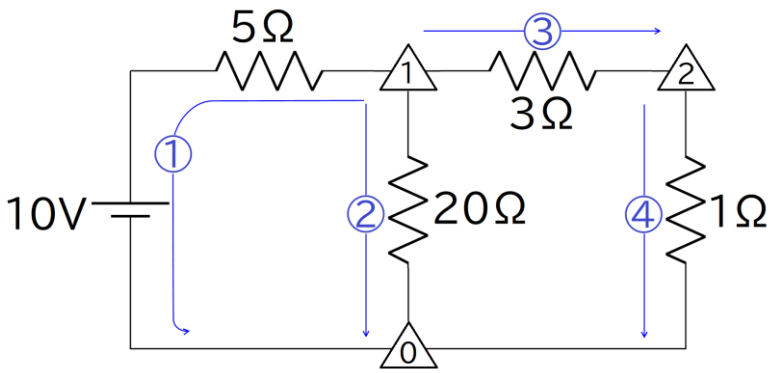

図 2.2.3　簡単な回路

図 2.2.3 に示すように、この場合にはノードは 0 番を数えなければ 2 つ、ブランチは 4 つとなる．ただしこの回路の場合には、電源が何なのかはわからないから、節 2.1 でこの回路を解いた場合にならって、電源は理想的な電圧源であるものとする．そしてブランチ 1 は 5Ω の抵抗と 10V の電源を含むようにする．

2.3 回路の接続関係をあらわす方法

前節では、回路を構成する要素をブランチ、その接続点をノード、というかたちで定義した. そして、オームの法則が支配するブランチにおける電圧と電流の関係、それから、キルヒホッフの法則が支配するノードにおける電流の合分流の関係を分けて考えるアプローチについて、図 2.2.2 の乾電池と抵抗 1 本だけの簡単な回路を具体例として説明した. このような単純な回路では、問題をややこしくしているだけに思われるかもしれない. しかし複雑な回路を統一的な方法で解く際には、このやり方が威力を発揮する. しかしそのためには、回路ごとに異なるブランチやノードそしてその接続関係をあらわす手段が必要となるのである.

2.3.1 回路の接続関係をあらわす行列

図 2.2.1 に示すように、ブランチ k の始点ノード ξ、終点ノード η のとき、次の a を定義する.

$$\begin{cases} a_{k\xi} = 1 \quad (\xi \geqq 1) \\ a_{k\eta} = -1 \quad (\eta \geqq 1) \\ a_{ki} = 0 \ \left(i \neq \xi, \eta,\ i \geqq 1\right) \end{cases} \tag{2.3.1}$$

着目するノードが、ブランチ k の始点ノードである場合には $a = 1$、ブランチ k の終点ノードである場合には $a = -1$ であり、そのいずれでもない、すなわち、つながっていない場合には $a = 0$ である. ただし始点ノードあるいは終点ノードが 0 で接地ないしグランドの場合には a は定義しない.

つぎに回路内の任意のブランチ k に着目して、このブランチ k に関係する a のノードごとの値をノード 1 からノード N まで順番に並べてみよう. そうして作られたベクトルを A_k とすると次のようになる.

$$A_k = (a_{k1} \quad \cdots \quad a_{k\xi} \quad \cdots \quad a_{k\eta} \quad \cdots \quad a_{kN}) = (0 \quad \cdots \quad 1 \quad \cdots \quad -1 \quad \cdots \quad 0) \tag{2.3.2}$$

このベクトル A_k は式 2.3.1 の定義により、その要素はブランチ k の始点ノード ξ に対応する要素で $a_{k\xi} = 1$、終点ノード η に対応する要素で $a_{k\eta} = -1$ となり、その他の要素はすべて 0 である. ただし、始点ノード ξ が 0 番でグランドないしは接地の場合には、$a_{k\xi}$ は存在せず、終点ノード η が 0 番でグランドないしは接地の場合には、$a_{k\eta}$ は存在しないことに注意が必要である.

さてそれでは、式2.3.2で示すベクトルを、回路を構成するすべてのブランチ1～Bについて、縦に順番に並べたらどうなるだろうか？それを次に行列 $\boldsymbol{A}$ として示す.

$$\boldsymbol{A}=\begin{pmatrix} a_{11} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1N} \\ \vdots & \ddots & & & & & \vdots \\ \vdots & & & & & & \vdots \\ a_{k1} & \cdots & a_{k\xi} & \cdots & a_{k\eta} & \cdots & a_{kN} \\ \vdots & & & & & & \vdots \\ \vdots & & & & & \ddots & \vdots \\ a_{B1} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{BN} \end{pmatrix} \tag{2.3.3}$$

行列 $\boldsymbol{A}$ は、各行が各ブランチに対応するから、その行数はBとなる. また、各列は各ノードに対応するから、列数はNとなる. だから行列 $\boldsymbol{A}$ は、B×Nの大きさである. そして行列 $\boldsymbol{A}$ の各行が各ブランチのノードとの接続関係を示し、それをすべてのブランチ分集めたものであるから、行列 $\boldsymbol{A}$ は、回路に含まれるすべてのブランチとすべてのノードの接続関係、すなわち回路全体の接続関係をあらわす行列なのである.

ここで行列 $\boldsymbol{A}$ について整理するために、pythonのプログラムで行列 $\boldsymbol{A}$ を作ってみよう.

リスト2.3.1：行列 $\boldsymbol{A}$ を作るpythonのコードイメージ

```
# Number of Nodes and Branches
n = int(input('Number of nodes    '))
b = int(input('Number of branches '))
print('node# =', n, 'branch# =', b)

def set_A(n, b):
    A = [[0 for j in range(n)] for i in range(b)]
    print('Input starting node and ending node of each branch.')
    for k in range(b):
        print('Branch#' + str(k+1))
        f1 = int(input('the starting node# '))
        if f1 > 0 : # ブランチkの始点
            A[k][f1 - 1] = 1
        t1 = int(input('the ending node#   '))
        if t1 > 0 : # ブランチkの終点
            A[k][t1 - 1] = -1
    return A

print('Matrix A ¥n', set_A(n, b))
```

リスト2.3.1のプログラムは、行列 $\boldsymbol{A}$ をつくるだけのプログラムだが、後で使えるように関数として用意しておくことにする. はじめにノード数nとブランチ数bを入力する. それで行列 $\boldsymbol{A}$ の大きさが決まる. そしてこれらを引数として行列 $\boldsymbol{A}$ を返す関数を定義する. 行

列 $\boldsymbol{A}$ は内包表記によって、大きさ b×n で値がすべて 0 の行列、ここでは 2 重のリスト、として初期化する．そのあとで、今度はブランチ 1 つずつについて for 文で回しながら、その始点ノードと終点ノードの番号を入力させて、その番号に対応する列の要素を = 1 ないしは = -1 で上書する．それをすべてのブランチについて行ったら for ループをぬけて、返り値 $\boldsymbol{A}$ を伴って終了する．

```
print('Matrix A ¥n', set_A(n, b))
```

では関数を呼ぶとともに直ちにその返り値を表示する．ブランチとノードの番号がリスト 2.2.1 同様に 1 つずれることに注意が必要．このつじつま合わせのために、for ループの中で始点、終点ノードの番号は入力された f1、t1 そのものではなく、f1 - 1 および t1 - 1 としている．このプログラムを起動して、図 2.2.2 の回路や、図 2.2.3 の回路を入力して、行列 $\boldsymbol{A}$ がどうなるかを見てほしい．

それから、ここで作った関数 set_A(n, b)は、この後で何度も使うことになる．そこで先に作った関数 set_branch(b)と併せて、別のファイル ELCA.py を作って置いておこう．ちなみにファイル名は好きにつけてもらってよいが、ここでは、ELectric Circuit Analisis の略で ELCA にした．

リスト 2.3.2：ELCA のコードイメージ

```
def set_A(n, b):
    A = [[0 for j in range(n)] for i in range(b)]
    print('Input starting node and ending node of each branch.')
    for k in range(b):
        print('Branch#' + str(k+1))
        f1 = int(input('the starting node# '))
        if f1 > 0 : # ブランチ k の始点
            A[k][f1 - 1] = 1
        t1 = int(input('the ending node#   '))
        if t1 > 0 : # ブランチ k の終点
            A[k][t1 - 1] = -1
    return A

def set_branch(b):
    Y = [[0.0 for j in range(b)] for i in range(b)]
    E = [0.0 for i in range(b)]
    print('Input spec of each branch.')
    for k in range(b):
        print('Branch#' + str(k+1))
        e = float(input('electro motive force(V)   '))
        r = float(input('electrical resistance(ohm) '))
        y = 1/r
        Y[k][k] = y
        E[k] = e
    return Y, E
```

リスト 2.3.3：ELCA を使って行列 $\boldsymbol{A}$ とブランチを入力する python のコードイメージ

```
from ELCA import *

# Number of Nodes and Branches
n = int(input('Number of nodes    '))
b = int(input('Number of branches '))
print('node# =', n, 'branch# =', b)

print('Matrix A ¥n', set_A(n, b))

Y, E = set_branch(b)
# display the spec of each branch
for i in range(b):
    print('branch#{:d}：electro motive force = {:6.3f}V admittance = {:6.3f}S' ¥
          .format(i+1, E[i], Y[i][i]))
```

リスト 2.3.3 はリスト 2.2.1 とリスト 2.3.1 の２つを併せて行うだけだ．冒頭で

```
from ELCA import *
```

によって、ELCA の中に定義された関数を全部まとめて import している．だから、set_A()と set_branch()はこのプログラムの中に定義したのと同じように実行できる．

2.3.2 電位と電圧の関係

ここで電位と電圧の関係 ― それは回路の接続関係が決まれば必然的に決まるものであるから ― について考えてみよう．回路を構成するノードの電位をノード１からノード N まで並べてベクトルであらわすと次のようになる．

$$\boldsymbol{W} = \begin{pmatrix} w_1 \\ \vdots \\ w_\xi \\ \vdots \\ w_\eta \\ \vdots \\ w_N \end{pmatrix} \tag{2.3.4}$$

ここでノード 0 は、グランドまたは接地で電位は 0 なので、ベクトル $\boldsymbol{W}$ には含めないこととする．いっぽう、回路を構成するブランチの両端の電圧も並べてベクトルであらわすと、

$$\boldsymbol{V} = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ \vdots \\ v_B \end{pmatrix} \tag{2.3.5}$$

ここで、電位と電圧の関係を考えてみよう. 電位の差が電位差であって電圧であることはすでに述べたが、図2.2.1に示すようにブランチ k について、

$$v_k = w_\xi - w_\eta \tag{2.3.6}$$

これは式2.3.4で与えられる電位のベクトル $\boldsymbol{W}$ から、その始点ノードと終点ノードに該当するノードの電位を、それぞれプラス(+)マイナス(−)の符号をつけて取り出したものにほかならない. それはまさしく、2.2.4 で定義した接続関係をあらわす行列 $\boldsymbol{A}$ によって、回路全体の電位と電圧の関係があらわせることを意味する.
式2.3.2の A_k と電位のベクトル $\boldsymbol{W}$ の内積は、

$$\begin{aligned} A_k \boldsymbol{W} &= (a_{k1} \quad \cdots \quad a_{k\xi} \quad \cdots \quad a_{k\eta} \quad \cdots \quad a_{kN}) \begin{pmatrix} w_1 \\ \vdots \\ w_\xi \\ \vdots \\ w_\eta \\ \vdots \\ w_N \end{pmatrix} \\ &= (0 \quad \cdots \quad 1 \quad \cdots \quad -1 \quad \cdots \quad 0) \begin{pmatrix} w_1 \\ \vdots \\ w_\xi \\ \vdots \\ w_\eta \\ \vdots \\ w_N \end{pmatrix} \\ &= w_\xi - w_\eta \end{aligned} \tag{2.3.7}$$

だから、

$$v_k = A_k \boldsymbol{W} \tag{2.3.8}$$

これをブランチ1からブランチBまで順に縦に並べれば、次式が成り立つ.

$$\begin{pmatrix} v_1 \\ \vdots \\ \vdots \\ v_k \\ \vdots \\ \vdots \\ v_B \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1N} \\ \vdots & \ddots & & & & & \vdots \\ \vdots & & & & & & \vdots \\ a_{k1} & \cdots & a_{k\xi} & \cdots & a_{k\eta} & \cdots & a_{kN} \\ \vdots & & & & & & \vdots \\ \vdots & & & & & \ddots & \vdots \\ a_{B1} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{BN} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_\xi \\ \vdots \\ w_\eta \\ \vdots \\ w_N \end{pmatrix} \tag{2.3.9}$$

$$\boldsymbol{V} = \boldsymbol{A}\,\boldsymbol{W} \tag{2.3.10}$$

このように回路全体にわたる電位と電圧の関係を簡単な式で示すことができた．逆にいうならば、行列 $\boldsymbol{A}$ は、回路の接続関係によって必然的に決まることがら(ここでは電位と電圧の関係なのだが、)を表現するように定義したものなのである.

念のためブランチの始点がグランド、あるいは、終点がグランドの場合についても見ておこう．まず始点がグランドの場合だが、この場合 $\xi = 0$ であって $\xi \geqq 1$ は満たされず、始点ノードに対応する $a_{k\xi}$ は行列 $\boldsymbol{A}$ の要素として存在しない．したがって、式 2.3.7 ないしは式 2.3.8 において、終点ノード η に対する $a_{k\eta} = -1$ と、他のノードに関する

$a_{ki} = 0\ \left(i \neq \xi, \eta、\ i \geqq 1\right)$ だけが存在して、

$$v_k = -w_\eta \tag{2.3.11}$$

これは図 2.2.1 で始点ノードの電位が 0 となる場合を考えれば当然である.
同様に終点がグランドの場合だが、この場合 $\eta = 0$ であって $\eta \geqq 1$ は満たされず、終点ノードに対応する $a_{k\eta}$ は行列 $\boldsymbol{A}$ の要素として存在しない. したがって、式 2.3.7 ないしは式 2.3.8 において、始点ノード ξ に対する $a_{k\xi} = 1$ と、他のノードに関する

$a_{ki} = 0\ \left(i \neq \xi, \eta、\ i \geqq 1\right)$だけが存在して、

$$v_k = w_\xi \tag{2.3.12}$$

これも図 2.2.1 で終点ノードの電位が 0 となる場合を考えれば当然である．いずれの場合にも、式 2.3.9 ないしは式 2.3.10 が成り立っている．ではここで python のプログラムで確かめておこう.

リスト 2.3.4：電位と電圧の関係をみる python のコードイメージ

```
from ELCA import *
import MatMath as mm

# Number of Nodes and Branches
n = int(input('Number of nodes    '))
b = int(input('Number of branches '))
print('node# =', n, 'branch# =', b)

A = set_A(n, b)
print('Matrix A ¥n', A)

W = []
# Input the Potential of each node
print('Input the potential of each node.')
for i in range(n):
    print('Node#' + str(i+1))
    w = float(input('the potential '))
    W.append(w)

print('Vector W =', W)

V = mm.dot(A, W)

# Display Branches
for i in range(b):
    print('branch#{:d}：Voltage = {:6.3f}V'.format(i+1, V[i]))
```

このプログラムでは、行列 $\boldsymbol{A}$ を作り、ノード電位 $\boldsymbol{W}$ を入力して、式 2.3.9 ないしは式 2.3.10 で示す電位と電圧の関係が本当に成り立っているかどうかを確かめる．冒頭で、

```
from ELCA import *
```

により、関数 set_A()を import している．ノード電位 $\boldsymbol{W}$ は空のリストで宣言してから、for 文でループして 1 番から n 番まで順に、ノード電位を入力してはベクトル $\boldsymbol{W}$ に append で組み上げていく．行列 $\boldsymbol{A}$ とベクトル $\boldsymbol{W}$ が得られると、V = mm.dot(A, W) で $\boldsymbol{A}$ と $\boldsymbol{W}$ の積を計算して、ブランチ電圧のベクトル $\boldsymbol{V}$ が算出される．説明が前後するが、この計算のために 2 行目で MatMath を mm と変名してインポートしている．ベクトル $\boldsymbol{V}$ が得られたら、各ブランチの電圧を出力して終了する．このプログラムを起動して、すでにノードの電位がわかっている図 2.2.2 の回路や、図 2.1.5 および図 2.2.3 の回路で各ブランチの両端の電圧が正しく表示され、式 2.3.10 が成り立つことを確認しておきたい．

2.4 回路におけるオームとキルヒホッフ

2.4.1 オームの法則

2.2.1 でも説明したオームの法則について再度考えてみよう．図 2.2.1 で示す回路の基本構成要素としてのブランチにおいて成り立つ関係は、

$$i = \frac{v - e}{r} = y(v - e) = y\,v - y\,e \tag{2.4.1}$$

回路全体が B 個のブランチからなる場合には、各々のブランチにおいて、この関係が成り立つから、

$$\begin{aligned} i_1 &= y_{11}v_1 - y_{11}e_1 \\ i_2 &= y_{22}v_2 - y_{22}e_2 \\ &\vdots \\ i_B &= y_{BB}v_B - y_{BB}e_B \end{aligned} \tag{2.4.2}$$

行列やベクトルを使ってあらわせば、

$$\begin{aligned} \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_B \end{pmatrix} &= \begin{pmatrix} y_{11}v_1 - y_{11}e_1 \\ y_{22}v_2 - y_{22}e_2 \\ \vdots \\ y_{BB}v_B - y_{BB}e_B \end{pmatrix} = \begin{pmatrix} y_{11}v_1 \\ y_{22}v_2 \\ \vdots \\ y_{BB}v_B \end{pmatrix} - \begin{pmatrix} y_{11}e_1 \\ y_{22}e_2 \\ \vdots \\ y_{BB}e_B \end{pmatrix} \\ &= \begin{pmatrix} y_{11} & 0 & \cdots & 0 \\ 0 & y_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_{BB} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_B \end{pmatrix} - \begin{pmatrix} y_{11}e_1 \\ y_{22}e_2 \\ \vdots \\ y_{BB}e_B \end{pmatrix} \end{aligned} \tag{2.4.3}$$

ここで回路を構成するブランチを流れる電流をまとめてベクトルで次のようにあらわす．

$$\boldsymbol{I} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_B \end{pmatrix} \tag{2.4.4}$$

また、回路を構成するブランチのアドミタンスを並べた行列を次のように定義する．

$$\boldsymbol{Y} = \begin{pmatrix} y_{11} & 0 & \cdots & 0 \\ 0 & y_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_{BB} \end{pmatrix} \tag{2.4.5}$$

そして式 2.4.3 の右辺の既知の値が並んだベクトルを次のように定義する.

$$\boldsymbol{S} = \begin{pmatrix} y_{11}e_1 \\ y_{22}e_2 \\ \vdots \\ y_{BB}e_B \end{pmatrix} \tag{2.4.6}$$

式 2.4.3 は、式 2.3.5 で定義した $\boldsymbol{V}$ も用いて次式となる.

$$\boldsymbol{I} = \boldsymbol{Y}\boldsymbol{V} - \boldsymbol{S} \tag{2.4.7}$$

ここで示した式 2.4.3、ないしは、式 2.4.7 は、各ブランチにおけるオームの法則をひとまとめにしてあらわすものであり、回路全体のオームの法則ということになる. オームの法則については、ブランチごとに独立した関係であるから、回路の接続関係をあらわす行列 $\boldsymbol{A}$ は回路全体のオームの法則において使われることはない.

[補足 2.4.1]
式 2.4.5 で回路を構成するブランチのアドミタンスを並べた行列として $\boldsymbol{Y}$ を定義し、その対角要素だけが各ブランチの抵抗に対応するアドミタンスの値をもち、それ以外はすべて 0 とした. 図 2.2.1 でブランチ k に対応するアドミタンスを Y_k ではなく Y_{kk} と添字を重ねたのは、この整合性のためである. しかしそもそも回路を構成するブランチのアドミタンスは電流や電圧と同じように、ベクトルであらわせるはずであって、行列にする理由があるのだろうか？
実はトランジスタなどのいわゆる能動素子が回路に含まれる場合には、それをあるブランチから別のあるブランチへの作用としてあらわす必要がある. そして作用ブランチと被作用ブランチの 2 つの相互の関係をあらわす手段として、式 2.4.5 で定義する行列 $\boldsymbol{Y}$ の対角要素以外の要素を使って、ブランチ間の相互コンダクタンスが記述されることとなるのである. 補足 2.2. 1 で言及した電圧制御電流源の特性はこれによって表現されることとなる.
ではここでプログラムで確かめておこう.

