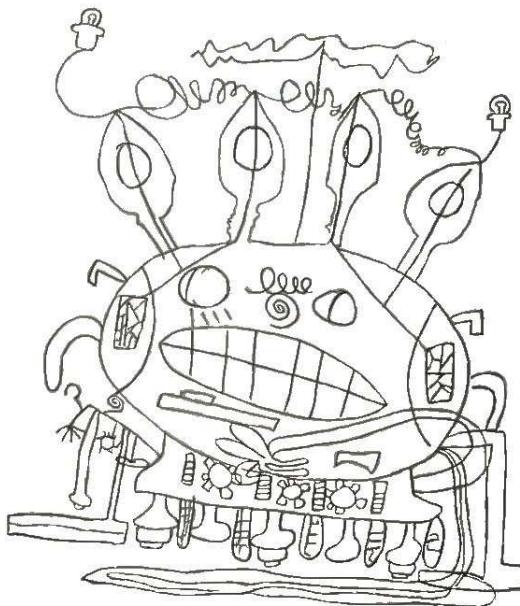


ロボット知能をつくる

Debian と LEGO を使って



Y. Honda

目次

第Ⅰ部 構成論的科学としてのロボット知能	13
第1章 反応行動のための知能	25
1.1 走性における感覚行動写像	26
1.1.1 非交差性線形写像	27
1.1.2 超音波センサーによる走性	28
1.1.3 交差性線形写像	29
1.2 ゲインパラメータの取り得る範囲	31
1.3 抑制性感覚運動写像	33
1.4 非線形感覚運動写像	33
1.5 感覚運動写像で行動するロボット	34
第2章 単純な反応行動	37
2.1 EV3 上で Debian(ev3dev)	37
2.1.1 インストールと起動	37
2.1.2 停止, 再起動	39
2.1.3 USB 有線 LAN を用いたセットアップ	40

2.1.4	USB ネットワーク	43
2.1.5	USB 無線 LAN	44
2.2	PVM	47
2.3	ev3c	47
2.4	クロスコンパイルと Makefile	48
2.4.1	感覚運動写像と PVM	51
第 3 章	収束する感覚行動写像	57
3.1	1 軸回転運動	58
3.2	数学的解析の概略地図	61
3.3	行列形式	63
3.4	行列の指數関数を用いた解	64
3.5	行列の固有値と固有ベクトル	67
3.6	状態ベクトルに対する解	70
3.7	状態ベクトルの具体的なかたち	71
3.8	無矛盾性の確認	75
3.9	振動	78
3.10	減衰振動	81
3.11	比例ゲインと積分ゲイン空間における相図	83
3.12	遷移行列を用いた漸化式	83
第 4 章	時間遅れのある反応行動	87
4.1	回転運動方程式	88
4.2	反応行動としての感覚運動写像	89

4.3	時間遅れと離散制御	96
4.4	ローターの回転運動における緩和時間	98
4.5	時間遅れを考慮した拡張遷移行列	102
4.6	フィボナッチ型遷移行列	106
4.7	時間遅れの固有値への繰り込み	109

第 II 部 具体的にロボット知能を構成するための道具 117

第 5 章	Debian をつかう	121
5.1	パッケージの管理	125
5.1.1	既にインストールされているパッケージの確認 .	125
5.1.2	DVD(CD) ドライブ内のパッケージ認識	126
5.1.3	パッケージ情報のアップデート	126
5.1.4	パッケージの検索	126
5.1.5	パッケージのインストール	127
5.1.6	ネットワークインストール	128
5.1.7	amd64 アーキテクチャで i386 アーキテクチャを使えるようにする	128
5.2	エディター vi	129
5.2.1	vi の動作設定	129
5.2.2	vi の使い方	130
5.2.3	GUI 上の vi	133
5.3	ネットワーク	134

5.3.1	IP アドレス	135
5.3.2	名前解決	136
5.3.3	経路解決	138
5.3.4	ネットワークファイルシステム NFS	139
5.4	kernel の構築とインストール	140
5.4.1	構築	141
5.4.2	インストール	142
第 6 章	無線 LAN	143
6.1	ツールのインストール	143
6.2	設定コマンド	144
6.3	シェルスクリプト	148
6.4	経路の追加	150
6.5	トラブルシューティング	152
6.5.1	SIOCSIFFLAGS: Operation not possible due to RF-kill	152
6.5.2	Network-Manager を使うと DNS が破壊される	153
第 7 章	並列計算ライブラリ PVM	155
7.1	インストールと開始	156
7.1.1	apt-get によるインストール	156
7.1.2	dpkg を用いたインストール	157
7.1.3	PVM を開始する	158
7.2	C 言語で PVM を利用する	162

7.2.1	ライブラリのインクルード	162
7.2.2	構成情報の取得	163
7.2.3	プロセスの生成 (spawn)	165
7.2.4	標準出力	168
7.2.5	データの送信と受信	168
7.2.6	遅延のないデータ送受信を行うために	172
7.2.7	spawnされたプログラムが正常終了しないで残っている場合	174
7.2.8	整数型データの通信プログラム例	174
7.3	トラブルシューティング	176
第 8 章	グラフ作成 (GNUPLOT)	179
8.1	gnuplot のインストール	182
8.2	起動と終了	183
8.3	データ描画 (2D)	183
8.4	文字の大きさ変更	188
8.5	制御コードファイル	189
8.6	対数スケール	190
8.7	EPS 出力	191
8.8	EPS 出力で複数のグラフを <code>replot</code> する	191
8.9	LaTeX 文章へのグラフの組み込み	192
8.10	EPS ファイル内の文字を日本語に置き換える	193
8.11	C 言語ソースコードから直接グラフを作る	196
8.12	簡易アニメーション	198

8.13 演算結果をプロットする	199
8.14 データを間引いてプロット	200
8.14.1 ブロック	200
8.14.2 every オプション	201
8.15 範囲指定してプロット	204
8.16 データを関数で fit する	205
付録 A ボードコンピュータ	207
A.1 BeagleBone Black	208
A.1.1 Debian のインストール	208
A.1.2 Debian のイメージを直接 microSD にコピー	212
A.1.3 ネットワークデバイスに付けられる番号	213
A.1.4 PIC とのシリアル (UART) 通信	213
A.1.5 USB 無線 LAN	223
A.1.6 i2c でセンサーの値を読み取る	226
A.1.7 PIN 配置	227
A.1.8 超音波センサー値を i2c で読み取る	235
A.1.9 BBB の電源	238
A.1.10 PWM でモーターを制御する	239
付録 B シリアルコンソール	243
B.1 シリアルポートの読み書き許可	243
B.2 minicom	244

付 錄 C 起動時に実行されるシェルスクリプト	247
C.1 insserv	247
C.2 update-rc.d	248
付 錄 D 分散バージョン管理システム gitlab	251
関連図書	255
付録	258
索引	258

第I部

構成論的科学としての
ボット知能

ここには、ロボット知能を考える上で基本的に必要になる事柄をまとめた。

反応行動のための単純な感覚運動写像を中心に、知能的な行動の創発という考え方を紹介する。

また、反応行動の基本として系が振動する場合について、基礎数学を用いた解析的な議論と数値解析的な方法も紹介する。時間遅れが存在しない場合、系が不安定になることは理論的にはありえず、せいぜい振動するだけであることを示す。

いっぽう、飛行ロボットの様な行動の特徴的な時間長さがミリ秒単位となるような、系の安定性を考える場合には、身体性がもつ時間遅れがもたらす影響が無視でない。むしろその行動を大きく左右する例を紹介する。反応の時間遅れは、系を大きく不安定にすることを示す。ロボット知能の研究とは、簡単に言えば人間の持っているような知能を創って、ロボットにそれをもたせようという試みと言えるであろう。似たような言葉として、人工知能というものがある。たとえばよく知られている例をあげると、コンピューターが将棋のプロ棋士に勝ったという話が話題になることがあることはご存知だろう。ここでいうコンピューターというのは、高速な演算処理をすることが可能な計算機ハードウェアと、そこにインストールされた、実際に将棋をさすプログラム（ソフトウェア）双方を一体としてコンピューターと呼んでいる。将棋プログラムは人工知能の典型的な例である。

指し手の数は膨大であるが、有限であることには違いない。巧妙に最善手を探し出すアルゴリズムを考えだされれば、記憶力も計算力も人間より桁外れの能力をもつコンピューターは人間の棋士においつき、いず

れ追い越してしまうであろう。将棋のような枠組みがはっきりと定義されている問題については、人工知能のエキスパートシステムはかなりの性能を発揮することが出来るのである。このような能力も、もちろん知能であることには違いない。

われわれ人間が遭遇する状況は、このちようく枠組みがはっきりしている場合ばかりとは限らない。むしろ、そうではない場合がほとんどであろう。判断の条件や状況は刻一刻と変化しているし、どこまで判断の条件に含めて考えればよいのかも曖昧である。そのような状況でも、われわれは何がしかの判断を下し、適切と思われる行動を起こしている。

知能がどのように構成されていて、どのようなメカニズムの下ではたらいているのか、はっきりわかっていれば、知能を創ることが可能かもしれない。しかし、それらがはっきり解明されていないばかりか、知能とは何か、どういうことなのかさえはっきりと定義できないのが現状である。

構成論的科学

そこで、最も知能的な存在であると考えられる人間を、今までの科学が用いてきた分析的な手法で研究しようとするのは、自然な考え方かもしれない。分析的な手法とは、いわば対象を徹底的に細かく細分化していき、それらの細分化された構成要素の働きの総体として、元の対象を理解しようとする方法である。例えば、物質の性質を、原子や分子が多数集まった総体として理解したり、半導体のように活用したりしよう

とする。このような物質科学とよばれる分野がもっともその典型的な例であろう。

ところが、知能の研究において分析的手法を用いるということは、おのずと脳を細分化して、その生物学的な性質や、電気的な機能などを調べることになる。このような分析的な努力は多くの研究者によって進められている。しかし、物質科学と同様な分析的な手法を用いることが困難であるということは、容易に想像できるであろう。たとえば、物質科学の場合のようにX線を照射して構造を調べたり電子顕微鏡でのぞいたりすることが、生きた脳に対して行えないということである。もし、行ったとしても、それらの外部刺激の影響が強すぎて、知能の働いている現場をとらえることができないということも考えられる。

そこで、構成論的科学として研究を進める方法を考える。まず「知能」を構成して、つまり、創って、それを実環境の中で行動させる。ここで、われわれ人間などが持っているような生物の知能と区別するために、構成論的に創ろうとする知能の方を「知能」と表した。「知能」はコンピューター内のいわばソフトウェアであるから、かたちとしては、その中にプログラムとして作り込むだけでも十分と思われるかもしれない。しかし、構成論的科学としてのロボット知能の研究においては、「知能」と「身体」をもったロボットが変化に富んだ実環境の中で行動することが必要であると考える（図1参照）。もちろん、その「知能」で多様な実環境の中を行動するのに十分であるとは考えられない。生物であれば、そのような知能をもった生物は十分環境に適応していないということで、淘汰されるであろう。すなわち実環境に適した知能（行動原理）をもった生物だけが生き残る。

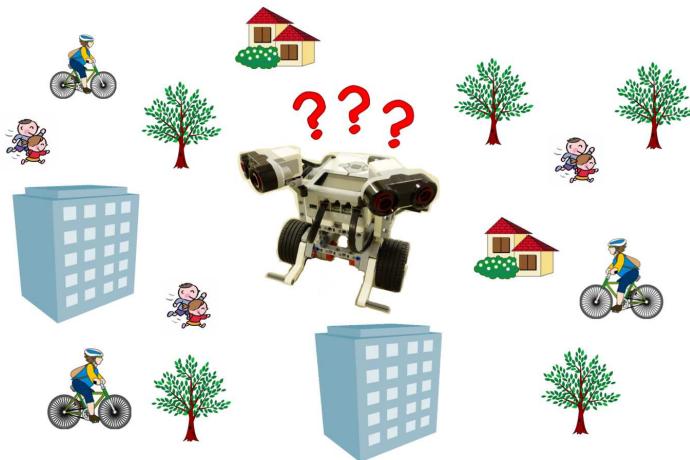


図 1: 環境の中で行動するロボット

いっぽう、ロボットの場合にはその「知能」を再構築することによって同じ身体をもつロボットを再利用することができる。構成論的科学とは、いばば「知能」と「身体」を最構成しながら、生物がたどった進化の過程を再現することを通じて、知能を理解しあるいは活用しようというアプローチである。

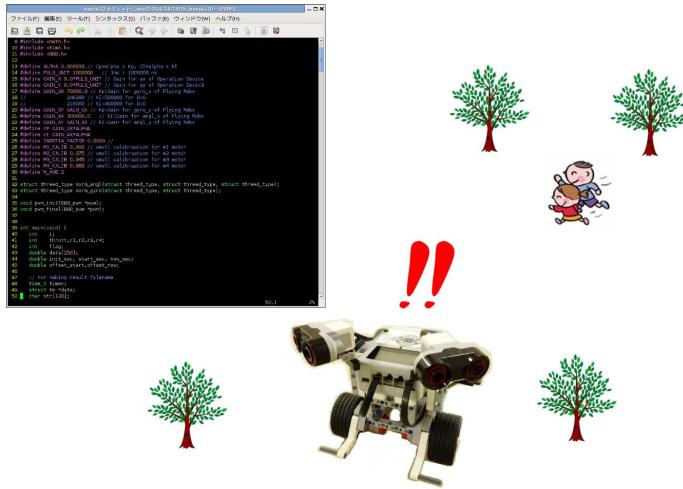


図2: 知能（プログラム）が進化するロボット

ロボットの身体性

上でも述べたように、構成論的に知能を研究しようとすると、その「知能」を持った機械がある環境の中で行動することが必要である。いわば脳だけではだめで、身体があって「知能」を活用して行動することが必須である。

「知能」を搭載する機械のことを、筐体、そして「知能」と筐体の総体のことを、ここではロボットと呼ぶことにしよう。ロボットという言



図 3: 構成論的科学としてのロボット知能の進化

葉の語感からは、アトムやアシモのような人間型の機械を想像してしまうかもしれない。本書では、広く「知能」を搭載した、行動（運動）することのできる機械（筐体）のことをロボットと呼ぶ。

自律運転の「知能」を搭載した自動車なども、単なる人間が操縦する機械ではなくて、もはやロボットと呼んでよいであろう。あるいは、自動飛行できるジャンボジェット機も、広い意味ではロボットと呼んだ方がよいかもしれない。

これらの例で言うと、自動車や飛行機の機械の部分のもつ性質のことを身体性と呼ぶ。また、筐体はロボットの「身体」と呼んでもよいであろう。

自動車と飛行機の例からも想像できるとおり、「知能」の構造や機能は身体性と大きく関わっている。あるいは、同じ「知能」であったとしても身体性の違いによって、生じる行動は大きく異なる可能性が高い。

創発

ロボットの行動は「知能」と「身体」だけで決定されるわけではない。どのような外部環境のなかでロボットが行動するかによって、生成される行動は異なる。いわば、「知能」、「身体」および環境の3者がダイナミックに相互作用するなかで、行動が現実に現れてくる。このことを創発とよぶ。

行動のための知能もそうであるが、様々なシステムはこの様に多くの構成要素からなっている。環境のなかでシステム全体が機能する時、それぞれの構成要素の機能の単なる和としてではなく、構成要素それぞれがもともと持っていないような機能が現れることを創発と言うこともできる。個別分断的に構成要素を観察しても、全体としての機能を理解することはできない。

ロボットが置かれる環境は、人間が置かれている環境とは一般に異なる可能性がある。たとえば、深海に潜るロボットであったり、原子炉の中に入るロボットであったり、空中を浮遊するロボットをイメージすれば、人間が普段置かれている環境条件とは異なる環境条件の中でロボットが行動することがわかる。

「知能」、「身体」および「環境」が相互作用することによって創発する行動がロボットインテリジェンスであるのだから、どのような環境条件の中で「知能」を実現するかということに依存して、結果として得られるロボット知能の形態は、人間のそれとは異なったものになることも十分考えられる。

飛行機の歴史から

飛行機の歴史を思い起こしてみる。鳥は、翼を利用して空を飛ぶ人間の先輩である。最初、人間は、翼が羽ばたくということに飛行のメカニズム原理が潜んでいると考えた。いまから振り返って考えると滑稽であるが、人工の翼を腕につけて多くの人々が飛行を試みたが、失敗に終わった。

その後、ライト兄弟につながる先人達は、鳥を徹底的に観察し、グライダーや飛行機を作つて飛んだ。失敗を繰り返して飛行機の発明に至るその過程は、まさに飛行のメカニズムを解明するための構成論的科学であると言える。実際に飛行機を作つて飛ばすという過程の中で、翼が空気中を前進することによって揚力が発生するという飛行のメカニズムが解明されたのである。

現代のジャンボジェット機を見れば分かる通り、鳥を観察して構成された飛行機は、鳥とは異なる形をしている。しかし、推進力を生み出す方法が異なっているのであって、揚力を生み出す原理は両者に共通しているのである。

ライト兄弟になれるのか

知能の研究の中で、まだわれわれはライト兄弟以前にいるということになる。知能ロボットを作つて、実環境の中で行動させてみるということを繰り返すことで前進しようとしている。先にも述べたとおり、このことを、構成論的科学と呼ぶ。一気に高度な知能を構成しようとするこ

とは、もちろん困難であろう。やはりもっとも基本的な、あるいは単純な知能から出発する。

そのために、ロボットの「知能」、すなわちロボット知能を大きく4つに分類して考えることからはじめる[1]。

ひとつ目は、反応行動のための知能である。感覚器やセンサーから入力してきた情報にたいしてある定型的な反応を示す行動のことである。ここで、注意が必要である。もっとも単純であると考えられる反応行動においても、生成される（創発される）行動が定形的であるとは限らない。定型的なのは、反応を決めるメカニズム（感覚行動写像）である。反応のメカニズムが定型的であったとしても、複数の感覚入力が組み合せられたり、環境との相互作用を通じて、複雑な行動が創発される可能性がある。

また、反応行動と混同しやすい言葉として、反射行動がある。反応行動の一種として反射行動が含まれる場合もあるが、反射行動とは、行動そのものが定形的な行動のことを指す。たとえば、熱いものに触ったときに、瞬間的に手を離す行動などである。

4つの知的行動の2つ目は、計画行動のための知能である。地図や経路情報をもとに行動の計画を構成する知能といえる。また、実際の行動から地図を構成する学習過程もこの知能に含めて考えられる。

3つ目が、適応行動のための知能である。変化する環境の下では、計画行動だけでは適応できない条件の変化に対応する能力が必要とされる。

4つ目として、協調行動のための知能がある。複数の知能を持った行動体が、それぞれの意図を理解し合いながら、目的を達成するための知能である。

これらの4つの知能は、それぞれの間に明確に線引きできるわけではなく、またそれが補いあったり、関連しあって全体として機能すると考えられる。ロボット知能を構成論的に考える上で、このように分けて考えることから出発しようとするわけである。本書では、これら4つの知能のなかで、主に反応行動のための知能に着目する。

Debian をつかう

ロボットの「知能」は、具体的にはコンピューター内のプログラムの形をとる。なおかつ、環境のなかで実際に行動するロボットに搭載されるコンピューターなので、PCなどとことなり、小型かつ軽量であることが必要である。そのために、ボードコンピューターを利用する。ボードコンピューターにネットワーク接続してプログラムなどの調整を行う必要があるが、ロボットに搭載されているということもあり、PCのような入出力装置が存在しない。つまり、キーボードとディスプレイがない状態でプログラミングを行う必要がある。実際にはボードコンピューターにそれらを接続して利用することも可能であるが、構成論的に「知能」を構築する過程で、入出力装置をロボットに接続したり切り離したりという作業を常に行なうことは、現実的ではない。

本書では、ボードコンピュータおよび基地となるPC上で利用するオペレーティングシステムとして Debian/GNU Linux（以下、単に Debian とよぶ）を用いる。Debian は GNU という名前がついていることからもわかるとおり、オープンソースとして管理されており、ロボット知能の研究のためには最適な OS のひとつとかんがえられる。

第1章 反応行動のための知能

ロボットは、図 1.1 に示したように、センサーを通じて環境からの情報を取り込む。その情報を基に行動を生成し、モーターを通して環境に働きかける。環境はロボットからの行動を受けて、そのダイナミクスにしたがって変化し、それを再びロボットがセンサーで受け取る。ロボットが環境の中で行動することは、このようなロボットと環境がつくるループを繰り返して行くことである。ロボットに着目すると、その構成

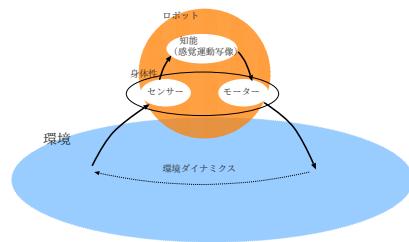


図 1.1: 環境と相互作用するロボット

要素はセンサー、感覚行動写像（知能）、モーターの 3 つに大別される。

動物でいえば、それぞれ感覚器官、中枢神経系、筋肉に対応する。

センサーをロボットのどの部分に、どのような角度でつけるのか、あるいはモーターの数や大きさ角度をどのように形成するのか。これらの配置は、実際につくり出されるロボットの行動に大きく影響するであろう。このような要因は、身体性と呼ばれる。

一方、行動を決定しているのは中枢神経系であり、ロボットであればコンピュータでそれを決定することになる。反応行動を実現するメカニズムの一つとして、感覚行動写像がある [1]。

1.1 走性における感覚行動写像

感覚入力は一つだけではなく、複数の値が考えられるのでベクトル \vec{s} でそれを表す。また、行動出力も複数ある場合を考えてベクトル \vec{r} で表す。従って、感覚行動写像は、感覚ベクトル \vec{s} から行動ベクトル \vec{r} への写像 f となる。

$$f : \vec{s} \rightarrow \vec{r} \quad (1.1)$$

たとえば、図 1.2 に示したように、2つのセンサーと、2つのモーターを持つ走行ロボットを考えてみる。左側のセンサー値を s_L 、右側のセンサー値を s_R とする。また、左側のモーター出力を r_L 、右側のモーター出力を r_R とする。感覚ベクトル \vec{s} と、行動ベクトル \vec{r} は、それぞれ

$$\vec{s} = \begin{pmatrix} s_L \\ s_R \end{pmatrix}, \quad \vec{r} = \begin{pmatrix} r_L \\ r_R \end{pmatrix} \quad (1.2)$$

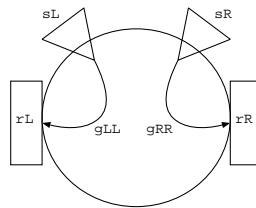


図 1.2: 2つのセンサーと2つのモーターをもつロボット例 (ncross)

と表すことができる。

1.1.1 非交差性線形写像

図 1.2 に示した走行ロボットは、感覚入力と行動出力が左右で非交差性結合をしている (noncrossover link) ので、このロボットを ncross ロボットと呼ぶことにしよう。

センサーからの入力値 s にゲイン g を掛けてそのまま出力値 r とする感覚行動写像の場合、

$$\begin{aligned} r_L &= g_{LL}s_L \\ r_R &= g_{RR}s_R \end{aligned} \tag{1.3}$$

と書ける（線形写像）。ただし、左 (L) のセンサーからの入力を左のモーターに出力する際のゲインを g_{LL} また、右 (R) のセンサーからの入力を右のモーターに出力する際のゲインを g_{RR} と表した。行列 \hat{G}_{ncross} を使っ

てこれを書くと,

$$\vec{r} = \hat{G}_{\text{ncross}} \vec{s} \quad (1.4)$$

$$\hat{G}_{\text{ncross}} \equiv \begin{pmatrix} g_{LL} & 0 \\ 0 & g_{RR} \end{pmatrix} \quad (1.5)$$

と, ゲイン行列 \hat{G}_{ncross} は, 2×2 の対角行列で表すことができる.

ゲインの値が正の場合, センサー値が大きくなればなるほど, モーター出力も大きくなるので, このことを興奮性結合とよぶ. すなわち, $g_{LL} > 0, g_{RR} > 0$ のことを興奮性結合という.

このようなセンサーとモーターで構成される比較的単純な走行ロボットにおいてお, センサーの種類によって様々な走性が生じる.

光センサーによる走性

図1.3に示したように, センサーを光センサーとして, 光源が左前方にある場合, 走行ロボットはどのような振る舞いをするだろう?

左側のセンサーの値の方が大きいので, 興奮性結合をもつロボットの場合, 左のモーターの方が出力がより大きくなるであろう. 結果として, 光から遠ざかろうとする行動が生じると考えられる.

1.1.2 超音波センサーによる走性

センサーが, 超音波センサー, すなわち距離センサーである場合にはどのような行動が生じるだろうか?(図1.4参照) 興奮性結合の場合, 右

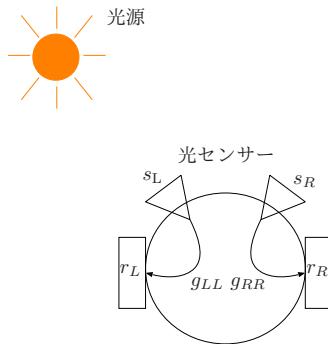


図 1.3: 光センサーをもつ走行ロボット (ncross-light) と光源

側のセンサーの方が障害物までの距離が大きいので、出力も大きく、結果として、障害物の方へぶつかっていく行動が生じると予想される。

1.1.3 交差性線形写像

図 1.5 に示した様に、センサーとモーターの結合が交差している場合を cross ロボットと呼ぶことにしよう。

このロボットの場合、出力 \vec{r} は

$$\vec{r} = \hat{G}_{\text{cross}} \vec{s} \quad (1.6)$$

$$\hat{G}_{\text{cross}} \equiv \begin{pmatrix} 0 & g_{RL} \\ g_{LR} & 0 \end{pmatrix} \quad (1.7)$$

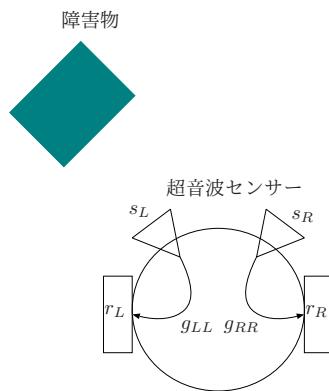


図 1.4: 超音波センサー（距離センサー）をもつ走行ロボット (ncross-ultra) と障害物

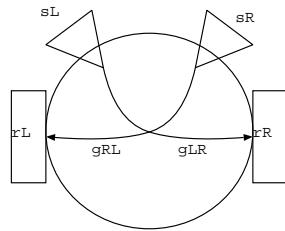


図 1.5: 交差性結合をもつ走行ロボット (cross)

と表される。

センサーが光センサーの場合、光がある方と逆のモーターの方が出力が大きくなるので、光の方に向かっていく走性を示すと予想される。

いっぽう、距離センサーを用いた場合、逆に障害物を避ける動きが予想される。

1.2 ゲインパラメータの取り得る範囲

ここまでは、簡単のためにゲインパラメータ $\{g_{LL}, g_{RR}\}$ などの値については詳しく述べなかった。これらの値が正の場合、興奮性結合、すなわち、センサー値が大きくなればなるほどモーター出力値も大きくなる場合を述べた。つまり、原理的にはモーター出力値はいくらでも大きな値を取りうる。

しかし、実際のロボットにおいてはモーターに対して無制限に大きな出力を動作させることは不可能である。センサー入力値の取り得る範囲を考慮して、モーター出力可能な範囲になるようにゲインパラメータの値を決める必要がある。

センサー値 s の値が以下の範囲の値をとるとする。

$$s_{\min} \leq s \leq s_{\max} \quad (1.8)$$

線形写像の場合、センサー値にゲインを掛けたものをモーター出力値と

するので、

$$\begin{aligned} gs_{\min} &\leq gs \leq gs_{\max} \\ gs_{\min} &\leq r \leq gs_{\max} \end{aligned} \tag{1.9}$$

である。

モーター出力値がとるべき範囲を

$$r_{\min} \leq r \leq r_{\max} \tag{1.10}$$

とすると、 g の最大値に関して、

$$\begin{aligned} gs_{\max} &\leq r_{\max} \\ g &\leq \frac{r_{\max}}{s_{\max}} \end{aligned} \tag{1.11}$$

という条件を満たさなければならない。ここで、センサー値はセンサーからの光の強さや、障害物までの距離の値であるから、 $s_{\max} > 0$ と考えてよいことを用いた。

同様に、 g の最小値について

$$g \geq \frac{r_{\min}}{s_{\min}} \tag{1.12}$$

でなければならない。

まとめると、ゲイン g は

$$\frac{r_{\min}}{s_{\min}} \leq g \leq \frac{r_{\max}}{s_{\max}} \tag{1.13}$$

の範囲内になければならない。

1.3 抑制性感覚運動写像

ここまででは、モーター出力値がセンサー入力値に比例して大きくなる線形写像を考えた。

ゲインの値 ($-g$) が負の値となる場合、センサー入力値が大きくなると、モーター出力値は小さくなる。モーター出力値が負になることが許される場合は、ここまで線形写像の表式をそのまま用いてもよいが、その値が負になることが出来ない場合には、定数 $c > 0$ を用いて

$$r = c - gs \quad (1.14)$$

とすることによってモーター出力値を正に保つことができる。この抑制性写像の場合にも、モーター出力値の取り得る範囲に応じて c と g の値の範囲を制限する必要があるのは、先に示した興奮性写像の場合と同様である。

1.4 非線形感覚運動写像

ここまででは、興奮性と抑制性の線形写像を用いる場合について述べた。非線形写像を用いることによって、より複雑な走性を創発する可能性がある。

たとえば、モーター出力値がセンサー値に反比例する以下のような場合、

$$r = \frac{g}{(s - s_{\min} + c)} \quad (1.15)$$

センサー値が大きくなればなるほどモーター出力が小さくなることは、抑制性線形写像と同じであるが、センサー値が小さい時にはモーター出力値が大きく変化し、センサー値が大きい場合にはモーター出力があまり変化しないという反応行動をつくることができる。

すなわち、この場合、

$$\frac{dr}{ds} = -\frac{g}{(s - s_{\min} + c)^2} \quad (1.16)$$

であるから、 $s_0 < s_1$ のとき

$$\left| \frac{dr}{ds}(s_0) \right| > \left| \frac{dr}{ds}(s_1) \right| \quad (1.17)$$

という性質をもつ感覚運動写像となる。

もうひとつの例として、興奮性および抑制性の2つの性質を同時にもつような写像の例をみてみよう。たとえば、

$$\begin{aligned} r &= \frac{c}{\cosh\{g(s - s_0)\}} \\ &= \frac{2c}{e^{g(s-s_0)} + e^{-g(s-s_0)}} \end{aligned} \quad (1.18)$$

の場合、 $s < s_0$ の領域では、興奮性であり、 $s > s_0$ の領域では抑制性の感覚運動写像となる（図1.6参照）。

1.5 感覚運動写像で行動するロボット

LEGO ロボットを使うと、ここまでに挙げた感覚行動写像で実際に走行するロボットを簡単につくることができる。第2章を参照してほしい。

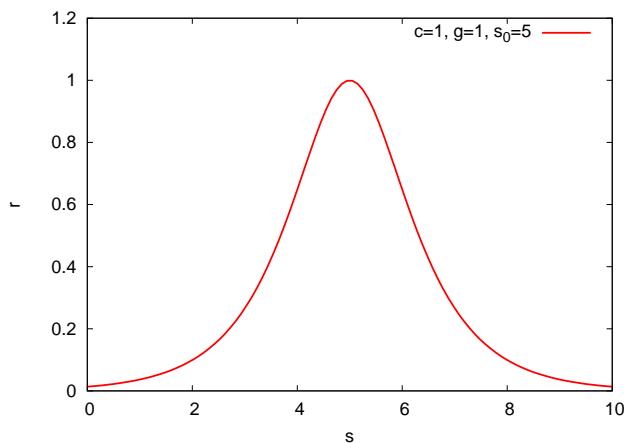


図 1.6: (1.18) 式による非線形感覚運動写像の例

第2章 単純な反応行動

NXT では、LEGO 社のファームウェアと Windows OS を用いてプログラミングしました。ここではその次世代である EV3 に Debian OS (ev3dev) を導入しロボット知能のプログラミングを行う方法を説明します。

2.1 EV3 上で Debian(ev3dev)

EV3 上で機能する Debian として ev3dev(www.ev3dev.org) があります。ev3dev は EV3 用に基本的な部分から再構築された linux でなので、各種のハードウェアドライバも、linux kernel の考え方に基づいてつくられているようです。したがって、ファイルの読み書きをするようにそれらをアクセスすることができます。

2.1.1 インストールと起動

イメージファイルをダウンロードし、microSD に焼付けて、EV3 の microSD スロットに挿入した後、スイッチを入れると ev3dev が起動します。

<https://github.com/ev3dev/ev3dev/releases> からイメージファイルをダウンロードします。例えば、ev3-ev3dev-jessie-2015-07-08.img.zip をダウンロードし、microSD カードが /dev/sdb というデバイスとして認識されている場合、

```
# unzip ev3-ev3dev-jessie-2015-07-08.img.zip  
# dd if=ev3-ev3dev-jessie-2015-07-08.img of=/dev/sdb
```

と実行するだけで、ev3dev 全体が microSD にインストールされます。Debian 全体なので、解凍にも dd にも数分から十数分の時間を要します。

microSD スロットに ev3dev イメージをコピーした microSD カードを挿入し、電源を入れると、数分の起動処理の後、図 2.1 のような画面が表示されれば正常起動が行われたことがわかります。数分間の起動処理

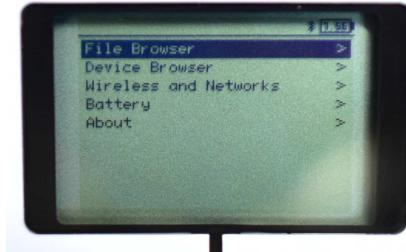


図 2.1: EV3 で microSD から ev3dev を起動した直後の液晶画面

を経たあとオレンジ色の LED が点滅したまま、図 2.1 の画面が表示されない場合があります。その場合には、図 2.2 に示した、A と B のボタ

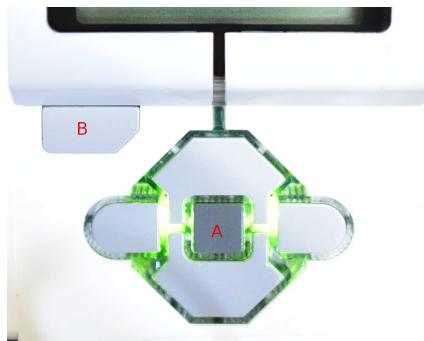


図 2.2: 起動リセットを実行する場合は、図中 A と B のボタンを同時に長押し（数秒）する。

ンを同時に長押しすると、システムがリセットされ、再起動処理が開始します。

ev3dev の起動が失敗するのは、1 番入力ポートに超音波センサーなどのセンサーを接続してある場合がほとんどです。センサーとの接続ケーブルを抜いた状態で起動すると、ev3dev の起動が途中で中断することは無くなります。

2.1.2 停止、再起動

図 2.2 における B ボタンを押すと、shutdown, reboot などを選択する画面が表示されます。A ボタンの上下にあるボタンでいずれかを選び、

A ボタンを押すと、シャットダウン（停止）あるいはリブート（再起動）プロセスが開始されます。

後述のように、USB ネットワークなどで、ev3dev にログインしたあとは、普通の Linux や Debian のように shutdown -h now のコマンドで、停止できます。

2.1.3 USB 有線 LAN を用いたセットアップ

USB-LAN アダプタ (Buffalo LUA3-U2-ATX) を USB ポートに挿すと、自動認識されます。

```
Wireless and Networks > All Network Connections > *Wired
```

を EV3 の画面で選び IPv4 のタブのなかで、IP address, Mask, Gateway を入力した後、一番下の apply を選ぶと、有線 LAN にアクセスできるようになります。IP address は 172.16.0.130 などのプライベートアドレスを用います。Mask は 255.255.0.0 とします。Gateway は、利用している環境に応じてルーターや NAT サーバなどのアドレスを入力します。

ssh などで Debian PC からログインした後は、普通の Debian として利用できます。

```
DebianPC> ssh root@aaa.bbb.ccc.ddd
```

root の初期パスワードは r00tme になっています。aaa.bbb.ccc.ddd は先に EV3 の画面で設定したプライベート IP アドレス、たとえば 172.16.0.130 などです。

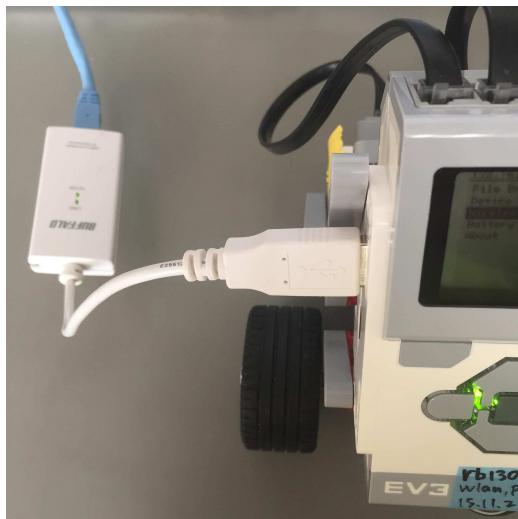


図 2.3: EV3 の USB ポートに有線 LAN を接続する。

/etc/resolv.conf のなかのアドレスを、たとえば 8.8.8.8¹ に変えると名前解決できるようになります。

```
root@ev3dev:~# vi /etc/resolv.conf
```

/etc/apt/sources.list のなかの URL を国内のミラーサーバに変更すると、パッケージのダウンロードなどが高速化されます。apt-get を用いてパッ

¹google public DNS(IPv4)。他に 8.8.4.4 がある

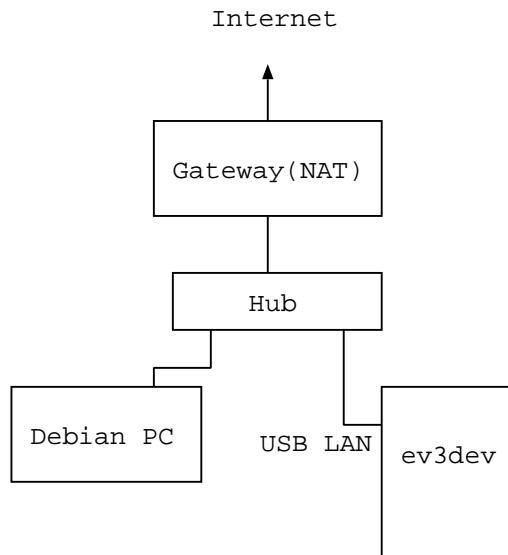


図 2.4: EV3 と Debian PC によるネットワーク構成

```
1 # Generated by Connection Manager  
2 nameserver 8.8.8.8
```

図 2.5: ev3dev における/etc/resolv.conf の例

ケージ情報をアップデートします。

```
1 #deb http://ftp.debian.org/debian jessie main contrib non-free
2 deb http://ftp.nara.wide.ad.jp/debian jessie main contrib non-free
3
4 deb http://ev3dev.org/debian jessie main
```

図 2.6: ev3dev における/etc/apt/sources.list の例

```
root@ev3dev:~# apt-get update
```

i386 や amd64 の CPU をもつ DebianPC に比べると、時間がかかりますが、それらの DebianPC と同様に apt-get を利用することができます。

gcc,pvm,git をインストールします。

```
root@ev3dev:~# apt-get install gcc
root@ev3dev:~# apt-get install pvm
root@ev3dev:~# apt-get install pvm-dev
root@ev3dev:~# apt-get install git-all
root@ev3dev:~# apt-get install make
root@ev3dev:~# apt-get install wireless-tools
```

2.1.4 USB ネットワーク

EV3 の miniUSB インターフェイスと Debian PC の USB インターフェイスを USB ケーブルで接続すると双方で、usb0 というインターフェイスとして認識されます。

Debian PC 側では、ifconfig コマンドなどを使って、IP アドレスなどを設定すれば、通常のネットワークインターフェイスと同様に使うこと

が出来ます（5.3節を参照）。

EV3側では、液晶画面上でUSBネットワークの確認や設定をします。TopメニューからWireless and Networks > USB > の順序で選択すると、USBネットワークの状態選択画面が表示されます。CDC (Inactive)の場合、もう一度それを選択して中央のAボタンを押すと、USBネットワークがActiveになります。IPアドレスも自動的に割り振られて、液晶画面の左上に表示されます。

自動的に割り振られたIPアドレスを変更したい場合は、

Wireless and Networks > All Network Connections > *Wired

とメニュー選択すれば、IPv4ネットワークのアドレスなどを手動で変更できます。

変更しない場合でも、液晶画面左上に表示されたIPアドレスに合わせて、Debian PC側でIPアドレスを設定すれば、pingやsshなどが利用できる様になります。

2.1.5 USB無線LAN

例として、NETGEARのWNA1100をEV3のUSBポートに挿入して、無線LANを利用する方法を示します。ファームウェアhtc_9271.fwを

http://wireless.kernel.org/download/htc_fw/から、ダウンロードします。バージョンがいくつかあり、常に更新されている。

基本的には最新のものを使うとバグや修正されてたり通信速度が改善している可能性があります。

htc_9271.fw を/lib/firmware に置きます。次に insmod コマンドで各モジュールをロードすると、WNA1100 が認識されます。

```
insmod /lib/modules/2.6.33-rc4/kernel/compat/compat.ko
insmod /lib/modules/2.6.33-rc4/kernel/net/wireless/cfg80211.ko
insmod /lib/modules/2.6.33-rc4/kernel/net/mac80211/mac80211.ko
insmod /lib/modules/2.6.33-rc4/kernel/drivers/net/wireless/ath/ath.ko
insmod /lib/modules/2.6.33-rc4/kernel/drivers/net/wireless/ath/ath9k\
    ath9k_hw.ko
insmod /lib/modules/2.6.33-rc4/kernel/drivers/net/wireless/ath/ath9k\
    ath9k_common.ko
insmod /lib/modules/2.6.33-rc4/kernel/drivers/net/wireless/ath/ath9k\
    ath9k.ko
insmod /lib/modules/2.6.33-rc4/kernel/drivers/net/wireless/ath/ath9k\
    ath9k_htc.ko
```

起動時に自動的に実行するためには、スクリプトとして保存しておき、/etc/init.d で insserv コマンドを実行して登録します。

wi-fi ネットワーク設定

EV3 の USB インターフェイスに、例えば NETGEAR WNA1100 を挿入すると、wlanX というインターフェイスとして自動的に認識されます。X の部分は 0, 1, … などの数字です。この順番を変更して、0 に戻す方法については、A.1.3 節を参照してください。

EV3 の液晶画面で

```
Wireless and Networks > Wi-Fi >
```

の順序でメニュー選択すると、周囲にある Wi-Fi 接続ポイントの ESSID が表示されますので、パスワードやキーが分かっている場合はそれらを選択します。IP アドレスは DHCP を利用して自動的に割り振られます。

これは EV3 をクライアントとして利用するだけならば設定が必要ないでの便利ですが、EV3 にログインしたり、ホストを指定して PVM でプロセス間通信をしたりする場合には不便です。ev3dev も Debian ですので、iwconfig などのコマンドを利用して Wi-Fi の設定を Ad-Hoc モードなどに固定して利用することも可能です。

iwconfig を利用するためには、

```
# dpkg -i libiw30_30-pre9-8_armel.deb  
# dpkg -i wireless-tools_30-pre9-8_armel.deb
```

これらの deb パッケージは、ネットワーク検索によって容易に探すことができますが、

```
https://packages.debian.org/jessie/armel/net
```

の中からリンクをたどることによってダウンロードできます。

ev3dev では、Wi-Fi は自動認識され DHCP でアドレスが取得されるようにデフォルトスクリプトが記述されているようです。無線 LAN 設定スクリプトを insserv コマンドで起動スクリプトに登録すれば、固定 IP を利用するスクリプトが自動実行されますが、デフォルトスクリプトが後から実行されると、DHCP 利用モードに戻ってしまいます。/etc/rc2.d などのディレクトリ内のスクリプトへのリンク名 SXXyyy などで、XX の数字をディレクトリ内で一番大きな数字にリネームしておくと、最後

に実行され、固定 IP がデフォルトで利用可能になります。

2.2 PVM

ev3dev にはデフォルトでは PVM はインストールされていないので、下記のように `dpkg` コマンドを使ってインストールできます。

```
# dpkg -i libpvm3_3.4.5-12.5_armel.deb  
# dpkg -i pvm_3.4.5-12.5_armel.deb
```

これらのパッケージは

```
https://packages.debian.org/jessie/libpvm3
```

などからダウンロードできます。

一般に、`dpkg` はパッケージ間の依存関係を解決してくれませんが、`iwconfig` と PVM を ev3dev 上にインストールする際には、依存関係の問題は生じません。

PVM の利用の仕方についてのより詳しい内容は、7 章を参照してください。

2.3 ev3c

ev3c は ev3dev が動いている EV3 上で、C 言語を使ってセンサーやモーターを制御するためのライブラリです。

```
$ git clone https://github.com/theZiz/ev3c.git
```

を実行すると、カレントディレクトリに ev3c/というディレクトリが作成され、その下に各種ライブラリやテストプログラムがダウンロードされます。

ev3c/というディレクトリがカレントディレクトリにできるので、そのディレクトリの中で make を実行すると、ライブラリやテストプログラムがコンパイルされて作成されます。Makefile の中で、コンパイラーは \$(CC) で参照されているので、クロスコンパイラーが CC という環境変数に指定されている必要があります。クロスコンパイラーの導入方法は後述します。

2.4 クロスコンパイルと Makefile

ev3dev が動いている EV3 に ssh を使ってログインすれば、他の Debian など Linux と同様のシェル環境を利用することができます。デフォルトでは C コンパイラーはインストールされていません。また、それをインストールしたとしても、EV3 に用いられている CPU では、PC と比べてコンパイルに非常に時間がかかります。

Debian PC 上で、EV3 用のプログラムをクロスコンパイルして、生成された実行形式ファイルを scp などを使って送ってから実行することで、センサーやモーターを利用することができます。

クロスコンパイラーを次の URL アドレスからダウンロードします。

```
https://sourcery.mentor.com/GNUToolchain/package4571/public  
/arm-none-linux-gnueabi  
/arm-2009q1-203-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
```

このクロスコンパイラは i386 環境用なので、 amd64 の環境では、 ia32-libs をインストールして、 i386 のバイナリが実行できる準備が必要です。

上記アーカイブを解凍すると、 arm-2009q1/ というディレクトリができるので、

```
# cp -rf arm-2009q1/ /usr/local/sbin/  
# export CC=/usr/local/sbin/arm-2009q1/bin/arm-none-linux-gnueabi-gcc
```

とすることで、 ev3c の Makefile を利用して、 make 可能になります。

図 2.7 に示した Makefile を、 アーカイブがコンパイルされた ev3c/ が存在するディレクトリ上に置いて make コマンドを実行すると、 TG に

```
1 TG=smmmap2  
2 CFLAGS = -O3 -lm -lpvm3  
3  
4 $(TG): $(TG).c ev3c/lib/ev3c.a  
5     $(CC) -o $@ $< ev3c/lib/ev3c.a $(CFLAGS)  
6  
7 clean:  
8     rm -rf $(TG)
```

図 2.7: ev3c/ というディレクトリがあるディレクトリでクロスコンパイルするための Makefile

指定された実行形式ができるので、それを scp などで ev3 に送れば、 実行できます。

また, gcc,pvm,pvm-dev,make がインストールされた ev3dev 上で, この Makefile を使って make を行うこともできます.

Makefile を使うと, 複数のソースコードから実行形式のターゲットをコンパイルして生成できます. make コマンドは, 複数のソースコードのうち, 更新されたものだけコンパイルを実行します. つまり, すでに最新のものは再コンパイルしないので, 時間の節約になります.

1 行目で, コンパイル後に作成されるターゲットを指定しています.

2 行目では, コンパイルオプションと, 数学ライブラリおよび PVM ライブラリを指定しています.

4 行目で, ターゲット (\$TG) が \$TG.c と /ev3c/lib/ev3c.a に依存していることを示しています.

5 行目で, このターゲットに対して行われるコンパイルを記述しています. \$@ は内部マクロで, ターゲット自身を意味します. \$_i は依存ファイルの左端のファイルを意味します. 今の場合, smmap2.c になります. ev3c アーカイブ(ev3c.a)とともにターゲットを生成しているのがわかります.

7, 8 行目は, make clean を実行した際に, ターゲットを削除することを意味します.

2.4.1 感覚運動写像と PVM

感覚運動写像による走行ロボット上のプログラム (smmap2.c) を Debian PC 上のプログラム (master2.c) から生成し、ロボットのセンサーで得たセンサー値をリアルタイムで master2 側に PVM で送信する例を示す。

master2.c

```
1 #include <stdio.h>
2 #include <pvm3.h>
3
4 #define PROGRAM "/root/smmap2"
5 #define SLAVE "sg154w"
6 #define MAXNHOST 256
7 #define SLEEP_TIME 200000
8 #define OUT_FILE "smmap2.xy"
9
10 #define sTAG 1
11 #define rTAG 2
12
13 int main(){
14     int i;
15     int info;
16     int nhost, narch, infos;
17     int tids[MAXNHOST];
18     int dum;
19     FILE *fp;
20     double rdum[256];
21     struct pvmhostinfo *hostp;
22
23
24     info = pvm_config(&nhost, &narch, &hostp);
25
26     printf("nhost=%d \n", nhost);
27     printf("narch=%d \n", narch);
28     for(i=0;i<nhost;i++)
29         printf("host name[%d]=%s \n", i, hostp[i].hi_name);
30
31     // SLAVE 一つだけ spawnする。
32     pvm_spawn(PROGRAM, (char**)0, 1, SLAVE, 1,tids);
33     printf("tid:%d\n",tids[0]);
```

```
34
35
36     fp=fopen(OUT_FILE, "w");
37     i=0;
38     while(1){
39         // SLAVEから送り返されてきたデータを受け取る.
40         if(pvm_nrecv(tids[0],rTAG)){
41             pvm_upkdouble(rdum,2,1);
42             printf("recv:%lf %lf\n",rdum[0],rdum[1]);
43             fprintf(fp,"%d %lf %lf\n",i,rdum[0],rdum[1]);
44         }
45
46         fflush(fp);
47         i++;
48         usleep(SLEEP_TIME);
49     }
50     fclose(fp);
51 }
52 }
```

smmap2.c(slave)

```
1  /*
2  2015 11.28 Yasushi Honda
3  cf. https://github.com/theZiz/ev3c
4  */
5
6 #include <stdio.h>
7 #include <math.h>
8 #include "ev3c/ev3c.h"
9 #include <pvm3.h>
10
11 // #define O_FILE "smmap2.xy"
12 #define I_MAX 300
13 #define SLEEP_TIME 200000
14 #define US_MAX 2550 // (mm)
15 #define MOTOR_MAX 500
16 #define B1 300 // (mm)
17 #define A1 0.005
18
19 #define rTAG 1
20 #define sTAG 2
21
22 #define SENSOR_PORT1 2
23 #define SENSOR_PORT2 4
24 #define SENSOR_MODE1 1
25 #define SENSOR_MODE2 1
26
27 int main(){
28     int i;
29     int j;
30     int left,right;
31     double us1d,us2d;
32     //FILE *fp;
33
34     int irecv;
35     int ptid;
36     double dum[256];
37
38     setlinebuf(stdout);
39
40     ptid=pvm_parent();
41
42     printf("こんにちは、超音波センサ値をsendします。 \n");
```

```
43 //Loading all sensors
44 ev3_sensor_ptr sensors = ev3_load_sensors();
45 ev3_sensor_ptr us1,us2;
46
47 us1=ev3_search_sensor_by_port(sensors,SENSOR_PORT1);
48 us2=ev3_search_sensor_by_port(sensors,SENSOR_PORT2);
49 ev3_open_sensor(us1);
50 ev3_open_sensor(us2);
51 ev3_mode_sensor(us1,SENSOR_MODE1);
52 ev3_mode_sensor(us2,SENSOR_MODE2);
53
54 ev3_motor_ptr motors = ev3_load_motors();
55 ev3_reset_motor(motors);
56 ev3_open_motor(motors);
57 ev3_reset_motor(motors->next);
58 ev3_open_motor(motors->next);
59
60 ev3_set_ramp_up_sp(motors, 10);
61 ev3_set_ramp_down_sp(motors, 10);
62 ev3_set_ramp_up_sp(motors->next, 10);
63 ev3_set_ramp_down_sp(motors->next, 10);
64 //fp=fopen(O_FILE,"w");
65 i=0;
66 while(i<I_MAX) {
67     ev3_update_sensor_val(us1);
68     ev3_update_sensor_val(us2);
69     us1d=(double)us1->val_data[0].s32;
70     us2d=(double)us2->val_data[0].s32;
71     right=MOTOR_MAX*(tanh(A1*(us1d-B1))+0.7)*0.5;
72     left=MOTOR_MAX*(tanh(A1*(us2d-B1))+0.7)*0.5;
73     printf("%d/%d \t %lf (-) %lf (-) : %d %d \n",i,I_MAX,us1d,us2d,left,right);
74     //fprintf(fp,"%lf %d \n",us1d,right);
75
76     dum[0]=us1d;
77     dum[1]=us2d;
78     pvm_initsend(0);
79     pvm_pkdouble(dum,2,1);
80     pvm_send(ptid,stAG);
81
82     /*
83     ev3_set_speed_sp(motors, left%1000);
84     ev3_set_speed_sp(motors->next, right%1000);
85     ev3_command_motor_by_name(motors, "run-forever");
86 }
```

```
87     ev3_command_motor_by_name(motors->next, "run-forever");
88     */
89
90     i++;
91     usleep(SLEEP_TIME);
92 }
93
94 //fclose(fp);
95
96 ev3_delete_sensors(sensors);
97 ev3_delete_motors(motors);
98 return 0;
99 }
```


第3章 収束する感覚行動写像

2014年ごろから、ドローンという言葉をニュースなどでも耳にするようになった。おもに、4つの回転翼をもつ飛行体（マルチコプター）のことをドローンと呼んでいるようである。単なるホビーとしてのラジコンから、空撮やインフラ調査撮影、また災害調査や農業への応用など爆発的な広がりを見せる勢いである。

このドローンは、単純な感覚運動写像で安定性を保っている飛行ロボットとみなすこともできる（図3.1参照）。

ここでは、飛行ロボットの1軸回転運動をとりだし、単純な2階微分方程式でモデル化する[10]。感覚運動写像[1]に時間遅れやノイズが存在しないと仮定した場合の、時間発展の振る舞いを解析する。

飛行ロボットの例として取り上げたが、この2階微分方程式は、摩擦のある振動や電気回路などをの挙動を表す方程式としてよく現れるものである。

解の求め方はいくつかある。変数分離や定数変化法を使って求める方法や、ラプラス変換して、それを部分分数にした後、逆変換して求める方法などが一般的かもしれない。ここでは、系の安定性を指數関数の指數部の値から直接議論しやすい行列形式を用いて解析する。



図 3.1: 著者が開発した飛行ロボットの一例

ここに示すように, 系に時間遅れが存在しない場合, 系は振動することはあっても, 不安定になることは無いことを理論的に示す. ここでいう不安定とは振動の振幅が徐々に大きくなって発散することを指す.