リスト 2.4.1：オームの法則の python のコードイメージ

```
from ELCA import *
import MatMath as mm

# Number of Branches
b = int(input('Number of branches '))
print('branch# =', b)

Y, E = set_branch(b)

V = []
for k in range(b):
    print('Branch#' + str(k+1))
    v = float(input('volatage between start and end '))
    V.append(v)

# Caluculate electric current
S = [Y[i][i] * e for i, e in enumerate(E)]
I = mm.sub(mm.dot(Y, V), S)

# Display
print('Matrix Y ¥n', Y)
print('Vector S =', S)
print('Vector E =', E)
print('Vector I =', I)
```

このプログラムは、式 2.4.3 ないし式 2.4.7 で示す回路全体のオームの法則を確かめる．そのためにブランチのアドミタンスの行列 $\boldsymbol{Y}$ や電圧源のベクトル $\boldsymbol{E}$ が必要だが、これらは ELCA に置いた関数 set_branch()を使って作る．ブランチの両端の電圧のベクトルは、この関数では作らないので、これとは別にブランチ 1 から b の両端電圧を for ループで回しながら順に入力して append()で組み上げる．それから、2 行目で mm と変名してインポートしておいた MatMath の関数を使って、オームの法則の計算を行う．まず定数項 $\boldsymbol{S}$ を、

```
S = [Y[i][i] * e for i, e in enumerate(E)]
```

で求める． $\boldsymbol{S}$ は python のリスト内包表記で作っているが、enumerate()を使えば引数 $\boldsymbol{E}$ の要素を 1 つずつその指標とともに取り出すことができる． $\boldsymbol{S}$ が求まるとあとは一気に、式 2.4.7 にしたがって、

```
I = mm.sub(mm.dot(Y, V), S)
```

でブランチの電流のベクトル $\boldsymbol{I}$ を計算する．そして結果を表示して終わる．本来は電圧のベクトル $\boldsymbol{V}$ は、回路の解析によって算出すべきものだが、このプログラムでは外から入力している．すでに解析ができている図 2.2.2 や、図 2.1.5 および図 2.2.3 の回路を入力して、式 2.4.7 の回路全体のオームの法則について確かめてほしい．

[補足 2.4.2]
ベクトル $\boldsymbol{S}$ は、行列 $\boldsymbol{Y}$ が対角要素だけの場合は、

```
S = mm.dot(Y, E)
```

でも求めることができる．しかし式 2.4.1 から式 2.4.7 を導く過程からわかるように、ベクトル $\boldsymbol{S}$ は、ブランチを流れる電流のうち、ブランチ自身の電圧源に起因する成分を自身の抵抗との関係において定数項として分離したものである．

2.4.2 キルヒホッフの法則

キルヒホッフの法則について具体例からはじめてみよう．図 2.2.2 で示した電池と抵抗だけからなる回路だった．そしてそこでキルヒホッフの法則は式 2.2.10 で示された．
ここで式 2.3.1 で定義された a がどうなるかを考えてみると、ノード 1 がブランチ 1 とブランチ 2 の両方の始点ノードなのだから、$a_{11} = 1$ かつ $a_{21} = 1$ であり、式 2.2.10 は次式と矛盾しない．

$$a_{11}i_1 + a_{21}i_2 = 0 \tag{24.9}$$

これをベクトルの内積であらわせば、

$$(a_{11} \quad a_{21})\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = 0 \tag{2.4.10}$$

ここで注意すべきは、$(a_{11} \quad a_{21})$ は式 2.3.2 で定義した行列 $\boldsymbol{A}$ を転置したものとなっていることである．さてここではじめから答えのわかっている極めて単純な例について考えたが、では一般的にはどうなのだろうか？
式 2.3.3 で定義した行列 $\boldsymbol{A}$ を i 列を明示して再掲する．

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & \cdots & a_{1i} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots & & \vdots \\ \vdots & & \vdots & & \vdots \\ \vdots & & a_{ki} & & \vdots \\ \vdots & & \vdots & & \vdots \\ \vdots & & \vdots & \ddots & \vdots \\ a_{B1} & \cdots & a_{Bi} & \cdots & a_{BN} \end{pmatrix} \tag{2.4.11}$$

ここで行列 $\boldsymbol{A}$ から i 列を抜き出して、そのベクトルを ${}^{t}A_i$ とおくと、

$$ {}^{t}A_i = (a_{1i} \quad \cdots \quad a_{ki} \quad \cdots \quad a_{Bi}) \tag{2.4.12} $$

その定義から ${}^{t}A_i$ の要素 $a_{ki}\ (1 \leqq k \leqq B)$ は、ノード i がブランチ k の始点となっている場合は $a_{ki} = 1$、終点となっている場合は $a_{ki} = -1$ である．そしてノード i と関係ないブランチに対応する場合は $a_{ki} = 0$ である．もちろんノード i にいくつブランチがつながっているかは回路次第であるから、 ${}^{t}A_i$ の要素の 1、-1、0 の数に制約はない．図 2.1.4 に示すようにブランチ k では、その始点ノードからその終点ノードに向かって電流 i_k が流れる．このことから、 ${}^{t}A_i$ はノード i に関する電流の出入りの関係 ― ブランチの始点となっていて流れ出す、ブランチの終点となっていて流れ込む ― の関係をあらわすことは明らかである．したがってノード i に関するキルヒホッフの法則は、次式であらわされる．

$$ (a_{1i} \quad \cdots \quad a_{ki} \quad \cdots \quad a_{Bi}) \begin{pmatrix} i_1 \\ \vdots \\ i_k \\ \vdots \\ i_B \end{pmatrix} = 0 \tag{2.4.13} $$

$$ {}^{t}A_i \, \boldsymbol{I} = 0 \tag{2.4.14} $$

回路の各部分でキルヒホッフの法則は成り立つから、回路全体について式 2.4.13 ないしは式 2.4.14 をノード 1 からノード N まで順に縦に並べれば、次式が成り立つ．

$$ \begin{pmatrix} a_{11} & \cdots & \cdots & \cdots & a_{B1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & & & \vdots \\ a_{1i} & \cdots & a_{ki} & \cdots & a_{Bi} \\ \vdots & & & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{1N} & \cdots & \cdots & \cdots & a_{BN} \end{pmatrix} \begin{pmatrix} i_1 \\ \vdots \\ \vdots \\ i_k \\ \vdots \\ \vdots \\ i_B \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \tag{2.4.15} $$

ここで左辺の行列は $\boldsymbol{A}$ を転置した行列となっている．すなわち、

$$
{}^{t}\boldsymbol{A} = \begin{pmatrix} a_{11} & \cdots & \cdots & \cdots & a_{B1} \\ \vdots & \ddots & & & \vdots \\ \vdots & & & & \vdots \\ a_{1i} & \cdots & a_{ki} & \cdots & a_{Bi} \\ \vdots & & & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{1N} & \cdots & \cdots & \cdots & a_{BN} \end{pmatrix} \tag{2.4.16}
$$

であって、もともとベクトル ${}^{t}A_i$ は、行列 $\boldsymbol{A}$ の i 列を抜き出したものであったから、行列 ${}^{t}\boldsymbol{A}$ の i 行を抜き出したものとなる．ここで、

$$
\mathbf{0} = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \end{pmatrix} \tag{2.4.17}
$$

とおけば、式 2.4.15 は次式となる．

$$
{}^{t}\boldsymbol{A}\,\boldsymbol{I} = \mathbf{0} \tag{2.4.18}
$$

このベクトルの要素数は N 個である．ともあれこれで、式 2.3.3 で定義した回路全体の接続関係をあらわす行列 $\boldsymbol{A}$ を用いて回路全体のキルヒホッフの法則をあらわせることがわかった．

これまでにリスト 2.3.1 で行列 $\boldsymbol{A}$ を作った．そしてリスト 2.3.4 で行列 $\boldsymbol{A}$ を使って電位と電圧の関係を確かめたから、今度は行列 $\boldsymbol{A}$ を使って電流を確かめる、すなわちキルヒホッフの法則を確かめる番である．ではさっそくプログラムで確かめてみよう．

リスト 2.4.2：キルヒホッフの法則の python のコードイメージ

```
from ELCA import *
import MatMath as mm

# Number of Nodes and Branches
n = int(input('Number of nodes    '))
b = int(input('Number of branches '))
print('node# =', n, 'branch# =', b)

A = set_A(n, b)
print('Matrix A ¥n', A)

I = []
print('Input electric current of each branch.')
for k in range(b):
    print('Branch#' + str(k+1))
    i = float(input('electric current '))
    I.append(i)

print('Electric Current Vector I =', I)

tA = mm.trn(A)
print('Transpose Matrix tA ¥n', tA)
Z = mm.dot(tA, I)
print('Zeros Vector 0 =', Z)
```

ノードとブランチの数を入力してから、ELCA に定義した関数 set_A()で行列 $\boldsymbol{A}$ を作る．いっぽうキルヒホッフの法則を確かめるのに必要な電流は、これとは別に入力する．先に空のリスト[]で宣言しておいて、for 文でループしながらブランチの電流を入力させて、append()によりリストに追加してベクトル $\boldsymbol{I}$ として組み上げる．ループを抜けて $\boldsymbol{I}$ ができるとこれを表示したのち、$\boldsymbol{A}$ の転置行列を、

```
tA = mm.trn(A)
```

で作って表示する．そして、式 2.4.15 ないし式 2.4.18 を、

```
Z = mm.dot(tA, I)
```

で計算して表示する．これらの計算のため MatMath を mm と変名して冒頭でインポートしている．この結果 $\boldsymbol{Z}$ がすべて 0 の要素のベクトルとなっていたら、キルヒホッフの法則が成り立っているということである．本来は電流のベクトル $\boldsymbol{I}$ は、回路の解析によって算出すべきものだが、このプログラムでは外から入力している．すでに解析ができている図 2.2.2 や、図 2.1.5 および図 2.2.3 の回路を入力して、式 2.4.15 ないし式 2.4.18 の回路全体のキルヒホッフの法則について確かめてほしい．

2.5 回路の解析方法

あらためて考えてみると、回路が決まったら確定するのは、それがどういう部品で構成され、それがどうつながれているかである．これをブランチとノードで言い換えるならば、構成するブランチのスペックと、ノードにおいてブランチがどう接続されているかが、与えられるのである．いっぽう、その回路が構成された結果として、各ノードの電位、各ブランチの電流、電圧などの状態が決まるのであり、電位と電圧の当たり前の関係とともに、それを支配する法則が、オームの法則とキルヒホッフの法則なのである．これら回路の状態を支配するものについては、ここまでの歩みの中で、一つ一つすべて確かめてきた．振り返るならば、リスト 2.3.2 で回路の接続を示す行列 $\boldsymbol{A}$ を作り、電位、電圧を与えて、式 2.3.10 の電位と電圧の関係が成り立つことを確かめた．そして次にリスト 2.4.2 でブランチのスペックを与えるとともに、ブランチ両端の電圧を与えて、式 2.4.7 のオームの法則が成り立つことを確かめた．最後にリスト 2.4.3 で、接続関係を示す行列 $\boldsymbol{A}$ を作るとともに、各ブランチの電流を与えて、式 2.4.18 のキルヒホッフの法則が成り立つことを確かめた．だから回路の状態はここまでに求めてきた関係式を解くことで解析できるはずである．それでは、ここまでの結果をふまえて、いよいよ電子回路の解析へと歩みを進めていこう．

上述のように、ここまでに回路を解析するのに必要な数式がそろっている．ここでそれをもう一度まとめると、電位と電圧の関係から式 2.3.10、オームの法則から式 2.4.7、そして、キルヒホッフの法則から式 2.4.18 である．

$$\boldsymbol{V} = \boldsymbol{A}\,\boldsymbol{W} \qquad (2.3.10\ \text{再掲})$$

$$\boldsymbol{I} = \boldsymbol{Y}\,\boldsymbol{V} - \boldsymbol{S} \qquad (2.4.7\ \text{再掲})$$

$${}^{t}\boldsymbol{A}\,\boldsymbol{I} = \boldsymbol{0} \qquad (2.4.18\ \text{再掲})$$

ではこれらの式から回路を解いていこう．式 2.4.7 の両辺に左から ${}^{t}\boldsymbol{A}$ をかけて、

$${}^{t}\boldsymbol{A}\,\boldsymbol{I} = {}^{t}\boldsymbol{A}\,(\boldsymbol{Y}\,\boldsymbol{V} - \boldsymbol{S}) = {}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} - {}^{t}\boldsymbol{A}\,\boldsymbol{S} \qquad (2.5.1)$$

式2.5.1と式2.4.18から、

$$ {}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} - {}^{t}\boldsymbol{A}\,\boldsymbol{S} = \boldsymbol{0} $$

$$ {}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} = {}^{t}\boldsymbol{A}\,\boldsymbol{S} \tag{2.5.2} $$

式2.5.2に式2.3.10を代入して、

$$ {}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{A}\,\boldsymbol{W} = {}^{t}\boldsymbol{A}\,\boldsymbol{S} \tag{2.5.3} $$

${}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{A}$ の逆行列を求めて、式2.5.3の両辺にかければ、左辺は $\boldsymbol{W}$ となって、

$$ \boldsymbol{W} = \left({}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{A} \right)^{-1} {}^{t}\boldsymbol{A}\,\boldsymbol{S} \tag{2.5.4} $$

式2.5.4から $\boldsymbol{W}$ が求まれば、これを式2.3.10に代入して、 $\boldsymbol{V}$ が求まり、その $\boldsymbol{V}$ を式2.4.7に代入して $\boldsymbol{I}$ が求まる．これで回路のすべてのノードの電位、そして回路のすべてのブランチの両端の電圧とそこを流れる電流がわかることになる．

それではいよいよ、電子回路の解析プログラムを作ることにする．まずは、この節で説明してきた解析方法をそのまま関数Analize()として定義する．ファイルELCA.pyに、関数set_A(), set_branch()をおいているから、そこに書き加えるのが良いだろう．

リスト2.5.1：電子回路の解析のための関数Analize()のpythonのコードイメージ

```
import MatMath as mm

def Analize(A, Y, E):
    S = [Y[i][i] * e for i, e in enumerate(E)]
    tA   = mm.trn(A)
    tAS  = mm.dot(tA, S)
    tAY  = mm.dot(tA, Y)
    tAYA = mm.dot(tAY, A)
    L    = mm.inv(tAYA)
    W    = mm.dot(L, tAS)
    V    = mm.dot(A, W)
    I    = mm.sub(mm.dot(Y, V), S)
    P    = mm.mul(mm.sub(V, E), I)
    return W, V, I, P
```

リスト 2.5.1 には、新たに定義する関数 Analize()だけを掲載したが、先に述べたように ELCA.py には、このほかに set_A()と set_branch()の 2 つの関数がすでに定義されていて、つごう 3 つの関数が置かれる．ベクトルや行列の演算を使うので冒頭では MatMath をインポートする．この関数 Analize()には引数として接続関係の行列 $\boldsymbol{A}$、アドミタンスの行列 $\boldsymbol{Y}$、起電力のベクトル $\boldsymbol{E}$ を渡す．これらを受け取って解析にはいると、定数項のベクトル $\boldsymbol{S}$、行列 $\boldsymbol{A}$ の転置行列 ${}^t\boldsymbol{A}$ をつくり、式 2.5.3 の右辺の ${}^t\boldsymbol{AS}$、左辺の ${}^t\boldsymbol{AYA}$ を作る．そして、

```
L = mm.inv(tAYA)
```

で tAYA の逆行列を求めて、式 2.5.4 にしたがって、

```
W = mm.dot(L, tAS)
```

でノード電位のベクトル $\boldsymbol{W}$ を求める．$\boldsymbol{W}$ が求まると、式 2.3.10 にしたがって、

```
V = mm.dot(A, W)
```

でブランチの電圧を算出し、式 2.4.7 にしたがって

```
I = mm.sub(mm.dot(Y, V), S)
```

でブランチの電流を算出する．プログラムでは、このあとにブランチの消費電力も

```
P = mm.mul(mm.sub(V, E), I)
```

で算出している．ここで消費電力は、$\boldsymbol{V}\times\boldsymbol{I}$ ではなくて、$(\boldsymbol{V}-\boldsymbol{E})\times\boldsymbol{I}$ としているが、ブランチの抵抗で熱として失われるものを電力としているのであり、ブランチの抵抗の両端の電圧は$\boldsymbol{V}$ ではなくて $\boldsymbol{V}-\boldsymbol{E}$ だからである．

計算が完了すると、ノード電位 $\boldsymbol{W}$、ブランチ電圧 $\boldsymbol{V}$、ブランチ電流 $\boldsymbol{I}$、そしておまけにブランチ消費電力 $\boldsymbol{P}$ を返り値として関数を終わる．

続いて電子回路の解析プログラムの本体部分である.

リスト 2.5.2：電子回路の解析の python のコードイメージ

```
from ELCA import *

# Number of Nodes and Branches
n = int(input('Number of nodes    '))
b = int(input('Number of branches '))
print('node# =', n, 'branch# =', b)

A = set_A(n, b)
Y, E = set_branch(b)
W, V, I, P = Analize(A, Y, E)

# Display Nodes
print()
for i in range(n):
    print('Node#{:d} potential = {:8.4f}V'.format(i+1, W[i]))

# Display Branches
print()
for i in range(b):
    print('branch#{:d}：voltage = {:8.4f}V current = {:8.4f}A power = {:8.4f}W' ¥
          .format(i+1, V[i], I[i], P[i]))
```

このプログラムでは回路の諸元を入力して、それをもとに算出したノードの電位やブランチの電圧、電流を表示する. 主要な機能はすでに ELCA に定義されているから、回路のノード数とブランチ数を入力して、ELCA の関数を順に呼出す. まずは、

```
A = set_A(n, b)
Y, E = set_branch(b)
```

で、接続関係の行列 $\boldsymbol{A}$、アドミタンスの行列 $\boldsymbol{Y}$、ブランチの起電力のベクトル $\boldsymbol{E}$ を得る.
それから解析 Analize()を走らせる. 解析が終わると、各ノードの電位を表示し、各ブランチの電圧、電流、電力を表示してプログラムが終了する.
長い長い道のりだったが、図 2.2.3 に示す回路を入力してみて、図 2.1.5 と同じ結果が得られることを確かめてほしい. それからできれば、電源や抵抗がもっとたくさんつながった回路でもプログラムが、動作してきちんと結果が得られることを確かめてほしい.

2.6 ブランチの定義の拡張

ここまで電圧源と抵抗だけで構成される回路を対象として回路解析の方法を考えてきた．しかし電子回路は、バイポーラトランジスタなどの能動素子を加えることではじめて、機能的になる．そこでここからは、能動素子をどのようにして表現し、これを含む回路を解析するのかについて検討する．むろんできることは、まだまだ限られているのであって、ここから先は非線形性や時間応答など、ここまでの内容では取り扱うことができない多くの難問が待ち構えている．そこであくまでも、ここまでの内容を基本としながら発展させていく方向性を探るにとどめておくこととする．

まずはじめにやらなくてはならないことは、図 2.2.1 に提示したブランチの構成要素に電圧制御電流源を加えることである．これをつかって、トランジスタをどう表現するかは、後で説明することとして、図 2.6.1 に電圧制御電流源を加えたブランチを始点ノード、終点ノードとともにあらわす．

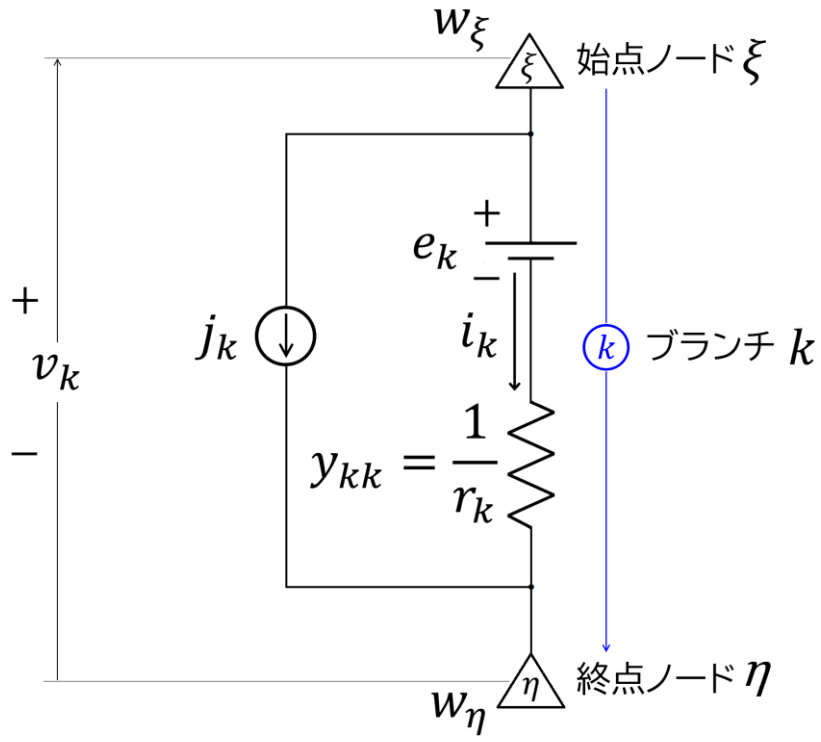

図 2.6.1　電圧制御電流源を加えたブランチ

ブランチの始点ノードξ、終点ノードηを結ぶ形で、これまでのブランチと並列に電圧制御電流源を設ける．電圧制御電流源は、その呼び名どおり、回路内のいずれかのブランチの電圧に制御されて、流れる電流の大きさが変わる電流源である．そしてそこを流れる電流は、j_k とする． i_k 同様に始点ノードξから終点ノードηに向かって流れる電流をプラス(+)、逆をマイナス(−)とする．これを加えたことによって、ブランチを流れる電流は $i_k + j_k$ となる．

ブランチの抵抗を流れる電流については、式 2.4.1 に示すように、その抵抗の両端の電圧が、 $v_k - e_k$ であり、

$$i_k = \frac{v_k - e_k}{r_k} = y_{kk}(v_k - e_k) \qquad \left(2.4.1 \text{ 再掲}\right)$$

いっぽう電圧制御電流源を流れる電流 j_k は、この制御ブランチが l のとき、そのブランチ l の両端の電圧が v_l ならば、

$$j_k = y_{kl} v_l \tag{2.6.1}$$

電圧制御電流源を加えたブランチに流れる電流は、
式 2.4.2 は次式で置き換えられる．

$$i_k + j_k = y_{kk} v_k + y_{kl} v_l - y_{kk} e_k \tag{2.6.2}$$

これを B 個のブランチについてまとめて、式 2.4.3 は次式で置きかえられる．

$$\begin{pmatrix} i_1 \\ \vdots \\ i_k \\ \vdots \\ i_B \end{pmatrix} + \begin{pmatrix} j_1 \\ \vdots \\ j_k \\ \vdots \\ j_B \end{pmatrix} = \begin{pmatrix} y_{11} & \cdots & \cdots & \cdots & \cdots & \cdots & y_{1B} \\ \vdots & \ddots & & & & & \vdots \\ \vdots & & & & & & \vdots \\ \vdots & \cdots & y_{kk} & \cdots & y_{kl} & \cdots & \vdots \\ \vdots & & & & & & \vdots \\ \vdots & & & & \ddots & \ddots & \vdots \\ y_{B1} & \cdots & \cdots & \cdots & \cdots & \cdots & y_{BB} \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ \vdots \\ v_l \\ \vdots \\ v_B \end{pmatrix} - \begin{pmatrix} y_{11} e_1 \\ \vdots \\ y_{kk} e_k \\ \vdots \\ \vdots \\ y_{BB} e_B \end{pmatrix} \tag{2.6.3}$$

式 2.4.5 の行列 $\boldsymbol{Y}$ は対角要素以外も 0 とは限らないから、一般的に、

$$\boldsymbol{Y} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1B} \\ y_{21} & y_{22} & \cdots & y_{2B} \\ \vdots & \vdots & \ddots & \vdots \\ y_{B1} & y_{B2} & \cdots & y_{BB} \end{pmatrix} \tag{2.6.4}$$

式 2.4.7 は次式で置き換えられる.

$$\boldsymbol{I} + \boldsymbol{J} = \boldsymbol{YV} - \boldsymbol{S} \tag{2.6.5}$$

また、式 2.4.18 のキルヒホッフの法則は、電流が $\boldsymbol{I}$ ではなくて $\boldsymbol{I} + \boldsymbol{J}$ なのだから、

$$^{t}\boldsymbol{A}\,(\boldsymbol{I} + \boldsymbol{J}) = \boldsymbol{0} \tag{2.6.6}$$

当然ながら電位と電圧の関係は変わらず式 2.3.10 である.

$$\boldsymbol{V} = \boldsymbol{A}\,\boldsymbol{W} \qquad \left(2.3.10\ \text{再々掲}\right)$$

ではこれらを解いていくことにする. 式 2.6.5 の両辺に左から $^{t}\boldsymbol{A}$ をかけて、

$$^{t}\boldsymbol{A}\,(\boldsymbol{I} + \boldsymbol{J}) = {}^{t}\boldsymbol{A}\,(\boldsymbol{YV} - \boldsymbol{S}) = {}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} - {}^{t}\boldsymbol{A}\,\boldsymbol{S} \tag{2.6.7}$$

式 2.6.6 と式 2.6.7 から、

$$^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} - {}^{t}\boldsymbol{A}\,\boldsymbol{S} = \boldsymbol{0}$$

$$^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{V} = {}^{t}\boldsymbol{A}\,\boldsymbol{S} \qquad \left(2.5.2\ \text{再掲}\right)$$

式 2.5.2 はそのまま成立するから、式 2.5.3、式 2.5.4 もそのまま成立して、$\boldsymbol{W}$ が求まる.
念のためそれを再掲する.

$$^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{A}\,\boldsymbol{W} = {}^{t}\boldsymbol{A}\,\boldsymbol{S} \qquad \left(2.5.3\ \text{再掲}\right)$$

$$\boldsymbol{W} = \left({}^{t}\boldsymbol{A}\,\boldsymbol{Y}\,\boldsymbol{A}\right)^{-1}\,{}^{t}\boldsymbol{A}\,\boldsymbol{S} \qquad \left(2.5.4\ \text{再掲}\right)$$

これを式2.3.10に代入して $\boldsymbol{V}$ が求まるのは2.5と同じ．さらにその $\boldsymbol{V}$ を式2.6.5に代入して $\boldsymbol{I}+\boldsymbol{J}$ が求まる．そして $\boldsymbol{I}$ と $\boldsymbol{J}$ は、式2.4.1ないし、式2.6.1によって別々に求めることもできる.
ともかくも、ここでわかったことは、電圧制御電流源をブランチに加えるには、行列 $\boldsymbol{Y}$ の非対角成分にしかるべき値を設定する以外、何もしなくてよいということである.