3.1 1 軸回転運動

4つのローターを用いた飛行ロボットは, 3次元空間を自由に並行移動し, 回転の自由度も3つの軸に関してそれぞれ持っているので, 合計6自由度の運動をする.

1 軸 (x-軸とする) の周りにおける回転運動のみを取り出して考えよう。図 3.2 に真横から飛行ロボットを見た図を示した。



図 3.2: 真横からみた飛行ロボット

また、図 3.3 に回転運動の概略図を示した。

水平状態からの回転角度 φ の運動方程式は、慣性モーメント I と回転軸の周りのトルク $l(L_1 - L_3)$ によって

$$I\ddot{\varphi} = l(L_1 - L_3) \quad (3.1)$$

と表される。ここで、 L_1, L_3 は 1 と 3 のそれぞれの位置における揚力である。また l は回転軸から 1 および 3 の位置までの距離を表す。

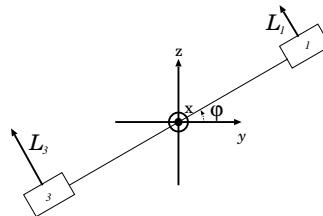


図 3.3: x -軸周りの飛行ロボットの回転角度 φ とローター 1,3 による揚力 L_1, L_3

いま感覚運動写像における感覚に当たるのは、ジャイロセンサーからの角速度 $\dot{\varphi}$ と、加速度センサーからの回転角度 φ の 2つであるとする。そして、運動に当たる出力は揚力 L_1, L_3 である。

線形写像を用いて感覚運動写像を

$$L_1 = -g_G \dot{\varphi} - g_A \varphi \quad (3.2)$$

$$L_3 = g_G \dot{\varphi} + g_A \varphi \quad (3.3)$$

とする。 $g_G > 0$ はジャイロセンサー値に対するゲイン、 $g_A > 0$ は加速度センサー値に対するゲインである。

揚力 (3.2),(3.3) を回転運動を表す (3.1) に代入すると、

$$I\ddot{\varphi} = -2lg_G \dot{\varphi} - 2lg_A \varphi \quad (3.4)$$

であるので、飛行ロボットの 1 軸回転運動は、

$$\ddot{\varphi} = -c_G \dot{\varphi} - c_A \varphi \quad (3.5)$$

と 2 階微分方程式で書ける。ただし、係数 c_G, c_A を

$$c_G \equiv \frac{2lg_G}{I} \quad (3.6)$$

$$c_A \equiv \frac{2lg_A}{I} \quad (3.7)$$

と定義した。 $l > 0, I > 0$ であるから、これらの係数もそれぞれ $c_G > 0, c_A > 0$ である。

時間 t を明示的に示して書き直せば、

$$\ddot{\varphi}(t) = -c_G\dot{\varphi}(t) - c_A\varphi(t) \quad (3.8)$$

となる。この微分方程式は典型的な 2 階微分方程式であり、 $\varphi(t)$ の解析的な関数を明示的に求めることが可能である。つまり、飛行ロボットの 1 軸周りの運動はノイズや筐体の歪による影響を除いて、厳密に数学的にあらかじめ予測することが可能なはずである。

実際には、時間遅れがあるため、飛行ロボットの運動は (3.8) 式で表される微分方程式の解とは異なった運動をするが、ここではひとまず、時間遅れがない場合、つまり (3.8) 式の性質を明らかにする。

3.2 数学的解析の概略地図

飛行ロボットの運動を数学的に解析するためには、行列の固有値やベクトルの一次独立性、また関数の級数展開など、いくつか数学的知識を必要とする。これらの知識は、線形代数や解析学で個別に学ぶ項目であるが、それらの知見が単なるバラバラな存在としてだけでなく、それぞ

これが関連して活躍し、飛行ロボットの運動という現実世界の問題を解析する上で結びつく。全体的な話の流れを図3.4に示した。

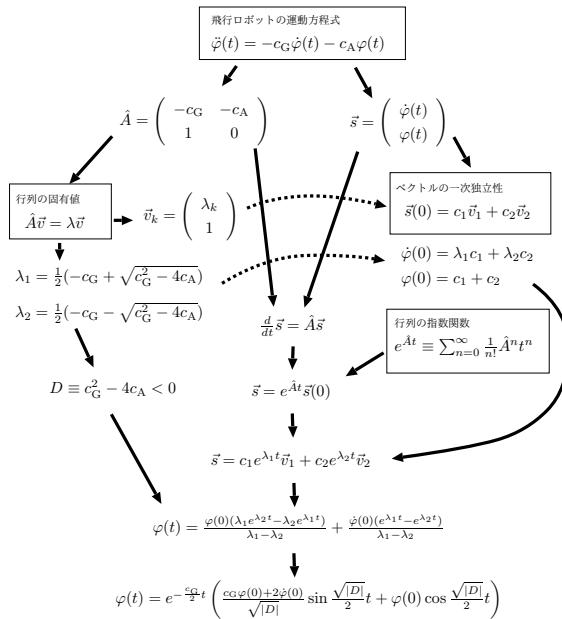


図3.4: 飛行ロボットの運動方程式に対する数学的解析地図。行列の固有値・固有ベクトル、ベクトルの1次独立性、および行列の指数関数などの数学的知識が組み合わせて活用される。

3.3 行列形式

2階微分方程式の解を求める方法はいくつかあるが、ここでは、行列形式を用いてその解析的な性質を調べることにしよう。行列の固有値がどのような値になるかを調べれば、運動が振動するのか、収束するのか、あるいは発散するのかということを、直接判定しやすくなる。

(3.8) 式から出発しよう。その左辺は、

$$\ddot{\varphi}(t) = \frac{d}{dt}\dot{\varphi}(t) \quad (3.9)$$

と書き改めることができる。また、すこしテクニカルな印象を受けるが、次の自明な式

$$\frac{d}{dt}\varphi(t) = \dot{\varphi}(t) \quad (3.10)$$

を(3.8)式と縦に並べて書くと、

$$\frac{d}{dt}\dot{\varphi}(t) = -c_G\dot{\varphi}(t) - c_A\varphi(t) \quad (3.11)$$

$$\frac{d}{dt}\varphi(t) = \dot{\varphi}(t) \quad (3.12)$$

となる。これらの微分方程式を行列形式で書くと

$$\frac{d}{dt} \begin{pmatrix} \dot{\varphi}(t) \\ \varphi(t) \end{pmatrix} = \begin{pmatrix} -c_G & -c_A \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \dot{\varphi}(t) \\ \varphi(t) \end{pmatrix} \quad (3.13)$$

と書ける。さらに、角速度 $\dot{\varphi}(t)$ と角度 $\varphi(t)$ は、飛行ロボットの状態を表す量であるから、状態ベクトル $\vec{s}(t)$ を

$$\vec{s}(t) \equiv \begin{pmatrix} \dot{\varphi}(t) \\ \varphi(t) \end{pmatrix} \quad (3.14)$$

と定義し、行列 \hat{A} を

$$\hat{A} \equiv \begin{pmatrix} -c_G & -c_A \\ 1 & 0 \end{pmatrix} \quad (3.15)$$

と定義すれば、(3.13) 式の行列形式は

$$\frac{d}{dt} \vec{s}(t) = \hat{A} \vec{s}(t) \quad (3.16)$$

と書きなおすことができる。つまり、(3.8) 式の 2 階微分方程式を行列 \hat{A} を導入することによって、より簡潔な (3.16) 式の 1 階微分方程式の形に書き改めることができた。

3.4 行列の指數関数を用いた解

(3.16) 式は時間 t に関する 1 階微分方程式のかたちをしている。その解 $\vec{s}(t)$ を具体的にもとめてみよう。 $\vec{s}(t)$ の n 階微分を $\vec{s}^{(n)}(t)$ と表す、つまり

$$\vec{s}^{(n)}(t) = \frac{d^n}{dt^n} \vec{s}(t) \quad (3.17)$$

と書き表すことにすると、テイラー級数を用いて、

$$\vec{s}(t) = \vec{s}(0) + \vec{s}^{(1)}(0)t + \frac{\vec{s}^{(2)}(0)}{2!}t^2 + \frac{\vec{s}^{(3)}(0)}{3!}t^3 + \dots \quad (3.18)$$

と書ける。”…”は、その先に無限に項がつづくことを表している。厳密には、この和が収束する条件を考慮する必要があるが、ここでは気に

しないでこのように表せるとして、先に進む。 $\vec{s}(t)$ を 1 回微分するたびに、 \hat{A} が掛けられるのであるから、 n 回微分するということは、 n 回 \hat{A} を掛けることに等しい。

$$\begin{aligned}
 \vec{s}^{(n)}(t) &= \frac{d^n}{dt^n} \vec{s}(t) = \frac{d^{n-1}}{dt^{n-1}} \frac{d}{dt} \vec{s}(t) \\
 &= \frac{d^{n-1}}{dt^{n-1}} \hat{A} \vec{s}(t) \\
 &= \frac{d^{n-2}}{dt^{n-2}} \hat{A} \frac{d}{dt} \vec{s}(t) \\
 &= \frac{d^{n-2}}{dt^{n-2}} \hat{A}^2 \vec{s}(t) \\
 &\quad \dots \\
 &= \hat{A}^n \vec{s}(t)
 \end{aligned} \tag{3.19}$$

ここで、行列 \hat{A} は時間 t に依存しないので、 \hat{A} と、微分演算子 $\frac{d}{dt}$ が交換可能であることを使った。これを、テイラー展開の式に使うと、

$$\vec{s}(t) = \vec{s}(0) + \hat{A} \vec{s}(0)t + \frac{\hat{A}^2 \vec{s}(0)}{2!} t^2 + \frac{\hat{A}^3 \vec{s}(0)}{3!} t^3 + \dots \tag{3.20}$$

となることがわかる。つまり、(3.16) 式が成り立つならば、(3.20) 式が成り立つことがわかった。

逆に、(3.20) 式が成り立つならば、(3.16) 式が成り立つことを示そう。

(3.20) 式の両辺を t で微分すると,

$$\begin{aligned}
 \frac{d}{dt} \vec{s}(t) &= \frac{d}{dt} \left(\vec{s}(0) + \hat{A} \vec{s}(0)t + \frac{\hat{A}^2 \vec{s}(0)}{2!} t^2 + \frac{\hat{A}^3 \vec{s}(0)}{3!} t^3 + \dots \right) \\
 &= \hat{A} \vec{s}(0) + \hat{A}^2 \vec{s}(0)t + \frac{\hat{A}^3 \vec{s}(0)}{2!} t^2 + \dots \\
 &= \hat{A} \left(\vec{s}(0) + \hat{A} \vec{s}(0)t + \frac{\hat{A}^2 \vec{s}(0)}{2!} t^2 + \dots \right) \\
 &= \hat{A} \vec{s}(t)
 \end{aligned} \tag{3.21}$$

つまり, (3.16) 式

$$\frac{d}{dt} \vec{s}(t) = \hat{A} \vec{s}(t)$$

が成り立つ. その過程で, “...”の中に無限に項がたくさんあるという事実を使った. どんなにたくさん項があったとしても, その数が有限であるかぎり, このようなことは起こらない. 無限にあるからこそ, (3.16) 式が得られたのである.

ここでわかったことを, 改めて確認すると,

$$\frac{d}{dt} \vec{s}(t) = \hat{A} \vec{s}(t) \tag{3.22}$$

が成り立つならば,

$$\vec{s}(t) = \vec{s}(0) + \hat{A} \vec{s}(0)t + \frac{\hat{A}^2 \vec{s}(0)}{2!} t^2 + \frac{\hat{A}^3 \vec{s}(0)}{3!} t^3 + \dots \tag{3.23}$$

がその解として成り立つということである. また, その逆も成り立つことも確認した. つまり, 両者はお互いに必要十分条件になっている, すなわち等価であることがわかる.

上の， $\vec{s}(t)$ の式をよく見ると，

$$\vec{s}(t) = \hat{f}(\hat{A}t)\vec{s}(0) \quad (3.24)$$

と書けることがわかる。ただし，

$$\hat{f}(\hat{A}t) \equiv \hat{I} + \hat{A}t + \frac{\hat{A}^2}{2!}t^2 + \frac{\hat{A}^3}{3!}t^3 + \dots \quad (3.25)$$

と定義した。ここで， $\vec{s}(0)$ は状態ベクトルの初期値であるが， $\hat{f}(\hat{A}t)$ という行列の右側に書かれていることに注意して欲しい。行列とベクトルの掛け算は，順序が入れ替わると意味が異なってくるのである。

このように， $\vec{s}(t)$ に対する解は，形式的には簡単なかたちとして求めることができた。しかしそく見ると，こまつたことに，その行列の掛け算を無限回まで行い，それをすべて足し合わせなければならないことがわかる。いくらコンピューターの演算処理が高速化したといえども，無限回掛け算や足し算を繰り返すことはできない。

つぎに示す行列の固有値を用いると，この問題は解決される。

3.5 行列の固有値と固有ベクトル

いま，行列 \hat{A} に対して，

$$\hat{A}\vec{v} = \lambda\vec{v} \quad (3.26)$$

を満たすベクトル \vec{v} とスカラー λ のことを，それぞれ \hat{A} の右固有ベクトル，および固有値という。これは， 2×2 の次元をもつ行列 \hat{A} について

てのみ言えることではなく、一般の正方行列に対しても成り立つ。行列 \hat{A} が対称行列であれば、右固有ベクトルと左固有ベクトルの区別はしなくとも、両者は同じものになるが、(3.15)式の行列の様に、非対称行列に関しては、右固有ベクトルと左固有ベクトルが異なるので、注意が必要である。ここではこのことに関して深入りはしない。ただし、後で具体的に求めるが、固有値、固有ベクトルは、行列の次数だけ存在するということは覚えておく必要がある。つまり、 2×2 の行列の場合、固有値が2つ、そしてそれぞれの固有値に対応して、固有ベクトルも2つ存在する。

右固有ベクトルと固有値を求める方法については、後回しにして、ここでは一旦それらが得られたとして、話を進めることにする。(3.25)式

$$\hat{f}(\hat{A}t) \equiv \hat{I} + \hat{A}t + \frac{\hat{A}^2}{2!}t^2 + \frac{\hat{A}^3}{3!}t^3 + \dots \quad (3.27)$$

の両辺に、右から \hat{A} の固有ベクトル \vec{v} を掛けると、

$$\begin{aligned} \hat{f}(\hat{A}t)\vec{v} &= \sum_{n=0}^{\infty} \frac{1}{n!} \hat{A}^n t^n \vec{v} \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} \hat{A}^n \vec{v} t^n \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} \lambda^n \vec{v} t^n \\ &= \left(\sum_{n=0}^{\infty} \frac{1}{n!} \lambda^n t^n \right) \vec{v} \\ &= e^{\lambda t} \vec{v} \end{aligned} \quad (3.28)$$

となる。固有値と固有ベクトルを使うと、行列をスカラーに置き換えられるので、行列の無限回掛け算という困難を指数関数 $e^{\lambda t}$ という形で、すっきり回避できるのである。

この式の意味するところは、行列 $\hat{f}(\hat{A}t)$ を \hat{A} の固有ベクトル \vec{v} に左から掛けることは、スカラー $e^{\lambda t}$ を掛けることに等しいということである。そこで、

$$\hat{f}(\hat{A}t) = e^{\hat{A}t} \quad (3.29)$$

と書き表すことにすれば、

$$e^{\hat{A}t} \vec{v} = e^{\lambda t} \vec{v} \quad (3.30)$$

という、自然な形で表すことが出来る。つまり、行列 \hat{A} の指数関数が、次の様にその無限級数で定義された。

$$e^{\hat{A}t} \equiv \hat{I} + \hat{A}t + \frac{\hat{A}^2}{2!}t^2 + \frac{\hat{A}^3}{3!}t^3 + \dots \quad (3.31)$$

さて、行列の固有値と右固有ベクトルはそれぞれ2つずつあるので、それらを λ_k, \vec{v}_k ($k = 1, 2$) と表す。 (3.28) 式から、

$$e^{\hat{A}t} \vec{v}_k = \vec{v}_k e^{\lambda_k t} \quad (k = 1, 2) \quad (3.32)$$

である。また、右固有ベクトル \vec{v}_1, \vec{v}_2 はそれぞれ、1次独立であり、任意のベクトルをその線形結合で表すことができる。

3.6 状態ベクトルに対する解

準備が整ったので、状態ベクトル $\vec{s}(t)$ の話に戻ろう。 (3.24) 式の $\vec{s}(0)$ を右固有ベクトルの線形結合で表す。

$$\vec{s}(0) = c_1 \vec{v}_1 + c_2 \vec{v}_2 \quad (3.33)$$

これを (3.24) 式に代入すると、

$$\begin{aligned} \vec{s}(t) &= e^{\hat{A}t} (c_1 \vec{v}_1 + c_2 \vec{v}_2) \\ &= c_1 e^{\lambda_1 t} \vec{v}_1 + c_2 e^{\lambda_2 t} \vec{v}_2 \end{aligned} \quad (3.34)$$

となり、状態ベクトルは、行列の固有値によってその時間変化が記述されることがわかる。言い換えると、行列の固有値から、飛行ロボットの運動を記述（予測）できるということである。

ここで、行列 \hat{A} の固有値と右固有ベクトルを求めておこう。行列の特性方程式¹を用いれば、即座に固有値に対する方程式を求めることができるが、ここでは素直に固有値、固有ベクトルの定義から、それらを求めてみる。 (3.26) 式に具体的に行列 (3.15) を用いると、

$$\begin{pmatrix} -c_G & -c_A \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (3.35)$$

と具体的に書くことができる。ここで、右固有ベクトル \vec{v} を

$$\vec{v} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (3.36)$$

¹ $\det(\hat{A} - \lambda \hat{I}) = 0$ を行列 \hat{A} に対する特性方程式よ呼び、これを満たす λ を求めれば、それが固有値となる。ただし、 \hat{I} は単位行列。 \det は行列式をとることを意味する。

と、 α, β を用いて表した。これらの値も、もちろん既知ではなく、固有値 λ を求める過程において同時に求められるべきものである。 2×2 行列とベクトルの積を展開してみると、

$$-c_G\alpha - c_A\beta = \lambda\alpha \quad (3.37)$$

$$\alpha = \lambda\beta \quad (3.38)$$

となる。(3.38) 式を (3.37) 式に代入すると、

$$-c_G\lambda\beta - c_A\beta = \lambda^2\beta \quad (3.39)$$

である。任意の β に対してこれが成り立つためには

$$\lambda^2 + c_G\lambda + c_A = 0 \quad (3.40)$$

である必要があり、2次方程式の解の公式を用いて固有値 λ は

$$\lambda_1 = \frac{1}{2} \left(-c_G + \sqrt{c_G^2 - 4c_A} \right) \quad (3.41)$$

$$\lambda_2 = \frac{1}{2} \left(-c_G - \sqrt{c_G^2 - 4c_A} \right) \quad (3.42)$$

と求められる。規格化条件 $|\beta| = 1$ がない場合には、 α, β の値は一意には定まらないが、その比は (3.38) 式によって与えられる。

3.7 状態ベクトルの具体的ななかたち

ここまで議論では、(3.41),(3.42) 式であたえられる固有値 λ_k が複素数になる可能性がある。同様に (3.38) 式から、右固有ベクトルも複素ベ

クトルになる可能性がある。したがって、(3.34)式で与えられる状態ベクトルも複素ベクトルになる可能性がある。状態ベクトルの要素は、もともと飛行ロボットの姿勢角度と角速度であるから、実数でなければならぬ。そもそも初期状態 $\vec{s}(0)$ が実数ベクトルであることからしても、奇妙に思える。そこで、初期状態を用いて状態ベクトルを具体的に表してみよう。初期状態 $t = 0$ においては、

$$\begin{pmatrix} \dot{\varphi}(0) \\ \varphi(0) \end{pmatrix} = c_1 \begin{pmatrix} \lambda_1 \\ 1 \end{pmatrix} + c_2 \begin{pmatrix} \lambda_2 \\ 1 \end{pmatrix} \quad (3.43)$$

であるので、この連立方程式から $\lambda_1 \neq \lambda_2$ の場合の c_1, c_2 を求めると、

$$c_1 = \frac{\lambda_2 \varphi(0) - \dot{\varphi}(0)}{\lambda_2 - \lambda_1} \quad (3.44)$$

$$c_2 = -\frac{\lambda_1 \varphi(0) - \dot{\varphi}(0)}{\lambda_2 - \lambda_1} \quad (3.45)$$

となる。したがって、これらを (3.34) 式

$$\vec{s}(t) = c_1 e^{\lambda_1 t} \vec{v}_1 + c_2 e^{\lambda_2 t} \vec{v}_2$$

に用いると、

$$\varphi(t) = \varphi(0) \left\{ \frac{\lambda_2 e^{\lambda_1 t} + \lambda_1 e^{\lambda_2 t}}{\lambda_1 - \lambda_2} \right\} + \dot{\varphi}(0) \left\{ \frac{e^{\lambda_1 t} - e^{\lambda_2 t}}{\lambda_1 - \lambda_2} \right\} \quad (3.46)$$

と得られる。無論、角速度 $\dot{\varphi}(t)$ も同様にして求めることが可能であるし、この式を時間微分して求めることもできる。

(3.46) 式には、行列 \hat{A} の固有値 λ_1, λ_2 が、まだそのまま含まれている。その値がどのような値をとっても、一般的に成り立つ式であると言える。

しかし、その値が複素数である場合には、実関数であるべき $\varphi(t)$ が本当に実数値を取る関数であるかどうか分かりにくい。

そこで、固有値を与える (3.41),(3.42) 式

$$\begin{aligned}\lambda_1 &= \frac{1}{2} \left(-c_G + \sqrt{c_G^2 - 4c_A} \right) \\ \lambda_2 &= \frac{1}{2} \left(-c_G - \sqrt{c_G^2 - 4c_A} \right)\end{aligned}$$

を具体的に用いてみよう。固有値の式からわかる通り $c_G^2 - 4c_A$ の値が正の時には、固有値の虚数部はゼロである。したがって、当然 $\varphi(t)$ の値も実数値となる。いっぽうその値が負の時には、固有値は複素数になる。
そこで、

$$D \equiv c_G^2 - 4c_A \quad (3.47)$$

と定義して、 $\varphi(t)$ の式を具体的に表してみると、

$$\begin{aligned}\varphi(t) &= \frac{\varphi(0)}{2} e^{-\frac{c_G}{2}t} \left\{ \frac{c_G}{\sqrt{D}} \left(e^{\frac{\sqrt{D}}{2}t} - e^{-\frac{\sqrt{D}}{2}t} \right) + \left(e^{\frac{\sqrt{D}}{2}t} + e^{-\frac{\sqrt{D}}{2}t} \right) \right\} \\ &\quad + \frac{\dot{\varphi}(0)}{\sqrt{D}} e^{-\frac{c_G}{2}t} \left(e^{\frac{\sqrt{D}}{2}t} - e^{-\frac{\sqrt{D}}{2}t} \right) \quad (3.48)\end{aligned}$$

となる。 $D < 0$ のときに虚数が現れる。

$$\sqrt{D} = i\sqrt{|D|} \quad (3.49)$$

ここで、 i は虚数単位である。これを用いると、

$$\begin{aligned}\varphi(t) &= \frac{\varphi(0)}{2} e^{-\frac{c_G}{2}t} \left\{ \frac{c_G}{i\sqrt{|D|}} \left(e^{\frac{i\sqrt{|D|}}{2}t} - e^{-\frac{i\sqrt{|D|}}{2}t} \right) + \left(e^{\frac{i\sqrt{|D|}}{2}t} + e^{-\frac{i\sqrt{|D|}}{2}t} \right) \right\} \\ &\quad + \frac{\dot{\varphi}(0)}{i\sqrt{|D|}} e^{-\frac{c_G}{2}t} \left(e^{i\frac{\sqrt{|D|}}{2}t} - e^{-i\frac{\sqrt{|D|}}{2}t} \right) \\ &= \varphi(0) e^{-\frac{c_G}{2}t} \left\{ \frac{c_G}{\sqrt{|D|}} \sin \frac{\sqrt{|D|}}{2}t + \cos \frac{\sqrt{|D|}}{2}t \right\} \\ &\quad + \frac{2\dot{\varphi}(0)}{\sqrt{|D|}} e^{-\frac{c_G}{2}t} \sin \frac{\sqrt{|D|}}{2}t\end{aligned}\tag{3.50}$$

となり、すべて実数と実数の三角関数を用いて表されるので、確かに $\varphi(t)$ は実数関数であることが確認できる。

この表式の物語っていることはどのようなことであろうか？大きき分けると、初期角度 $\varphi(0)$ に関する項と、初期角速度 $\dot{\varphi}(0)$ に関する項の 2 つの項から全体が構成されていることがわかる。