[補足2.6.1]
式2.6.5で、定数項 $\boldsymbol{S}$ は、抵抗の両端の電圧が v ではなくて、$v-e$ であることに関連して、その起電力分 e に起因する．だから $\boldsymbol{S}$ を求める際には、式2.4.6で定義した通り、$\boldsymbol{Y}$ の対角成分と、その対応するブランチの起電力分 e で算出しなくてはならない．この点はすでに補足2.4.2で述べたが、節2.5までのように $\boldsymbol{Y}$ が対角成分以外はすべて0の場合には問題にならない．しかし、はじめから原理通りのインプリメントをしておかないと、あとになって「結果は出るけれどもなんだかおかしい」といったたぐいのたちの悪いバグをまねき、しばしば膨大な時間と労力をデバグのために費やすこととなってしまうから要注意である.

2.7 ダイオード

電圧制御電流源を加えることで能動素子の基本的な増幅動作は表すことができるのだが、現実の回路で使われる素子は半導体であって、その基本的な動作はダイオードの挙動を無視しては表すことができない．ここではまず、そのダイオードをプログラムであらわすことから始めてみよう．

2.7.1 ダイオードの特性

電子の電荷を q[C]、ボルツマン定数を k、絶対温度を T[K]、結合係数を n として、熱電圧 V_T は次式であらわされる．

$$V_T = \frac{nkT}{q} \tag{2.7.1}$$

そして、この V_T を用いて、PN 接合の電圧電流特性は、電圧を V、電流を I とし、逆方向飽和電流を I_0 とおくと、次式であらわされる．

$$I = I_0\left(e^{\frac{V}{V_T}} - 1\right) \tag{2.7.2}$$

もちろん式 2.7.2 であらわされるのは理想的なダイオードであって、現実のダイオードではこの式から外れていくし、ダイオードの品種や個別のばらつきや使用条件によっても特性は変わってくる．しかしここでは、モデルとしてある程度使えればよいと割り切ることとして、まずはダイオードの基本特性をプログラミングしていこう．
コンピュータのプログラムで扱う際に式 2.7.2 はあまり好ましくない．というのも、式の右辺の指数部分 $e^{\frac{V}{V_T}}$ が比較的大きな値であるのに対して、これと掛け合わされる I_0は非常に小さな値である．そこで、この式を次のように変形する．

$$I = e^{\frac{V}{V_T}+\ln(I_0)} - I_0 \tag{2.7.3}$$

では、プログラムをその実行結果とともに示そう．

リスト 2.7.1：ダイオードの基本特性

```
mport matplotlib.pyplot as plt
from ufiesia import MatMath as mm
import math

def diode(v, vt=0.026, i0=3e-9):
    i = math.exp(v / vt + math.log(i0)) - i0
    return i

V = mm.linspace(-1, 1, 101)
I = [diode(v) for v in V]

for v, i in zip(V, I):
    print('{:6.3f}{:18.2f}'.format(v, i))

plt.plot(V, I, label='current')
plt.legend()
plt.ylim(0, 1000)
plt.show()
```

このプログラムでは、ダイオードの電圧に対して電流を求めて、結果をグラフ表示する．表示のために matplotlib を使うから、冒頭でインポートしている．このほか Python の標準モジュールで数学的な計算を担う math もインポートしている．これは式 2.7.3 の対数や指数の計算を行いたいためだけの目的である．ダイオードの特性は関数 diode()として定義する．引数 v はもちろん電圧だが、このほかに、vt, i0 の 2 つの引数を与える．デフォルト値は、熱電圧は式 2.7.1 の一般的な値として vt=0.026 にしたが、i0 の方は様々で決めにくい．とはいえ普通の Si ダイオードの逆方向飽和電流が数 nA 程度なので i0=3e-9 としておいた．関数の中では、

```
i = math.exp(v / vt + math.log(i0)) - i0
```

で式 2.6.10 の計算を行って、電流 i を返り値として返すだけである．

プログラムの本体では、まず -1～1 の 101 個の電圧データを用意する．それから内包表記で電圧を一つずつ取り出して、関数 diode()で電流に変換して、101 個の電圧に対応する電流のリスト $\boldsymbol{I}$ を作る．

```
V = mm.linspace(-1, 1, 101)
I = [diode(v) for v in V]
```

これを matplotlib でグラフ表示する．ただここで、

```
plt.ylim(0, 1000)
```

で y 軸の範囲を指定しているが、これは式 2.7.2 ないし式 2.7.3 であらわされるダイオードの指数関数が、どこまで値の範囲を変えても急激に立ち上がる様子を見るために、いろいろ変えてみることをお勧めする．

説明が長くなったが、このプログラムの実行結果を以下に示す．電圧がマイナスの間は電流が流れず、プラスになってほどなく急激に電流が増えていて、ダイオードの整流作用が再現されていることがわかる．しかし同時に、この結果が現実的ではないこともわかる．すなわち、数式どおりの理想的ダイオードを現実とは違う無限に電流の供給できる環境で測定したら、こうなるという結果を示すに過ぎない．

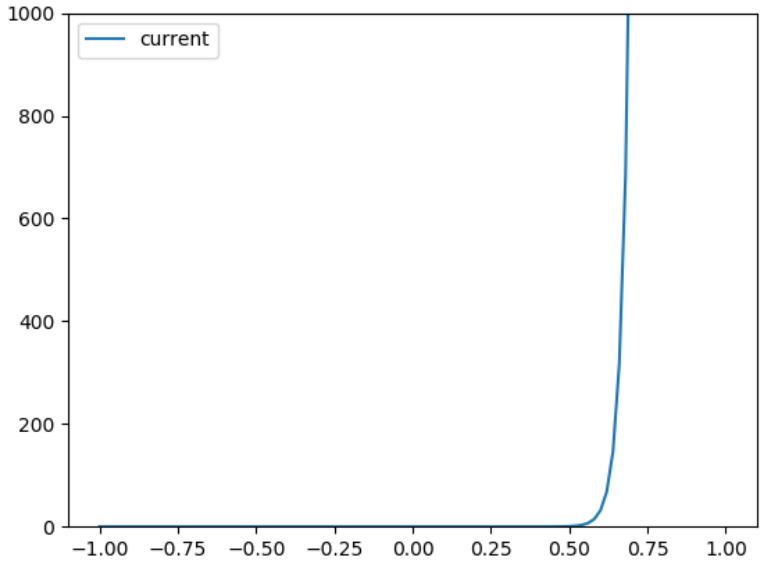

図 2.7.1 ダイオードの特性

2.7.2 ダイオードを回路へ組み込む

ダイオードの特性がプログラムであらせたので、これを回路に組み込んでいくことにする．そのために必要なことは、これまで行ってきた行列 $\boldsymbol{A}$、$\boldsymbol{Y}$、ベクトル $\boldsymbol{E}$、$\boldsymbol{W}$、$\boldsymbol{V}$、$\boldsymbol{I}$ で回路を扱う枠組みに整合するように、回路全体の中で１つとは限らないダイオードをあらわすことである．そこで図 2.7.2 に示すように通常のブランチの構成要素のうち、抵抗の部分をダイオードで置き換えることとする．どのブランチがダイオードかと、そのブランチの向きに対するダイオードの pn 接合の方向が自由に指定できるようにするために、これを指定するベクトルを用意する．こうしておけば、後でトランジスタなどの能動素子を組み込む際にも、都合が良い．細かい点だが、ダイオードのブランチでも電圧源があるから、ダイオードの電圧電流特性を電圧方向でシフトすることもできる．

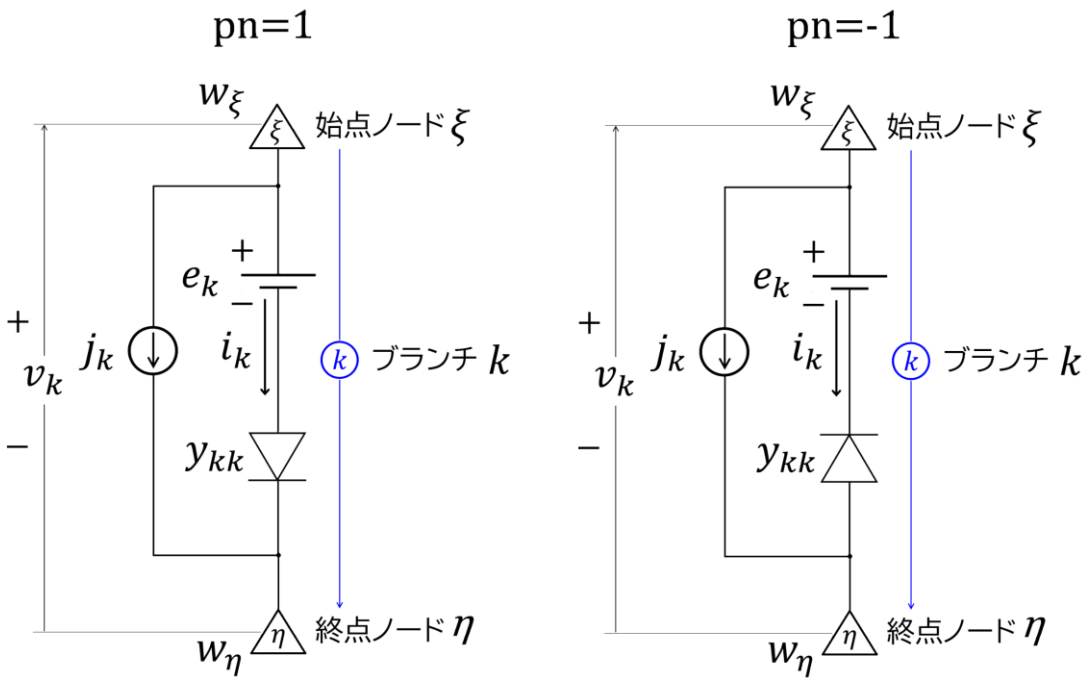

図 2.7.2　ダイオードのブランチ

これらの対応のため回路の中でダイオードをあらわす関数は複雑になる．しかし基本的にその電圧電流特性はリスト 2.7.1 の関数 diode()と同じである．とまれその関数をリスト 2.7.2 に示そう．リスト 2.7.1 の関数 diode()と区別するために、大文字で始めて関数名は Diode()とした．

リスト 2.7.2：ダイオードを回路に組み込むための関数

```
def Diode(Y, E, pn, V, Vt=0.026, Is=3e-9):
    Vf = mm.sub(V, E)
    Vf = mm.mul(Vf, pn)
    Vf = mm.where(mm.eq(Vf, 0), 1e-12, Vf)

    lnIs = mm.log(Is)
    X = mm.div(Vf, Vt)
    Z = mm.add(X, lnIs)
    Z = mm.where(mm.gt(Z, 100), 100, Z)
    Z = mm.exp(Z)
    I = mm.sub(Z, Is)

    vY = mm.div(I, Vf)
    vY = mm.where(mm.lt(vY, 1e-12), 1e-12, vY)
    vY = mm.where(mm.gt(vY, 1e12), 1e12, vY)
    mpn = mm.vec2mat(pn)
    Yd  = mm.vec2mat(vY)
    Yd  = mm.where(mm.eq(mpn, 0), Y, Yd)

    I = mm.mul(I, pn)

    return Yd
```

関数 Diode()では、先の関数 diode()とおなじく引数として電圧 $\boldsymbol{V}$ を与えるが、 $\boldsymbol{V}$ は回路全体の電圧をあらわすベクトルである．また、引数に $\boldsymbol{Y}$、$\boldsymbol{E}$、pnが加えられている．引数 $\boldsymbol{Y}$ は回路全体のアドミタンスの行列をこの関数に渡す．関数 Diode()は、回路の中でダイオードがどこにあるかを引数 pnで知り、同時にそのダイオードにかかる電圧を引数の $\boldsymbol{V}$ で知る．そして、ダイオードごとの電流を算出して、それをアドミタンスに変換する．その結果を引数として受けたアドミタンス $\boldsymbol{Y}$ と併せて、ダイオードを反映した回路のアドミタンスの行列 $\boldsymbol{Y_d}$ を作って、関数の返り値として返す．pn 接合の向きはベクトル形状の引数 pnで指定するが、回路内のブランチに対応して、$pn = 1$で pn 接合順方向のダイオード、$pn = -1$で pn 接合逆方向のダイオード、そして$pn = 0$はいずれでもない、つまりダイオード以外の通常のブランチを表す．また引数 I_s はダイオードの逆方向飽和電流、引数 offset は電圧をシフトする指定を行い、デフォルトではスカラ値を与えているが、使う際にはいずれもベクトルで与えてブランチごとに個別に指定できるようにした．これは回路内にトランジスタをはじめとして様々な素子がある場合に、個々の異なる特性を表すためであるが、MatMath の機能として形状の異なる変数同士の演算をサポートしているので、それに依っている．すでに多くを説明したが、初めの 2 行で図 2.7.2 に示すブランチの電圧源を除き、pnに応じた向きにする．これにより順方向を基準にしてダイオードにかかる電圧を Vf として求める．

```
    Vf = mm.sub(V, E)
    Vf = mm.mul(Vf, pn)
```

それから、後のアドミタンスを求める際に0除算をさけるため、電圧が0の場合には極小値1e-12にしておく.

```
Vf = mm.where(mm.eq(Vf, 0), 1e-12, Vf)
```

また、ダイオードの特性の計算はベクトル形状に対応してMatMathの関数で行う. I_s の自然対数を取るところから始めて、式2.7.3の通りに計算するが、指数関数がオーバーフローしないための対策としてexp()の前に100を上限にしている.

```
lnIs = mm.log(Is)
X = mm.div(Vf, Vt)
Z = mm.add(X, lnIs)
Z = mm.where(mm.gt(Z, 100), 100, Z)
Z = mm.exp(Z)
I = mm.sub(Z, Is)
```

電流が求まると電流を電圧で割ってアドミタンスを求める.

```
vY = mm.div(I, Vf)
```

この時点では、求まったアドミタンスは各ブランチに対応したベクトル形状だ. また後述の繰り返し計算の途中で、偶々発生する極端な状況として次の2行で、vYが負の値を含め極小値になってしまった場合と、極大値になってしまった場合ををを回避する.

```
vY = mm.where(mm.lt(vY, 1e-12), 1e-12, vY)
vY = mm.where(mm.gt(vY, 1e12), 1e12, vY)
```

それから次の３行で、もとの行列Yと形状を合わせたうえで、ダイオード部分のアドミタンスの値を計算で得られた値で置き換える.

```
mpn = mm.vec2mat(pn)
Yd  = mm.vec2mat(vY)
Yd  = mm.where(mm.eq(mpn, 0), Y, Yd)
```

この関数Diode()では、返り値はアドミタンスだけだから無用な操作だが、電圧をpnで指定した向きにかえたことに対応して電流の向きを合わせる１行を敢えて最後に残しておいた. もし電流 $\boldsymbol{I}$ を使うような拡張を行った場合には、これを忘れると負性抵抗のようになって意図しない挙動を示すから注意が必要だ.

```
I = mm.mul(I, pn)
```

Ydを返り値として関数Diode()が完了する.

さて、これでダイオードを含む回路の解析の準備の一つが整った. しかしここで困ったことに、これまでの逆行列を用いて解く方法では解けない. それというのも、リスト2.7.1の関数diode()やリスト2.7.2の関数Diode()に示すように、ダイオードを流れる電流 $\boldsymbol{I}$ は印加電

圧 V の指数関数となっているからである．当然ながらアドミタンス Y は印加電圧 V の指数関数となっている．これまでは、回路が決まればアドミタンス Y が先に与えられるということを前提として回路の解析を行ってきた．リスト 2.5.1 の関数 Analize()でいうならば、引数として接続関係の行列 A、アドミタンス Y、起電力 E を与えて、返り値として電位 W、電圧 V、電流 I、電力 P を得ていたのである．いっぽうダイオードは、電圧 V が与えられないとアドミタンス Y が決まらないのである．しかし、このアドミタンスが電圧 V の関数で、関数 Analize()では解けないことに対して、いささか安易ではあるものの、繰り返し計算によって対応することができるのではないだろうか？ つまり、ダイオードに印加される電圧に仮の値を与えてアドミタンス Y を求め、この求まった Y を用いて関数 Analize()によって解析を行って V を求め、その求まった V を用いて再び Y を求めて、また関数 Analize()で解析するということを繰り返すのである．その際に注意すべきポイントとして、関数 Analize()の解析の結果得られた V は依然として仮の値だということを忘れてはならない．なぜなら Analize()に与えた Y は仮の V から求めた仮の Y だからだ．しかし、この仮の V も仮の Y もどこかに収束して一定の値になったならば、もはやそれは仮ではなくて、解析結果として採用してよいと言えるだろう．だから、回路解析は V や Y が収束するまで繰り返す必要があるし、そもそも収束するように繰り返さなければならない．その収束を助けるために、ここでは仮の V の与え方にちょっとした工夫が必要となる．その工夫を含めてダイオードを組み込んだ回路の解析をプログラムで示そう．といっても、図 2.7.3 に示すダイオード 1 つの他は電源を担うブランチ 1 つだけのごくごく簡単な回路である．

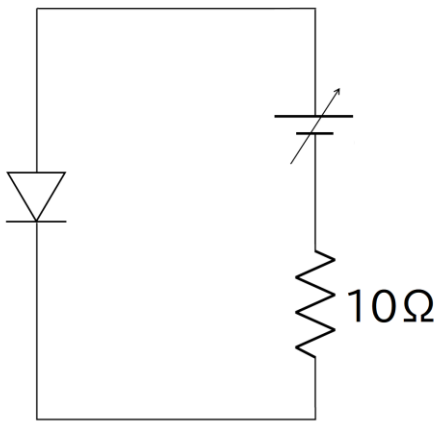

図 2.7.3 ダイオード 1 つと電池 1 つ

リスト 2.7.3：ダイオードを含む回路の解析

```
from ELCA import *

n = 1; b = 2
A = [[1], [1]]
Y = [[0.01, 0.0], [0.0, 0.1]]
E = [0.3, 0.0]
pn = [1, 0]

inputs = mm.linspace(-1, 1, 21)
Vs, Is, Ys = [], [], []
itr = 300; lr=0.1
for inp in inputs:
    E[1] = inp
    V_ppl = [0.5 for i in range(b)]
    for i in range(itr):
        W, V, I, P = Analize(A, Y, E)
        V_ppl = mm.sub(V_ppl, mm.mul(lr, mm.sub(V_ppl, V)))  # 移動平均
        Y = Diode(Y, E, pn, V_ppl)

        Vs.append(V[0])
        Is.append(I[0]*100)

    print('voltage{:6.3f}V current{:6.3f}mA'.format(V[0], I[0]*1000))

import matplotlib.pyplot as plt
plt.plot(Vs, label='voltage')
plt.plot(Is, label='current(10mA)')
#plt.plot(Vs, Is, label='V-I')
plt.legend()
plt.grid()
plt.show()
```

すでに述べたように、このプログラムでは、図 2.7.3 のダイオード 1 個とこれに電圧を加えるブランチ 1 つのごく簡単な回路で、加える電圧を変えて電流を観測する．リスト 2.7.2 で定義した関数 Diode()は、リスト 2.5.1 で定義した関数 Analize()などとともに、ファイル ELCA.py に置いて、それを冒頭でインポートする．そして、ごく簡単な回路なので回路をあらわす行列、ベクトルを次の 5 行で直接記述した．

```
n = 1; b = 2
A = [[1], [1]]
Y = [[0.01, 0.0], [0.0, 0.1]]
E = [0.3, 0.0]
pn = [1, 0]
```

n、b、$\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$ については、これまで何度も出てきたものであるから復習を兼ねて、図 2.7.3 と見比べながら、その意味を考えてほしい．pn=[1, 0]は、ブランチ 1 が順方向に接続されたダイオードであることを示す．入力となる電圧は、

```
inputs = mm.linspace(0, 2, 101)
Vs, Is = [], []
itr = 300; lr=0.1
```

で、0～2V の 0.02V 刻みの 101 個の値と結果を記録する器を用意するとともに、繰り返し計算の回数と、その中で使う移動平均の学習率を指定する．for ループの中だが、その中にもう一つ for ループがある．外側の for ループは、inputs に用意した電圧から一つずつ取り出して、それを

```
E[1] = inp
```

でブランチ 2 の電圧源として設定する．その後の、

```
V_ppl = [0.5 for i in range(b)]
```

は、ブランチ電圧の移動平均のベクトルを 0.5 で初期化する．0.5 にしておくと多少収束が早いようだ．そのあとに内側の for ループがあり、この内側の for ループがダイオードに関連して収束計算を担う．

```
for i in range(itr):
    W, V, I, P = Analize(A, Y, E)
    V_ppl = mm.sub(V_ppl, mm.mul(lr, mm.sub(V_ppl, V))) # 移動平均
    Y = Diode(Y, pn, V_ppl, offset=Th)
```

ループの中では関数 Analize()で、$\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$ からノード電位 $\boldsymbol{W}$、ブランチの電圧 $\boldsymbol{V}$、電流 $\boldsymbol{I}$、電力 $\boldsymbol{P}$ を算出するが、ここで E[1]には先に設定した入力電圧が設定されている．いっぽう $\boldsymbol{Y}$ は、はじめは関数 Diode()によるダイオードのアドミタンスが反映されていない．そして、電圧 $\boldsymbol{V}$ の移動平均 V_{ppl}を求めている．これは学習率を lr として移動平均を求める式

$$V_{ppl} = V_{ppl} - lr \times \left(V_{ppl} - V\right) \tag{2.7.4}$$

にしたがい、MatMath の関数を使ってベクトルで求めている．そして求まった V_{ppl}を関数 Diode()に $\boldsymbol{Y}$、pn、offset=Th とともに引数として渡して、ダイオードを反映した新たなアドミタンス $\boldsymbol{Y}$ を得て、その新たな $\boldsymbol{Y}$ を用いて再び関数 Analize()から繰り返す．このときに、電圧 $\boldsymbol{V}$ をそのまま使わずに、移動平均 V_{ppl}にすることが、この繰り返し計算を収束に導くためになくてはならない．あとは、結果を記録し、ダイオードのブランチ 1 の電圧と電流を出力して、最後にグラフ表示して終わる．

ここでまずはこの収束計算がうまくいっているかどうかを確かめておこう．次の 2 行が電圧と電流を記録するが、

```
Vs.append(V[0])
Is.append(I[0]*100)
```

この 2 行はリスト 7.3 では内側の for ループの中にある．だから、収束計算の毎回の結果が Vs、Is に記録される．そしてこれにより収束の様子が確認できる．この結果を図 2.7.4 に示す．なお電流は記録の際に×100 しているから 10mA の単位である．

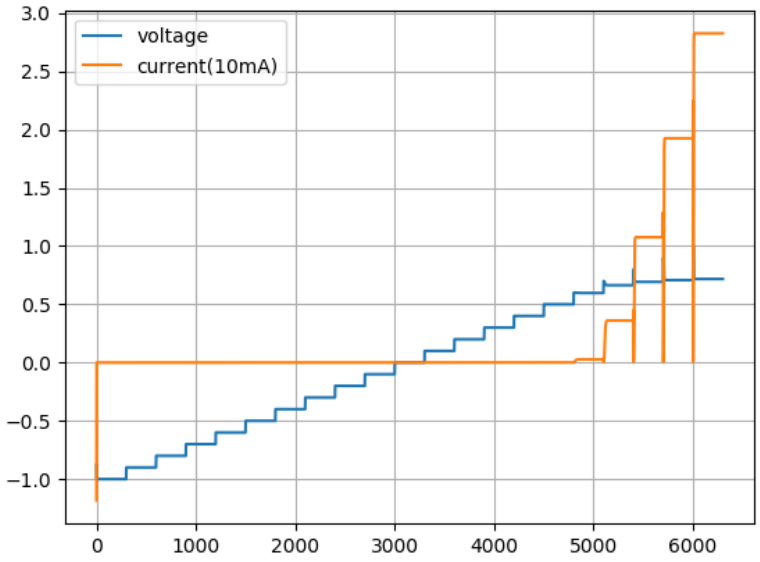

図 2.7.4 ダイオードを回路に組み込んだ結果 1

図 2.7.4 で横軸は -1～1 の 21 個のデータの一つのデータにつき 300 回ずつ計算したことに対応して 0～6300 となっている．さて電圧の変化は一番初めにアドミタンスの初期値の影響で大きな段がつくものの、その後は階段状にきれいに上がっていく．そして 5400 回から階段にとんがりがついて、段差が小さくなっていく．これは 0～5400 の手前までは収束計算がいらないのに対し、5400～6300 の範囲では収束計算によって正しい電圧が得られていることを示す．いっぽう電流も、0～4800 の範囲でほとんど 0(グラフを拡大すると 4500～4800 でも少し流れている)だが、5100～6300 では急激に電流が増えていて、かつ、各データに対応する 300 回の計算のはじめはオーバーシュートのようになっているがすぐに収束していることがわかる．ちなみに関数 Diode()に移動平均の V_{ppl} ではなく、もとの $\boldsymbol{V}$ を与えると、得られ

る Y は極端な値となって振動し続けるのが観測される．ここでは、繰り返し回数 itr=300、学習率 lr=0.1 としたが、これらの値とともに、変えてみてどうなるかを試してみてほしい．

次に、電圧と電流を記録する次の 2 行、

```
Vs.append(V[0])
Is.append(I[0]*100)
```

このインデントを 1 段左にずらすと、今度は、収束計算が終わったあとの結果が記録される．そして、次の 3 行のコメントアウトしてある行と、そうでない行のコメント#を入れ替える．

```
plt.plot(Vs, label='voltage')          →    #plt.plot(Vs, label='voltage')
plt.plot(Is, label='current(10mA)')    →    #plt.plot(Is, label='current(10mA)')
#plt.plot(Vs, Is, label='V-I')         →    plt.plot(Vs, Is, label='V-I')
```

そうすると今度は、ダイオードの電圧電流特性をあらわすグラフが得られる．グラフを滑らかにするために、次の行で指定するデータの数を増やして粒度も小さくする．

```
inputs = mm.linspace(-1, 1, 21)        →    inputs = mm.linspace(-1, 1, 101)
```

そうして図 2.7.5 を得る．

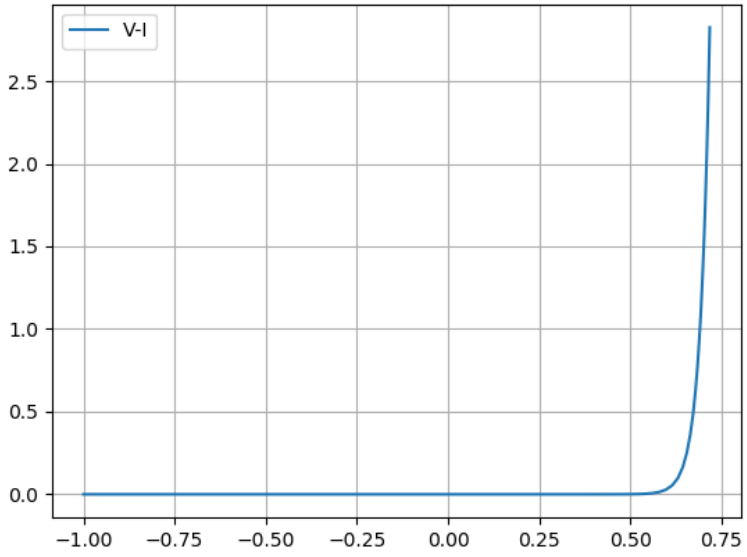

図 2.7.5 ダイオードを回路に組み込んだ結果 2

当然ながら、きれいなダイオードの特性図となっている．なお縦軸は 10mA の単位であり、グラフはおおよそ 0～300mA の範囲を示している．上記で説明しなかったが、プログラムの冒頭で、

```
E = [0.3, 0.0]
```

としたが、このベクトル E[0]は、ダイオードをあらわすブランチ内部の電圧源である．この値を変えると、それに応じて図 2.7.5 に示すダイオードの電圧電流特性を左右にずらすように変化させることができる．つまり電圧電流特性の電流が流れ始める電圧にオフセットを入れることができる．せっかくダイオードの特性を理論通りに実装したことに反するようだが、理論を尊重しつつもモデルとして現実に即したものを実現するための妥協として許される範囲だろう．そしてこれは後にトランジスタや FET を組み込む際に役に立つ．ともあれ、ダイオードそのものが理想的なものであるのは変わりないが、現実的な特性グラフとなった．ここでは簡単のため、電源の電圧を変えるようにしたが、より現実的には図 2.1.7 でモデル化したように電源は乾電池として電圧源は固定で、図 27.3 の 10Ωの固定抵抗の代わりに 1kΩ程度の可変抵抗とするのが良いだろう．ここでは割愛するが、リスト 2.7.3 をもとにすれば簡単に試すことができるからやってみてほしい．

[補足 2.7.1]

リスト 2.7.3 で Y=[[0.01, 0.0],[0.0, 0.1]]としている．行列 $\boldsymbol{Y}$ の要素 Y[0][0]は図 2.7.3 のダイオードに対応する．そしてその値を Y[0][0]=0.01 としている．ダイオードだから、この値は初期値であって、あとから関数 Diode()によって上書きされて、アドミタンスが電圧に応じた値に決まる．だから Y[0][0]の値はなんでも良さそうだ．実際に Y[0][0]=0.0 としても、このプログラムは問題なく走る．しかし $\boldsymbol{Y}$ の対角成分に 0 が置かれるのは注意が必要だ．それというのも複雑な回路の場合には、解析がうまくいかなくなってしまう恐れがあるからだ．リスト 2.5.1 の関数 Analize()の中で、L = mm.inv(tAYA)となっているところがある．これは式 2.5.4 に対応し、回路解析の要である．ここで逆行列を求める対象は、行列 ${}^t\boldsymbol{AYA}$ なのだが、$\boldsymbol{Y}$ の対角成分に 0 があると、場合によっては ${}^t\boldsymbol{AYA}$ の対角成分が 0 となって、行列が正則ではなくなり、逆行列が求められなくなってしまう場合がある．これはいつでもそうなるわけではないが、あらかじめダイオードに対応するブランチのアドミタンス $\boldsymbol{Y}$ の初期値を非 0 にしておけば避けられることなので、そうすべきである．

あとさきになるが、リスト 2.3.2 で示した関数 set_branch()をダイオードも入力できるように拡張しておこう.

リスト 2.7.4：ブランチのダイオードを含む拡張

```
def set_branch(b, diode=False):
    Y = [[0.0 for j in range(b)] for i in range(b)]
    E = [0.0 for i in range(b)]
    pn = [0.0 for i in range(b)]
    print('Input spec of each branch.')
    for k in range(b):
        print('Branch#' + str(k+1))
        if diode:
            Dtype = input('If this is a diode, specify the type [diode/LED] => ')
        else:
            Dtype = None
        if Dtype in ('led', 'LED', 'l', 'L'):
            print('LED')
            pn[k] = 1
            e = 1.4
        elif Dtype in ('diode', 'Diode', 'd', 'D'):
            print('diode')
            pn[k] = 1
            e = 0.3
        else:
            e = float(input('electro motive force  '))
            r = float(input('electrical resistance '))
            y = 1/r
        Y[k][k] = y
        E[k] = e

    if diode:
        return Y, E, pn
    else:
        return Y, E
```

従来の関数と互換のためにデフォルト引数 diode=False として、diode=True が与えられた場合に限って、返り値に pn が加えられるようにした. また LED も電子回路で使われることが多いから、ダイオードの１種として電圧オフセットを変えて入力できるようにした.

2.8 トランジスタ

電圧制御電流源を節 2.6 で、そしてダイオードを節 2.7 で、組み込むことができたので、いよいよトランジスタを組み込んでいくことにしよう．これによって増幅回路などを作ることができる．ところでトランジスタにはいくつかの種類があるが、ここではバイポーラトランジスタと、電界効果トランジスタ（Field effect transistor，FET）、その中でも接合型 FET の 2 つについて実装していくこととする．そして以降、単にトランジスタといえば、バイポーラトランジスタを指し、FET は接合型電界効果トランジスタを指すことにする．また、トランジスタや FET の構造や動作原理については他に譲ることとして、ここではトランジスタや FET を図示する場合の記号を示すにとどめることにする．

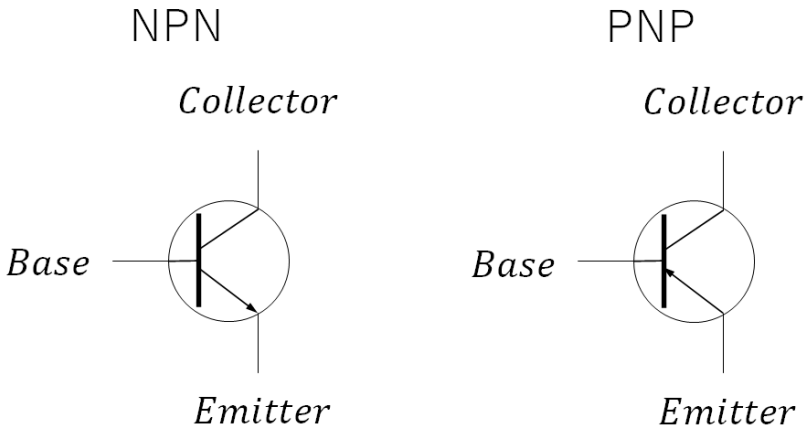

図 2.8.1 トランジスタをあらわす記号

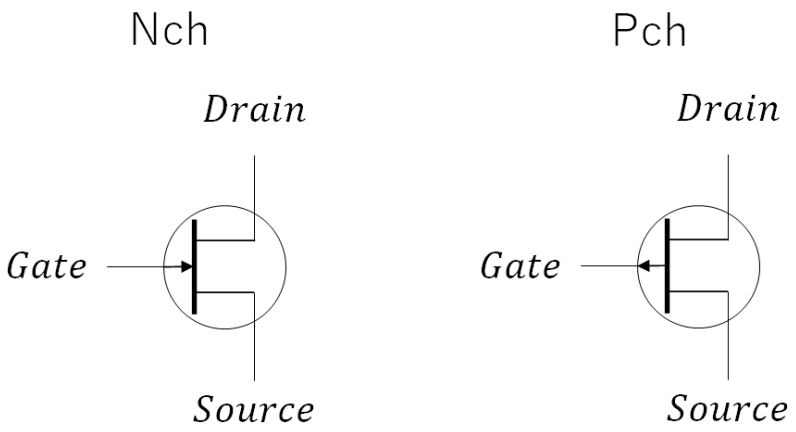

図 2.8.2 FET をあらわす記号

2.8.1 トランジスタの等価回路

トランジスタをモデル化するのに、用途によって応じて様々な等価回路が提案されているが、ここでは主に直流特性に焦点をあてて、図 2.8.3 に示すように等価回路を決めることにする.

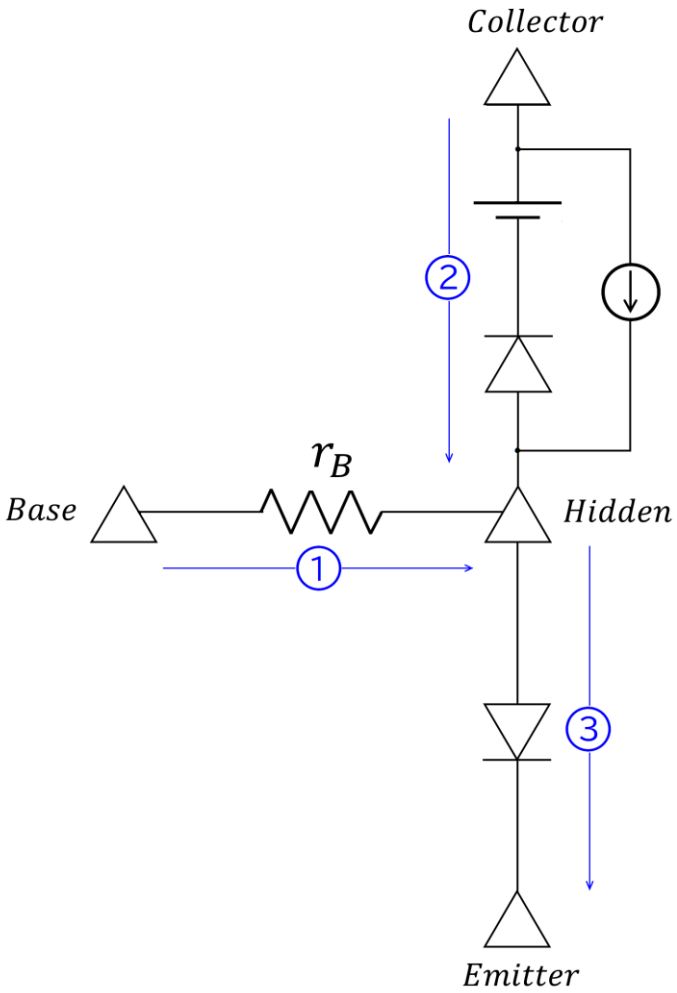

図 2.8.3 トランジスタの等価回路

図 2.8.3 の等価回路には、外部との接点となるノードが 3 つと、内部に閉じたノードが 1 つで合計 4 つのノードがある. 外部との接点はトランジスタの 3 つの端子、ベース、コレクタ、エミッタであり、内部のそれは Hidden とした. いっぽうブランチは 3 つあり、それぞれ始点ノード→終点ノードとしてあらわすと、ブランチ 1 はベース→Hidden、ブランチ 2 はコレクタ→Hidden、ブランチ 3 は Hidden→エミッタである.

この図に示すのは図 2.8.1 の左図の NPN 接合型トランジスタで、ベースが P 型半導体、コレクタとエミッタが N 型半導体となっている．図 2.8.1 右図の PNP 接合型トランジスタでは、ベースが N 型半導体、コレクタとエミッタが P 型半導体となっている．PNP の場合には、ブランチ 2 の電圧源の向きとダイオードの向き、および、ブランチ 3 のダイオードの向きがいずれも図 2.8.3 とは逆になる．

抵抗やダイオードにはないトランジスタの動作として、ブランチ 2 の電圧制御電流源がある．この電流を制御するのはブランチ 1 であるが、ブランチ 1 の両端電圧にしたがって、電流が増減する．トランジスタのコレクタ・エミッタ間に流れる電流 I_{ce} の大きさは、ベースに流れ込む電流 I_{be} に依存し、この比を示す値を直流電流増幅率と呼び h_{fe} であらわす．すなわち、

$$h_{fe} = \frac{I_{ce}}{I_{be}} \tag{2.8.1}$$

しかしここでは、電流 I_{be} をブランチ 1 の抵抗での電圧降下によって電圧に換算したうえで、このブランチ 1 を制御ブランチとした場合の相互コンダクタンスを回路解析に用いることにする．ブランチ 1 の両端電圧を V_{bh} 抵抗を r_b とおけば、

$$I_{be} = \frac{V_{bh}}{r_b} \tag{2.8.2}$$

だから式 2.8.1 は次のように変形できる．

$$I_{ce} = h_{fe} I_{be} = h_{fe} \frac{V_{bh}}{r_b} \tag{2.8.3}$$

ここで、

$$g_m = \frac{h_{fe}}{r_b} \tag{2.8.4}$$

とおけば式 2.8.3 は、

$$I_{ce} = g_m V_{bh} \tag{2.8.5}$$

式 2.8.5 は相互コンダクタンスと電圧の積になっており、I_{ce} を担うブランチ 2 の電圧制御電流源は、相互コンダクタンスを式 2.8.4 の g_m として、ブランチ 1 の電圧によって制御されることとなる．

2.8.2 トランジスタを回路に組み込む

2.8.1 でトランジスタの等価回路が決まったので、これを回路に組む込んでいくこととする．まずはトランジスタの各端子と内部の Hidden ノードの接続を行うためのプログラムだが、これをリスト 2.8.1 に示す．

リスト 2.8.1：トランジスタの接続

```
def set_A_trs(n, t):
    A = [[0 for j in range(n+t)] for i in range(3*t)]
    print('各Transistorの接続ノードを入力せよ')
    for k in range(t):
        print( str(k+1) + '番目のTransistor')
        base = int(input('Node where Base connected      => '))
        emit = int(input('Node where Emitter connected   => '))
        clct = int(input('Node where Collector connected => '))
        if base > 0:
            A[3*k][base-1]   = 1
        if clct > 0:
            A[3*k+1][clct-1] = 1
        if emit > 0:
            A[3*k+2][emit-1] = -1

        # hidden node is n+k
        A[3*k][n+k]   = -1 # base to hidden
        A[3*k+1][n+k] = -1 # collector to hidden
        A[3*k+2][n+k] =  1 # hidden to emitter

    return A
```

リスト 2.8.1 の関数 set_A_trs()は、入力にしたがって回路の接続関係を示す行列 $\boldsymbol{A}$ を作る．引数にトランジスタ数 t だけでなく、ノード数 n を与える．この n はトランジスタを組み込む回路のトランジスタ以外の部分のノード数である．トランジスタの各端子はトランジスタ以外から、すなわち抵抗や電圧源などからなる回路のどこかのノードに接続され、またトランジスタ内部の Hidden ノードは、これらの後に新たなノードとして追加されるから、すでに定義されたノードの数が必要となるのである．関数の中ではじめに行列 $\boldsymbol{A}$ を 0 で初期化する．このとき、ブランチはトランジスタ 1 個につき 3 つ必要で、いっぽうノードは、トランジスタ 1 個につき 1 つの Hidden ノードが必要なので、行列 $\boldsymbol{A}$ の大きさは行数 3×t、列数 n+t である．そしておさらいすると、行列 $\boldsymbol{A}$ の要素は始点ノードと終点ノードについて、

A[ブランチ番号][始点ノード番号] = 1

A[ブランチ番号][終点ノード番号] = -1

と±1 である．これをふまえてトランジスタの 3 つの端子の接続ノードにしたがってすべての要素が 0 で初期化されている行列 $\boldsymbol{A}$ に値を入れて完成させていく．for ループでトランジ

スタ 1 個ずつ接続ノードの入力を要求する．入力されるノードの番号は 1 から始まるのに対し、プログラムの中では 0 から始まるので、入力された数値を -1 した数値が接続点のノード番号となる．いずれの端子もノード 0 すなわちグランドへの接続は行列 $\boldsymbol{A}$ にあらわれないから、それを判断したあとで、ベースとコレクタは接続点がブランチの始点なので 1、エミッタは接続点が終点なので -1 を行列 $\boldsymbol{A}$ にセットする．ループ中で k 番のトランジスタを扱うときに、各ブランチ 3*k 番、3*k+1 番、3*k+2 番はそれぞれ図 2.8.3 の等価回路に示すブランチ 1、2、3 に対応する．ここもプログラム内では番号が 0 から始まっていることに注意が必要だ．また、Hidden ノードに関しては、図 2.8.3 のブランチ 1、2 の終点で -1、ブランチ 3 の始点で 1 を行列 $\boldsymbol{A}$ にセットする．これで行列 $\boldsymbol{A}$ が出来上がるので、これを返り値として関数 set_A_trs()は完了する．