もうすこし $\varphi(t)$ の式の意味するところを見やすくするためにここで、減衰係数 γ を

$$\gamma \equiv \frac{c_G}{2}\tag{3.51}$$

と定義し、角振動数 ω を

$$\omega \equiv \frac{\sqrt{|D|}}{2}\tag{3.52}$$

と、それぞれ定義すると、

$$\varphi(t) = \varphi(0) e^{-\gamma t} \left(\frac{\gamma}{\omega} \sin \omega t + \cos \omega t \right) + \frac{\dot{\varphi}(0)}{\omega} e^{-\gamma t} \sin \omega t\tag{3.53}$$

というかたちに整理できる。ただし、ここまで話しあくまでも $D < 0$ の場合にたいするものである。

c_G, c_A は、運動方程式においては、その意味がわかり易かった。しかし、この時間発展の式のなかでそれを直接使うと、煩雑なかたちとなる。むしろ、 γ, ω を用いて表した方が、よりその振る舞いがイメージしやすくなる。

3.8 無矛盾性の確認

ここまで、 $D = c_G^2 - 4c_A < 0$ の条件が満たされる場合について $\varphi(t)$ の具体的な表式をもとめた。どうやって、最終結果 $\varphi(t)$ に間違いがないことを確認すればよいだろう？

(3.53) 式が、実際に (3.8) 式

$$\ddot{\varphi}(t) = -c_G\dot{\varphi}(t) - c_A\varphi(t)$$

を満たすことを確認する。 $\varphi(t)$ を時間微分して、 $\varphi(0), \dot{\varphi}(0)$ および三角関数の種類別にまとめると、

$$\dot{\varphi}(t) = -\left(\frac{\gamma^2}{\omega} + \omega\right)\varphi(0)e^{-\gamma t} \sin \omega t \quad (3.54)$$

$$-\frac{\gamma}{\omega}\dot{\varphi}(0)e^{-\gamma t} \sin \omega t \quad (3.55)$$

$$+\dot{\varphi}(0)e^{-\gamma t} \cos \omega t \quad (3.56)$$

となる。さらに時間微分して、 $\ddot{\varphi}(t)$ を求めるとき、

$$\begin{aligned}\ddot{\varphi}(t) &= \frac{\gamma}{\omega}(\gamma^2 + \omega^2)\varphi(0)e^{-\gamma t} \sin \omega t \\ &\quad - (\gamma^2 + \omega^2)\varphi(0)e^{-\gamma t} \cos \omega t \\ &\quad + \left(\frac{\gamma^2}{\omega} - \omega\right)\dot{\varphi}(0)e^{-\gamma t} \sin \omega t \\ &\quad - 2\gamma\dot{\varphi}(0)e^{-\gamma t} \cos \omega t\end{aligned}\tag{3.57}$$

と求められる。これが、(3.8)式の左辺である。ここで、

$$\gamma = \frac{c_G}{2}, \quad \omega = \frac{\sqrt{|D|}}{2}, \quad D = c_G^2 - 4c_A < 0$$

を使うと、

$$\gamma^2 + \omega^2 = c_A \tag{3.58}$$

という関係が成り立つ。

さらに、これらの関係を使うと、(3.8)式の右辺は、

$$-c_G\dot{\varphi}(t) - c_A\varphi(t) = -2\gamma\dot{\varphi}(t) - (\gamma^2 + \omega^2)\varphi(t) \tag{3.59}$$

と表される。ここに $\dot{\varphi}(t)$ および $\varphi(t)$ の各項を代入すればよい。すこし煩雑になるので、右辺について、各項べつにまとめると、

1. $\varphi(0)e^{-\gamma t} \sin \omega t$ の係数 : $\frac{\gamma}{\omega}(\gamma^2 + \omega^2)$
2. $\varphi(0)e^{-\gamma t} \cos \omega t$ の係数 : $-(\gamma^2 + \omega^2)$
3. $\dot{\varphi}(0)e^{-\gamma t} \sin \omega t$ の係数 : $\frac{\gamma^2}{\omega} - \omega$

4. $\dot{\varphi}(0)e^{-\gamma t} \cos \omega t$ の係数 : -2γ

となり、たしかに $\ddot{\varphi}(t)$ の各項の係数

$$\begin{aligned}\ddot{\varphi}(t) &= \frac{\gamma}{\omega}(\gamma^2 + \omega^2)\varphi(0)e^{-\gamma t} \sin \omega t \\ &\quad - (\gamma^2 + \omega^2)\varphi(0)e^{-\gamma t} \cos \omega t \\ &\quad + \left(\frac{\gamma^2}{\omega} - \omega\right)\dot{\varphi}(0)e^{-\gamma t} \sin \omega t \\ &\quad - 2\gamma\dot{\varphi}(0)e^{-\gamma t} \cos \omega t\end{aligned}$$

と一致する。

$\varphi(t)$ の具体的表式は、その出発点にした 2 階微分方程式をたしかに満たしていることが確認できた。行列形式や固有ベクトルなどの道具をつかって、たどり着いた目的地がもとの出発点と矛盾していないことが分かった。

数学的な論理展開に象徴されるような、「論理」というものは、意外と脆い側面がある。それぞれの論理には、それが成り立つ条件というものがある。そのような条件付き論理を、いくつも重ねて用いてたどり着く結論というものには、当然それぞれの条件が何重にも課されているので、ただでさえ危うい。さらに加えて、無矛盾性が確認されていない論理展開というものは、そうやすやすと信じてよいとは言えないものである。

3.9 振動

はなしを， $\varphi(t)$ すなわち，飛行ロボットの傾き角度の具体的な振るまいにもどそう。 $D = c_G^2 - 4c_A < 0$ の場合には， $\varphi(t)$ は (3.53) 式

$$\varphi(t) = \varphi(0)e^{-\gamma t} \left(\frac{\gamma}{\omega} \sin \omega t + \cos \omega t \right) + \frac{\dot{\varphi}(0)}{\omega} e^{-\gamma t} \sin \omega t \quad (3.60)$$

と表される。

c_G, c_A は，われわれが選べるパラメーターであった。いま， $c_G = 0$ と選ぶと， $\gamma = c_G/2 = 0$ であるから，

$$\varphi(t) = \varphi(0) \cos \omega t + \frac{\dot{\varphi}(0)}{\omega} \sin \omega t \quad (3.61)$$

である。また，これを時間微分すると

$$\dot{\varphi}(t) = -\varphi(0)\omega \sin \omega t + \dot{\varphi}(0) \cos \omega t \quad (3.62)$$

が得られる。

これは，式からも分かるとおり，振幅が初期状態 $\varphi(0), \dot{\varphi}(0)$ に依存する時間的な振動運動を表す。角振動数 ω は，いま $c_G = 0$ であるから， $\omega = \sqrt{|D|}/2 = \sqrt{c_A}$ と， c_A の値だけで決まる。ただし， $D < 0$ でなければならないので， $c_A > 0$ の条件を満たさなければならない。

$\omega = \pi, 3/2\pi, 1/2\pi$ の場合の， $\varphi(t), \dot{\varphi}(t)$ の時間変化を図 3.5 に表した。 ω の値が大きいほど振動の周期が短い。 ω の値は， c_A の値だけできまるのであるから， c_A の値，すなわち，PI 制御の比例ゲインはロボット運動の振動周期を左右するパラメーターであることがわかる。

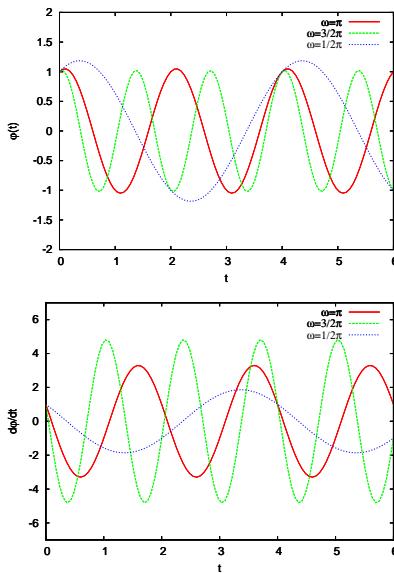


図 3.5: $c_G = 0$ の場合, $\varphi(t), \dot{\varphi}(t)$ の時間変化は振動をつづける.

$\varphi(t)$ と $\dot{\varphi}(t)$ はおたがいに位相がすこしずれた, 似たような振動を続けることがわかった.

ここで, つぎの量を計算してみると,

$$\{\omega\varphi(t)\}^2 + \dot{\varphi}(t)^2 = \omega^2\varphi(0)^2 + \dot{\varphi}(0)^2 \quad (3.63)$$

と, 右辺は時間 t に依存しない一定値となることがわかる. つまり, $\omega\varphi(t)$

を横軸, $\dot{\varphi}(t)$ を縦軸としてグラフを描くと, 半径が $\sqrt{\omega^2\varphi(0)^2 + \dot{\varphi}(0)^2}$ の円となることを意味している.

実際に, それらを図3.6に描画した. 確かに, 円が描かれた.

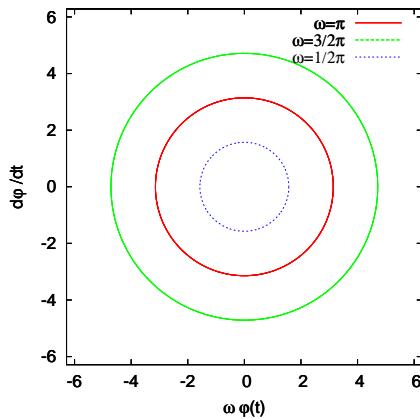


図3.6: $\omega\varphi(t), \dot{\varphi}(t)$ をそれぞれ横軸と縦軸に表した空間を相空間と呼ぶ. 運動は, 相空間の軌道として表される. 振動運動は, 相空間における円軌道となる.

この空間のことを, 相空間と呼び, 運動状態はこの空間内の1点として表される. また, 運動の時間発展は, 相空間内の軌道として描かれることになる. $c_G = 0$ の場合の振動運動は, 円軌道として表される.

3.10 減衰振動

具体的な時間発展の式

$$\varphi(t) = \varphi(0)e^{-\gamma t} \left(\frac{\gamma}{\omega} \sin \omega t + \cos \omega t \right) + \frac{\dot{\varphi}(0)}{\omega} e^{-\gamma t} \sin \omega t$$

が意味するところを理解するために、各項についてそれぞれくわしく見てみる。

第1項目は初期角度 $\varphi(0)$ がゼロでない時に現れる項である。その全体に $e^{-\gamma t}$ が掛かっているので、 $\gamma > 0$ すなわち $c_G > 0$ の場合には、初期角度の効果は時間が経てば減衰していくことを意味している。そして、減衰係数が大きければ大きいほど、速く減衰する（図 3.7(a) 参照）。次

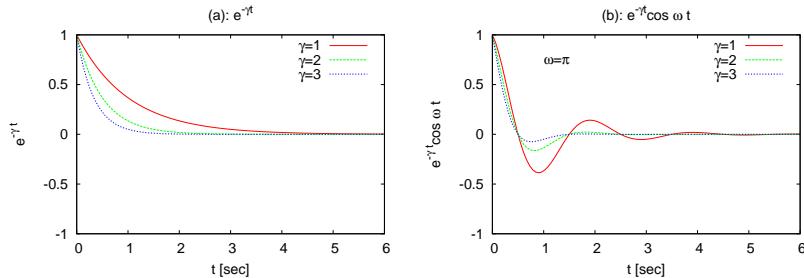


図 3.7: (a) 指数関数的な減衰の例。減衰係数 γ が大きくなるほど、速く減衰する。(b) 振幅が指数関数的に減衰する \cos 関数の例。

に、この初期角度の減衰項の中の、 \cos 関数で表された項がどのように時間変化するのかを図 3.7(b) に示した。例として角振動数 $\omega = \pi$ の場合

を示した。振動しながら、その振幅が減衰していくことがわかる。このような関数の振る舞いの場合に、 $e^{-\gamma t}$ のことを包絡関数、あるいは包絡線と呼ぶ。

$\varphi(t)$ の時間発展を表す式の、第1項にはもうひとつの項が含まれていることを忘れてはならない。この項は

$$e^{-\gamma t} \frac{\gamma}{\omega} \sin \omega t \quad (3.64)$$

というかたちをしている。この項自身の減衰の様子を図3.8(a)に示した。この項が \cos 関数の減衰に加えられたものが実際の振動減衰の時間発展

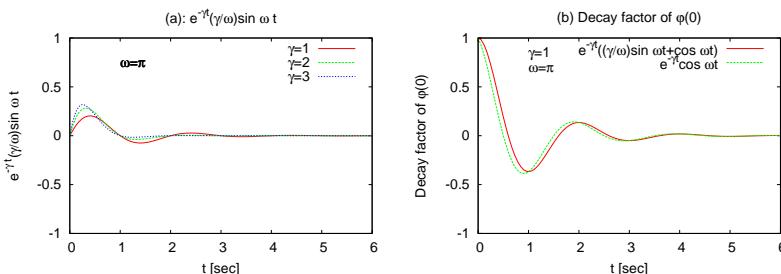


図 3.8: (a) $e^{-\gamma t}(\gamma/\omega) \sin \omega t$ の減衰の様子。 (b) $e^{-\gamma t}((\gamma/\omega) \sin \omega t + \cos \omega t)$ の減衰の様子。

(図3.8(b)参照)である。 \cos 関数だけの場合と比べて、やや位相が遅れた減衰を示すが、結局指数関数的に減衰する包絡線によってその振動は抑えられて、急速にゼロに収束する様子がわかる。

さらに初期角速度の項があるが、この項は初期角度の \sin 関数と類似

のかたちをしており、その時間発展の様子も、まったく同様の振る舞いをすることが容易に理解できる。

3.11 比例ゲインと積分ゲイン空間における相図

次に、固有値を与える(3.41))式と、 c_G, c_A の値を用いて、 λ_1 が実際にどのような値をとるかを図示してみよう。それらの実数部の正負によって、状態ベクトルの発散、収束が決まる。図3.9に λ_1 の実部と虚部の値を c_G, c_A の関数として示した。まず、虚部を見ると、 $c_A < c_G^2/4$ の領域では、 $\text{Im}(\lambda_1)=0$ なので、系は振動しない。一方 $c_A > c_G^2/4$ の領域では、 $\text{Im}(\lambda_1) > 0$ となり、系は振動する。 $\text{Im}(\lambda_1)$ の大きさがその振動数に対応するが、 $c_A = c_G^2/4$ から離れれば離れるほど、振動数が大きくなることがわかる。

つぎに、実数部を見るとすべての領域で負の値になる。包絡曲線は図に示したすべての領域でゼロに収束することがわかる。境界 $c_A = c_G^2/4$ の近傍において、比較的 $|\text{Re}(\lambda)|$ のあたいが大きくなる、すなわち速く収束する傾向にあることがわかる。

3.12 遷移行列を用いた漸化式

このように、系の振る舞いは行列の固有値を求める問題に帰着され、その固有値がもともとの系のゲインの値 c_G, c_A によって決まることがわ

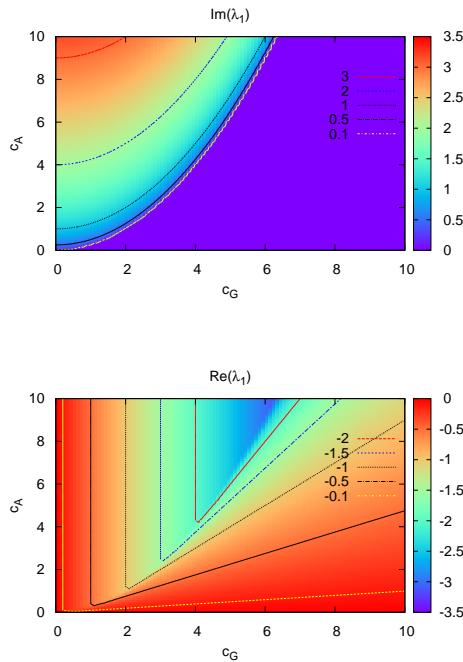


図 3.9: c_G, c_A の関数としての $\text{Im}(\lambda_1)$ と $\text{Re}(\lambda_1)$

かる。すでに明示的に系の振る舞いが表現されているが、ここでは漸化式の形式を用いても以上の議論を定式化可能であることを示す。

微分方程式 (3.16) から、微小時間 Δt に対して

$$\Delta \vec{s}(t) = \hat{A} \vec{s}(t) \Delta t \quad (3.65)$$

が成り立つ。これを用いると Δt 後、すなわち $t + \Delta t$ における状態ベクトル $\vec{s}(t + \Delta t)$ は

$$\begin{aligned} \vec{s}(t + \Delta t) &= \vec{s}(t) + \Delta \vec{s}(t) \\ &= \vec{s}(t) + \hat{A} \vec{s}(t) \Delta t \\ &= (\hat{I} + \Delta t \hat{A}) \vec{s}(t) \end{aligned} \quad (3.66)$$

となる。ここで、 \hat{I} は単位行列

$$\hat{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (3.67)$$

である。また、スカラー量 Δt と行列やベクトルとの積は、順序に依存しない性質を用いた。 $\vec{s}_{i+1} = \vec{s}(t + \Delta t)$, $\vec{s}_i = \vec{s}(t)$ と置き換えて書くと (3.66) 式は

$$\vec{s}_{i+1} = \hat{T} \vec{s}_i \quad (3.68)$$

と漸化式のかたちに書くことが出来る。ただし、ここで遷移行列 \hat{T} を

$$\hat{T} \equiv \hat{I} + \Delta t \hat{A} \quad (3.69)$$

と定義した。行列 \hat{A} の右固有ベクトル \vec{v} を用いると

$$\begin{aligned} \hat{T} \vec{v} &= (\hat{I} + \Delta t \hat{A}) \vec{v} \\ &= (1 + \Delta t \lambda) \vec{v} \end{aligned} \quad (3.70)$$

であるから、遷移行列 \hat{T} の固有値 η は

$$\eta = 1 + \Delta t \lambda \quad (3.71)$$

である。したがって、元の \hat{A} の固有値 λ は、遷移行列の固有値 η を用いて

$$\lambda = \frac{\eta - 1}{\Delta t} \quad (3.72)$$

と求められる。

この関係を用いれば、漸化式(3.68)から、系の振る舞いを決める \hat{A} 行列の固有値 λ を求めることができる。

第4章 時間遅れのある反応行動

回転翼飛行ロボットは回転運動に関して3つの自由度を持っているが、その中の一つだけを取り出して着目する。その運動方程式を求め、ホバリングを目標姿勢とするPI制御について説明する。

小型の飛行ロボットの姿勢変化は比較的短い時間に変化する。そのため、信号伝達やノイズ除去演算による時間遅れが相対的に無視できない[8, 9, 10]。また、同様の理由で、制御の離散性も影響が大きいと考えられる。

さらに、ローター角速度の緩和時間も、時間遅れや離散時間間隔と同程度の大きさを持つので、これらがすべて考慮されていなければならぬと考えられる。

また、制御に必要なパラメーターの決定には、飛行シミュレーターの存在が不可欠なので、すべてを明示的な数式によって表現しておくことが重要と考える。

4.1 回転運動方程式

回転翼飛行ロボットは、図 4.1 に示したように、4 つのローター ($k = 1, 2, 3, 4$) を持つ。ローター 1,3 を結ぶ軸を y - 軸、ローター 2,4 を結ぶ

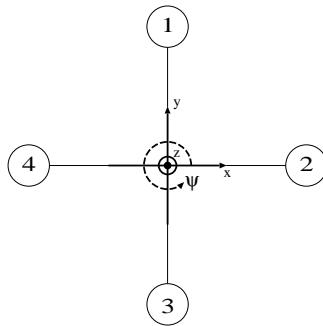
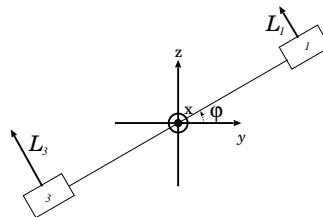


図 4.1: 4 回転翼飛行ロボット上の xy 座標軸およびローター ($k = 1, 2, 3, 4$) の配置。

軸を x - 軸とする。 z - 軸は xy 平面に垂直とする。いま、飛行ロボットの 1 軸 (x -軸とする) の周りだけで回転運動を考える（図 4.2 参照）。飛行ロボットの x -軸周りの慣性モーメントを I_x 、また回転角度を φ とすると回転運動方程式は

$$I_x \ddot{\varphi} = \tau_x \quad (4.1)$$

である。ここで、 τ_x は 2 つのローターによる揚力差から生じる、 x -軸周りのトルクである。すなわち、 L_k を k 番目のローターによって生じた

図 4.2: x - 軸周りの飛行ロボットの回転角度 φ とローター 1,3 による揚力 L_1, L_3

揚力とすると,

$$\tau_x = r(L_1 - L_3) \quad (4.2)$$

である。ここで、 r は飛行ロボットの中心から、ローターまでの距離である。揚力 L_k は、一般にローター角速度 ω_k の 2 乗に比例する。すなわち、

$$L_k = a_L \omega_k^2 \quad (4.3)$$

である。図 4.3 に、実際に測定したローター角速度と、その揚力の関係をプロットした。

4.2 反応行動としての感覚運動写像

反応行動とは、単純化して言えば、環境情報（の一部）を感知して、それに応じて行動を起こすことを指す。それに伴って環境が変化し、そ

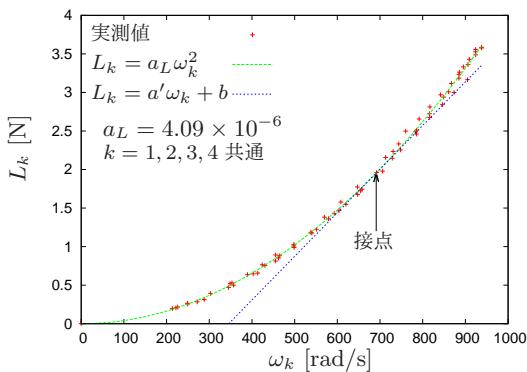


図 4.3: 揚力 L_k は、ローター角速度 ω_k の 2 次関数である。ローターの番号 $k = 1, 2, 3, 4$ に対して共通。

れをまたロボットが感知して …, という風に、ロボットと環境が相互作用のループを続けていく（図 1.1 参照）。

ここで少し誤解が生じるかもしれない。小さなロボットがちょっと行動したぐらいで「環境」が変化するであろうか？という疑問である。例えば、光を好む反応行動を起こすロボットがあるとする。ここで光だけを「環境」とすると、ロボットが移動しても「環境」は変化しない。そうではなくて、ロボットと「環境」全体を含めて環境とみなす。すると、ロボットが光の方に近づく行動を取れば、ロボットが感知する光は変化（増加）するはずである。

飛行ロボットで言えば、ローター回転速度を変化させて、姿勢や位置が変化すれば、ロボット自身が感知する角度や角速度が変化する。それ

らの情報まで含めて環境情報と考えるわけである。

反応行動をとるための構成要素は、大きく分けると3つある。環境を感じる部分、感知した情報に応じて行動を決定する部分、そして行動を起こす部分の3つである。飛行ロボットにおいて、それぞれの役割を担う要素を、図4.4に示した。

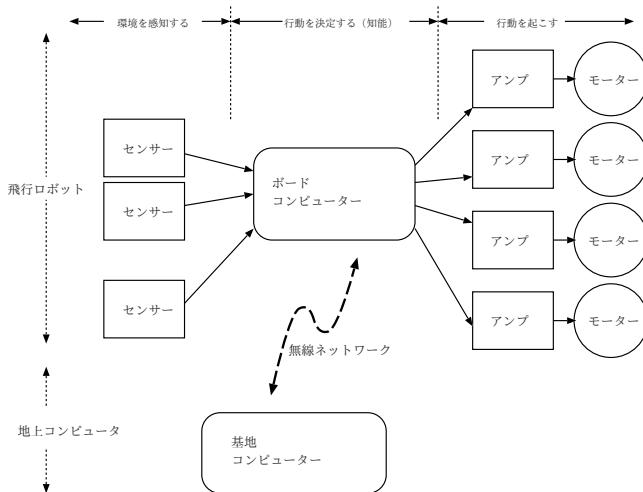


図4.4: 飛行ロボットのシステム構成概略図

環境を感じる部分は、センサーである。人間で言えば、五感の感覚器に相当する部分である。

行動の決定は、ボードコンピューターが主に担う。ただし、生物と異なる点は、その部分が飛行ロボットの身体内部にある必要性は必ずしも

ないということである。ボードコンピューターは、無線ネットワークを通じて、基地コンピューターと情報のやり取りが可能なので、行動の決定は基地コンピューターが行なって、その結果を行動を起こす部分に渡すことも可能である。

反応行動の範囲であれば、行動を決定するための手続きに要する計算量が比較的少ないことが予想されるので、この必要はあまりないかもしれない。しかし、計画行動、適応行動といったより複雑な知能が要求される判断においては、ボードコンピューターの計算処理能力では間に合わないかもしれない。基地コンピューターの計算処理能力を動員する必要があるかもしれないし、またそれが可能である。この点は、生物の知覚と大きく異なり、ロボット知能の可能性を感じる部分の一つである。

行動を起こす部分は、アンプとモーターから成る。地上を走行したり、歩行したりするロボットでは、この部分をどのような構造と機能にするかが大きな比重を占めると考えられる。いっぽう、飛行ロボットでは、この部分は比較的単純な構造と機能であると言える。

ただし、この部分のもつ時間遅れの要素は、十分考慮されなければならない。ローターは、アンプからの信号によって、回転速度が決められるが、その信号の値に応じて瞬間に（時間遅れおよび緩和時間なしで）回転速度が希望の値に変化するわけではない。

ローター制御値と角速度の関係

緩和時間がロボットの運動に要する時間に比べて十分に小さいと見なせれば、ローターの角速度は、アンプへの制御値 Ω_k だけの関数とみなし

ても妥当であると考えられるが、実際は、後節 4.4 で述べるように、緩和時間が数十ミリ秒ほどなので、無視できない。したがって、ローターの角速度 ω_k は、 Ω_k と時刻 t の関数である。

$$\omega_k = f(\Omega_k, t) \quad (4.4)$$

この関数 f の具体的な形は、ローター、モーターおよびアンプの種類に大きく依存するので、実測によって決定する。まず、十分に時間が経った時のローターの角速度を実測し ω_p と表すことにする。すなわち、

$$\omega_p \equiv f(\Omega, \infty) \quad (4.5)$$

と定義する。この ω_p が、次節のローターの回転運動における、角速度の目標値である。実測の結果、この $f(\Omega, \infty)$ は Ω の 1 次関数で近似できることが分かっている（図 4.5 参照）。

$$\omega_p = a\Omega + b \quad (4.6)$$

一次関数による感覚運動写像

次に、 Ω の値を $\varphi, \dot{\varphi}$ などの値に応じて、目標姿勢を達成するために決定しなければならない。すなわち、

$$\Omega = \pi(\varphi, \dot{\varphi}, \ddot{\varphi}, \dots) \quad (4.7)$$

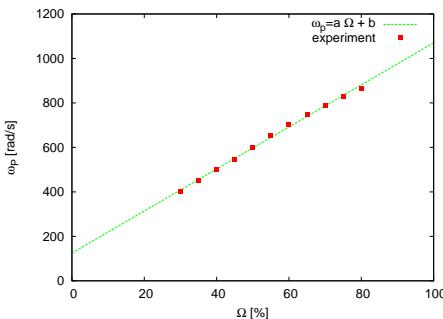


図 4.5: $f(\Omega, \infty)$ の実測値をつかったフィッティング. $a = 9.453[\text{rad/s\%}]$, $b = 124.81[\text{rad/s}]$

$\varphi, \dot{\varphi}, \ddot{\varphi}, \dots$ を感覚, Ω をそれに対する応答であるとみなせば, 関数 π は, 感覚運動写像と言う事もできる [1]. また, $\varphi, \dot{\varphi}, \ddot{\varphi}, \dots$ を状態, Ω をそれに対する行動とみなせば, 強化学習の用語を用いて π を決定論的方策関数と言う事もできる [6].