続いて、ブランチの設定を行うためのプログラムをリスト 2.8.2 に示す．

リスト 2.8.2：トランジスタのブランチの設定

```
def set_branch_trs(t, rb=50, Thc=0.2, The=0.2):
    Y = [[0.0 for j in range(3*t)] for i in range(3*t)]
    E = [0.0 for i in range(3*t)]
    pn = [0 for i in range(3*t)]

    print('各 Transistor のデータを入力せよ')
    for k in range(t):
        print( str(k+1) + '番目の Transistor')
        NPN  = -1 if input('Type(NPN/PNP)') in ('p', 'P', 'pnp', 'PNP') else 1
        print('NPN =', NPN)
        hfe  = float(input('current amplification factor (hfe) '))

        # branch base to hidden
        Y[3*k][3*k]     = 1/rb            # rb:input resistance of base

        # branch collector to hidden
        pn[3*k+1]       = -NPN            # backward connection
        E[3*k+1]        = Thc*NPN         # threshold
        Y[3*k+1][3*k+1] = 0.01

        # Transconductance
        Y[3*k+1][3*k]   = hfe/rb          # hfe -> Transconductance

        # branch hidden to emitter
        pn[3*k+2]       = NPN             # forward connection
        E[3*k+2]        = The*NPN         # threshold
        Y[3*k+2][3*k+2] = 0.01

    return Y, E, pn
```

リスト 2.8.2 の関数 set_branch_trs()は、リスト 2.8.1 の関数 set_A_trs()と対になって、トランジスタを回路に組み込むが、set_A_trs()がトランジスタの接続を作るのに対し、set_branch_trs()はトランジスタ内部のブランチの諸元を入力にしたがって設定する．基本的な引数は回路内のトランジスタ数 t で、このほかの引数はデフォルト値をそのまま使うことを想定している．以下図 2.8.3 を参照しながら説明を読んでほしい．

関数の中では、まず $\boldsymbol{Y}$、 $\boldsymbol{E}$、pn を引数 t すなわち、トランジスタ数に応じた大きさで初期化する．各トランジスタのブランチの設定は for ループの中で 1 つずつ行うが、入力を求めるのはトランジスタの極性が NPN なのか PNP なのかと、トランジスタの電流増幅率 h_{fe} であり、他は基本的に引数のデフォルト値を使ってブランチ諸元を確定する．

ブランチ 1 についてはデフォルト引数 rb をアドミタンスに換算して $\boldsymbol{Y}$ に設定するだけだ．

続いてブランチ 2 については、入力されたトランジスタの極性に応じて $pn = 1$ または $pn = -1$ に設定してダイオードであることを極性とともに指定する．それから起電力 $\boldsymbol{E}$ をデフォルト引数 Thc=0.2 を受けて設定するが、このときトランジスタの極性を配慮して Thc*NPN としている．アドミタンス $\boldsymbol{Y}$ は初期値として 0.01 を設定しておく．ブランチ 2 には電圧制御電流源もあるから、この相互コンダクタンスを設定する．このとき入力されるのは h_{fe}だから、式 2.8.4 によって g_m に変換して設定する．

残るブランチ 3 についてだが、まず入力されたトランジスタの極性に応じて $pn = 1$ または $pn = -1$ に設定してダイオードであることを極性とともに指定する．それからデフォルト引数 The=0.2 で起電力 E を設定する．ブランチ 2 同様に $\boldsymbol{Y}$ も 0.01 を初期値として設定しておく．以上の諸元の設定の際のインデクスだが、ループは、

```
    for k in range(t):
```

でトランジスタに対して 0 から始まる番号が k に入っている．ブランチはトランジスタ 1 つにつき 3 つあるから、k の指すトランジスタのブランチは、ブランチ 1 が 3*K、ブランチ 2 が 3*k+1、ブランチ 3 が 3*k+2 となる．したがってたとえば、相互コンダクタンスを設定する対象の $\boldsymbol{Y}$ は、ブランチ 1 がブランチ 2 の電圧制御電流源を制御するから、Y[3*k+1][3*k]となる．すべてのトランジスタの諸元が設定し終わると $\boldsymbol{Y}$、$\boldsymbol{E}$、pnを返り値として、関数 set_branch_trs()が完了する．

回路を解析するにあたって、関数 set_A_trs()の返り値 $\boldsymbol{A}$ や、関数 set_branch_trs()の返り値 $\boldsymbol{Y}$、$\boldsymbol{E}$、pnは、トランジスタ以外のそれとマージする必要があるが、それについては後述する．

2.8.3 FETの等価回路

本節2.8ではじめに述べたように、ここでは接合型FETを扱うことにして、その等価回路を示す．トランジスタ同様に、アナログ回路での実用性を考えて、図2.8.4に示すようにした．

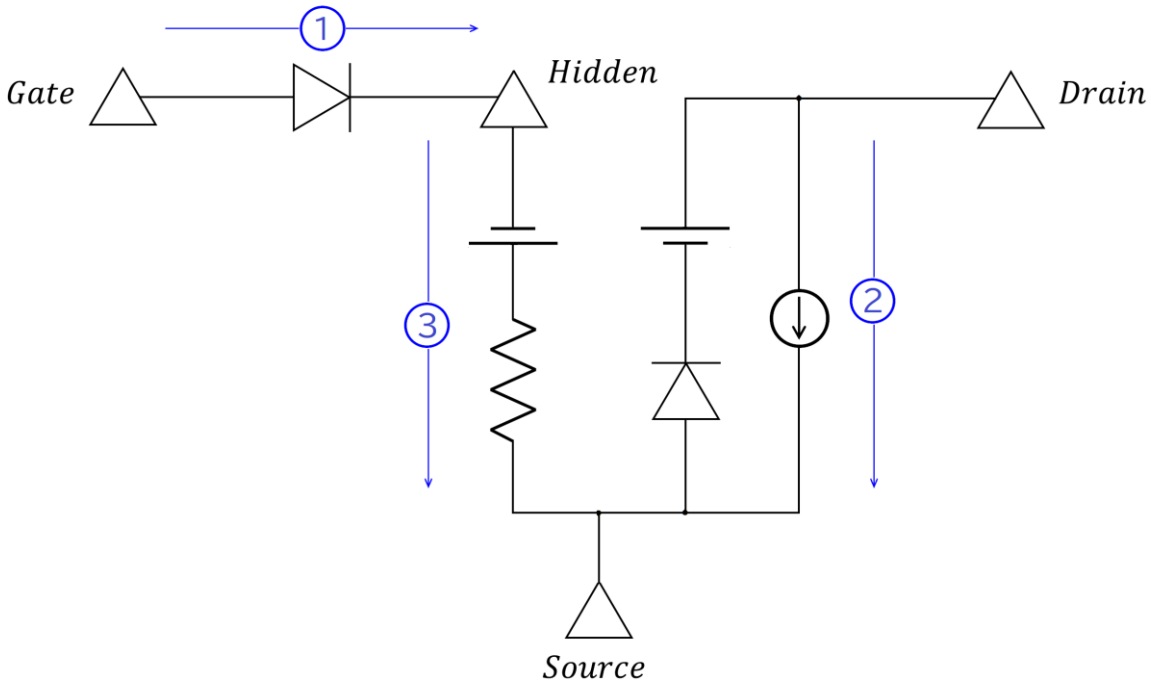

図2.8.4 FETの等価回路

図2.8.4の等価回路には、外部との接点となるノードが3つと、内部に閉じたノードが1つで合計4つのノードがある．外部との接点はFETの3つの端子、ゲート、ドレイン、ソースであり、内部のそれはHiddenとした．いっぽうブランチは3つあり、それぞれ始点ノード→終点ノードとしてあらわすと、ブランチ1はゲート→Hidden、ブランチ2はドレイン→ソース、ブランチ3はHidden→ソースである．

この図に示すのは図2.8.2の左図のNch FETで、ドレイン、ソース間のN型のチャネルを、それとPN接合されるP型のゲートとなっている．図2.8.2右図のPch FETでは、チャネルがP型で、ゲートはN型である．Pchの場合には、ブランチ1のダイオードの向き、ブランチ2の電圧源の向きとダイオードの向き、および、ブランチ3の電圧源の向きがいずれも図2.8.4とは逆になる．

FETもトランジスタ同様にブランチ2に電圧制御電流源がある．この電流を制御するのがブランチ1なのもトランジスタ同様だ．しかし図2.8.3と図2.8.4に違いがあるように、動作が異なる．FETのドレイン・ソース間に流れる電流 I_{ds} の大きさは、ゲート・ソース間の電圧 V_{gs} に依存する．すなわち V_{gs} によって I_{ds} を制御する電圧制御素子なのである．ここでは、$V_{gs}=0$ のときに流れるドレイン飽和電流 I_{dss}(drain saturation current)を含めるように等価回路と、これに関係する式を決めることとする．
図2.8.4のブランチ3に設けた電圧源は、$V_{gs}=0$ のときにちょうど $I_{ds}=I_{dss}$ となるように、あらかじめHiddenノードのソースに対する電位を調整する役割を担う．すなわちHiddenノードの電位を、NchFETでは下げ、PchFETでは上げて、ドレイン・ソース電流を制御するブランチ1の電圧にオフセットを与えるのである．この電圧は逆向きの電圧源であるから $-V_{th}$ とおく．そして I_{ds} を制御するブランチ1の電圧を V_{gh} とおけば、

$$V_{gs}=V_{gh}-V_{th} \tag{2.8.6}$$

そして、

$$\begin{aligned} I_{ds} &= g_m\ V_{gh} \\ &= g_m(V_{gs}+V_{th}) \end{aligned} \tag{2.8.7}$$

$V_{gs}=0$ のときの I_{ds} が I_{dss} なのだから、

$$I_{dss}=g_m\ V_{th} \tag{2.8.8}$$

よって、

$$V_{th}=\frac{I_{dss}}{g_m} \tag{2.8.9}$$

I_{dss} の値はFETの品種によって違い、また個々のチップによっても異なるが、データシートなどで $V_{gs}=0$ のときの g_m の値とともに得られることが多い．だから式2.8.9によって V_{th} を得て、式2.8.7と併せて図2.8.4の等価回路でFETの動作をあらわすことができる．
ところでこの等価回路では、図2.8.3のトランジスタ同様に、ブランチ2に電圧制御電流源に並列に逆接続のダイオードを設けている．これはFETの構造上は存在しないダイオードだが等価回路の実用性のために設けた．式2.8.7で V_{gs} を大きくしていくと、係数 g_m にしたがって I_{ds} が際限なく大きくなっていく．そうすると増幅回路などで回路に供給される電圧によって頭打ちになるはずの電流を超えて電流が増え続けてしまうが、これを避けるために役立つ．流れる電流による電圧降下で、ドレインの電位がソースのそれを下回るようになった場合に、ブランチ2のダイオードが短絡する．そうすると電圧制御電流源の電流をブラン

チ 2 の中で受けて、ブランチの外にはそれ以上は電流が流れなくなり、結果的に外からドレインに流れ込む電流が頭打ちになるのである.

実はもう一つ実用上対処が必要な問題がある．FET のゲートには逆バイアスを加えるのは通常の動作の範囲で普通に行われることで、それゆえに FET の入力インピーダンスは大きくて、FET の特徴の一つとなっている．しかしゲートに加えられる逆方向の電圧が大きくなっていった場合の挙動が、このままではうまくないのである．ややこしいので図 2.8.4 の Nch の場合で説明する．ゲート・ソース間の電圧 $V_{gs}=0$ ならば、ドレイン・ソース間電流は I_{dss}となる．そしてこのときブランチ 1 の電圧 $V_{gh}=V_{th}$ である．そこからゲートの電位を下げていくと、やがてブランチ 1 の電圧 $V_{gh}=0$ となる．そしてこのときドレイン・ソース間電流 $I_{ds}=0$ となる．この状態は実際の FET ではピンチオフと呼ばれ、電流を担うキャリアが無くなって、ドレイン・ソース電流が流れなくなった状態である．当然ながら、さらにゲート電位を下げていっても、やはりドレイン・ソース電流は流れない．しかし式 2.8.7 をモデル化した等価回路では、ゲート電位を下げていくと、ブランチ 1 の両端電圧にしたがって、電圧制御電流源が際限なく逆方向の電流を生じてしまう．これでは現実の回路の動作と大きく乖離してしまうから、V_{gh} が逆方向の場合にはドレイン・ソース電流が流れないようにしなくてはならない．これを式であらわすと次のようになる．なお次式は NchFET の場合である.

$$I_{ds}=\begin{cases} g_m V_{gh} & (V_{gh}>0) \\ 0 & (V_{gh}\leqq 0) \end{cases} \tag{2.8.10}$$

この式は $V_{gh}=0$ で不連続で、こういうかたちの関数をランプ関数という．これをそのまま等価回路上にあてはめて扱うことは難しいが、回路解析の過程でこの対処を行うこととする．これまでの解析の枠組みを考慮して式 2.8.10 は次のように読み替えておこう.

$$I_{ds}=g'_m V_{gh} \qquad g'_m=\begin{cases} g_m & (V_{gh}>0) \\ 0 & (V_{gh}\leqq 0) \end{cases} \tag{2.8.11}$$

つまり g_m をステップ関数にするのであるが、具体的な対処方法については後述する．一見簡単な FET ではあるが、トランジスタにはない難しさがあって、モデルとして何を評価したいかによるが工夫が必要となる．いずれにせよ説明してきたのは、あくまでも FET のデバイスの特性を正確に表すものではなく、回路に簡単に組み込んで、その回路中での挙動が実用になるかどうかの観点でまあまあ使えるといった類いのものである.

[補足 2.8.1]
ドレイン・ソース間電圧を高くしていった場合に、空乏層が広がってチャネル幅が減少していき、ついに空乏層によってチャネルが埋められて、それ以上ドレイン電流が増えない飽和領域に達するところもピンチオフと呼ぶ．これは明らかにここで使ったピンチオフと違う現象である．紛らわしいので両者を区別して、ここでのピンチオフはゲートピンチオフと呼ぶ場合がある．

[補足 2.8.2]
等価回路におけるピンチオフは $V_{gh}=0$ で $I_{ds}=0$ となった状態とした．このとき、$V_{gs}=V_{th}$ となっている．ところが実際の FET ではピンチオフは $V_{gs}=V_p$ で起きる． 次の補足 2.8.3 で詳述するが、FET の特性をどう近似するかの問題である．

$V_{th}=\frac{V_p}{2}$ の関係となっているが、直線近似による近似では、こうするのが妥当と考える．

[補足 2.8.3]
飽和領域ではドレイン・ソース間電流は二乗近似が良くあてはまり、ピンチオフ電圧を V_p とおいて V_{gs} と I_{ds} の関係は次式であらわされる．

$$I_{ds}=I_{dss}\left(1-\frac{V_{gs}}{V_p}\right)^2 \tag{2.8.12}$$

いっぽう g_m はゲート・ソース間電圧 V_{gs}に対するドレイン電流の変化の大きさとして定義されるから、これを微分して、

$$g_m=\frac{\partial I_{ds}}{\partial V_{gs}}=-\frac{2I_{dss}}{V_p}\left(1-\frac{V_{gs}}{V_p}\right) \tag{2.8.13}$$

ここで $V_{gs}=0$ での g_m を g_{m0} とおくと、

$$g_{m0}=-\frac{2I_{dss}}{V_p} \tag{2.8.14}$$

I_{dss} の値は、それと $V_{gs}=0$ で測定条件を合せた g_m の値、すなわち g_{m0} とともに提供されることが多い．これがあればピンチオフ電圧 V_p を知ることが出来る．

$$V_p=-\frac{2I_{dss}}{g_{m0}} \tag{2.8.15}$$

そして I_{dss} と V_p がわかれば、飽和領域においては任意のゲート・ソース間電圧 V_{gs}に対するドレイン電流を式 2.8.12 によって求めることが出来る．

いっぽう等価回路では、ブランチ 1 とブランチ 3 が直列になっていて、式 2.8.6 で V_{gs} を V_{gh} と V_{th} に分けた．そして V_{th} を式 2.8.9 により I_{dss} から定まる定数とし、常に $g_m = g_{m0}$ であるものとして、式 2.8.7 で I_{ds} があらわされるとした．すなわち V_{gh} に対して g_m（実際には g_{m0}）を係数として直線で I_{ds} を近似したのである．これはちょうど上記の二乗近似に $V_{gs} = 0$ のところで接する直線となる．$V_p = 2\,V_{th}$ の関係になっていることも確かめられる．そして最後に式 2.8.10 ないし 2.8.11 でピンチオフの対応をしたのである．この様子を確認しておこう．リスト 2.8.3 にそのためのプログラムを図 2.8.5 にその結果を示す．ここではプログラムの説明は省略するが、プログラム中の数式をここでの説明と見比べてほしい．

リスト 2.8.3：FET の特性

```
import MatMath as mm
import matplotlib.pyplot as plt

Vgs = mm.linspace(-1.5, 0.5, 101)
idss = 0.01
gm0 = 0.02
vp = - 2 * idss / gm0
vth = idss / gm0

Ids, Ids2 = [], []
for vgs in Vgs:
    ids = idss * (1 - vgs / vp) ** 2 if vgs > vp else 0

    vgh = vgs + vth
    gmm = gm0 if vgh > 0 else 0
    ids2 = gmm * vgh

    Ids.append(ids)
    Ids2.append(ids2)

plt.plot(Vgs, Ids, label='Ids')
plt.plot(Vgs, Ids2, label='Ids2')
plt.grid()
plt.legend()
plt.show()
```

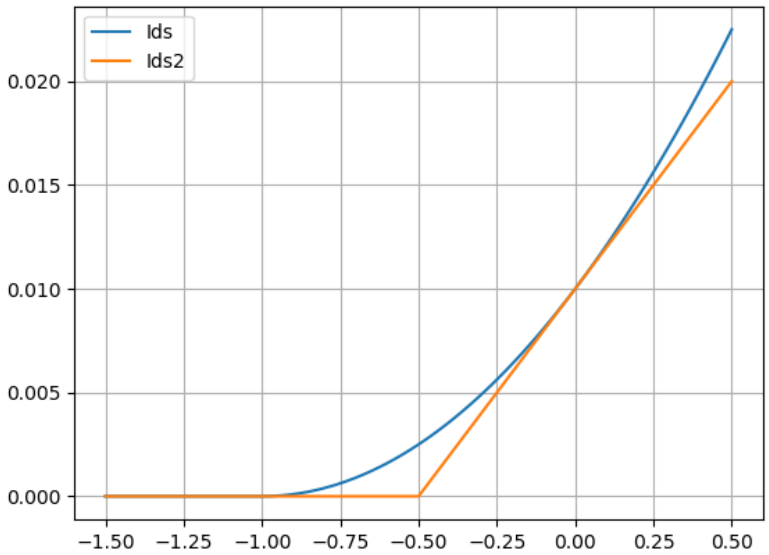

図 2.8.5 FET の特性

図 2.8.5 の横軸はゲート・ソース間電圧 V_{gs} 縦軸はドレイン・ソース間電流 I_{ds} である. そして二乗近似と、図 2.8.4 に対応する直線近似の 2 つが描かれている. 二乗近似に対して、縦軸の切片 $I_{dss} = 0.01$ 横軸の切片 $V_{th} = \frac{V_p}{2} = -0.5$ の直線で 1 次近似していて、$V_{gs} = 0$ で二次曲線に接していることがよくわかる. $V_{gs} = 0$ の近傍では直線近似でも二乗近似と隔たりは少ない. いっぽうここから離れて、たとえば $V_{gs} = \frac{V_p}{2}$ ～ あたりでは、二乗近似では I_{ds} がまだ流れているが直線近似では 0 となっている. だから FET がカットオフするあたりでの挙動はあまりうまく表せないことを念頭においておく必要がある.

2.8.4 FET を回路に組み込む

2.8.3 で FET の等価回路が決まった．そこで、トランジスタ同様に、回路に組み込んでいくこととする．まずは接続関係をあらわす行列 $\boldsymbol{A}$ を作る．これをリスト 2.8.4 に示す．

リスト 2.8.4：FET の接続

```
def set_A_fet(n, t):
    A = [[0 for j in range(n+t)] for i in range(3*t)]
    print('各 FET の接続ノードを入力せよ')
    for k in range(t):
        print( str(k+1) + '番目の FET')
        gt   = int(input('Node where Gate connected     '))
        so   = int(input('Node where Source connected   '))
        dr   = int(input('Node where Drain connected    '))
        if gt > 0:
            A[3*k][gt-1]   = 1
        if so > 0:
            A[3*k+1][so-1] = -1
            A[3*k+2][so-1] = -1
        if dr > 0:
            A[3*k+1][dr-1] = 1

        # hidden node is n+k
        A[3*k][n+k]   = -1
        A[3*k+2][n+k] =  1

    return A
```

リスト 2.8.4 の関数 set_A_fet()は、リスト 2.8.1 の関数 set_A_trs()のトランジスタの場合とほとんど同じである．違うのは、端子の名称と Hidden ノードの接続だけである．トランジスタの場合同様に、for ループで FET 1 個ずつ、ゲート、ソース、ドレインの接続ノードの入力を求めて、変数 gt、so、dr に入れ、それぞれ-1 した値をおのおののノード番号として、行列 $\boldsymbol{A}$ の要素にアクセスして、始点ノードは 1、終点ノードは -1 をその $\boldsymbol{A}$ の要素の値として設定する．トランジスタの場合と違ってソースにブランチ 2 と 3 がいずれも終点としてつながることや、Hidden ノードは、ブランチ 1 の終点かつブランチ 3 の始点であることを考えれば、特に難しいところはないはずだ．

続いて、FET を構成するブランチの設定だが、これをリスト 2.8.5 に示す.