感覚運動写像 π の最も直感的で基本的な方法は, 反応を感覚の 1 次関数で構成する方法である.

$$\begin{aligned} \Omega_1 &= -K_G(\dot{\varphi} - \dot{\varphi}_a) - K_A(\varphi - \varphi_a) \\ &\quad - K_D(\ddot{\varphi} - \ddot{\varphi}_a) + c(\varphi) \end{aligned} \tag{4.8}$$

$$\begin{aligned} \Omega_3 &= +K_G(\dot{\varphi} - \dot{\varphi}_a) + K_A(\varphi - \varphi_a) \\ &\quad + K_D(\ddot{\varphi} - \ddot{\varphi}_a) + c(\varphi) \end{aligned} \tag{4.9}$$

ここで, $\varphi_a, \dot{\varphi}_a$ および $\ddot{\varphi}_a$ は目標姿勢 (aim) を表す. これら 3 つの項にお

けるゲインパラメーター K_G, K_A および K_D の値は、目標姿勢を達成するために調節されなければならない。それらの値は、飛行ロボットの形状や、ローター、モーターの種類などに依存するものであるし、目標姿勢の種類によっても変化するであろう。実験による、試行錯誤でそれらを決定する方法もあるし、限界感度法などの種々の方法も考案されている。遺伝的アルゴリズムや、教師データが準備できる場合であれば、ニューラルネットを用いることも考えられる。

これら、3つの項は、ロボットの傾きや角速度を制御する項であるが、一方、 $c(\varphi)$ は、飛行ロボットを重力に逆らって持ち上げるために必要な項である。この項は、傾きや角速度には影響を与えないが、ゼロにしてしまうと飛行ロボットは飛行状態を維持できないため、必要な項である。

ホバリング

いま、飛行ロボットのホバリング、すなわち

$$\varphi_a = 0, \quad \dot{\varphi}_a = 0, \quad \ddot{\varphi}_a = 0 \quad (4.10)$$

を目標姿勢とすると、

$$\Omega_1 = -K_G\dot{\varphi} - K_A\varphi - K_D\ddot{\varphi} + c(\varphi) \quad (4.11)$$

$$\Omega_3 = +K_G\dot{\varphi} + K_A\varphi + K_D\ddot{\varphi} + c(\varphi) \quad (4.12)$$

である。さらに、 $K_D = 0$ と固定した PI 制御の場合、

$$\Omega_1 = -K_G\dot{\varphi} - K_A\varphi + c(\varphi) \quad (4.13)$$

$$\Omega_3 = +K_G\dot{\varphi} + K_A\varphi + c(\varphi) \quad (4.14)$$

という式でローター操作値 Ω が決定される。

4.3 時間遅れと離散制御

ロボットの回転角度（傾き） φ と、その角速度 $\dot{\varphi}$ は時刻 t に依存するので、それを明示的に表すと(4.13), (4.14)式は、

$$\Omega_1(t) = -K_G\dot{\varphi}(t) - K_A\varphi(t) + c(\varphi(t)) \quad (4.15)$$

$$\Omega_3(t) = +K_G\dot{\varphi}(t) + K_A\varphi(t) + c(\varphi(t)) \quad (4.16)$$

と表される。 φ は加速度センサー、また $\dot{\varphi}$ はジャイロセンサーで観測する。 Ω_i を計算してローターが実際に回転を変化させ始める時刻 t は、それらセンサー値の信号伝達やノイズ除去の後なので、センサーでそれらを観測した瞬間よりも、少し遅れているはずである。逆の言い方をすれば、ローターは現在 t より少し前 $t - \delta$ のセンサー情報を元に決められた操作値を受け取ることになる。

$$\begin{aligned} \Omega_1(t) &= -K_G\dot{\varphi}(t - \delta_G) - K_A\varphi(t - \delta_A) \\ &\quad + c(\varphi(t - \delta_A)) \end{aligned} \quad (4.17)$$

$$\begin{aligned} \Omega_3(t) &= +K_G\dot{\varphi}(t - \delta_G) + K_A\varphi(t - \delta_A) \\ &\quad + c(\varphi(t - \delta_A)) \end{aligned} \quad (4.18)$$

ここで、ジャイロセンサーで観測される $\dot{\varphi}$ の時間遅れを δ_G また、加速度センサーで観測される φ の時間遅れを δ_A と表した。信号伝達による遅

れ（むだ時間）は約 30[ms] であることが分かっている [8]. また、ローパスフィルターなどによるノイズ除去演算によって、さらに時間遅れは増加する. 30 個のデータによる単純移動平均によってセンサー値のノイズを低減すると、合計約 280[ms] ほどの時間遅れが生じると考えられる [8].

さらに、ボードコンピューターは常に連続的に操作値をモーターに送り続けているわけではなく、1 秒間に 30~50 回程度の頻度で離散的にモーターを制御している. したがって、その間隔は 20~33[ms] である. この値は、先に述べた時間遅れや、後ほど述べる、ローター回転運動の緩和時間とほぼ同等の大きさを持っている. そのためこの制御時刻の離散性を考慮しなければならない. この離散的な制御の時間間隔を Δ とすると、時刻 t ($j\Delta \leq t < (j+1)\Delta$) におけるローター 1,3 の操作値は、

$$\begin{aligned} \Omega_1(t) = & -K_G \dot{\varphi}(j\Delta - \delta_G) - K_A \varphi(j\Delta - \delta_A) \\ & + c(\varphi(j\Delta - \delta_A)) \end{aligned} \quad (4.19)$$

$$\begin{aligned} \Omega_3(t) = & +K_G \dot{\varphi}(j\Delta - \delta_G) + K_A \varphi(j\Delta - \delta_A) \\ & + c(\varphi(j\Delta - \delta_A)) \end{aligned} \quad (4.20)$$

$$\text{ただし } j = 1, 2, 3, \dots \quad (4.21)$$

である.

4.4 ローターの回転運動における緩和時間

ローターおよびモーターの慣性モーメントを一体として J で表すことにする。この節における議論は、4つのローター全てにおいて共通していることなので、ローターを区別する添字 i を省略して、ローターの角速度を ω で表す。また、その目標値を ω_p とする。

運動方程式

ω を変化させる角加速度は、 $\omega_p - \omega$ に比例する回転トルクによって与えられると考えられる。すなわち、

$$J\dot{\omega} = \alpha(\omega_p - \omega) \quad (4.22)$$

という微分方程式にしたがって、ローターの角速度は変化すると考えられる。 α はモーター、ローターおよびアンプの性能を反映したパラメータである。

角速度の時間変化

この微分方程式の同次形

$$J\dot{\omega} = -\alpha\omega \quad (4.23)$$

の解は

$$\omega = Ae^{-\frac{\alpha}{J}t} \quad (4.24)$$

である。非同次微分方程式である(4.22)式の解を求めるために、 A を定数ではなく、 t の関数であると考えると、 $\omega(t) = A(t)e^{-\frac{\alpha}{J}t}$ であるから、

$$\begin{aligned}\dot{\omega} &= \dot{A}e^{-\frac{\alpha}{J}t} - \frac{\alpha}{J}Ae^{-\frac{\alpha}{J}t} \\ J\dot{\omega} &= J\dot{A}e^{-\frac{\alpha}{J}t} - \alpha Ae^{-\frac{\alpha}{J}t} \\ J\dot{\omega} &= J\dot{A}e^{-\frac{\alpha}{J}t} - \alpha\omega\end{aligned}\quad (4.25)$$

(4.25)式と(4.22)式を比較すると、

$$\alpha\omega_p = J\dot{A}e^{-\frac{\alpha}{J}t} \quad (4.26)$$

であることがわかる。 \dot{A} を時間積分すると、

$$\begin{aligned}\dot{A} &= \frac{\alpha}{J}\omega_p e^{\frac{\alpha}{J}t} \\ A &= \omega_p e^{\frac{\alpha}{J}t} + B\end{aligned}\quad (4.27)$$

B は t に依存しない定数である。(4.27)式を(4.24)式に用いると、

$$\begin{aligned}\omega &= (\omega_p e^{\frac{\alpha}{J}t} + B)e^{-\frac{\alpha}{J}t} \\ \omega &= \omega_p + Be^{-\frac{\alpha}{J}t}\end{aligned}\quad (4.28)$$

定数 B は ω の初期値 $\omega(0)$ によって決まる。

$$\begin{aligned}\omega(0) &= \omega_p + B \\ B &= \omega(0) - \omega_p\end{aligned}\quad (4.29)$$

これを、(4.28)式に用いると、

$$\omega(t) = \omega_p + (\omega(0) - \omega_p)e^{-\frac{\alpha}{J}t} \quad (4.30)$$

となる。ローターの慣性モーメント J は、およその値は推測できるが、サイズが10数センチで、重量も数グラムなので、2点吊り法で正確にその値を計測することは、空気抵抗や摩擦の影響が大きく、困難であると考えられる。また、 α の値は、ローター、モーターおよびアンプの性能、あるいはバッテリーの性能や起電力にも依存する可能性があり、先見的にその値を求めるることは、これも困難である。

そこで、

$$\tau \equiv \frac{J}{\alpha} \quad (4.31)$$

と τ を定義する。これを、(4.30) 式に用いると、 $\omega(t)$ は

$$\omega(t) = \omega_p + (\omega(0) - \omega_p)e^{-\frac{t}{\tau}} \quad (4.32)$$

というかたちで、時間依存すると考えられる。 τ がローター角速度の時間変化を特徴付ける緩和時間である。

角速度実測値からもとめられる緩和時間

図4.6に、ローター操作値を40%から50%に増やした場合のローター回転速度の実測値を示した。また、その時間変化に関して、(4.32)式を使って、最小二乗法フィッティングを行った結果も同時に示した。このように、ローター角速度が増加する場合、すなわち、 $\omega_p > \omega(0)$ の場合の緩和時間 τ_+ は約48.5[ms]であることがわかる。

いっぽう、ローター操作値を50%から45%に下げた場合のローター角速度の実測値およびそのフィッティング結果を図4.7に示した。ロー

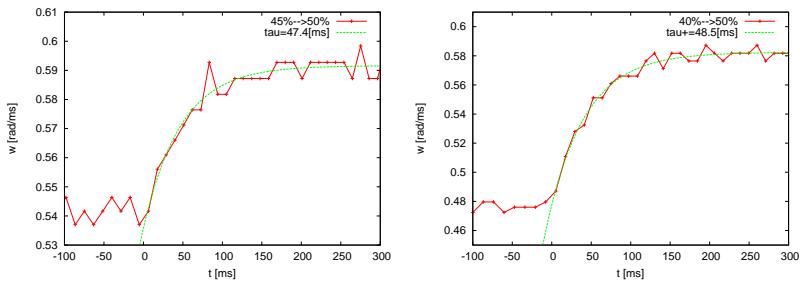


図 4.6: ローター操作値を 45%から 50%に増やした場合のローター回転速度の時間変化. $\tau_+ = 47.4[\text{ms}]$. ローター操作値を 40%から 50%に増やした場合のローター回転速度の時間変化. $\tau_+ = 48.5[\text{ms}]$.

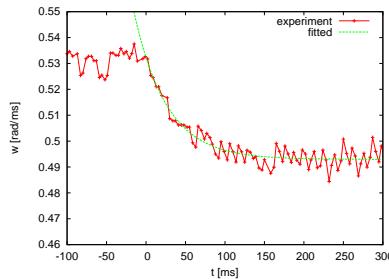


図 4.7: ローター操作値を 50%から 45%に下げた場合のローター回転速度の時間変化. $\omega_p = 0.493[\text{rad/ms}]$, $\omega(0) = 0.532[\text{rad/ms}]$, $\tau_- = 41.4[\text{ms}]$

タ一角速度が減少する場合, すなわち, $\omega_p < \omega(0)$ の場合の緩和時間 τ_- は約 41.4[ms] となり, τ_+ の値よりも大きい. すなわち, 角速度を減速

する際には、その緩和時間は加速する場合よりも、6[ms]ほど長くなる傾向がある。つまり、ローターの減速には、加速よりも時間がかかることがわかる。

4.5 時間遅れを考慮した拡張遷移行列

3章では、時間遅れがない場合に、線形近似と行列形式を用いて系の挙動を解析した。実際の飛行ロボットでは、データ転送時間やアンプの遅延時間など、時間遅れ δ が無視できない。つまり、現在の状態の変化率が、少し昔 $t - \delta$ における状態によって決まるというわけである。これを微分方程式で表すと、

$$\frac{d}{dt}\dot{\varphi}(t) = -c_G\dot{\varphi}(t - \delta) - c_A\varphi(t - \delta) \quad (4.33)$$

となる。この式からは、時間遅れがないときのように行列の指數関数を用いて解を容易に求めることができない。

制御理論によれば、フィードバック系のダイナミクスが微分方程式で記述できる場合、それをラプラス変換することによって伝達関数を求める。時間遅れがない場合、伝達関数は有理関数となり、それを逆ラプラス変換することによって、系の振る舞いを明示的に求めることができる。

しかし時間遅れが緩和時間などの時定数と同程度の長さ存在し、無視できない場合、伝達関数は有理関数とはならず、逆ラプラス変換で系の振る舞いを明示的に記述できない。振動応答を求め、ボード線図からその安定性などが議論されるが、時間遅れが無い系に比べて一般に制御が

困難であると言われている。その対処法としては、大別すると以下の3通りの方法がとられている。

ひとつ目は、伝達関数をパディ近似などで有理関数に近似するというものである。いったん有理関数のかたちに近似された伝達関数は逆ラプラス変換などを通じて詳細に検討可能となる。2つ目は、スミス法などのように、予測器を系に挿入することによって、見かけ上時間遅れをフィードバックループの外に出した形で、予測制御を行うものである。

また、3つ目として、現実の制御システムでよく用いられる方法がある。実際に制御系を動かしてみて、振動の周期などを観測し、その観測データに基づいてゲインを調節するという、限界感度法などが用いられる。しかし、この経験的な方法も時間遅れが存在する制御システムにおいてはその有効性が低いと言われている。だいいち、どのような挙動をするか不明な状態で、飛行ロボットを飛ばしてみて、その様子を観察することは非常に危険なので、いまの場合、この方法を用いることには無理があると言えるであろう。

そこで、ここでは、先に示した行列形式の方法を用いて時間遅れのある制御系を記述し、遷移行列の固有値問題としてその振る舞いを見通すことを試みる。まず、時間遅れがゼロではないが、非常に微小である場合を考える。いずれにせよ、時間遅れが無い場合のように単純な指数関数のかたちでは状態を記述できないので、系の時間発展を遷移行列によるベクトルの漸化式として表すことにする。遷移行列は時間遅れがない場合にすでに現れたが、時間遅れがある場合、遷移行列の次元を拡張するという自然なかたちでその影響を記述できる。遷移行列の次元が増えるわけであるから、その固有値と固有ベクトルを求める手続きは複雑化

するわけだが、いったん固有値が求まれば、その性質によって制御系の振る舞いが直接記述されるという利点があると考えられる。

行列形式の具体的な記述に戻ろう。時間遅れ δ を含む運動方程式(4.33)に加えて、以下の自明な式

$$\frac{d}{dt}\varphi(t) = \dot{\varphi}(t) \quad (4.34)$$

をまとめて行列形式で書くと

$$\frac{d}{dt} \begin{pmatrix} \dot{\varphi}(t) \\ \varphi(t) \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \dot{\varphi}(t) \\ \varphi(t) \end{pmatrix} + \begin{pmatrix} -c_G & -c_A \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \dot{\varphi}(t-\delta) \\ \varphi(t-\delta) \end{pmatrix} \quad (4.35)$$

ここで、 \hat{A}_0, \hat{A}_1 を

$$\hat{A}_0 \equiv \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \quad (4.36)$$

$$\hat{A}_1 \equiv \begin{pmatrix} -c_G & -c_A \\ 0 & 0 \end{pmatrix} \quad (4.37)$$

と定義すれば、(4.35)式は、

$$\frac{d\vec{s}(t)}{dt} = \hat{A}_0\vec{s}(t) + \hat{A}_1\vec{s}(t-\delta) \quad (4.38)$$

となる。

微小時間 $dt = \Delta t$ また、状態ベクトル $\vec{s}(t)$ の微小変化量 $d\vec{s}(t) = \Delta\vec{s}(t)$ と書くと、

$$\frac{\Delta\vec{s}(t)}{\Delta t} = \hat{A}_0\vec{s}(t) + \hat{A}_1\vec{s}(t-\delta) \quad (4.39)$$

$$\Delta\vec{s}(t) = \hat{A}_0\vec{s}(t)\Delta t + \hat{A}_1\vec{s}(t-\delta)\Delta t \quad (4.40)$$

である。両辺に $\vec{s}(t)$ を加えると、

$$\vec{s}(t) + \Delta \vec{s}(t) = \vec{s}(t) + \hat{A}_0 \vec{s}(t) \Delta t + \hat{A}_1 \vec{s}(t - \delta) \Delta t \quad (4.41)$$

となる。ここで、

$$\vec{s}_i = \vec{s}(t) \quad (4.42)$$

$$\vec{s}_{i+1} = \vec{s}(t) + \Delta \vec{s}(t) \quad (4.43)$$

というように、状態ベクトル \vec{s} が離散的に発展していくとみなす。また、いま時間遅れが微小で $\delta = \Delta t$ であるとき、

$$\vec{s}(t - \delta) = \vec{s}(t - \Delta t) = \vec{s}_{i-1} \quad (4.44)$$

とみなせる。したがって、時間発展式 (4.41) は、

$$\vec{s}_{i+1} = \vec{s}_i + \hat{A}_0 \vec{s}_i \Delta t + \hat{A}_1 \vec{s}_{i-1} \Delta t \quad (4.45)$$

$$\begin{pmatrix} \vec{s}_{i+1} \\ \vec{s}_i \end{pmatrix} = \begin{pmatrix} \hat{I} + \hat{A}_0 \Delta t & \hat{A}_1 \Delta t \\ \hat{I} & \hat{0} \end{pmatrix} \begin{pmatrix} \vec{s}_i \\ \vec{s}_{i-1} \end{pmatrix} \quad (4.46)$$

と表される。 \hat{A} 行列が \hat{A}_0 と \hat{A}_1 のふたつの行列にわかれたので、 \hat{A} 行列の固有値および固有ベクトルを直接用いた議論がここではできない。

そこで、次のように微小パラメータ ε を用いて拡張遷移行列 $\hat{T}(\varepsilon)$ 導

入する。

$$\begin{aligned}\hat{T}(\varepsilon) &\equiv \begin{pmatrix} \hat{I} + \hat{A}_0 \Delta t \varepsilon & \{(1 - \varepsilon)\hat{A}_0 + \hat{A}_1\} \Delta t \\ \hat{I} & \hat{0} \end{pmatrix} \\ &= \begin{pmatrix} \hat{I} & \hat{A} \Delta t \\ \hat{I} & \hat{0} \end{pmatrix} + \begin{pmatrix} \hat{A}_0 \Delta t \varepsilon & -\hat{A}_0 \Delta t \varepsilon \\ \hat{0} & \hat{0} \end{pmatrix} \quad (4.47)\end{aligned}$$

ここで、 \hat{I} は単位行列、 $\hat{0}$ は要素がすべてゼロのゼロ行列である。ベクトルのベクトルや、行列の行列という見慣れない形式が現れたが、全体としてそれぞれベクトルと行列になるので、この簡略化された表現を用いることにする。

さらに、 \hat{H} を

$$\hat{H} \equiv \begin{pmatrix} \hat{A}_0 \Delta t & -\hat{A}_0 \Delta t \\ \hat{0} & \hat{0} \end{pmatrix} \quad (4.48)$$

と定義する。

4.6 フィボナッチ型遷移行列

$\varepsilon = 0$ の場合、漸化式は

$$\begin{pmatrix} \vec{s}_{i+1} \\ \vec{s}_i \end{pmatrix} = \begin{pmatrix} \hat{I} & \hat{A} \Delta t \\ \hat{I} & \hat{0} \end{pmatrix} \begin{pmatrix} \vec{s}_i \\ \vec{s}_{i-1} \end{pmatrix} \quad (4.49)$$

と書き表すことができる。

(4.49) 式内の行列は 2×2 行列のように記述されているが、その行列要素が 2×2 行列なので、全体として 4×4 行列を表す。ベクトルについても同様である。

この漸化式 (4.49) の行列要素がスカラーで、すべて 1 の場合は、フィボナッチ数列という有名な数列を表す漸化式となり、その性質は詳しく知られている。その類似性から、(4.49) 式に現れる行列をフィボナッチ型遷移行列 \hat{T}_F と呼ぶことにしよう。

$$\hat{T}_F \equiv \begin{pmatrix} \hat{I} & \hat{A}\Delta t \\ \hat{I} & \hat{0} \end{pmatrix} \quad (4.50)$$

フィボナッチ型遷移行列 \hat{T}_F と \hat{H} を用いることで、

$$\hat{T}(\varepsilon) = \hat{T}_F + \varepsilon \hat{H} \quad (4.51)$$

$\hat{T}(1)$ が (4.46) 式の漸化式における遷移行列を表す。また、 $\hat{T}(0)$ がフィボナッチ型の遷移行列を表す。

固有値

フィボナッチ型遷移行列の固有値 η と右固有ベクトルを求める。右固有ベクトル v_F^r を

$$v_F^r = \begin{pmatrix} \vec{\alpha} \\ \vec{\beta} \end{pmatrix} \quad (4.52)$$

と書くことになると,

$$\begin{pmatrix} \hat{I} & \hat{A}\Delta t \\ \hat{I} & \hat{0} \end{pmatrix} \begin{pmatrix} \vec{\alpha} \\ \vec{\beta} \end{pmatrix} = \eta \begin{pmatrix} \vec{\alpha} \\ \vec{\beta} \end{pmatrix} \quad (4.53)$$

と表すことができる。掛け算を実行してみると、

$$\vec{\alpha} + \Delta t \hat{A} \vec{\beta} = \eta \vec{\alpha} \quad (4.54)$$

$$\vec{\alpha} = \eta \vec{\beta} \quad (4.55)$$

となるが、(4.55) 式を (4.54) 式に代入すると、

$$\eta(1 - \eta) \vec{\beta} + \Delta t \hat{A} \vec{\beta} = \vec{0} \quad (4.56)$$

ここで、行列 \hat{A} の右固有ベクトル \vec{v}_1, \vec{v}_2 を使って、

$$\vec{\beta} = \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 \quad (4.57)$$

と線形結合によって $\vec{\beta}$ を表すと、

$$\{\eta(1 - \eta) + \lambda_1 \Delta t\} \beta_1 \vec{v}_1 + \{\eta(1 - \eta) + \lambda_2 \Delta t\} \beta_2 \vec{v}_2 = \vec{0} \quad (4.58)$$

である。 \vec{v}_1, \vec{v}_2 は 1 次独立であるから、任意の β_1, β_2 に対して、この等式が成り立つためには、

$$\eta(1 - \eta) + \lambda_1 \Delta t = 0 \quad (4.59)$$

$$\eta(1 - \eta) + \lambda_2 \Delta t = 0 \quad (4.60)$$

でなければならない。これらは、2次方程式であるから、解の公式をもちいて簡単に解が得られ、

$$\eta_k = \frac{1}{2} \left(1 \pm \sqrt{1 + 4\lambda_k \Delta t} \right) \quad (k = 1, 2) \quad (4.61)$$

という4つの固有値が得られる。この固有値の性質を調べれば、飛行ロボットの状態変化、すなわち振動、収束および発散が予測できる。

4.7 時間遅れの固有値への繰り込み

感覚運動写像による系の振る舞いを \hat{A} 行列による微分方程式で記述すると、その固有値 λ の値から、系の振る舞いがわかる。この微分方程式を、対応する漸化式の形に書き直すと、系の振る舞いは遷移行列を用いて表される。もちろん、遷移行列の固有値と、 \hat{A} 行列の間には関係性があるので、遷移行列の固有値から \hat{A} 行列の固有値は求められる。

まず、微小な時間遅れ $\delta = \Delta t$ があると、それが無い場合から遷移行列がどのように変化するかを調べる。その固有値から \hat{A} 行列の固有値を求めれば、それは微小な時間遅れが繰り込まれた λ の値を意味する。この繰り込み変換を $\delta = n\Delta t$ となるまで繰り返せば、その時えられた λ の値は、時間遅れ δ に対応する \hat{A} 行列の固有値を意味する。そのとき、 $\delta = n\Delta t$ の値を保ったまま、 $\Delta t \rightarrow 0, n \rightarrow \infty$ の極限をとる必要がある。

微小時間遅れによる \hat{A} 行列固有値の変化

$\delta = \Delta t$ のときのフィボナッチ型遷移行列に対する固有値 η_k は,

$$\eta_k = \frac{1}{2} \left(1 \pm \sqrt{1 + 4\lambda_k \Delta t} \right) \quad (4.62)$$

と求められている。3.12節で述べた様に、遷移行列の固有値 E_k と \hat{A} 行列の固有値の間には,

$$\lambda_k = \frac{E_k - 1}{\Delta t}$$

という関係がある。

いま、 $E_k = \eta_k$ であることに注意すると,

$$\frac{\eta_k - 1}{\Delta t} \quad (4.63)$$

の値が時間遅れ δ によってどのように影響を受けるかを調べることによって、時間遅れの効果を考えよう。この式によると、解 η_k の中で

$$\eta = \frac{1}{2} \left(1 + \sqrt{1 + 4\lambda \Delta t} \right) \quad (4.64)$$

が重要な意味を持つことが予想される。ここでは、 $k = 1, 2$ のどちらの場合でも共通する議論なので、それを省略して示す。

(4.64)式の形の η をそのまま用いると、考えにくいので、元の特性方程式

$$\eta(1 - \eta) + \lambda \Delta t = 0 \quad (4.65)$$

に戻って、その解が Δt によって 1 からどのように変位するか調べよう。

η の Δt による 1 階微分を $\dot{\eta}$ で表すこととする。この特性方程式 (4.65) の両辺を Δt で微分すると、

$$\begin{aligned} \dot{\eta}(1 - \eta) & - \eta\dot{\eta} + \lambda = 0 \\ \dot{\eta} &= \frac{\lambda}{2\eta - 1} \end{aligned} \quad (4.66)$$

となる。いま、1 からの変位を考えているので、 $\eta \neq 1/2$ と仮定しても良いであろう。

ここで、(4.64) 式をつかうと、

$$2\eta - 1 = \sqrt{1 + 4\lambda\Delta t} \quad (4.67)$$

なので、

$$\dot{\eta} = \frac{\lambda}{\sqrt{1 + 4\lambda\Delta t}} \quad (4.68)$$

のことから、 η は

$$\begin{aligned} \eta &\simeq 1 + \dot{\eta}\Delta t \\ &= 1 + \frac{\lambda}{\sqrt{1 + 4\lambda\Delta t}}\Delta t \end{aligned} \quad (4.69)$$

と、求められる。系の振る舞いを特徴づける量を求めるために、これを (4.63) 式に用いると、

$$\frac{\eta - 1}{\Delta t} \simeq \frac{\lambda}{\sqrt{1 + 4\lambda\Delta t}} \quad (4.70)$$

となる。

この(4.70)式の左辺の意味は、わずかな時間遅れ Δt によって修正された固有値 $\lambda + \Delta\lambda$ であると解釈できる。すなわち、

$$\lambda + \Delta\lambda = \frac{\lambda}{\sqrt{1 + 4\lambda\Delta t}} \quad (4.71)$$

という λ の時間発展方程式とみなせる。

時間遅れのある系における λ

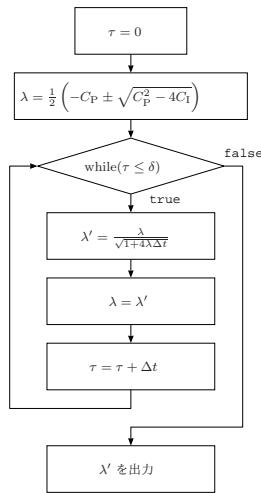
(4.71)式を $\delta = n\Delta t$ となるまで n 回くり返し用いて得られた λ は、時間遅れ δ に対応する λ であると考えられる。

つまり図4.8に示した計算を行う。最終的に得られる λ' が時間遅れ δ に対応する \hat{A} 行列の固有値に対応すると考えられる。

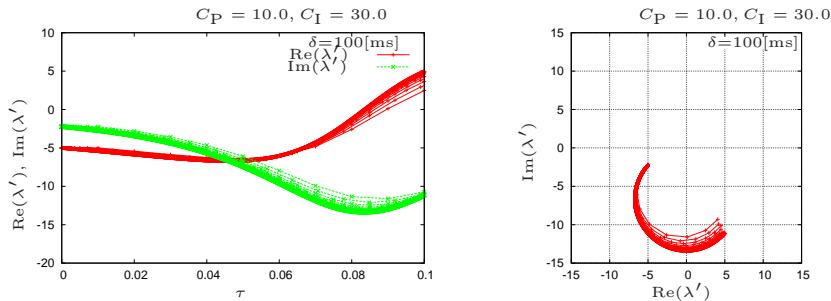
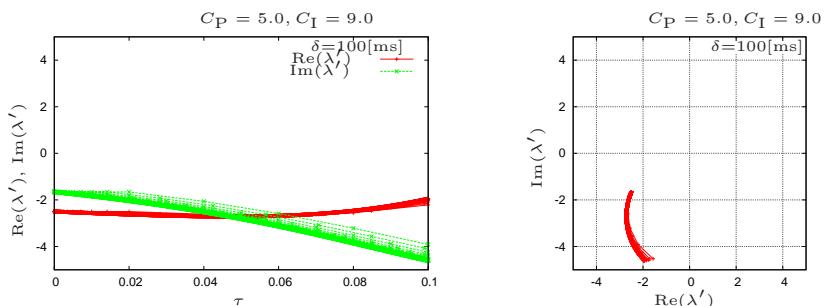
図4.9に様々な Δt の値に対する、この手続きによって求められた λ' について示した。 Δt の値を徐々に小さくしていくと、変化の様子は一定の振る舞いに落ち着くので、 λ' の Δt 依存性は、収束していると考えられる。

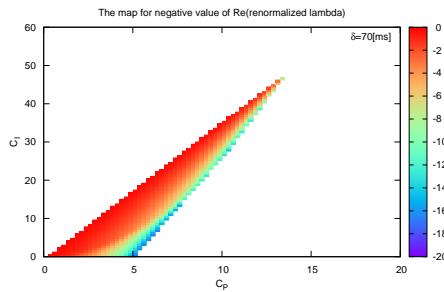
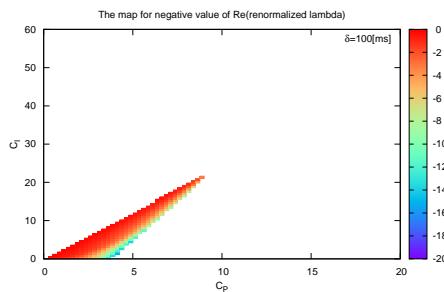
$\tau = 0$ における値が時間遅れが無い場合の λ である。実部、虚部ともに負の値なので、減衰振動をする。いっぽう、 $\tau = \delta$ まで修正された $\text{Re}(\lambda')$ の値は正である。すなわち、時間遅れの効果によって、減衰振動が発散振動に変化したことを意味する。

さまざまな C_P, C_I の値に対して同様の計算を行い、 $\text{Re}(\lambda') < 0$ となる領域を図4.11に示した。時間遅れがない場合には、ゲイン領域のあらゆる値に対して機体の振る舞いは収束したが、時間遅れが存在すること

図 4.8: \hat{A} 行列の固有値 λ に対する繰り込みアルゴリズム

によって、運動が収束するのは非常に限られたゲイン領域であることが分かる。

図 4.9: λ の時間遅れによる変化。 $\text{Re}(\lambda') > 0$ となる場合の例図 4.10: λ の時間遅れによる変化。 $\text{Re}(\lambda') < 0$ となる場合の例

図 4.11: 修正された λ' の実部が負になる領域地図 $\delta = 70[\text{ms}]$ 図 4.12: 修正された λ' の実部が負になる領域地図 $\delta = 100[\text{ms}]$

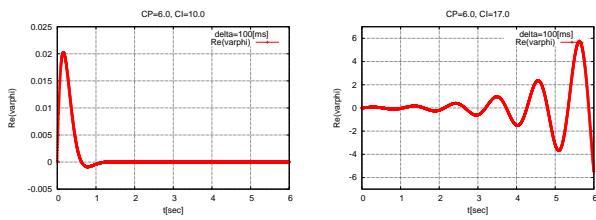


図 4.13: 時間遅れが繰り込まれた λ' を用いた状態 $\varphi(t)$

第II部

具体的にロボット知能を構成するための道具

ここでは、実際にロボットをつくって、環境のなかで行動させるため、またそれを観察するために必要な道具となる事柄をまとめた。

LEGO Mindstorm EV3 や BeagleBone Black などのハードウェア的な内容からそれらを利用するためには必要なオペレーティングシステム Debian OS についての内容まで様々な内容を含む。

それぞれについて詳しく正確な記述は、ここでは求めず、あくまでもロボットをつくって動かすために必要な部分をまとめたものであると思って欲しい。

それぞれの章や節は、関連性があるが、比較的独立しているので、マニュアルとして利用することも可能であると思う。

第5章 Debian をつかう

Debian とは、ファイル管理やネットワーク環境などコンピューターのオペレーティングシステム機能を提供するフリーソフトウェアパッケージ群のことです。ここでは、Debian GNU/Linux¹ のことを単に Debian と呼ぶことにします。

日常的に利用するデスクトップ環境はもちろん、サーバー機能など、29,000 以上のソフトウェアを提供しており、日本語利用環境² も充実しています。

ロボットインテリジェンスの研究に欠かせないボードコンピューターでは、ARM アーキテクチャの CPU が使われることが多いです。Debian は ARM 用のパッケージも提供しているので、i386 や AMD64 アーキテクチャをつかったノート PC などと同じ要領でボードコンピューターにインストールやアップデートなどの管理が可能であるため、さまざまな OS が混在する状況よりも、円滑な作業が可能です。

Debian はフリーな OS であることが特長です。ここでフリーとは、自

¹<http://www.debian.org/>

²<http://www.debian.or.jp/>

由という意味です³。しかし、Debian は無料で入手できるので、事実上、無料という意味も含まれてます。

下記に Debian のインストール DVD のイメージがあるので、ダウンロードしてインストール DVD を作ります。

<http://cdimage.debian.org/debian-cd/current/amd64/iso-dvd/>

インストールの方法には、おおきく 2 通りあります。インストーラーだけが含まれた CD イメージだけをダウンロードして、それをを利用してインストーラーを起動し、パッケージはネットワークからダウンロードしながら Debian をインストールする方法がひとつめです。ネットワークインストール用の CD イメージはサイズが小さくダウンロードが容易です。いっぽう、パッケージも含んだ DVD イメージをダウンロードし、インストール DVD を作り、それをつかって Debian をインストールする方法もあります。DVD-1 にインストーラー機能が含まれています。

ネットワークインストールの方法においては、大量のソフトウェアパッケージをインターネットを通じてダウンロードしなければならないので、インストール時間はネットワークの通信速度に依存します。どのようなパッケージを選択するかによって構築される Debian のサイズは異なりますが、デスクトップ環境やフォントまで含めると 4G バイト程度、BeagleBone Black などのボードコンピューターに CUI 環境を構築すると 1G バイトほどのハードディスク領域を消費するだけで Debian 環境を構築できます。

³http://www.debian.org/social_contract#guidelines

つまり Debian は、microSD や USB メモリのなかに簡単に構築できてしまします。デフォルトで GUI を含んだ、Debian 以外の巨大な OS を使わず、Debian をロボットインテリジェンス研究に利用する理由のひとつに、この OS のサイズが大きすぎず、自由にパッケージ管理することで、その大きさをコントロールすることができるという点もあります。

なぜこれほどのOSが無料なのか

Debianをはじめ「フリー」とよばれるソフトウェアをつくっている開発者は、多大な労力かけてそれを作っている。一般には労力に対して金銭で報酬が支払われるが、Debian開発者はそれを求めていないとすることである。

ソフトウェア開発者は、自分の作ったソフトウェアに誇りをもっており、それが世界中で活用されるということに価値を見出しているのである。ソフトウェアの実態は文字記号の羅列であり、簡単に複製することができる。それが記述されているハードウェア(HDやメモリーなど)が壊れても、壊れていないハードウェアにコピーされたソフトウェアは100%もとのソフトウェアと同じ機能を備えており、損なわれることはない。

Debianを含めたGNUソフトウェアは自由な複製や修正を保証したライセンスの下に管理されている。複製が禁止されたソフトウェアは、それがインストールされているハードウェアが壊れ、その配布元がなくなれば地球上から消滅する。しかし、複製が許されたソフトウェアはだれかが複製するかぎり消滅しない。自らが生み出したものが、いわば「永遠の命」をもった存在になるのである。これ以上の報酬があるだろうか。

ロボット知能の構成論的研究のなかでDebianを活用することは、こういった精神のソフトウェア開発者に対する報酬の一部であると著者は考えている。

5.1 パッケージの管理

Debian の大きな特長の一つに、ソフトウェア管理の容易さがあります。一通り Debian のインストールが終了したあとに、新たに別のソフトをインストールしたり、すでにインストールしてあるソフトを削除したりすることが、コマンドから簡単に実行できます。Debian では、これらの操作を `dpkg`, `apt-get` または `aptitude` コマンドをつかって行います。

5.1.1 既にインストールされているパッケージの確認

たとえば、`g++`に関するパッケージで、すでにインストールされているものを表示するには、

```
# dpkg -l | grep g++
```

と入力します。ここで、プロンプト#は、root ユーザーになって作業をすることを意味します。実行結果例は、たとえば

```
ii  g++          4:4.7.2-1      amd64      GNU C++ compiler
ii  g++-4.7     4.7.2-5       amd64      GNU C++ compiler
```

となります。ふたつの `g++`に関するパッケージがインストールされていることが分かります。

5.1.2 DVD(CD) ドライブ内のパッケージ認識

DVD(CD) ドライブに Debian のサイトからダウンロードして作った DVD を挿入します。ドライブを認識させるためには、

```
# apt-cdrom ident
```

その後、以下のコマンドでそれを認識させる必要があります。

```
# apt-cdrom add
```

認識させたい DVD が複数ある場合には、DVDを入れ替えてこのコマンドを繰り返す必要があります。認識された DVD の情報は、/etc/apt/sources.list の中に記述されています。

5.1.3 パッケージ情報のアップデート

パッケージのインストールを実行するまえに、情報をアップデートします。

```
# apt-get update
```

5.1.4 パッケージの検索

たとえば、gvim に関するパッケージを検索する場合、

```
# apt-cache search gvim
```

と実行します。実行結果例は

```
vim-athena - Vi IMproved - enhanced vi editor - with Athena GUI  
vim-gui-common - Vi IMproved - Common GUI files  
netrik - vi ライクなキーバインディングによるテキストモード WWW ブラウザ  
vim-gnome - Vi IMproved - 強化版 vim エディタ - GNOME2 GUI 付き  
vim-gtk - Vi IMproved - 強化版 vim エディタ - GTK2 GUI 付き
```

です。出力結果が長く、折り返されて複数行に渡っていて見づらい場合もありますが、一番ひだりに表示されているのが、パッケージ名です。ここでは、vim-athena, vim-gui-common, netrik, vim-gnome, vim-gtk の 5 つの gvim に関するパッケージがあることが分かります。

aptitude を用いる場合、

```
# aptitude search gvim
```

と search オプションを用います。

5.1.5 パッケージのインストール

例として、vim-gnome をインストールしてみます。

```
# apt-get install vim-gnome
```

あるいは、

```
# aptitude install vim-gnome
```

とします。aptitude コマンドは、apt-get コマンドと似ていますが、パッケージ間の依存関係の解決能力が改善されています。

5.1.6 ネットワークインストール

/etc/apt/sources.list の記述。

```
#deb cdrom:[Debian GNU/Linux testing _Wheezy_ - Official Snapshot \
amd64 DVD Binary-1 20121119-04:55]/ wheezy contrib main
...
deb http://ftp.nara.wide.ad.jp/debian/ wheezy main contrib non-free
```

PROXY の環境変数への設定。

```
# export http_proxy=proxy.murooran-it.ac.jp:8080
```

5.1.7 amd64 アーキテクチャで i386 アーキテクチャを使えるようにする

amd64 アーキテクチャで Debian をインストールすると、i386 でしか動かないソフトウェアが起動しない。具体的には、command not foundなどのメッセージが表示されるだけになる。

ia32-libs をインストールすれば、実行可能になる。

```
# dpkg --add-architecture i386  
# apt-get update  
# apt-get install ia32-libs
```

5.2 エディター vi

ロボットインテリジェンスの研究を行う場合、ボードコンピュータ上のプログラムファイルや、設定ファイルを編集することが必要となる。ボードコンピュータは(Graphical User Interface(GUI)を持たず、メインメモリもPCなどとくらべて大きくなないので、viエディタを利用する必要がある。

viはDebian GNU/Linuxの元となったUNIXオペレーティングシステムが開発された当初から標準的に装備されており、使用メモリーも少なく動作も高速である。なおかつ迅速かつ高度な編集機能を十分に備え持っているので、ロボットインテリジェンスの研究にに限らず、組み込みシステムなどの開発においてもviエディタが大きな役割を果たす。

5.2.1 viの動作設定

viの動作を設定するファイルは、`~/.exrc`である。ただし、`~`は、ホームディレクトリを意味する。たとえば、ユーザー名 porco の場合`/home/porco`のことである。

`.exrc`ファイル内に

```
set nu
```

と書いておくと、起動したときに、自動的に行番号が表示される。

gvim が実行された時の、初期状態は、`~/.gvimrc` ファイルの内容によって指定できる。例えば、

```
set nu
set fileencodings=euc-jp,iso-2022-jp,sjis,utf8
set columns=150
set lines=40
colorscheme torte
syntax on
```

このように指定すると、行番号などが有効となる。

5.2.2 vi の使い方

vi は 2 つのモードを持つ。一つは、コマンドモードで、このモード内でカーソル移動や、文字消去、コピー&ペーストなどを行う。もう一つが、インプットモードで、キーボードから入力された文字列がそのままエディタに入力される。

モードの切り替え

モードの切り替えは、文字”i”と ESC キーで行う。コマンドモードの状態で、”i”を押すと、カーソル位置に文字を入力できる入力モードに

モードが切り替わる。いっぽう、入力モードの状態で ESC キーを押すとコマンドモードに戻る。(図 5.1 参照)

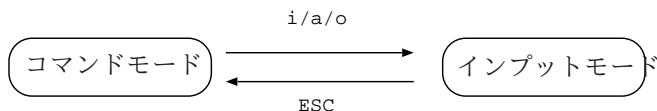


図 5.1: vi(gvim) におけるモードの切り替え

また、カーソル位置から一つだけ後ろの位置に文字入力したい場合には、”i”のかわりに”a”を押す。行末に文字を追加したい場合などに用いる。

入力行を増やす場合には、行末に移動してリターンキーを押してもよいが、コマンドモードで”o”を入力すると、次の行が追加され入力モードとなる。

編集コマンド

コマンドモードでよく使う編集コマンドキーを表 5.1 にまとめた。

一つだけ文字列置換における注意点を挙げておく。置換コマンドの行末にある、/gc の g は 1 行の中で複数個置換することを意味する。c は置換を毎回確認することを意味する。

これらのコマンドを覚えることで、キーボードのニュートラルポジションから大きく手を移動させずにファイルの編集が可能となる。マウ

表 5.1: vi でよく使う編集コマンド

機能	詳細	コマンド
終了	終了 破棄終了	:q :q!
保存	保存 保存終了	:w :wq
カット	一文字カット 一行カット <i>n</i> 行カット	x dd dnd
移動	下に移動 上に移動 右に移動 左に移動 行末に移動 行頭に移動 <i>n</i> 行目に移動	j (↓) k (↑) l (→) h (←) \$ ^ <i>n</i> G
コピー・ペースト	1 行ヤンク (コピー) 現在行から <i>n</i> 行目までヤンク 現在行から <i>x</i> 行分ヤンク ペースト	yy ynG yxy p
検索・置換	文字列検索 全文置換 範囲指定置換 (xxx-yyy 行目)	/文字列 :%s/文字列/置換文字列/gc :xxx,yyy/s/文字列/置換文字列/gc
ファイル間移動	ファイルを開く 最初のファイルに戻る	:e ファイル名 .rew
行連結		J
シェルコマンド		:!コマンド

スを使って文字の選択やペーストをおこなう必要はない。またプルダウンメニューから機能を選択したりする必要もない。そういうた操作は、

直感的なので、少數回ならば生産性の低下は軽微だが、ロボットインテリジェンスの研究のように、ボードコンピュータの GUI がない Debian 上でプログラムなどを編集することが必要な場合、これらのコマンドを習得することが必須である。また、これらのコマンドを一旦使いこなせるようになれば、通常のマウスなどをつかったファイル編集と比べて、短時間で大量の文字列編集が可能となるため、作業効率は飛躍的に高まる。

ここに紹介した機能以外に、vi はコマンドのマップなど、驚くほど多機能であり、なおかつ高速に動作する。ロボットインテリジェンスの研究においては、その活用が必須と言えるであろう。

5.2.3 GUI 上の vi

vi は元々、Character User Interface(CUI)⁴ 上で利用することを前提としてつくられたエディタであるが、GUI で日本語（全角文字）を編集する場合には Gnome 版の gvim が安定している。

```
$ gvim body.tex
```

⁴コマンドラインインターフェイス (CLI) とも呼ばれる。

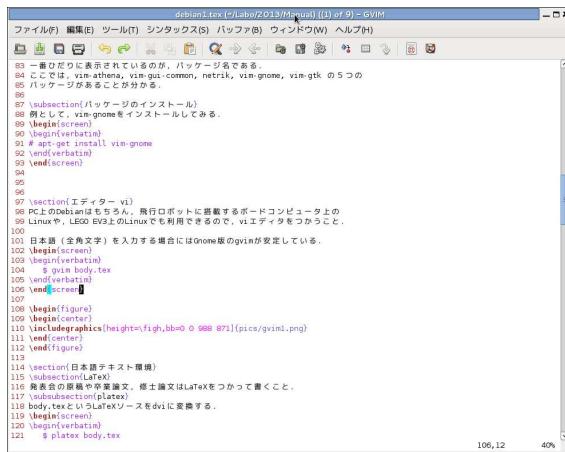


図 5.2: gvim の実行例.

5.3 ネットワーク

TCP/IP プロトコルで Debian PC とボード PC などをネットワーク接続する。ネットワーク接続では、大まかに言って、物理接続、インターフェイスのドライバ、IP アドレス、名前解決 (DNS, /etc/hosts) および経路解決 (route) がそれぞれ適切に設定されている必要がある。

ここでは、ネットワークインターフェイスが eth0 として認識されている場合について、IP アドレスの設定以降を説明する。

5.3.1 IP アドレス

設定ファイル

/etc/network/interfaces の例.

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
allow-hotplug eth0
iface eth0 inet static
    address 172.16.0.xxx
    netmask 255.255.0.0
    network 172.16.0.0
    broadcast 172.16.255.255
    gateway 172.16.0.1
    dns-nameservers 157.19.201.59
```

インターフェイスの確認

ifconfig コマンドを利用する.

```
# ifconfig
```

始動と停止

ネットワークインターフェイス eth0 の停止と始動. 以下のコマンドを実行すると, /etc/network/interfaces ファイルの内容が読み込まれて, インターフェイス eth0 の停止と始動が実行され

```
# ifconfig eth0 down  
# ifconfig eth0 up
```

5.3.2 名前解決

IP アドレスをすべて記憶しておけば, 基本的にネットワークアクセスは可能である. しかし, IP アドレスは数字の羅列なので, 記憶しにくい. そこで, IP アドレスとホスト名を対応付けておけば, ホスト名で IP アドレスを指定できるので, 利用しやすい.

hosts の利用

IP アドレスと, ホスト名の対応表は, /etc/hosts ファイルの中に記述する. /etc/hosts ファイルの例

```
127.0.0.1 localhost
172.16.0.1 maple
172.16.0.2 wiki
172.16.0.18 arm18
172.16.0.42 arm42
172.16.10.18 arm18w
172.16.10.48 tama
172.16.0.50 maple50
172.16.0.51 maple51
172.16.10.51 castle
```

通信の確認

通信の確認は、ping コマンドによって行う。自分の Debian PC から例えば maple にパケットを送って、戻ってくるまでの時間が msec 単位で表示されれば、通信に成功している。何度も繰り返し表示されるが、CTRL+c を入力すると停止する。

```
# ping maple
```

結果例.