リスト 2.8.5：FET のブランチの設定

```
def set_branch_fet(t, Thd=0.5, yhs=0.857):
    Y = [[0.0 for j in range(3*t)] for i in range(3*t)]
    E = [0.0 for i in range(3*t)]
    pn = [0 for i in range(3*t)]

    print('各 FET のデータを入力せよ')
    for k in range(t):
        print( str(k+1) + '番目の FET')
        Nch  = -1 if input('Type(N/P)') in ('p', 'P', 'pch', 'Pch') else 1
        print('Nch =', Nch)
        gm   = float(input('Transconductance(milli-mho)    '))
        idss = float(input('Drain cut-off Curennt(Idss mA) '))
        vth = idss / gm              # Vth related to bias of gate

        # branch gate to hidden
        pn[3*k]         = Nch
        Y[3*k][3*k]     = 0.01

        # branch drain to source
        pn[3*k+1]       = -Nch       # backward connection
        E[3*k+1]        = Thd*Nch    # threshold
        Y[3*k+1][3*k+1] = 0.01

        # Transconductance
        Y[3*k+1][3*k]   = gm / 1000  # transconductance

        # branch hidden to source
        E[3*k+2]        = -vth*Nch   # threshold
        Y[3*k+2][3*k+2] = yhs        # 0.857 but must be less than rg*1000

    return Y, E, pn
```

リスト 2.8.5 の関数 set_branch_fet()は、トランジスタの場合のリスト 2.8.2 の関数 set_branch_trs()に相当する FET の内部のブランチの諸元を設定する関数である．対応する図は 2.8.4 である．関数は FET の数 t を基本的な引数として呼ばれ、他のデフォルト引数は、そのデフォルト値を使うことを想定している．FET の場合も、for ループの中で FET ごとに、ブランチの設定を行う．FET も極性があるので、それを入力として求めて、NchFET か PchFET かを変数 Nch=±1(NchFET なら 1、PchFET なら-1)であらわして使う．このほか g_mと I_{dss} を入力として求めるが、いずれも単位が milli-mho と mA なので、V_{th} は単位が相殺してそのまま割り算で求めておく．

ブランチ 1 は順接続のダイオードとして pn=Nch を設定し、Y=0.01 を初期値として設定しておく．ブランチ 2 についても、変数 Nch の値とデフォルトの電圧オフセットを使って、pn、$\boldsymbol{E}$ を逆接続のダイオードとして設定する．Y=0.01 も設定しておく．またブランチ 2 は電圧制御電流源の相互コンダクタンスを 1/1000 して単位を合わせて設定する．残るブランチ 3 は等価回路の説明で述べたように、先に I_{dss} と g_m から求めた V_{th}を電圧源に設定する．このとき電圧源の向きが NchFET ではマイナス(−)、PchFET ではプラス(+)であることに注意が必要だ．

すべての FET のブランチの諸元が設定し終わると $\boldsymbol{Y}$、$\boldsymbol{E}$、pnを返り値として、関数 set_branch_fet()が完了する．FET の場合も回路を解析するにあたって、$\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$、pn は、FET 以外のそれとマージする必要があるが、それについては後述する．

2.9 トランジスタや FET を含む回路の解析

節 2.8 までに、電圧源や抵抗からなる回路と、そこからの発展として、ダイオードやトランジスタ、そして、FET を回路に組み込む方法を検討し、そのための関数を用意した．ここからはこれらをまとめて一つの回路として解析できるようにしていこう．

2.9.1 トランジスタや FET とこれ以外の回路の結合

リスト 2.8.1 と 2.8.2 でトランジスタの、そしてリスト 2.8.4 と 2.8.5 で FET の、それぞれ接続関係をあらわす行列 $\boldsymbol{A}$ とブランチの諸元をあらわす行列並びにベクトルの $\boldsymbol{Y}$、$\boldsymbol{E}$、pnを入力にしたがって作る関数を用意した．そして遡ってリスト 2.3.2 で、トランジスタや FET 以外の電圧源と抵抗からなる部分について $\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$ を入力に応じて作る関数を用意した．リスト 2.7.4 ではダイオードにも対応した．これらの関数を使いつつ、回路全体をまとめるようにするにあたり、各行列やベクトルをどう連結したらよいかを検討しよう．
まず行列 $\boldsymbol{A}$ を作る関数は、set_A()、set_A_trs()、set_A_fet()の 3 つだ．引数としてノード数 N、ブランチ数 B、トランジスタ数 T、FET 数 F を用意してこれらの関数を呼んだ場合を考える．そして返り値の行列 $\boldsymbol{A}$ の行数と列数を、対象とするブランチとノードとともにまとめると、表 2.9.1 のようになる．

表 2.9.1

関数	対象となるブランチ	行数	対象となるノード	列数
set_A(N,B)	一般	B	一般	N
set_A_trs(N,T)	トランジスタ内部	3*T	一般と Hidden	N+T
set_A_fet(N+T,F)	FET 内部	3*F	一般と Hidden	N+T+F

set_A_trs()や set_A_fet()で行列 $\boldsymbol{A}$ を作るトランジスタや FET の内部は、それぞれその等価回路である図 2.8.3 と図 2.8.4 に示すように 1 個につき 3 つのブランチから構成されるので、素子の数に対して 3 倍のブランチがあって、これが行列 $\boldsymbol{A}$ の列数となる．いっぽうノードについては、トランジスタや FET の端子が回路中のどこか、すなわち一般のノードのどれかと接続されるから、その接続点である一般のノードを、内部の Hidden ノードとともに対象として行列 $\boldsymbol{A}$ を作る．そのため一般のノードを含み、さらに Hidden ノードが素子 1 個について 1 つだから、素子の数と同数のノードが追加される．そしてトランジスタと FET で Hidden ノードが重なってはいけないから、この表のようにトランジスタを FET よりも先に設定することとすれば、関数 set_A_fet()の引数のノード数は n=N+T としなければならない．それでは一般のブランチに関する行列を Ab、トランジスタに関して At、FET に関して Af とし

て、接続関係の行列 $\boldsymbol{A}$ を作る関数の返り値を見ていくことにする．なお、行列の要素のインデクスは行方向と列方向のそれが区別しやすいように、間に’,’(カンマ)をいれてあらわす．

関数 set_A(N,B)の返り値は、

$$\boldsymbol{Ab} = \begin{pmatrix} ab_{1,1} & \cdots & ab_{1,N} \\ \vdots & \ddots & \vdots \\ ab_{B,1} & \cdots & ab_{B,N} \end{pmatrix} \qquad (ZZZ.9.1)$$

関数 set_A_trs(N,T)の返り値は、

$$\boldsymbol{At} = \begin{pmatrix} at_{1,1} & \cdots & at_{1,N} & at_{1,N+1} & \cdots & at_{1,N+T} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ at_{3*T,1} & \cdots & at_{3*T,N} & at_{3*T,N+1} & \cdots & at_{3*T,N+T} \end{pmatrix} \qquad (2.9.2)$$

関数 set_A_fet(N+T,F)の返り値は、

$$\boldsymbol{Af} = \begin{pmatrix} af_{1,1} & \cdots & af_{1,N} & 0 & \cdots & 0 & af_{1,N+T+1} & \cdots & af_{1,N+T+F} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ af_{3*F,1} & \cdots & af_{3*F,N} & 0 & \cdots & 0 & af_{3*F,N+T+1} & \cdots & af_{3*F,N+T+F} \end{pmatrix} \qquad (ZZZ.9.3)$$

Af の N+1 列から N+T 列の範囲は、トランジスタの Hidden ノードと FET の各ブランチの接続を示すことになるから、すべて値 0 で接続なしである．

回路を解析するためには、Ab、At、Af の 3 つを 1 つにする必要がある．足りない分は 0 で埋めて、列数の多いほうに合わせて縦に積み上げて、次のようにすればよい．

$$\boldsymbol{A} = \begin{pmatrix} ab_{1,1} & \cdots & ab_{1,N} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & & & & \vdots \\ ab_{B,1} & \cdots & ab_{B,N} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ at_{1,1} & \cdots & at_{1,N} & at_{1,N+1} & \cdots & at_{1,N+T} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ at_{3*T,1} & \cdots & at_{3*T,N} & at_{3*T,N+1} & \cdots & at_{3*T,N+T} & 0 & \cdots & 0 \\ af_{1,1} & \cdots & af_{1,N} & 0 & \cdots & 0 & af_{1,N+T+1} & \cdots & af_{1,N+T+F} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ af_{3*F,1} & \cdots & af_{3*F,N} & 0 & \cdots & 0 & af_{3*F,N+T+1} & \cdots & af_{3*F,N+T+F} \end{pmatrix} \qquad (2.9.4)$$

これで行列 $\boldsymbol{A}$ の形が決まった．それは同時にトランジスタと FET が回路を扱う中で、どう置かれるかを決めたことになる．

続いて各ブランチを設定する関数だ．これも一般、トランジスタ、FET それぞれで、set_branch()、set_branch_trs()、set_branch_fet()の 3 つがある．ブランチに関しては接続関係のように互いに参照しあう関係はなくて、それぞれ独立だからわかりやすい．そこでさっそく、関数を呼ぶ際の引数をブランチ数 B、トランジスタ数 T、FET 数 F として、返り値をみていこう．返り値 $\boldsymbol{Y}$、$\boldsymbol{E}$、pnのそれぞれについて一般とトランジスタと FET を b、t、f で区別すると次のようになる．

関数 set_branch(B,diode=True)を行った返り値は、

$$\boldsymbol{Yb} = \begin{pmatrix} yb_{1,1} & 0 & \cdots & 0 \\ 0 & yb_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & yb_{B,B} \end{pmatrix} \qquad \boldsymbol{Eb} = \begin{pmatrix} eb_1 \\ eb_2 \\ \vdots \\ eb_B \end{pmatrix} \qquad \boldsymbol{pnb} = \begin{pmatrix} pnb_1 \\ pnb_2 \\ \vdots \\ pnb_B \end{pmatrix} \tag{2.9.5}$$

set_branch_trs(T)の返り値は、

$$\boldsymbol{Yt} = \begin{pmatrix} yt_{1,1} & \cdots & yt_{1,T} \\ \vdots & \ddots & \vdots \\ yt_{T,1} & \cdots & yt_{T,T} \end{pmatrix} \qquad \boldsymbol{Et} = \begin{pmatrix} et_1 \\ \vdots \\ et_T \end{pmatrix} \qquad \boldsymbol{pnt} = \begin{pmatrix} pnt_1 \\ \vdots \\ pnt_T \end{pmatrix} \tag{2.9.6}$$

set_branch_fet(F)の返り値は、

$$\boldsymbol{Yf} = \begin{pmatrix} yf_{1,1} & \cdots & yf_{1,F} \\ \vdots & \ddots & \vdots \\ yf_{F,1} & \cdots & yf_{F,F} \end{pmatrix} \qquad \boldsymbol{Ef} = \begin{pmatrix} ef_1 \\ \vdots \\ ef_F \end{pmatrix} \qquad \boldsymbol{pnf} = \begin{pmatrix} pnf_1 \\ \vdots \\ pnf_F \end{pmatrix} \tag{2.9.7}$$

行列 $\boldsymbol{A}$ を回路全体について組み上げる際に、一般のブランチ、トランジスタの内部のブランチ、FET の内部のブランチの順に並べたから、互いに重ならないようにその順で、 $\boldsymbol{Y}$ も $\boldsymbol{E}$ も pnも順番に並べればよい．

$$\boldsymbol{Y} = \begin{pmatrix} yb_{1,1} & 0 & \cdots & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & yb_{2,2} & \cdots & 0 & & & & & & \\ \vdots & \vdots & \ddots & \vdots & & & & & & \\ 0 & 0 & \cdots & yb_{B,B} & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & 0 & yt_{1,1} & \cdots & yt_{1,T} & 0 & \cdots & 0 \\ \vdots & & & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \cdots & 0 & yt_{T,1} & \cdots & yt_{T,T} & 0 & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & yf_{1,1} & \cdots & yf_{1,F} \\ \vdots & & & & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & yf_{F,1} & \cdots & yf_{F,F} \end{pmatrix} \tag{2.9.8}$$

$$\boldsymbol{E} = \begin{pmatrix} eb_1 \\ eb_2 \\ \vdots \\ eb_B \\ et_1 \\ \vdots \\ et_T \\ ef_1 \\ \vdots \\ ef_F \end{pmatrix} \qquad \boldsymbol{pn} = \begin{pmatrix} pnb_1 \\ pnb_2 \\ \vdots \\ pnb_B \\ pnt_1 \\ \vdots \\ pnt_T \\ pnf_1 \\ \vdots \\ pnf_F \end{pmatrix} \tag{2.9.9}$$

これで回路を解析するに必要なものの揃えかたが分かった．そこでこれをプログラムにしておこう．プログラミングにあたり、ばらばらに用意した関数を必要に応じて呼び出し、回路の入力から解析まで、さらに一旦入力した回路の保存や保存したものを読込んで評価するなど発展を考えて、これらをまとめる python のクラスを用意して進めていくことにしよう．

リスト 2.9.1：回路をまとめるクラスと回路の入力メソッド

```
class ElectricCircuit:
    def set(self):
        t = int(input('Number of Transistors '))
        f = int(input('Number of FETs        '))
        n = int(input('Number of nodes       '))
        b = int(input('Number of branches    '))
        print('node# ', n, 'branch# ', b, 'transistor# ', t, 'FET# ', f)
        self.size = n, b, t, f

        Ab = set_A(n, b)
        At = set_A_trs(n, t)
        Af = set_A_fet(n+t, f)
        Yb, Eb, pnb = set_branch(b, diode=True)
        Yt, Et, pnt = set_branch_trs(t)
        Yf, Ef, pnf = set_branch_fet(f)

        A = mm.v_stack(Ab, At)
        A = mm.v_stack(A, Af)
        Y = mm.cross_concatenate(Yb, Yt)
        Y = mm.cross_concatenate(Y, Yf)
        E = Eb + Et + Ef
        pn = pnb + pnt + pnf

        self.A = A
        self.Y = Y
        self.E = E
        self.pn = pn

        return A, Y, E, pn
```

これまでに関数として定義してきた、set_A()、set_branch()、Analize()、Diode()、set_A_trs()、set_branch_trs()、set_A_fet()、set_branch_fet()を置いたファイル ELCA.py に、書き加えるかたちでクラス ElectricCircuit、そしてその最初のメソッド set()を定義する．ELCA.py では MatMath はインポート済みの前提である．メソッド set()ではトランジスタ数 t、FET 数 f とこれら以外のノード数n、ブランチ数bの入力を求め、これを受けて、行列 $\boldsymbol{A}$ を作る関数set_A()、set_A_trs()、set_A_fet()と、ブランチを設定する関数 set_branch()、set_branch_trs()、set_branch_fet()を呼出す．関数を実行した結果として受けた返り値は、MatMath の関数を使って組み上げる．MatMath の関数 v_stack()は、列数の短いほうの行列に 0 でパディングして、列数を合わせて縦に積上げる．また、MatMath の関数 cross_concatenate()は、行列を重ならないように斜めに結合し、空いた部分はやはり 0 で埋める．またベクトル $\boldsymbol{E}$ と pnについては、python のリスト形式のまま取り扱っているから、加算記号'+'は、そのままリストの結合、すなわちベクトルの結合となる．これらの操作で行っている結合のしかたは先に説明した通りだ．あとは得られた $\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$、pnをインスタンス変数に保存するとともに返り値として返す．

2.9.2 トランジスタや FET を含む回路の解析

リスト 2.9.1 でトランジスタや FET を含む回路をまとめるクラス ElectricCircuit とその回路の入力メソッド set()を定義した．そこで次は用意された回路の解析であり、そのメソッド convergent_analysis()を定義する．これをリスト 2.9.2 に示す．

リスト 2.9.2：トランジスタや FET を含む回路の解析メソッド

```
class ElectricCircuit:

    def convergent_analysis(self, itr=300, lr=0.1):
        A = self.A
        Y = self.Y
        E = self.E
        pn = self.pn
        V_ppl = [0.5 for e in E]

        eyeY = mm.eye(len(Y))
        nondY = mm.sub(Y, mm.mul(Y, eyeY))
        trnsY = mm.mul(mm.trn(pn), nondY)

        for i in range(itr):
            W, V, I, P = Analize(A, Y, E)
            V_ppl = mm.sub(V_ppl, mm.mul(lr, mm.sub(V_ppl, V)))
            Yd = Diode(Y, E, pn, V_ppl)
            diagY = mm.mul(Yd, eyeY)

            Yjdge = mm.lte(mm.mul(trnsY, V), 0)
            Yvccs = mm.mul(nondY, Yjdge)

            Y  = mm.add(diagY, Yvccs)

        self.W = W
        self.V = V
        self.I = I
        self.P = P

        return W, V, I, P
```

解析はこれまでどおり関数 Analize()を使う．しかしリスト 2.7.3 で示したように、回路にダイオードを含むことに対応するとともに、FET のピンチオフの対応が必要だ．前者の繰り返し計算で収束させていくやり方は、節「2.7 ダイオード」で確認済みだが、後者については、ここでやり方を確認していこう．両者の扱いは必ずしも同じでないから厄介だ．しかし幸いにも、前者は行列 $\boldsymbol{Y}$ の対角要素に関するのに対し、後者は行列 $\boldsymbol{Y}$ の非対角要素に関するものなので、これを分けて扱えば、それぞれの対処を別々に行い、かつ、それを 1 つにまとめることができる．

それではリスト 2.9.2 を見ていこう.
メソッド convergent_analysis()のデフォルト引数に itr=300 と lr=0.1 があるが、いずれもダイオードの収束計算に使う. $\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$、pnはインスタンス変数をいちいち self をつけて呼ぶのは面倒だからこうしただけだ. しかし $\boldsymbol{Y}$ だけは、後で出てくる for ループの中で使う $\boldsymbol{Y}$ は別物であるから注意が必要だ. V_ppl はダイオードの収束計算に使う. そのあと次の 3 行で解析の準備を行う. ここで行列 $\boldsymbol{Y}$ の対角要素と非対角要素を分けるのと、ピンチオフの判断に使う行列を作る.

```
eyeY = mm.eye(len(Y))
nondY = mm.sub(Y, mm.mul(Y, eyeY))
trnsY = mm.mul(mm.trn(pn), nondY)
```

eyeY は $\boldsymbol{Y}$ と同じ大きさの単位行列である. その eyeY を使って mm.mul(Y, eyeY)で $\boldsymbol{Y}$ の対角要素を抽出している. このかけ算は要素同士のかけ算であって行列積ではない. これで抽出された $\boldsymbol{Y}$ の対角要素を元の $\boldsymbol{Y}$ から引き算して、 $\boldsymbol{Y}$ の非対角要素を nondY として抽出している. これによって相互コンダクタンスだけが抽出される.
ピンチオフの判断に使う行列は少々ややこしい. pnはダイオードを示すベクトルだが、これを縦ベクトルにしてから先に求めた nondY と掛け合わせる. MatMath のベクトルと行列の演算ではベクトルが行列に拡張される. だから trnsY は、相互コンダクタンスに pnにしたがって±符号をつけた行列になる. このとき pnは縦ベクトルにしてから掛けているから、被制御ブランチ＝電圧制御電流源のあるブランチの pnによって符号が決まる.
図 2.8.4 の FET の等価回路で見てみよう. 相互コンダクタンスは電圧制御電流源があるブランチ 2 に関わる. そしてこのブランチは逆接続のダイオードなので、NchFET の場合 $pn = -1$、PchFET では $pn = 1$である. そうすると FET のブランチ 2 に対応する要素の相互コンダクタンスに NchFET ではマイナス(-)、PchFET ではプラス(+)の符号をつけた行列が trnsY に作られることになる. もともと Y の要素はアドミタンスや相互コンダクタンスなのであって、すべて 0 か正の値だ. だから trnsY の要素の符号は電圧制御電流源の極性をあらわすことができる. 逆に言うならば、電圧制御電流源とその極性をあらわすことができれば何でも良いのであって、たまたま FET にもトランジスタにも電圧制御電流源のあるブランチは逆接続ダイオードとなっているから、計算の容易さも考慮してダイオードを示すベクトル pn を使って trnsY を作っているのである.
さて以上で解析の準備ができたので、for ループでダイオードの収束計算を行う. ループの中のはじめの 3 行は、変数名がちょっと違うが、リスト 2.7.3 の最内ループと同じだ.

```
W, V, I, P = Analize(A, Y, E)
V_ppl = mm.sub(V_ppl, mm.mul(lr, mm.sub(V_ppl, V)))
Yd = Diode(Y, E, pn, V_ppl)
```

```
diagY = mm.mul(Yd, eyeY)
```

まずはリスト 2.5.1 で定義した関数 Analize()を使って $\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$ から $\boldsymbol{W}$、$\boldsymbol{V}$、$\boldsymbol{I}$、$\boldsymbol{P}$ を求める．このとき引数で与える $\boldsymbol{Y}$ は、はじめはメソッド set()で設定して self にあったものだが、ループ中で更新していくので、計算の進行とともに変わっていく．いっぽう $\boldsymbol{A}$ と $\boldsymbol{E}$ は変わらない．電圧は移動平均 V_ppl をとるのも同じだ．それからリスト 2.7.2 で定義した関数 Diode()で電圧に応じたアドミタンスを求める．次の 4 行目はリスト 2.7.3 にはなかったものだが、先に述べたように相互コンダクタンスは別の扱いをするため、これを除いた要素、すなわち行列 $\boldsymbol{Y}$ の対角要素だけを抽出して diagY としている．これでダイオードに対する計算は完了する．

引き続き次の 2 行でピンチオフに対応して相互コンダクタンスを調整する．

```
Yjdge = mm.lte(mm.mul(trnsY, V), 0)
Yvccs = mm.mul(nondY, Yjdge)
```

Yjdge は、ループに入る前に求めておいた trnsY を電圧 V と掛け合わせて、それが 0 以下かどうかを求めたものだ．電圧 V は横ベクトルのままで trnsY と掛け合わせている．このとき MatMath の関数によって横ベクトルが縦に拡張されて trnsY と要素ごとに掛け合わされる．そうすると制御ブランチの電圧が符号のついた相互コンダクタンスと掛け合わされることになる．これに対して先に説明したように trnsY を作るときには被制御ブランチの pnを使ったから、trnsY は電圧制御電流源それ自身の極性を示す．つまり電圧制御電流源の極性に対して、制御ブランチに加わる電圧が順方向か逆方向かを見ていることになるのである．また図 2.8.4 の FET の等価回路で見ていくことにする．trnsY のブランチ 2 の行でブランチ 1 の列の要素に、符号付きの相互コンダクタンスが設定されている．この値は、NchFET ではマイナス(-)、PchFET ではプラス(+)である．そしてブランチ 1 の電圧が NchFET でプラス(+)、PchFET でマイナス(-)ならば、trnsY と FET のブランチ 1 の電圧とを掛け合わせた値は、マイナス(-)であり、その逆ならばプラス(+)になる．つまり両者を掛け合わせた値がマイナス(-)ならば順方向、プラス(+)ならば逆方向の電圧が FET のブランチ 1 にかかっていることになるのである．逆方向の電圧はピンチオフを意味する．そこでこの trnsY と $\boldsymbol{V}$ を掛けた結果の正負を負ならば True、正ならば False の行列を作ってピンチオフの判断結果の行列 Yjdge としている．その Yjdge に応じて、つまり True ならば相互コンダクタンスは元の値を使い、False ならば 0 にしている．すなわち順方向は、相互コンダクタンスの値に従って電圧制御電流源が電流を生み出し、逆方向ではピンチオフしているとして、電流を生み出さないようにするのである．