```
64 bytes from maple (172.16.0.1): icmp_req=1 ttl=63 time=2.19 ms
64 bytes from maple (172.16.0.1): icmp_req=2 ttl=63 time=1.53 ms
64 bytes from maple (172.16.0.1): icmp_req=3 ttl=63 time=1.59 ms
64 bytes from maple (172.16.0.1): icmp_req=4 ttl=63 time=1.73 ms
64 bytes from maple (172.16.0.1): icmp_req=5 ttl=63 time=1.54 ms
^C
```

DNS, PROXY

ローカルなホストは/etc/hostsの中に列記しておけばよいが、世界中のホストのIPアドレスをそこの列記するわけにはいかない。URLなど自分が知らないホストのIPアドレスはドメインネームサーバに問い合わせせる。

ドメインネームサーバ(DNS)のIPアドレスを/etc/resolv.confの中に記述する。たとえばDNSが aaa.bbb.ccc.ddd の場合、/etc/resolv.confのなかに

```
nameserver aaa.bbb.ccc.ddd
```

と記述する。

また、ブラウザなどで指定するプロクシーサーバー(PROXY)は

```
http://proxy.xxxx.ac.jp:8080/
```

のように指定する。最後の8080はポート番号を表す。

5.3.3 経路解決

デフォルトルーター(hoge)を追加する。

```
# route add default gw hoge eth0
```

詳しくは、6.4節を参照してください。

5.3.4 ネットワークファイルシステム NFS

ネットワークファイルシステム (NFS) を用いると、有線 LAN や無線 LAN で接続されたリモート Debian マシン上の外部記憶装置（HD や SD カードなど）や、個々のディレクトリがあたかもローカルな Debian マシン上に存在するかの様に利用することが可能である。

NFS を利用するためには、サーバー側でディレクトリを export し、そのディレクトリをクライアント側で mount する必要がある。

exports

サーバ Debian マシン hoge0 で、/etc(exports ファイルに NFS クライアントを登録する。

```
/home/honda/work    hoge1(rw,sync)  hoge2(rw,sync)
```

NFS サーバに/etc(exports の内容を反映させる。

```
hoge0# /etc/init.d/nfs-kernel-server restart
```

mount

クライアント側で、

```
hogeh1# mount -t nfs hogeh0:/home/honda/work /home/honda/work
```

を実行することで、NFS マウントされる。もちろんディレクトリ名はこの例と同じでなくてもよい。

5.4 kernel の構築とインストール

Linux ボードコンピュータなどに新たなハードウェアを接続したり、こまかいシステムの動作をチューニングするために、最新の kernel をダウンロードしてコンパイルする必要が生じる場合がある。自分の使う Linux にとって必要な kernel 構成を適切に行うためには、相当な慣れと時間を必要とするが、成功した際の果実は、他に得がたいものであるからチャレンジしてみる価値はある。

kernel のソースコードは、<https://www.kernel.org/pub/linux/kernel/> からダウンロードできる。最新のバージョンを用いることが推奨されるが、デバイスのファームウェアとの関係などから、低いバージョンのものを使う必要がある場合もある。上記の URL には過去の kernel バージョンもあるので、状況に適したものを選ぶ。

作業ディレクトリでアーカイブを解凍する。ソースコードは膨大な量で、そこから生成されるバイナリも相当な容量となる。HDD の空きスペースに余裕のあるディレクトリを利用しないと容量を超えてしまう恐れがあるので注意が必要である。

5.4.1 構築

kernel は以下の手順でつくる。make menuconfig と make-kpkg を実行するためには必要なパッケージがインストールされていなければならぬが、以下を実行してみてエラーメッセージがでなければ既にそれらがインストールされているということである。必要なパッケージは後述する。

```
# make menuconfig  
...  
# make-kpkg clean  
# make-kpkg --initrd --revision xxxxx kernel_image
```

-initrd オプションは初期 ram disk を使ったブートのために必要である。

最初の make menuconfig で、カーネルのコンフィグレーションを決める。Wireless Network Device など数多くのドライバなどを kernel 自身に組み込むのか、module として外部において、必要な場合にロード (insmod) するのか、あるいは使わないのかを選択できる。

make menuconfig を実行するためには libncurses5-dev がインストールされている必要がある。

```
# aptitude install libncurses5-dev
```

また、make-kpkg を実行するためには kernel-package をインストールする。

```
# aptitude install kernel-package
```

先にも述べたが、kernel の構築にはかなり時間を要する。どれぐらいの時間が必要かはケースバイケースだが、たとえば kernel 3.18 を Intel Core i7 で make-kpkg すると約 1 時間ほどで終了する。

5.4.2 インストール

コンパイルで構築されたカーネルは Debian パッケージとして linux-xxx.deb というファイル名で保存されている。xxx の部分には make-kpkg 実行時に指定した revision などが記述されている。

保存先は、make-kpkg を実行した作業ディレクトリのひとつ上のディレクトリである。dpkg -i コマンドを実行して新しい kernel をインストールする。

```
# cd ..
# dpkg -i linux-xxx.deb
```

複数の OS を管理するブートローダー (GRUB など) が使われている場合には、そのコンフィグレーションも自動的に生成される。システムの再起動時に、ここでインストールした kernel ヴァージョンを選択すれば、自分でチューニングした Linux が起動する。

第6章 無線LAN

Debian OS の間の無線ネットワーク通信は、他の UNIX/Linux OS と同様に TCP/IP に基づいたものである。無線 LAN を利用するには、大きく分けてハードウェアの認識、アドレスやキー認証の設定、有線 LAN からのルーティングの変更が必要である。

6.1 ツールのインストール

apt-get あるいは、aptitude が使える状態であれば、それらを使って wireless-tools パッケージをインストールする。

```
# aptitude install wireless-tools
```

なお、LEGO EV3 の Native Linux では aptitude などが利用できないので、iwconfig と iwlist コマンドのソースコードをダウンロードし、コンパイルしてそれらを作成する必要がある。

ev3dev は Debian ベースの LEGO OS なので、通常の Debian と同じようにインストールや設定が可能である。

6.2 設定コマンド

無線 LAN 接続を実現するためには、カーネル・モジュールによるインターフェイスの認識、モード、ESSID、チャンネルおよび暗号化パスワードの設定、さらに IP アドレスの設定がそれぞれ必要である。

インターフェイスの認識は、各ハードウェアに依存する部分もあるので、それぞれのボードコンピュータなどの節を参照してほしい。

ここでは、モード、ESSID、チャンネルなどについて主に説明する。

モード

無線 LAN には ad-hoc, managed および master の 3 つのモードがある。ad-hoc モードは、2 台の無線 LAN の間で 1 対 1 の通信をするばあいに利用する。

master モードと managed モードを用いると、2 台の間の通信はもちろん可能であるが、3 台以上の環境でも無線 LAN 通信が可能である。ルーターあるいは、アクセスポイントになる無線 LAN を master モードにするが、master モードすなわちアクセスポイントに対応した無線 LAN ハードウェアを用いる必要がある。Armadillo-300 および Armadillo-420+AWL13 は master モードになることができる。

master モードのアクセスポイントにアクセスする子機は managed モードに設定する。一般に 3 台以上で master-managed モードで無線 LAN 環境を利用するよりも、ad-hoc モードで 1 対 1 通信するほうが通信速度が

速くなる傾向がある。ロボットと Debian PC との通信など、速度が要求される通信の場合にはどのモードを選択するか考慮が必要である。

以下にインターフェイス wlan0 を ad-hoc モードにする例を示す。

```
# iwconfig wlan0 mode ad-hoc
```

すでに IP アドレスが割り当てられているインターフェイスについてはモードの切り替えができないので、いったんそれを down する必要がある。

```
# ifconfig wlan0 down  
# iwconfig wlan0 mode ad-hoc
```

IP アドレス

1. wlan0 の手動起動例

```
# ifconfig wlan0 172.6.10.xxx
```

2. /etc/network/interfaces を使って無線 LAN インターフェイスを起動

```
# ifconfig wlan0 up
```

なお、wlan0 は ath0 など、デバイスに割り当てられるインターフェイス名に依存する。また、wlan0 を停止するには、

```
# ifconfig wlan0 down
```

3. アクセスポイントをスキャン

```
# iwlist wlan0 scan | more
```

| more は、長い行数にわたる表示を、1 ページづつに分けて表示することを指定する。スペースキーをおすと、次のページが表示される。たとえば、

```
Cell 02 - Address: 00:80:92:3A:9F:E1
  Channel:2
  Frequency:2.417 GHz (Channel 2)
  Quality=58/70  Signal level=-52 dBm
  Encryption key:off
  ESSID:"arm18"
  Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 6 Mb/s; 9 Mb/s
            11 Mb/s; 12 Mb/s; 18 Mb/s
  Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
  Mode:Master
  ....
```

のような表示が現れれば、無線 LAN インターフェイスは、正常に電波をキャッチしている状態である。6 行目の表示から、ESSID が arm18 であることが分かる。アクセスしたいルーターに合わせて、以下の essid を指定する。

チャンネル, ESSID

1. チャンネルの指定

利用する周波数帯を channel を用いて指定する。iwlist コマンドで表示された channel となるだけ重複しないチャンネルを利用する。
3 チャンネル以上離れたチャンネルを利用するすることが推奨されている。

```
# iwconfig wlan0 channel 2
```

2. essid を指定.

```
# iwconfig wlan0 essid arm18
```

デフォルトルーターの指定

たとえば、arm18w というマシンをルーターに選ぶ場合、詳細は、下記 6.4 を参照すること。

```
# route add default gw arm18w
```

6.3 シェルスクリプト

前節で説明したコマンドを実行すれば、アクセスポイントに無線 LAN 接続される。しかし、電波状況が悪い場合等、1回の実行で接続されない場合もある。何度か実行すれば接続されるが、煩雑さは免れない。接続されるまで、つまり、Access Point: にアクセスポイントの MAC アドレスが表示されるまで、これらのコマンドを繰り返すスクリプトを以下に示す。

root になって、以下のシェルスクリプトを実行する。このシェルスクリプトは、無線 LAN 接続が確立すれば、自動的に停止する。最大繰り返し回数は、変数 MAX に指定されている。

```
#!/bin/sh
# (c) 2013.12.3 Y.Honda
# name of wireless interface
INTF="wlan0"
# essid for the interface
ESSID="arm18"
# router host name
ROUTER="arm18w"
# mac address of access point
MAC="00:80:92:3A:9F:E1"
# IP address for the interface
ADDR="172.16.10.XX"
# period between commands
PROD="3s"
# max number of iteration
MAX=5

ifconfig $INTF down
echo "mode"
iwconfig $INTF mode managed
sleep $PROD

AP=`iwconfig | grep Mode`
echo $AP
APP=`echo $AP | sed -e 's/ //g' `
CNT=0
while [ `echo $APP |grep Not-Associated` ]
do
    if test $CNT -gt $MAX
    then
        break
    fi
    CNT=`expr $CNT + 1`
    echo "CNT=$CNT"

    ifconfig $INTF $ADDR
    sleep $PROD

    echo "channel"
    iwconfig $INTF channel 2
    sleep $PROD

    echo "essid"
    iwconfig $INTF essid $ESSID
    sleep $PROD

    echo "ap"
    iwconfig $INTF ap $MAC
    sleep $PROD

    AP=`iwconfig | grep Access`
    echo $AP
    APP=`echo $AP | sed -e 's/ //g' `
done
route add default gw $ROUTER $INTF
```

iwconfig コマンドを実行して、以下に示したように、Access Point の MAC アドレスが表示されれば、正常接続が完了したことを示す。

```
eth0      no wireless extensions.

lo       no wireless extensions.

wlan0    IEEE 802.11abgn  ESSID:"arm18"
          Mode:Managed  Frequency:2.417 GHz  Access Point: 00:80:92:3A:9F:E1
          Bit Rates=54 Mb/s  Tx-Power=15 dBm
          Retry long limit:7  RTS thr:off  Fragment thr:off
          Encryption key:off
          Power Management:off
          Link Quality=58/70  Signal level=-52 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
          Tx excessive retries:0  Invalid misc:2016  Missed beacon:0
```

6.4 経路の追加

ネットワークのアクセス経路の表示や追加は route コマンドを使って行う。

```
# route
```

例えば、以下の様に表示される。

カーネル IP 経路テーブル						
送信先サイト	ゲートウェイ	ネットマスク	フラグ	Metric	Ref	使用数 インタフェース
default	arm18w	0.0.0.0	UG	0	0	wlan0
localnet	*	255.255.0.0	U	0	0	wlan0

localnet 以外は、すべて wlan0 を通してゲートウェイ arm18w をアクセスする設定になっている。

経路を追加・削除するためには、経路情報うち受信先サイト、ゲートウェイ、メトリック、ネットマスク、インターフェイスを指定する。ルートを追加する場合、

```
# route add -net 受信先サイト gw ゲートウェイ metric メトリック netmask ネットマスク インターフェイス
```

という書式となる。ゲートウェイを省略するとゲートウェイは”*”となる。受信先サイトには、ネットワークアドレスや、localnet を指定する。ゲートウェイには、ホスト名や IP アドレスを指定する。メトリックと言うのは、仮想的な距離を意味し、同じ受信先サイトが複数登録されている場合、メトリックが相対的に小さい方の経路が選択される。ネットマスクは、IP アドレスの中でどこまでネットワークアドレスとして認識し、どこをホストアドレスとして認識するかを指定する。たとえば、255.255.255.0 の場合、最初の 3 バイトがネットワークアドレス、さいごの 1 バイトがホストアドレスを意味することになる。

ネットワーク 172.16.10.xxx をゲートウェイ arm18w を通じてアクセスするには、route コマンドで以下の様に経路を追加する。

```
# route add -net 172.16.10.0 gw arm18w metric 0 netmask 255.255.255.0 \\  
wlan0
```

route コマンドで、経路を再表示すると、

```
172.16.10.0      arm18w      255.255.255.0    UG      0      0      0      wlan0
```

という行が追加されて表示される。

6.5 トラブルシューティング

6.5.1 SIOCSIFFLAGS: Operation not possible due to RF-kill

と表示されて、ifconfig などが実行できない場合。

```
# rfkill list
```

を実行して、Wireless LAN の状態を調べます。下記の様に blocked: no となっていれば使える状態です。

```
0: phy0: Wireless LAN
    Soft blocked: no
    Hard blocked: no
3: hci0: Bluetooth
    Soft blocked: no
    Hard blocked: no
```

一方、yes の場合には

```
# rfkill unblock wifi
# rfkill unblock all
```

などとして、ブロックを解除します。Hard blocked: yes の場合には Wireless のスイッチが切れているので、スイッチを入れる必要があります。

6.5.2 Network-Manager を使うと DNS が破壊される

GNOME などで Network-Manager を利用すると、無線 LAN の設定などが GUI を用いて切り替えたりできるので便利である。しかし、Network-

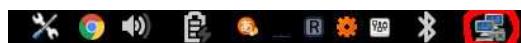


図 6.1: GNOME のタスクバーに表示された Network-Manager

Manager は、`/etc/network/interfaces` と矛盾した IP アドレスなどを使う場合があるので、注意が必要である。

また DNS として Network-Manager 内の情報を使うため、`/etc/resolv.conf` の情報が削除されることがある。その場合には、`resolv.conf` を元に戻さなければならない。

第7章 並列計算ライブラリ PVM

Paralell Virtual Machine(PVM) は、もともと大規模な学術計算を並列計算するために大学や研究機関が協力して開発したライブラリ群です。その経緯から、Debian と同じくオープンでフリーな存在として定着しており、Debian パッケージとしてもインストールすることができます。

PVM は、TCP/IP などのネットワークの専門知識を必要とせず C 言語から関数を呼び出す形で利用することができます。また、子プロセスからさらに自由にプロセスを生成することができるなど柔軟性にも富んでいます。

ロボット知能の研究では、環境の中で行動するロボットの状態を把握したりロボットに人間側の意志を伝えたりする必要が生じます。それらのデータ通信に PVM を利用することができます。

また、数多くのロボットが協調行動をする場合には、それぞれのロボットを制御するプログラムをそれぞれのロボット自身のなかで実行する必要があります。PVM のもつ spawn 機能を利用すれば、それが簡単に実現できます。

いくつかの pvm 関数を用いることで簡単にデータ通信が可能となること以外に、通信速度が比較的高速に実行出来るというメリットもあり

ます。そのためロボットと、時間的な遅れなしに、リアルタイムにデータ通信することも可能になります。

PVM ライブラリは、非常に多くの機能と関数を持っていますが、ここではデータの送受信を行うための基本的な関数だけを取り上げます。

7.1 インストールと開始

PVM を Debian パッケージを利用することで、インストールする方法を説明します。また、PVM は実際にプログラムを実行する前に、PVM に参加するマシンを登録しておく必要があるので、その方法を示します。

7.1.1 apt-get によるインストール

PVM に関するパッケージ pvm, libpvm3, pvm-dev をインストールします。

```
# apt-get install pvm
# apt-get install libpvm3
# apt-get install pvm-dev
```

dpkg コマンドを用いて個別にパッケージファイルをインストールする場合と異なり、依存関係を気にする必要がありません。

7.1.2 dpkg を用いたインストール

なんらかの理由で、`apt-get` コマンドによってインストールできない場合には、`dpkg` コマンドをもちいてパッケージファイルから直接インストールします。以下の 6 つのパッケージをコピーあるいはダウンロードしてインストールします。

1. libtinfo-dev
2. libreadline6-dev
3. libreadline-dev
4. pvm
5. libpvm3
6. pvm-dev

具体的な `dpkg` コマンドの使い方は以下の通りです。

```
# dpkg -i 「パッケージ名」.deb
```

「パッケージ名」の部分には上記パッケージ名と、バージョン番号が入ります。

`apt-get` コマンドによるインストールの場合と異なり、`dpkg` コマンドはパッケージの依存関係を自動解決してくれないので、インストールの順序に注意する必要があります。

7.1.3 PVM を開始する

PVM の起動と終了, ホストの追加, ホストの確認の方法などについてそれぞれ説明します.

起動と終了

pvm コマンドを実行すると, pvm プロンプトが起動します.

```
% pvm
```

また, hosts ファイルを指定して起動すると, pvm に参加するホストを自動的に読み込むことができます.

```
% pvm hosts
```

hosts というのは, pvm でプロセスを起動するホストの名前をリストアップしたファイル名のことです. べつに名前は hosts でなければならぬ訳ではありません. たとえば以下のようない内容のファイルであればファイル名は自由です.

```
host1  
host2
```

ここに指定するリモートホストでは ssh が使用できるように設定されていなければなりません. パスワードを聞いてくるので, それに答えます.

```
pvm>
```

というプロンプトが表示されれば正常に始動しています。sshの認証プロセスを自動化しておけば、パスワード入力の手間を省けます。

```
pvm>quit
```

と入力することによって、pvm プロンプトから抜け出せます。PVM 自体は終了していません。バックグラウンドで pvmd が動いている状態です。
バックグラウンドもふくめて pvmd を完全に止めるには

```
pvm>halt
```

と入力します。

ここで注意することは、/etc/hosts ファイルに記述してある hostname と IP address の値が、実際のホスト名と IP アドレスと必ず一致しないければなりません。PVM はそれらの矛盾を検知するとエラーを生じて起動されません。

ホストの追加

PVM を開始してから、新たにホスト (host3) を追加することも出来ます。

```
pvm>add host3
```

sshの場合と同じ、host3 のパスワードが必要となります。ssh で login できる状態になっている必要があります。すなわち、host3 にユーザー登録がされている必要があります。

ホストの確認

現在 PVM で利用可能なホストの一覧を得るには、conf コマンドを使います。

```
pvm>conf
```

たとえば、下記のような結果になれば、host1 と host2 という 2 つのホストが PVM に参加していることを確認できます。

```
pvm> conf
conf
2 hosts, 2 data formats
      HOST      DTID      ARCH      SPEED      DSIG
    host1    40000  LINUX64     1000  0x00408c41
    host2    80000  LINUXARM    1000  0x00408841
```

実行しているプロセスの確認と停止

ps コマンドを用いて書くホストが実行中のプロセスを表示することができます。a オプションですべてのプロセスを表示できます。

```
pvm> ps a  
ps a  
HOST      TID   FLAG 0x COMMAND  
host1    40002   4/c -  
host2    80001   6/c,f /tmp/2dovr  
host2    80002   6/c,f /tmp/pixy  
host2    80003   6/c,f /tmp/feeler
```

たとえば、このような結果になった場合、/tmp/2dovr, /tmp/pixy, /tmp/feeler というプロセスが host2 上で実行されています。これらは、pvm_spawn 関数を用いて生成されたプロセスであるため、その名前が明示されています。

一方、host1 上で実行されているプロセスは、その名前が明示されず、- で表されています。これは、host1 のプロンプトで実行したプログラムを表しています。

想定外のプロセスが実行中になっている場合、期待通りのデータ通信が行われないなど、障害が生じるので、以下の reset コマンドを用いて、すべてのプロセスを停止させてから新たなプロセス生成を行うようにします。

```
pvm>reset
```

個別のプロセスを停止させたい場合には、kill コマンドで TID を指定します。

```
pvm>kill 80006
```

この例のように、たとえば 80006 を指定した場合、ps a コマンドで実行

中のプロセスを確認すると、

```
pvm> ps a
ps a
      HOST      TID  FLAG 0x COMMAND
rock108    40003      4/c -
bbb140w    80004    6/c,f /tmp/2dovr
bbb140w    80005    6/c,f /tmp/pixy
```

たしかに、/tmp/feeler というプロセスは停止したことが確認できます。

それ以外のコマンドは

```
pvm>help
```

で調べることができます。

7.2 C 言語で PVM を利用する

ここからは、C 言語を用いて pvm を使うための方法を説明します。前節で説明したとおり、PVM が起動している状態で、プログラムの中から pvm 関数呼び出します。

7.2.1 ライブラリのインクルード

プログラムで PVM ライブラリ (API) を利用するためには、pvm3.h の include が必要です。ソースコードの最初の部分で、

```
#include <pvm3.h>
```

とします。

また、コンパイルの際には、pvm3 ライブラリを指定します。たとえば、`hoge.c` という C 言語プログラムを作成し、それをコンパイルして `hoge` という実行ファイルを作成する場合の例は、以下のようになります。

```
% gcc hoge.c -lpvm3 -o hoge
```

7.2.2 構成情報の取得

PVM は、複数のマシンを一つのバーチャルマシンとして使うためのライブラリです。その構成状況をプログラム内から調べることによって、それに応じたプロセスの生成などが可能となります。

構成を返す関数

`pvm_config` 関数を使います。

```
int info=pvm_config(int *nhost,
                     int *narch,
                     struct pvmhostinfo **hostp)
```

それぞれの、引数の意味は以下の通りです。

- **nhost**

バーチャルマシンにおけるホスト数.

- **narch**

現在使用中のデータフォーマットの種類.

- **hostp**

ホスト情報を保持する構造体の配列へのポインタ. この構造体 pvmhostinfo は pvm3.h の中で以下の様に定義されています.

```
struct pvmhostinfo {  
    int hi_tid;      /* pvmmd tid */  
    char *hi_name;  /* host name */  
    char *hi_arch;  /* host arch */  
    int hi_speed;   /* cpu relative speed */  
    int hi_dsig;    /* data signature */  
};
```

構造体のそれぞれのメンバーの意味は

hi_tid: タスク ID

hi_name: 名前

hi_arch: アーキテクチャ

hi_speed: 相対的速度

hi_dsig: 最大パケット長

です.

プログラム例

PVM の構成情報を調べて、コンソールにそれを表示する部分は以下の例のようになります。

```
...
int info;
int nhost, narch, infos;
struct pvmhostinfo *hostp;

info = pvm_config(&nhost, &narch, &hostp);

printf("nhost=%d \n",nhost);
printf("narch=%d \n",narch);
for(i=0;i < nhost;i++)
    printf("host name[%d]=%s \n",i,hostp[i].hi_name);
...
```

7.2.3 プロセスの生成 (spawn)

PVM では、どのプロセスからでも、別のプロセスを生成できます。`pvm_spawn` 関数を使います。英語の `spawn` は、「産卵する」という意味なので、イメージしやすいですね。

具体例を以下に示します。

```
pvm_spawn(PROGRAM,      // 実行ファイルの名前
          (char**)0,   // PROGRAM に渡す引数
          0,           // flag=0:すべてのホストに。flag=1:指定したホストに
          "",          // where, プロセスを生成するホスト。flag=0 の場合は ""
          nprocess,    // 起動する slave の数
          tids);       // 生成されたタスクの ID (配列) が返される
```

`pvm_spawn` 関数が成功すると、`tids` という 1 次元配列に、`spawn` した

タスク ID が配列として返って来ます。後ほど、`pvm_send` でデータを送るプロセスを指定する場合に、そこに格納された ID でプロセスを指定します。

ホストマシンの指定

このなかで、`flag` の値によってプロセスを生成するホストマシンが決まります。`flag=0` の場合、PVM が全ホストにたいして、自動的にプロセス生成します。`flag=0` の場合には、`where` の値は無視されます。

たとえば、上の例のように、`flag` の値を 0 として、`nprocess` の値を `pvm_config` で調べたホストの数に等しくしておくと、PVM は自動的に各ホストで `PROGRAM` を実行し始めます。

`flag=1` の場合には、次の引数 `where` に指定されたホストでプロセスが生成されます。`where` には、ホスト名を指定します。DNS を用いない場合には、ホスト名は、`/etc/hosts` ファイルに記述されていなければなりません。

`spawn` される実行ファイル

`spawn` される実行ファイルの名前は、`PROGRAM` で指定します。デフォルトでは、PVM は環境変数 `PVM_ROOT` に指定されたディレクトリの元にある実行ファイルを探しに行きます。

`PVM_ROOT` に実行プログラムを毎回置くこともできますが、`PROGRAM` の値として、ディレクトリの絶対パスとファイル名を指定する

こともできます。

```
#define PROGRAM "/home/taro/slave"
#define N 10
...
int tids[N];
nhost=1;
...
pvm_spawn(PROGRAM, (char**) 0, 1, "host2", nhosts, tids);
...
```

この例では、タスクの数として 10 と固定したので、PVM の構成に加えるホスト数がそれを超えないように注意する必要があります。

flag=1 として、プロセス起動ホストを "host2" と指定し、nhost=1 ので、ひとつだけ /home/taro/slave というプロセスが生成されます。

tids 配列には spawn されたタスクのタスク ID が格納されます。それらのタスクにデータを送る際には、この tids が必要となります。

spawn される実行ファイル、上の例では /home/taro/slave というファイルは spawn をしようとするホスト上に存在し、実行可能な状態でなければなりません。たとえば pvm_spawn 関数で指定する flag が 0 の場合には、pvm に参加しているすべてのホストでそのプログラムが実行されるので、そのすべてのホストの /home/taro/slave が実行可能な状態にある必要があります。

spawn するホストを指定する場合は、その指定したホストに実行ファイルが存在しなければなりません。

7.2.4 標準出力

pvm_spawn で生成された slave プロセスからの標準出力は、 slave 側の pvmd を通して、 master の /tmp/pvml.[uid] の中に書き込まれます。しかし、それぞれのホストの pvmd どうしが通信する際に、標準入出力用のバッファの内容が吐き出されて (flush) いなければ、出力はうまく /tmp/pvml.[uid] に反映されない場合があります。

これを解決するためには、 slave 側のプログラムの先頭付近で

```
setlinebuf(stdout);
```

とバッファリングモードを指定しておけば、 printf を実行するたびに、バッファの内容が吐き出され、標準出力は /tmp/pvml.[uid] に正確に反映されます。

7.2.5 データの送信と受信

PVM を使う場合に限らず、並列計算・データ処理において、中心的な役割を果たし、もっとも頻繁に使用される機能は、データの送出と受信です。ごく単純化して言えば、並列処理とは、全体の処理を小さなブロックに分割して、それぞれを別々のマシンで処理し、結果を集めて出力することと言えます。あるいは、次の処理にまわすということの繰り返しであると言えます。

分割して送り、受け取って処理し、その結果を送り返すという処理、つまり、データ送受信を繰り返すことになります。

PVM では、配列をパケットにパックして送出し、受け取り側がそのパケットをアンパック（配列に戻す）という形式でデータ送受信を実現します。以下では、具体例に沿って説明します。

タスク ID(tid)

pvm ではデータの通信をタスク（プロセス）間で行います。そのためには、通信の相手と自分の同定（Identify）が必要です。それを tid と呼ばれる数値で行います。

自分自身のタスク ID を知るためには

```
int mytid;  
mytid=pvm_mytid();
```

という関数を使います。また、自分を生成（spawn）した tid を知るためには、

```
int ptid;  
ptid=pvm_parent();
```

という関数を使います。

parent 側にデータを送り返す場合など、この tid を指定します。

TAG を指定してデータの送信

データを相手プロセスに送信するためには、pvm_initSend 関数、pvm_pkXXXX 関数、そして \pvm_send 関数をそれぞれ実行しなけれ

ばなりません。

`pvm_initsend(int encoding)` 関数は、送信バッファをクリアし、データ（メッセージ）のパックの送信バッファを作ります。`encoding` の値はエンコーディング方法を指定する値ですが、デフォルト値`encoding=0` を用います。この関数は、`pvm_pkXXXX` 関数を呼ぶ前に必ず実行します。そうしないと、送信バッファがクリアされず古いデータの次に、新しいデータが追加されてしまうため、アンパックした際に、正しくデータが読み取れない場合があります。

つぎに、`pvm_pkXXXX()` 関数で、データを送信バッファにパックします。`XXXX` の部分には、`int` や `double` が来ます。たとえば、倍精度浮動小数点データを送信する場合、

```
pvm_pkdouble(double *data, int nitem, int stride)
```

と、3つの引数を指定します。パックするデータ(`double data`)は、変数そのものではなく、そのポインタ(`&data`)で指定します。配列変数の場合、配列名そのものを指定します。これは、受け取る際のアンパックにおいても同様です。`pvm_pk` 関数の2番めの整数引数は、パックする配列のサイズ(`nitem`)を指定します。3番目の引数は、その間隔(`stride`)を指定します。配列を `nitem`だけすべてパックするばかりには、1を指定します。たとえば、2を指定すると、1つおきにデータがパックされます。

`pvm_send` 関数では、`tids` の指定以外に、通信 TAG を指定する必要があります。同じプロセス間で、種類の違う通信が行われる可能性があ

るので、それらを識別するためです。

下記に整数 int 型と char 型の配列を送信する例を示します。