そしてループの最後に、

```
Y  = mm.add(diagY, Yvccs)
```

で、対角成分＝ダイオードに関して電圧に応じて調整したアドミタンスと、非対角成分＝電圧制御電流源の逆バイアス対応をした相互コンダクタンスを合わせて、次のループでの計算に渡す．この段階でもはや $\boldsymbol{Y}$ はループに入るときに用意したものとは違っている．またピンチオフに対応する操作で、相互コンダクタンスに関わる $\boldsymbol{Y}$ の要素が 0 に強制されると、制御ブランチの在処がわからなくなる．だからループの中で必ず元の相互コンダクタンスを見る必要があって、trnsY はループに入る前にその外で作ったものを毎回引用するようにしている．なお一連の処理で、相互コンダクタンスがトランジスタに関するものなのか、それとも FET に関するものなのかは区別していない．トランジスタの場合には、図 2.8.3 に示す等価回路では、ベース・hidden ノード間のブランチ 1 には、電圧制御電流源に逆方向の電流を生じさせるような電圧は、その構成上かからない．このため FET のために行う対処はトランジスタでは不要であるし、悪さをすることもないのである．

ループはデフォルト引数で指定する階数だけ繰り返して終了する．ループを抜けるとインスタンス変数に結果を反映するとともに返り値 $\boldsymbol{W}$、$\boldsymbol{V}$、$\boldsymbol{I}$、$\boldsymbol{P}$ でメソッドを完了する．

2.9.3 回路解析の結果の表示や保存や読込

解析が終わったら結果を表示したい．クラスとして ElectricCircuit を定義したので、トランジスタや FET を含む回路の情報を使って、そのインスタンス変数の値を抽出して表示しよう．リスト 2.9.3 に結果を表示するメソッドを示す．

リスト 2.9.3：トランジスタや FET を含む回路の解析結果を表示するメソッド

```
class ElectricCircuit:

    def display(self):
        n, b, t, f = self.size
        # Display Nodes
        print()
        for i in range(n):
            print('Node#{:d} potential = {:7.4f}V'.format(i+1, self.W[i]))

        # Display Branches
        print()
        for i in range(b):
            print('branch#{:d}：voltage = {:7.4f}V current = {:7.4f}A power = {:7.4f}W' ¥
                  .format(i+1, self.V[i], self.I[i], self.P[i]))

        # Display Transistors
        if t > 0:
            print()
            for k in range(t):
                l = b + 3 * k
                print('Transistor#{:d} Vbe{:7.3f}V Vce{:7.3f}V Ib {:7.3f}mA Ic {:7.3f}mA'¥
                      .format(k+1, self.V[l]+self.V[l+2], self.V[l+1]+self.V[l+2],¥
                              self.I[l]*1000, self.I[l+1]*1000))

        # Display FETs
        if f > 0:
            print()
            for k in range(f):
                l = b + 3 * t + 3 * k
                print('FET#{:d}        Vgs{:7.3f}V Vds{:7.3f}V Ids{:7.3f}mA'¥
                      .format(k+1, self.V[l]+self.V[l+2], self.V[l+1], self.I[l+1]*1000))

        print()
```

とくに説明の必要なところはないと思うが、n, b, t, f = self.size で、トランジスタや FET 以外のノード数、ブランチ数、そしてトランジスタ数、FET 数がすべてわかる．リスト 2.9.1 の set()メソッドで説明したように、トランジスタも FET も 1 個につき 3 つのブランチで、それを順に積上げているから、どのブランチがトランジスタのそれか、あるいは FET のそれかを計算して、図 2.8.3 のトランジスタや図 2.8.4 の FET の等価回路にしたがって、基本的に

見ておきたい項目を並べるだけである．プログラムとしては簡単だが、回路解析の際にいちいち考えるのは面倒なので、こうしてメソッドを作っておくと良い．
set()メソッドで回路を入力するのは結構大変だ．そこで入力した回路を保存したり、保存したものを読込んだりする機能があると便利だ．リスト 2.9.4 に回路を保存するメソッド save()と保存したものを読込むメソッド load()を示す．

リスト 2.9.4：回路を保存するメソッドと保存したものを読込むメソッド

```
class ElectricCircuit:

    def save(self, file_name, title=None):
        # params は辞書形式(export_params で取得したもの)
        params = {}
        params['title'] = title
        params['size'] = self.size
        params['A'] = self.A
        params['Y'] = self.Y
        params['E'] = self.E
        params['pn'] = self.pn
        with open(file_name, 'wb') as f:
            pickle.dump(params, f)
        print('モデル緒元をファイルに記録しました=>', file_name)

    def load(self, file_name, dump=False):
        with open(file_name, 'rb') as f:
            params = pickle.load(f)
        title = params.pop('title', None)
        print(title)
        self.size = params['size']
        self.A = params['A']
        self.Y = params['Y']
        self.E = params['E']
        self.pn = params['pn']
        print('モデル緒元をファイルから取得しました<=', file_name)
        if dump:
            print(params)
        return self.A, self.Y, self.E, self.pn
```

これも取り立てて説明の必要なところはないと思う．pickle を使うことで、辞書にまとめた回路の情報をそのまま保存・読込することができる．行列やベクトルとしてインスタンス変数の $\boldsymbol{A}$、$\boldsymbol{Y}$、$\boldsymbol{E}$、pnに加え size があればよいから、これらを対象としている．一応表題も 'title'として設定できるようにしておいた．

2.10 回路解析の実行例

ここまで長々と準備とその説明に労力を費やしてきたが、いよいよこれを使って回路を解析してみよう.

リスト 2.10.1：トランジスタや FET を含む電子回路の解析プログラム

```
from ELCA import *

file_name='test_x-x'
elca = ElectricCircuit()

elca.set()

print('¥nStart analizing...')
elca.convergent_analysis()
elca.display()

if input('保存しますか？(y/n)=> ') in ('y', 'Y'):
    elca.save(file_name)
```

準備に時間をかけた分プログラムは簡単だ. ELCA.py にクラス ElectricCircuit をはじめとして必要なものは揃っているから、それをインポートして、そのメソッドを実行していけば良い. 回路情報を保存するファイル名は 'test_x-x' にしたが、自身で決めればよい. クラス ElectricCircuit をインスタンス化して elca とした. elca.set()で回路を入力する. あとは elca.convergent_analysis()で回路解析し elca.display()で結果を表示、最後に入力した回路を必要に応じて保存する.

2.10.1 トランジスタ 1 石増幅回路

さっそくリスト 2.10.1 のプログラムを起動して、トランジスタ 1 石の増幅回路を入力してみよう. 図は 2.10.1 に用意した. ブランチ 1 の起電力を 0.62V、トランジスタの $h_{fe} = 100$を入力して次の結果が得られるはずだ.

リスト 2.10.2：トランジスタ 1 石増幅回路実行結果

```
Start analizing...

Node#1 potential =  0.5733V
Node#2 potential =  9.9953V
Node#3 potential =  5.3255V

branch#1 : voltage =  0.5733V current = -0.0000A power =  0.0000W
branch#2 : voltage =  9.9953V current = -0.0047A power =  0.0000W
branch#3 : voltage =  4.6698V current =  0.0047A power =  0.0218W

Transistor#1 Vbe  0.573V Vce  5.326V Ib   0.047mA Ic   4.670mA
```

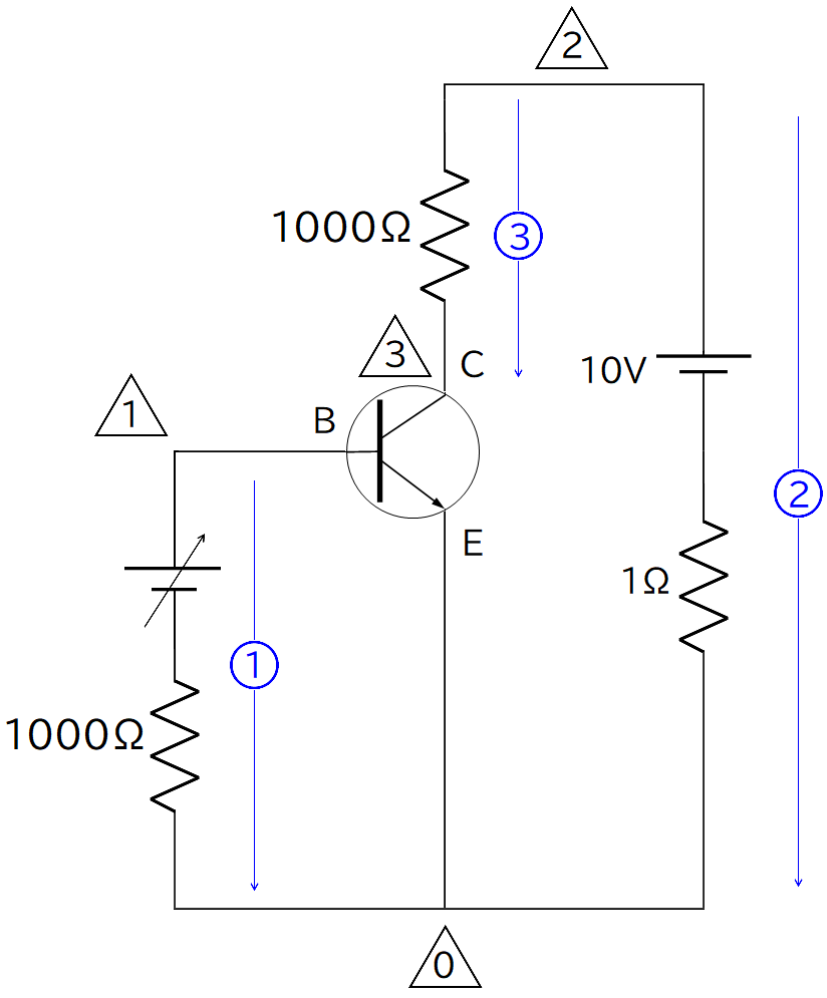

図 2.10.1 トランジスタ 1 石の増幅回路

これだけでは面白くないので、リスト 2.10.1 のプログラムを実行してファイル’test_10-1’に回路の情報を保存しておこう．保存された回路の情報を使って、図 2.10.1 のブランチ 1 の電圧源の値を変えたときに、どうなるのかを見てみよう．そのプログラムをリスト 2.10.3 に示す．

リスト 2.10.3：電子回路の解析プログラムで回路の特性を可視化

```
from ELCA import *

file_name='test_10-1'
elca = ElectricCircuit()
elca.load(file_name)

inputs = mm.linspace(0.0, 1.5, 101)
outputs1, outputs2, outputs3, outputs4 = [], [], [], []
print('  input  output  Ice(mA) Ibe(mA)')
for inp in inputs:
    elca.E[0] = inp                    # 入力をセット
    W, V, I, P = elca.convergent_analysis()

    outputs1.append(V[4]+V[5])       # Vce
    outputs2.append(I[4]*1000)       # Ice
    outputs3.append((V[3]+V[5])*10)  # Vbe
    outputs4.append(I[3]*10000)      # Ibe

    print('{:7.3f} {:7.3f} {:7.3f} {:7.4f}'¥
          .format(inp, W[2], I[4]*1000, I[3]*1000))

import matplotlib.pyplot as plt
plt.plot(inputs, outputs1, label='Vce')
plt.plot(inputs, outputs2, label='Ice(mA)')
plt.plot(inputs, outputs3, label='Vbe(10mA)')
plt.plot(inputs, outputs4, label='Ibe(100μA)')
plt.legend()
plt.grid()
plt.show()
```

ファイル名に’test_10-1’を指定し、ElectricCircuit を elca とインスタンス化してファイルから回路の情報を読込む．電圧源の値は MatMath の linspace()関数を用いてリスト変数 inputs に 0.0～1.5 の 101 個のデータを用意する．記録用の器を用意し、途中経過の出力のタイトルをプリントして for ループに入る．for ループでは inputs からデータを 1 つずつ取り出して elca.E[0]にセットする．これは図 2.10.1 のブランチ 1 の電圧源だ．これで入力電圧可変の調査ができる．あとは elca.convergent_analysis()で解析するが、返り値は表示と記録に使うので変数 $\boldsymbol{W}$、$\boldsymbol{V}$、$\boldsymbol{I}$、$\boldsymbol{P}$ で受ける．ループを回しながら append()で記録用のリストの

器に追加していくとともに表示していく．その際に例えば、コレクタ・エミッタ間電圧は、図 2.8.3 のトランジスタの等価回路でブランチ 2 とブランチ 3 の電圧を足し合わせたものだし、ベース・エミッタ間も同様だ．そしてノードやブランチの番号がプログラムの中では 0 から始まることに注意が必要だ．ともかく inputs に用意したデータを評価し終えるとループを抜けて、記録したデータをグラフ表示する．グラフを見やすくするためにデータを記録する際に単位や倍率を適宜調整する必要もある．多少厄介なことが多いが、結果は得るものが大きい．これを図 2.10.2 に示す．

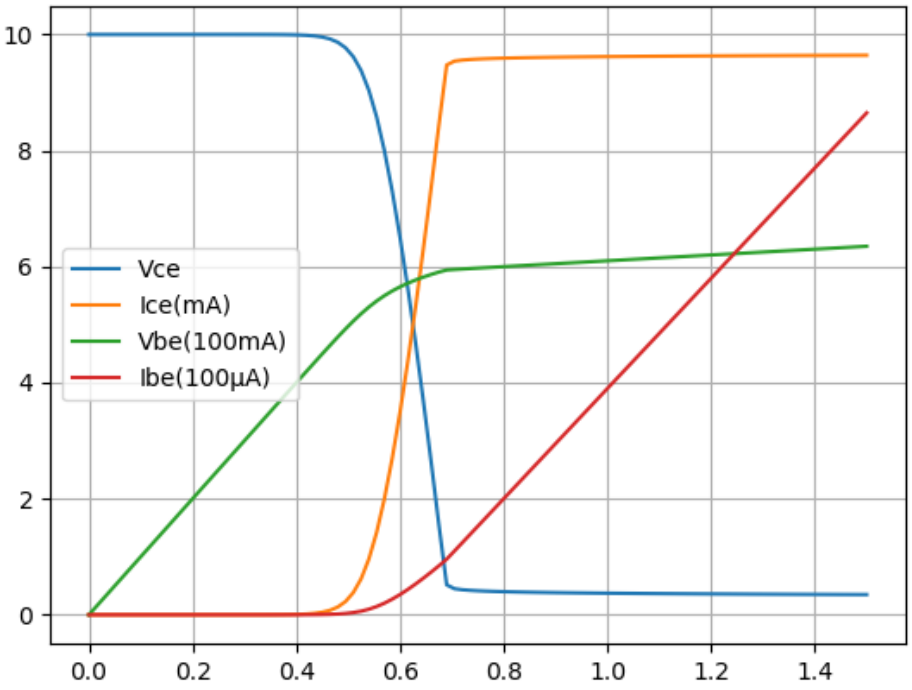

図 2.10.2 トランジスタ 1 石の増幅回路の動作

入力が 0.0V から次第に大きくなるが、はじめはベース・エミッタ間電圧だけが大きくなっていき、他は変化しない．しかし入力が 0.5V を超えたあたりからベース電流 Ibe が流れ始めて、それと同時にコレクタ電流 Ice が急に流れだして、コレクタ・エミッタ間電圧 Vce が 10V から急激に低下するようす．また入力が 0.7V くらいで Vce は 0V より少しプラスの電圧で下げ止まり、このとき Ice も 10mA より少し少ないところで頭打ちになっている．などなど、実際の回路の動作をうまくシミュレートしたものになっている．

2.10.2 FET 1石増幅回路

図2.10.3に示すようにトランジスタをFETに置き換えたらどうなるだろう？

リスト2.10.1のプログラムを起動して、FETに合うように抵抗値は変えて、今度はこれを入力してみよう．起動する際にfile_name='test_10-3'としてトランジスタとは別のファイルに回路情報を保存するようにしよう．

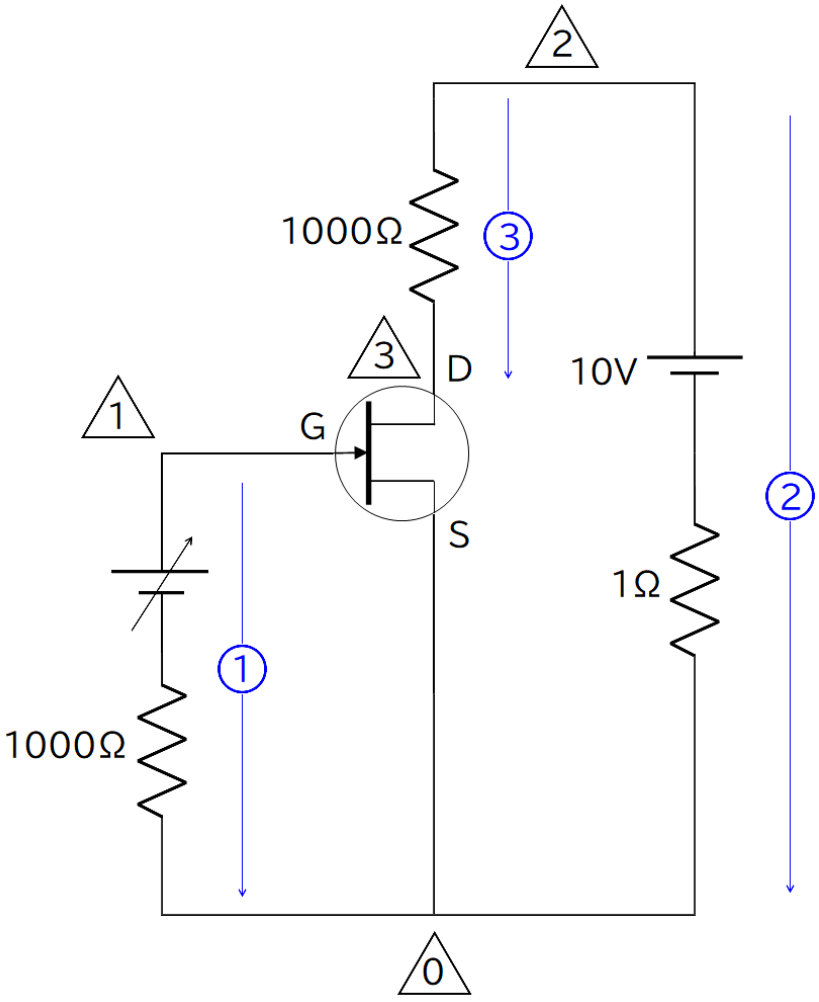

図2.10.3 FET 1石の増幅回路

ブランチ1の起電力は0.0V、FETの g_m =100m℧、 I_{dss}=5mAを入力して次の結果が得られるはずだ.

リスト2.10.4：FET 1石増幅回路実行結果

```
Start analizing...

Node#1 potential = -0.0000V
Node#2 potential =  9.9950V
Node#3 potential =  4.9968V

branch#1：voltage = -0.0000V current = -0.0000A power =  0.0000W
branch#2：voltage =  9.9950V current = -0.0050A power =  0.0000W
branch#3：voltage =  4.9982V current =  0.0050A power =  0.0250W

FET#1        Vgs -0.000V Vds  4.997V Ids  4.998mA
```

これも回路の特性を見ておこう．少々面倒くさいが、リスト2.10.3で観測のために選んだブランチを、図2.8.4の等価回路と図2.10.3を見比べながら置き換えて走行させればよい．ここでは結果だけ示すことにする.

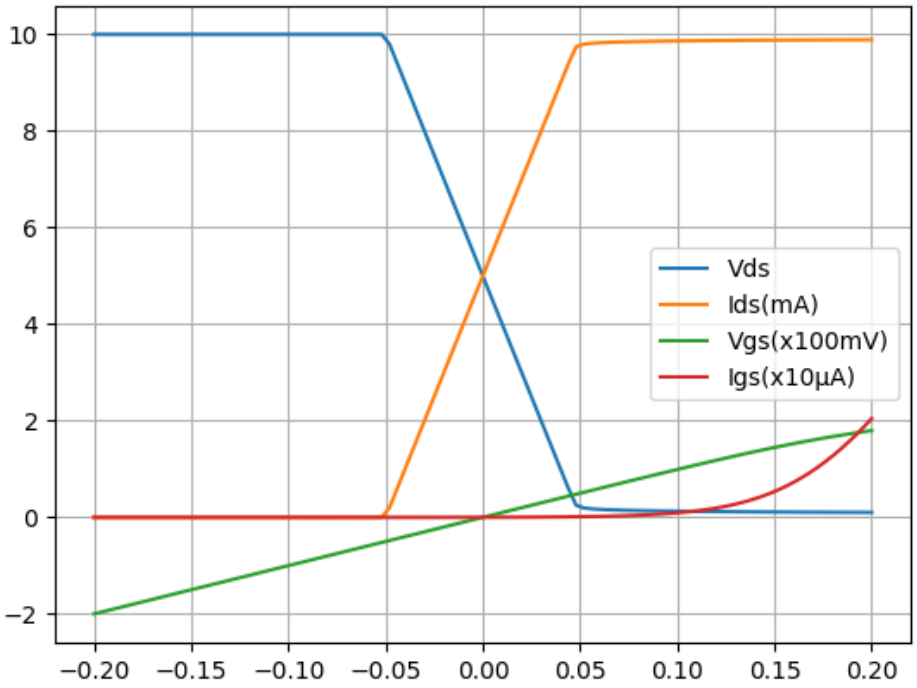

図 2.9.4 FET 1 石の増幅回路の動作

節 2.8.3 で説明したように FET の特性をかなり単純化したのがそのまま結果に表れている．しかし、入力電圧を 0 にした時に I_{dss} で指定した 5mA がそのまま I_{ds} になっていること、10V の電源とドレイン間の抵抗を 1kΩとしているのに対応した 10mA を少し下回ったところで I_{ds} が頭打ちになっていること、入力電圧を上げていくと途中からゲート電流が流れだしていること、などなど、これも回路の動作をうまくシミュレートしたものになった．少なくとも模式的にはこういう動作だよと、電子回路の初心者には勉強になるだろう．

2.10.3 FET 2石増幅回路

さて次は少し実用的なアンプだ．といっても、レトロな電池駆動のMCカートリッジ用ヘッドアンプだ．今も入手可能なFETを使っていて部品数も少なく、あまりお金はかからないけれど、味わい深い音がしてオーディオ好きでも十分満足させる．

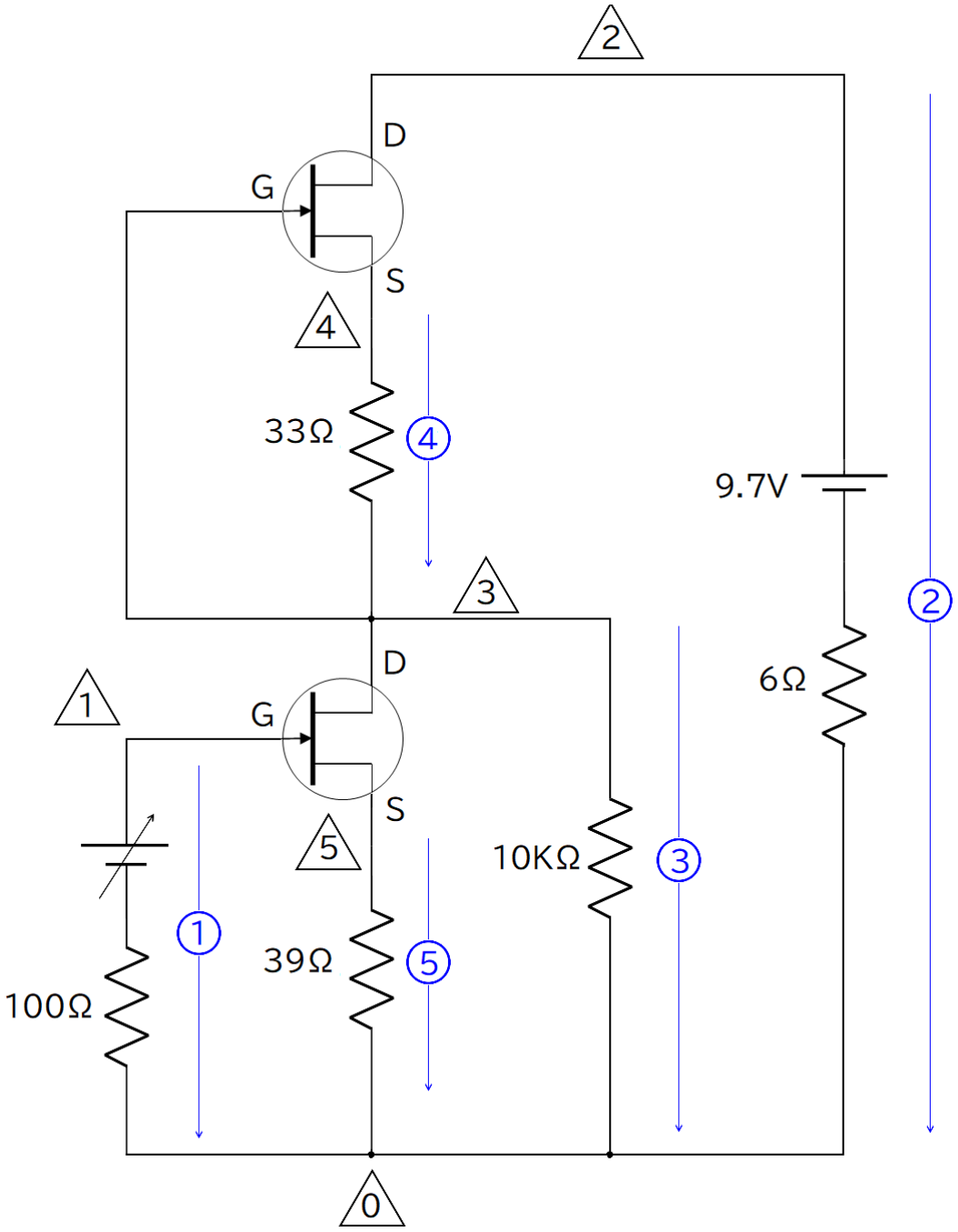

図 2.10.5 FET 2石の増幅回路

FET はいずれも 2SK68A で、 g_m=20m℧、 I_{dss}=10mA．電池は 006P で大雑把に起電力は 9.7V、内部抵抗 6Ωとした．入力抵抗は図では 100Ωとしているが、MC カートリッジをつなぐことを想定している．実際に作るときはゲートが静電破壊を起こさない程度の高抵抗を入れておくと良い．006P 電池は内部抵抗が大きく良い電源ではないから、図 2.10.5 の上側の FET は定電流動作をさせて、下側の FET を電源の変動から切り離している．下側の FET の交流負荷はグランドとの間の 10KΩとなるから、電源のインピーダンス上昇にも電圧変動にも強い回路となっている．実質的に単純な FET 1 つでの増幅で、良い部品を選べばきれいな音がするはずだ．

リスト 2.10.1 を起動してこの回路を入力した場合の実行結果を次に示す．これも起動する際にファイル名を指定して回路情報を保存するようにしよう．

リスト 2.10.5：FET 2 石増幅回路実行結果

```
Start analizing...

Node#1 potential = -0.0107V
Node#2 potential =  9.6645V
Node#3 potential =  4.6650V
Node#4 potential =  4.8689V
Node#5 potential =  0.2167V

branch#1：voltage = -0.0107V current = -0.0001A power =  0.0000W
branch#2：voltage =  9.6645V current = -0.0059A power =  0.0002W
branch#3：voltage =  4.6650V current =  0.0005A power =  0.0022W
branch#4：voltage =  0.2039V current =  0.0062A power =  0.0013W
branch#5：voltage =  0.2167V current =  0.0056A power =  0.0012W

FET#1        Vgs -0.227V Vds  4.448V Ids  5.450mA
FET#2        Vgs -0.204V Vds  4.796V Ids  5.916mA
```

ノード 1 に少しマイナスの電圧が出ているが、これは FET の等価回路から漏れている電圧と考えられる．ノード 3 から出力を取り出すのだが、その電圧が電源に対してうまく真ん中くらいになっている．そして両 FET ともドレイン電流が 5～6mA で程よいところで動作している．さてそれでは、これも回路の特性のグラフを作ろう．ここでは結果だけ示すことにする．

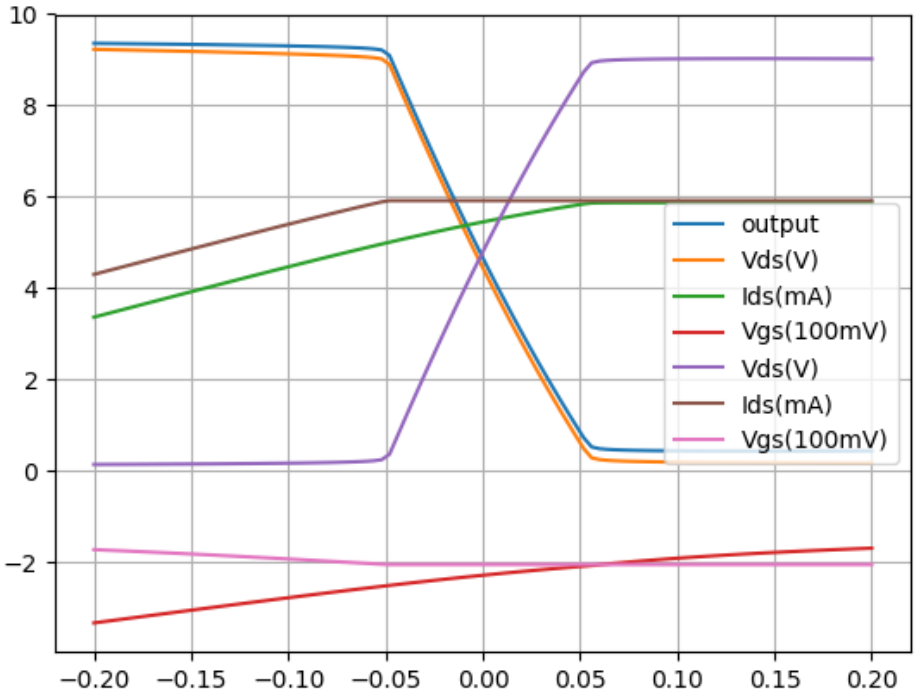

図 2.10.6 FET 2 石の増幅回路の動作

入力 0V を中心に、電源電圧の範囲で出力がフルスイングしている．そしてその±0.05V くらいの入力の範囲で十分にリニアな動作をして、だいたい 100 倍くらいのゲインがある．入力が-0.05V くらいから上の範囲で、上側の FET が I_{ds} が 6mA くらいの定電流動作をしているのも確認できる．このアンプはまだ持っているが当時は評価がうまくできていなくて、動作点がこれとは少しずれていた．今回あらためて評価して、改善点も見つかったので、実際に試してみたいものである．

2.10.4 オーディオアンプ

ではここで本格的なオーディオアンプの回路を解析してみよう．さっそく図 2.10.7 に回路図を示す．

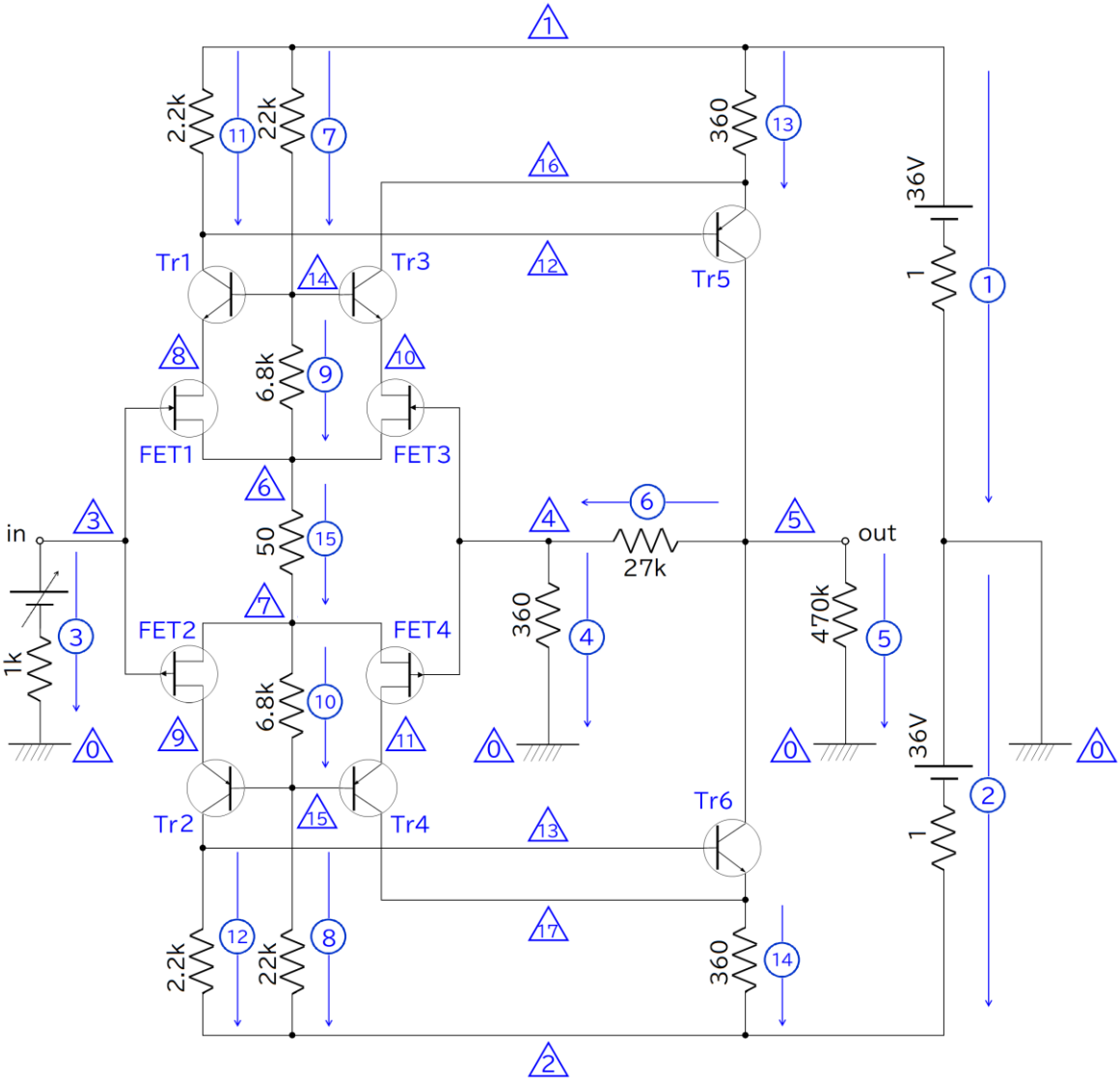

図 2.10.7 オーディオアンプの回路

この回路の解析では計算がうまく収束しないという問題に遭遇した．この解決には多くの試行錯誤を繰り返したのだが、簡単な方法で解決する．その方法は勾配クリッピングである．クラス ElectricCircuit のメソッド convergetnt_analysis()をリスト 2.9.2 に示した．そこでは回路解析は収束計算によって結果を求めている．その際に関数 Analize()で求めたブランチ

電圧 V から、その移動平均 V_{ppl} を求める．そして V_{ppl} を関数 Diode()に引数として与えて、ダイオードのアドミタンスを得ている．このブランチ電圧 $\boldsymbol{V}$ の移動平均 V_{ppl} を求める際に、勾配クリッピングを適用するのである．とまれ、もとよりブランチ電圧 $\boldsymbol{V}$ は、ダイオードのアドミタンスの計算において不安定で、それゆえに移動平均 V_{ppl} を使うのだが、これに能動素子の相互コンダクタンスを電圧依存で変える操作が加わって、さらに厳しい状況となる．そして収束計算がうまくいかない状況では、ブランチ電圧 $\boldsymbol{V}$ が収束計算のループの計算のたびに大きく変動し、しばしば極端な値となっている様子が観察される．そういう状況下で電圧 $\boldsymbol{V}$ の収束の方向を見出すことが困難となっていると考えられる．そこで、移動平均の算出での勾配に上限を設けて、その範囲で移動平均の更新を行うのである．電圧 $\boldsymbol{V}$ からその移動平均を求めるのは、式 2.7.4 に示したが、学習率 lr、勾配 $V_{ppl} - V$だった．この勾配の L2 ノルムは、

$$L2n = \sqrt{\sum \left(V_{ppl} - V\right)^2} \tag{2.10.1}$$

ここで勾配の上限を g_clip とおいて、$L2n \geqq g\_clip$ すなわち勾配の L2 ノルムが上限を超えている場合には、

$$rate = \frac{g\_clip}{L2n} \tag{2.10.2}$$

として、V_{ppl} の更新は、

$$V_{ppl} = V_{ppl} - lr \times rate \times \left(V_{ppl} - V\right) \tag{2.10.3}$$

いっぽう、$L2n < g\_clip$ すなわち勾配の L2 ノルムが上限未満の場合には、

$$V_{ppl} = V_{ppl} - lr \times \left(V_{ppl} - V\right) \qquad \left(2.7.4 \text{ 再掲}\right)$$

とすることによって、勾配 $V_{ppl} - V$ が大きくなっても g_clip で指定された値を上限として V_{ppl} の更新に使われる変化量が抑えられるのである．ここで V も V_{ppl} も複数の要素からなることに注意が必要だ．そして、$V_{ppl} - V$ の大きさを見るのに、個々の値ではなくて、全体の L2 ノルムを使って判断する．だから $V_{ppl} - V$ のどれか 1 つでも突出した大きな値となっていたら、すべての V_{ppl} の更新が抑制されることになる．ではこれを適用した解析メソッドを示そう．

リスト 2.10.6：L2 正則化を適用した解析メソッド

```
class ElectricCircuit:

    def convergent_analysis(self, itr=300, lr=0.1, g_clip=10.0):
        A = self.A
        Y = self.Y
        E = self.E
        pn = self.pn
        V_ppl = mm.mul([0.5 for e in E], pn)

        eyeY = mm.eye(len(Y))
        nondY = mm.sub(Y, mm.mul(Y, eyeY))
        trnsY = mm.mul(mm.trn(pn), nondY)

        for i in range(itr):
            W, V, I, P = Analize(A, Y, E)

            l2n = mm.Sum(mm.exp(2, mm.sub(V_ppl, V))) ** 0.5
            rate = g_clip / (l2n + 1e-7) if l2n >= g_clip else 1
            V_ppl = mm.sub(V_ppl, mm.mul(lr*rate, mm.sub(V_ppl, V)))

            Yd = Diode(Y, E, pn, V_ppl)
            diagY = mm.mul(Yd, eyeY)

            Yjdge = mm.lte(mm.mul(trnsY, V), 0)
            Yvccs = mm.mul(nondY, Yjdge)

            Y  = mm.add(diagY, Yvccs)

        self.W = W
        self.V = V
        self.I = I
        self.P = P

        return W, V, I, P
```

リスト 2.9.2 に示した解析メソッドと基本的には同じだから、違う部分だけ説明する. 引数に勾配クリッピングの閾値 g_clip=10.0 を加えた. そして for ループの中で上述したように、移動平均を求める際に L2 ノルムを使う.　MatMath の関数を使うので数式はややこしいが、式 2.10.1 に従い L2 ノルムを求め.

```
l2n = mm.Sum(mm.exp(2, mm.sub(V_ppl, V))) ** 0.5
```

次にこの L2 ノルムと閾値 g_clip とを比較するが、rate = 1 ならば式 2.10.3 は式 2.7.4 と同じだから、それも含めて rate を求める.

```
rate = g_clip / (l2n + 1e-7) if l2n >= g_clip else 1
```

こうして l2n が閾値 g_clip を超えたら rate は式 2.10.2 の値、閾値未満ならば rate=1 として

おき、あとは rate を学習率 lr に掛けたものを学習率として、移動平均 V_ppl を求める.

```
V_ppl = mm.sub(V_ppl, mm.mul(lr*rate, mm.sub(V_ppl, V)))
```

移動平均が求まれば、後の操作はリスト 2.9.2 と変わらない.

ではこれを使って図 2.10.7 の回路を解析しよう. Tr1～Tr4 の hfe=500、Tr5,Tr6 の hfe=100、FET1～FET4 の g_m=40m℧、 I_{dss}=10mA とした. この結果を図 2.10.8 に示す.

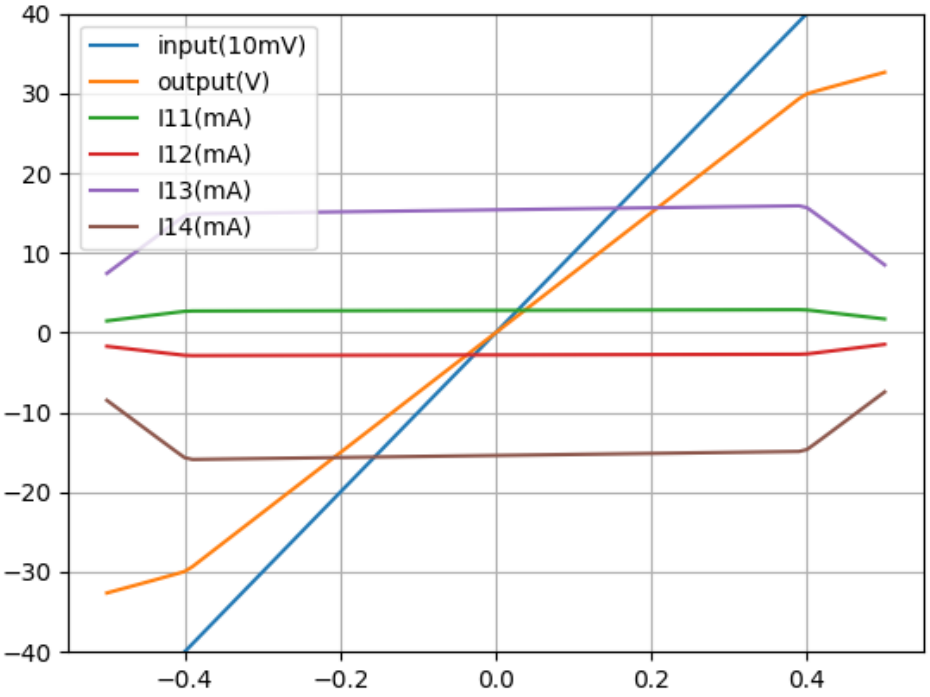

図 2.10.8 オーディオアンプの動作

入力を-0.5V～0.5V の範囲で振って動作を見たが、入力±400mV の範囲で出力±30V となっていて設計通りのゲインでリニアリティーがとても良く、多くを語る必要はない結果となった. 実際のアンプは、フォノイコライザアンプにしている. このため実機は、図 2.10.7 のブランチ 4 とブランチ 6 は RIAA 補正のために抵抗とコンデンサを組み合わせたものになっている. また図 2.10.7 ではアンプ入力を 1KΩとしているが、実機では 47KΩを介して接地してある. これは入力に接続する MM カートリッジの出力インピーダンスを想定した値だ. さらに

アンプ出力は次段の入力インピーダンスが 470K オームなので、図 2.10.7 では、これをつないだ状態を想定している.

[おわりに]

実は、この章の電子回路の解析のプログラムの起源は、40 年以上前にさかのぼる. 当時、私はシャープの MZ-2000 という 8 ビットの Z80A プロセッサを搭載した一体型のパーソナルコンピュータを購入した. MZ-2000 は起動すると BASIC インタプリタがいきなり立ち上がる、いわば BASIC 専用マシンで、プログラムを入力しなければ何もできない機械だった. それでも嬉しくて、ちょっとしたゲームなど作ったり、電車の模型を制御したりしてみたりして遊んでいた. しかしほどなく有限要素法を学んだ新入社員が職場にやってきて、彼に刺激されて勉強会などするうち、類似する手法で電子回路を解析できることを知るに至った. しかし何しろ Z80A という CPU は今とは比較にならない遅さで、メモリもうろ覚えだが 64KB くらいしかなかったと思う. 数値計算ライブラリもない中でプログラミングだったから、逆行列を求めるにも、求め方を勉強して四則演算とループで 1 ステップずつ書かざるを得なかった. しかしかえってその苦労をしてよかった気がしている. ともかくできたプログラムでアンプの設計に挑み、某誌主催の自作アンプコンテストなどに持って行ったりした. そして作ったプログラムも紹介記事を書いて某誌に投稿したりした. このプログラムの紹介は没になってしまった. そしてそれからほどなく私自身、コンピュータのハードウェアの開発にのめりこんでいくこととなり、すっかりアンプの製作からもプログラミングからも本当に長い間離れてしまっていた. しかし最近になって、そのとき投稿した原稿が残っているのを見つけた. それが本章のきっかけとなった. そして当時の回路解析方法が今も通用し、それどころか、コンピュータ・シミュレーションの手法の根底をなす技術となっていることを見出すに至った. プログラミング言語も今を意識して Python として刷新したが、敢えて既存のライブラリは使わずに、MatMath を作って使うようにした. また当時は考えられなかったような計算能力の向上があって収束計算が可能となり、2.7 ダイオードから以降の内容は新たに加えた.

余談とはなるが、2.10.4 で紹介したアンプは、製作当時に愛機として長く使えるものをと思っていたが今も現役である. 自作アンプコンテストではそれなりの評価だった. しかし、レコードから音楽の奥深さを伝え、その廃れることのない価値は、新たな発見をもたらすことはあっても陳腐化することはない.