```
#include <string.h>
#define TAG 1
...
char str[256];

strcpy(str,"最近、寒くなつきましたね");
int i_dum;
for(i=0;i < nprocess;i++){
    pvm_initsend(0);           //送信バッファなどの初期化
    i_dum=strlen(str);
    pvm_pkint(&i_dum,1,1);    //整数（ポインタ）を1つパック
    pvm_pkbyte(str,strlen(str),1); //文字配列をその長さだけパック
    pvm_send(tids[i],TAG);    //タスク ID 番号 i のプロセスに送信.
}
```

initsend は、send を実行する前に毎回必ず必要なので、注意してください。

受信側の例

```
#define TAG 1
...
int ptid;
int my_tid;
ptid=pvm_parent();
mytid=pvm_mytid();
...
pvm_recv(ptid,TAG);           //親タスクからのパケットを受信.
pvm_upkint(&str_len,1,1);    //整数(ポインタ)をアンパック.(配列の長さ)
pvm_upkbyte(str,str_len,1);  //文字配列をその長さだけアンパック
str[str_len]='\0';           //文字列末を代入
printf("%d:%s\n",mytid,str);
...
```

受信では、送信とは逆に受信、アンパックという順序でデータを取り出します。

ptid で指定されたプロセスから recv し、アンパックしてコンソールへ出力しています。コンソール上へ表示された文字は、PVM を開始したホストの/tmp/pvml.XXXX に出力されるので、tail コマンドなどで確認できます。XXXX は PVM を開始したユーザー ID です。

```
$ tail /tmp/pvml.XXXX
```

7.2.6 遅延のないデータ送受信を行うために

前節の例では、受信側のソースコードにおいて、pvm_recv 関数を用いました。この関数は、受信バッファにデータが到着していない場合

には、受信バッファにデータが入るまでプロセスを停止します。つまり、プログラムはそこから先に進みません。このことをブロッキングと呼びます。データが到着するまで、一切の計算を停止させたい場合、すなわち、データの同期をとりたい場合には、これは便利な機能です。

いっぽう、データの同期をとる必要がなく、バッファにデータがない場合には次の計算などの処理に移りたい場合には、困ります。その場合には、`pvm_nrecv` 関数を用います。

逆に、`pvm_recv` 関数あるいは、`pvm_nrecv` 関数が実行される前に、受信バッファーに大量のデータが蓄積された場合、`recv` 関数は古いデータから順番にデータを取り出します。`pvm` はバッファにたまっているデータを捨てません。

センサーなどの値を連続的に受け取る場合、受け取り側のプログラムの実行速度が遅いと、送られてきたセンサーデータなどが受信バッファに大量に蓄積します。この場合、`recv` 関数は古いデータから読み込んでいくので、リアルタイムなセンサー値が読み取れることになります。

これを解決する簡単な方法は、`send` 側のプロセスにおいて `sleep` 関数などを用いて、実行速度を落とします。`recv` 側の実行速度が十分に `send` 側の速度より早ければ、受信バッファーにはつねに最新のデータがたもたれます。

7.2.7 **spawn**されたプログラムが正常終了しないで残っている場合

各プロセスが何らかの理由で正常に終了しなかった場合には、PVMをリセットしてください。

```
pvm> reset
```

を実行します。プロセスが残っているかどうかは、

```
pvm> ps a
```

で調べることができます。

7.2.8 整数型データの通信プログラム例

master 側

```
1 #include <stdio.h>
2 #include <pvm3.h>
3
4 #define PROGRAM "/root/sensor_test1"
5 #define SLAVE "rb143"
6 #define MAXNHOST 256
7
8 #define sTAG 1
9 #define rTAG 2
10
11 int main(){
12     int i;
13     int info;
14     int nhost, narch, infos;
15     int tids[MAXNHOST];
16     int dum;
```

```
17     int rdum;
18     struct pvmhostinfo *hostp;
19
20
21     info = pvm_config(&nhost, &narch, &hostp);
22
23     printf("nhost=%d \n", nhost);
24     printf("narch=%d \n", narch);
25     for(i=0;i<nhost;i++)
26         printf("host name[%d]=%s \n", i, hostp[i].hi_name);
27
28     // SLAVE 一つだけ spawnする.
29     pvm_spawn(PROGRAM, (char**)0, 1, SLAVE, 1,tids);
30     printf("tid:%d\n", tids[0]);
31
32     /*
33     dum=tids[0];
34     pvm_initsend(0); // 初期化は send の前に必ず必要.
35     pvm_pkint(&dum, 1, 1);
36     pvm_send(tids[0], sTAG);
37
38     // SLAVEから送り返されてきたデータを受け取る.
39     pvm_recv(tids[0], rTAG);
40     pvm_upkint(&rdum, 1, 1);
41
42     printf("send:%d recv:%d\n", dum, rdum);
43     */
44
45 }
```

slave 側

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define rTAG 1
5 #define sTAG 2
6
7 int main(){
8     int irecv;
9     int ptid;
10    int dum;
11 }
```

```
12     ptid=pvm_parent();
13
14     printf("こんにちは、整数をrecvします。\\n");
15
16     pvm_recv(ptid,rTAG);
17     pvm_upkint(&irecv,1,1);
18     printf("受け取った数字:%d",irecv);
19
20     dum=irecv;
21     pvm_initsend(0);
22     pvm_pkint(&dum,1,1);
23     pvm_send(ptid,sTAG);
24
25 }
```

7.3 トラブルシューティング

PVM の開始やプログラムのコンパイルなどの際によく起こる問題と解決法を簡単に示します。

1. 各マシンで, /etc/hosts 内のホスト名と /etc/hostname で指定されるホスト名に矛盾が無いこと。矛盾があると, PVM がフリーズする場合があります。
2. PVM の開始の際, chmod a+rwx /tmp をしておくことが必要な場合があります。
3. sshd はインストールされている必要があります。環境変数で rsh を指定する場合にはリモートシェルが使える環境が整備されていなければなりません。

4. `#include <pvm3.h>` をするためには、`pvm-dev` がインストールされている必要があります。

第8章 グラフ作成(GNUPLOT)

数値データなどをグラフにするには, `gnuplot`¹ を使います。データの時間変化を二次元的なグラフに描画するだけでなく、変数が2つある場合のデータを3次元的なグラフにしたり、媒介変数を使って立体的にデータを可視化することも可能です。さまざまなコード例がネット上²に例示されており、目的に応じてそれらを参考にすると役立つでしょう(図8.1参照)。

`gnuplot`では、どのようにグラフを描画するかのコマンドなどを一連のコードとして保存しておいてそれを `load` することによってグラフを作成できます。複雑なコマンドなどをグラフをつくる度に入力する必要はありません。コードファイルとして保存をしておいて、それを `load` するようにしましょう。

作成されたグラフを画像として出力して、WEB上などに表示できます。また、EPS形式で保存し、論文などのLaTeXファイルに読み込むことで、文章のなかにグラフを示すことができます。本書に用いたグラフもそのようにして作成したものです。

¹<http://www.gnuplot.info/>

²<http://gnuplot.sourceforge.net/demo/>

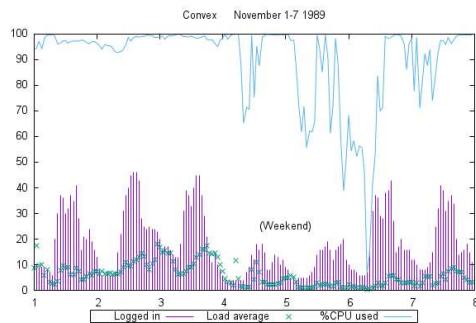


図 8.1: gnuplot で複数のデータをグラフ化する

```

1 # set terminal png transparent nocrop enhanced size 450,320 font "arial,8"
2 # set output 'using.2.png'
3 set key bmargin center horizontal Right noreverse enhanced autotitle box \
4     lt black linewidth 1.000 dashtype solid
5 set style data lines
6 set title "Convex      November 1-7 1989"
7 set xrange [ 1.00000 : 8.00000 ] noreverse nowriteback
8 x = 0.0
9 ## Last datafile plotted: "using.dat"
10 plot 'using.dat' using 3:4 title "Logged in" with impulses,\ 
11      'using.dat' using 3:5 t "Load average" with points,\ 
12      'using.dat' using 3:6 t "%CPU used" with lines

```

図 8.2: 図 8.1 を gnuplot で描画するためのコード

また、3次元的なグラフを表面表示したり、同時に等高線を表示することも可能です。3次元的なグラフは複雑な構造になることが多いので、ひとつの角度からそれをながめるだけではそれを容易に理解できない場合があります。Debian³ デスクトップ上で3D表示されたグラフは、マウスで表示角度を変更しながらその構造を確認できるので、理解の助けとなります。図8.3に示したコードをgnuplotにloadしてみてることを

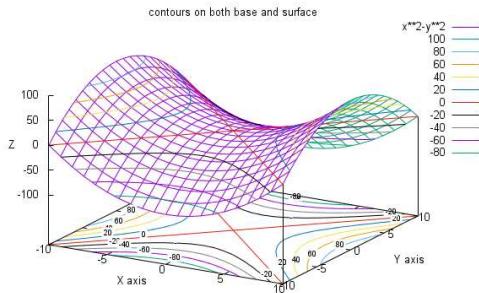


図8.3: 3次元構造を可視化する

お勧めします。

³<https://www.debian.org/index.ja.html>

```
1 # set terminal png transparent nocrop enhanced size 450,320 font "arial,8"
2 # set output 'contours.7.png'
3 set key at screen 1, 0.9, 0 right top vertical Right noreverse enhanced \
4     autotitle nobox
5 set style textbox opaque margins 0.5, 0.5 noborder
6 set view 60, 30, 1, 1.1
7 set samples 20, 20
8 set isosamples 21, 21
9 set contour both
10 set cntrlabel format '%8.3g' font ',7' start 5 interval 20
11 set cntrparam levels auto 10
12 set style data lines
13 set title "contours on both base and surface"
14 set xlabel "X axis"
15 set ylabel "Y axis"
16 set zlabel "Z "
17 set zlabel offset character 1, 0, 0 font "" textcolor lt -1 norotate
18 splot x**2-y**2 with lines, x**2-y**2 with labels boxed notitle
```

図 8.4: 図 8.3 を gnuplot で描画するためのコード

8.1 gnuplot のインストール

gnuplot は Debian のパッケージになっているので、`apt-get`を使ってインストールします。

```
# apt-get install gnuplot
# apt-get install gnuplot-x11
```

8.2 起動と終了

コンソールから gnuplot を起動すると、対話的にグラフを作成できます。

```
% gnuplot
```

gnuplot プロンプトを終了しコンソール（シェル）にもどるためにには。

```
gnuplot> exit
```

を実行します。

8.3 データ描画(2D)

たとえば、以下のようなデータファイル(data.xy)を描画します。

1	0.5	7.8	4.3
2	0.8	6.9	2.1
3	1.1	2.0	3.5
4	1.5	1.9	3.9
5	2.0	0.8	4.2

データを描画するためのコマンドは plot です。

```
gnuplot> plot 'data.xy'
```

このように対話的に plot を用いると、描画結果は 画面上に描画されます。 data.xy ファイル内の 1 列目が横軸、2 列目が縦軸としてグラフが描画されます。

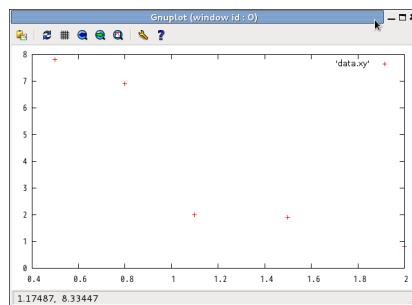


図 8.5: 単純な plot 結果

点のサイズが小さいので、大きくしてみましょう。ポイントサイズを 3 にしてみます。plot コマンドに ps 3 を追加します。

```
gnuplot> plot 'data.xy' ps 3
```

さらに、点と点を線でつないでみましょう。

```
gnuplot> plot 'data.xy' ps 3 with lp
```

だいぶ、見やすくなってきました。

つぎに横軸と縦軸にラベルをつけます。

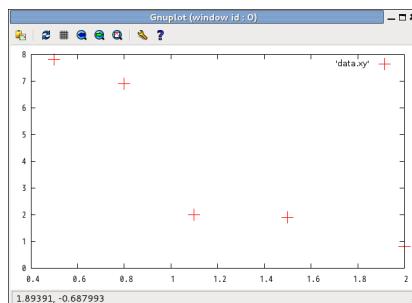


図 8.6: 点のサイズを大きく (3) する

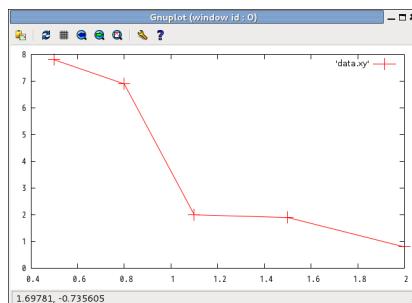


図 8.7: 点と点を線でつなぐ

```
gnuplot> set xlabel 'Time(s)'  
gnuplot> set ylabel 'Angle(rad)'  
gnuplot> plot 'data.xy' ps 3 with lp
```

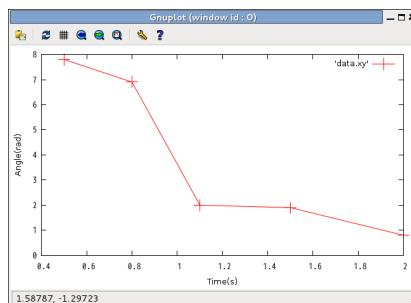


図 8.8: 横軸と縦軸にラベルをつける

描画範囲を指定するには、`set xrange`, `set yrange` を実行します。

```
gnuplot> set xrange [0:3]
gnuplot> set yrange [0:9]
```

`data.xy` の 2 列目も描画するれば、2 つのグラフを同時に一つの図の中に示すことができます。

```
gnuplot> plot 'data.xy' u 1:2 ps 3 w lp, 'data.xy' u 1:3 ps 3 w lp
```

ここで、`with` というオプションを `w` と省略しました。同様に、`u` も実は `using` の略です。この様にオプションは省略形で短く記述することができます。

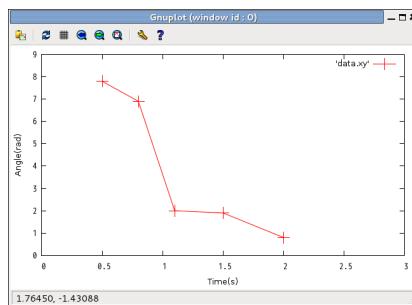


図 8.9: 描画する範囲を指定する。

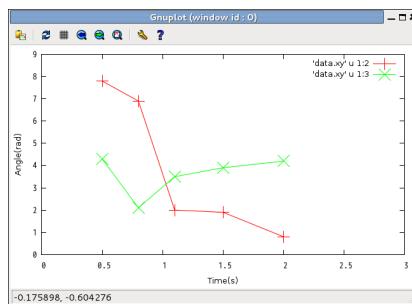


図 8.10: 2 つのデータを同時に描画する。

条件の異なるデータを比較したい場合には、このように、同時に表示するようにします。

8.4 文字の大きさ変更

gnuplot のデフォルトでは、論文掲載用、あるいは発表用としては、文字の大きさが小さい場合があります。set terminal で文字フォントと大きさを指定します。

```
gnuplot> set terminal wxt 0 font "Helvetica,15"
```

ここでは、set terminal の行の中でフォントを指定したので、出力され

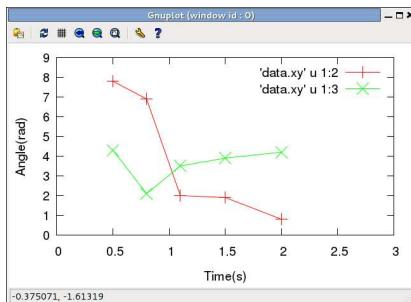


図 8.11: フォントを "Helvetica,15" に指定。

る文字すべてにわたって、このフォントとその大きさが摘要されます。一方、set xlabel font "Helvetica,15" のように個別の出力項目について、フォントとその大きさを指定することも可能です。

デフォルトでは、データファイル名がグラフタイトルとして表示されるので、グラフの意味に対応して title を変更します。

```
plot 'data.xy' u 1:2 ps 3 w lp t "Sound sensor", \
      'data.xy' u 1:3 ps 3 w lp t "Light sensor"
```

plot コマンドの行が長くなる場合には、この例の様に、行末に \ をつけ

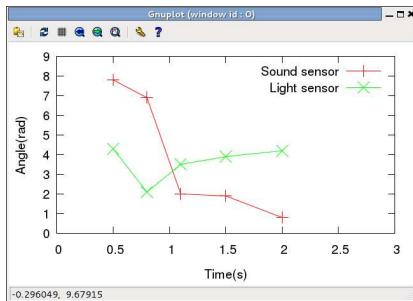


図 8.12: title を付ける

ると、次の行にその続きをつづけることができます。

8.5 制御コードファイル

データを描画する度に、これらの入力を行うのは手間がかかるので、これらの入力をすべてコードファイルとして保存しておくことができます。たとえば、vi エディタを使って次の内容のファイル data.gp を作ります。この例では、data.gp というファイルに保存しましたが、もちろん別のファイル名でもかまいません。いっぽんに、データファイルはどん

```
1 set terminal wxt 0 font "Helvetica,15"
2 set xlabel 'Time(s)'
3 set ylabel 'Angle(rad)'
4 set xrange [0:3]
5 set yrange [0:9]
6 plot 'data.xy' u 1:2 ps 3 w lp t "Sound sensor", \
7      'data.xy' u 1:3 ps 3 w lp t "Light sensor"
```

図 8.13: gnuplot の制御コードファイル例

どんどん増えていくので、どのようなデータのための gnuplot コードか分か
るように、ファイル名を工夫することが必要です。

セーブされた data.gp ファイルを load することによって、グラフに反
映させます。

```
gnuplot> load "data.gp"
```

data.gp ファイル内には、先に実行した plot コマンド、set xlabel など
の他にも、gnuplot を制御するための様々なパラメータなどを記述する
ことができます。

8.6 対数スケール

非常に大きな数や、非常に小さい数を同時に描画したい場合、対数ス
ケールを用います。

```
gnuplot> set logscale <x/y> <基底>
```

8.7 EPS 出力

描いたグラフを EPS 形式で出力すためには data.gp ファイル内で set terminal を postscript に変更し, set output を指定します。この様に記述

```
1 set terminal postscript eps color enhanced font "Helvetica,20"
2 set output "data.eps"
3 set xlabel 'Time(s)'
4 set ylabel 'Angle(rad)'
5 set xrange [0:3]
6 set yrange [0:9]
7 plot 'data.xy' u 1:2 ps 3 w lp t "Sound sensor", \
8      'data.xy' u 1:3 ps 3 w lp t "Light sensor"
```

図 8.14: グラフを EPS 出力するための gnuplot の制御コードファイル例

された, data.gp ファイルを先述のように load すると, data.eps というファイルが自動生成されます。

8.8 EPS 出力で複数のグラフを **replot** する

terminal が postscript の場合, plot のあと, replot を実行すると, 最後の replot されたグラフのみが EPS ファイルに書き込まれ, それ以前のグラフは現れません。それを回避するために, いったん

```
gnuplot> set output '/dev/null'  
gnuplot> plot f(x)  
gnuplot> set output 'file.eps'  
gnuplot> replot g(x)
```

とします。最初のプロットは/dev/nullにプロットし、最後の replot の直前で、output ファイルを指定します。

8.9 LaTeX 文章へのグラフの組み込み

EPS ファイルができたので、いよいよ LaTeX 文章ファイルにグラフを組み入れてみよう。プリアンブル部で次のように記述します。プリアンブルとは、\documentclass と \begin{document} のあいだの部分のことです。

```
\usepackage[dvips]{graphicx,psfrag}
```

つぎに、LaTeX ファイル内で実際に EPS ファイルを読み込んでグラフを表示したい場所で\includegraphics を用います。

```
\begin{figure}[h]  
  \begin{center}  
    \includegraphics[width=\figw]{data.eps}  
  \end{center}  
  \caption{\label{fig:data_eps}gnuplot から出力された EPS グラフ}  
\end{figure}
```

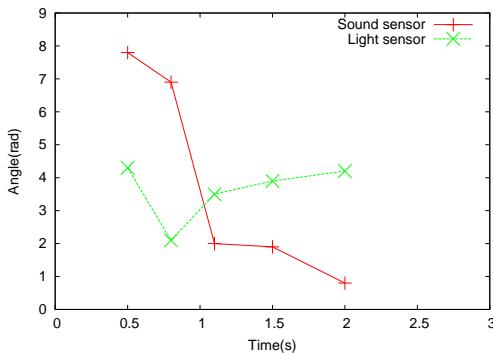


図 8.15: gnuplot から出力された EPS グラフ

8.10 EPS ファイル内の文字を日本語に置き換える

```
\begin{figure}[h]
\psfrag{Time(s)}{\footnotesize 時間 (s)}
\psfrag{Angle(rad)}{\footnotesize 角度 (rad)}
\psfrag{Sound sensor}{\tiny 音センサー}
\psfrag{Light sensor}{\tiny 光センサー}
\begin{center}
\includegraphics[width=0.8\textwidth]{data.eps}
\end{center}
\caption{\label{fig:data_psfrag} EPS ファイル内の文字を日本語に置き換える}
\end{figure}
```

この機能を用いるためには, dvipdfmx ではなく, dvips と ps2pdf を使

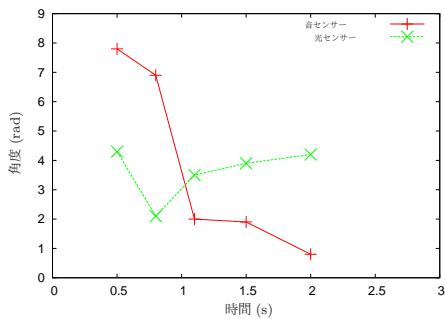


図 8.16: EPS ファイル内の文字を日本語に置き換える

う必要がります。

また、フォントの選択の都合などで pdf ファイルの生成にどうしても dvipdfmx をつかいたい場合には、図の部分だけ図 8.17 の様なファイルを用意して eps ファイルを作りおいてから本文の方にその eps ファイルを読み込むと良いでしょう。

```
1 \documentclass{jarticle}
2 \special{papersize=200.00mm,200.00mm}
3
4 \usepackage[dvips]{graphicx,psfrag}
5 \usepackage{color}
6
7 \newlength{\xadd}
8 \setlength{\xadd}{0mm}
9 \addtolength{\textwidth}{\xadd}
10 \addtolength{\oddsidemargin}{-0.5\xadd}
11 \addtolength{\evensidemargin}{-0.5\xadd}
12
13 \newlength{\figw}
14 \setlength{\figw}{0.9\textwidth}
15
16
17 \begin{document}
18 \pagestyle{empty}
19
20 \psfrag{Time(s)}{\footnotesize 時間(s)}
21 \psfrag{Angle(rad)}{\footnotesize 角度(rad)}
22 \psfrag{Sound sensor}{\tiny 音センサー}
23 \psfrag{Light sensor}{\tiny 光センサー}
24 \begin{center}
25   \includegraphics[width=0.8\textwidth]{data.eps}
26 \end{center}
27
28 \end{document}
```

図 8.17: psfrag と ps2pdf コマンドを使って eps ファイルを生成するための LaTeX コード

8.11 C言語ソースコードから直接グラフを作る

gnuplotでグラフを描画する作業は、通常は、シェルから gnuplot を起動し、その中でコマンドを実行することによって行います。あるいは、gnuplot の制御ファイルを作って、それを load コマンドで gnuplot に読み込むことでグラフを作ります。

ここでは、C言語のパイプオープン (popen) 機能を利用して、シミュレーションなどで作成したデータファイルを、直接C言語のソースコードによってグラフにする方法を紹介します。この方法を用いると、シェルから gnuplot のコマンドを実行したり、制御ファイルを編集したりする作業が省略できます(図 8.18)。

```
1 #define RESULTS "result_xxxx.xy"
2 // 計算結果を保存するファイル名
3 #define OUTPUTEPS "result_xxxx.eps" // グラフのepsファイル名
4
5 FILE* fp; // 計算結果を出力するためのファイルポインター
6 FILE* gp;
7 // gnuplotをパイプオーブンしてコマンドを渡すためのポインター
8
9 fp=fopen(RESULTS,"w"); // ファイルRESULTSを開く
10
11 ...
12 fprintf(fp,"%f %f %f\n", t, res1, res2);
13 // 計算結果をRESULTSに書き込む
14 ...
15
16 fclose(fp); // 計算が終了したら, RESULTSを閉じる.
17 ...
18
19 gp=popen("gnuplot","w"); // プロセスgnuplotを開く.
20 fprintf(gp,"set term post eps enhanced color 'Helvetica,20'\n");
21 // ターミナルタイプをepsにして, 文字の大きさを20ptとする.
22 fprintf(gp,"set output '%s'\n",OUTPUTEPS);
23 // グラフの出力先をOUTPUTEPSにする.
24 fprintf(gp,"set xlabel 'tau'\n");
25 // 横軸のラベル xlabelを指定する.
26 fprintf(gp,"set ylabel 'lambda'\n");
27 // 縦軸のラベル ylabelを指定する.
28 fprintf(gp,"plot '%s' using 1:2 with lp title 'Re(lambdaP)', \
29 '%s' using 1:3 with lp title 'Im(lambdaP)' \
30 \n",RESULTS,RESULTS);
31 // gnuplot の plotコマンドでグラフを描画する.
32 // 複数のグラフを同時に描画する場合には, 複数行に分けて記述した方が
33 // 分かりやすい.
34 pclose(gp); // gnuplotに対するプロセスピンターを閉じる
```

図 8.18: `popen` を使って, C 言語から直接 `gnuplot` を使う例

8.12 簡易アニメーション

たとえば、以下のような内容の制御ファイル”anime.gp”を作って、

```

1 xwidth=600
2 ywidth=400
3 set term x11 1 title "Accel" size xwidth,ywidth position 0,0
4 pi=3.141592
5 set yrange [-pi:pi]
6 plot '< tail -100  opedev9.0.dat' u 1:2 w l t "ax", \
7      '< tail -100  opedev9.0.dat' u 1:3 w l t "ay", \
8      '< tail -100  opedev9.0.dat' u 1:4 w l t "az"
9 set term x11 2 title "Gyro" size xwidth,ywidth position xwidth+10,0
10 set yrange [*:*]
11 plot '< tail -100  opedev9.0.dat' u 1:5 w l t "gx", \
12      '< tail -100  opedev9.0.dat' u 1:6 w l t "gy", \
13      '< tail -100  opedev9.0.dat' u 1:7 w l t "gz"
14 pause 0.2
15 reread

```

図 8.19: tail コマンドと reread を使って、簡易アニメーションを行うための gnuplot コード

```
% gnuplot anime.gp
```

を実行すると、データーファイルの最後の 100 行が常に描画されます。データファイルの行末に、どんどんデータを追加していくけば、リアルタイムにそのデータがグラフに描かれます。

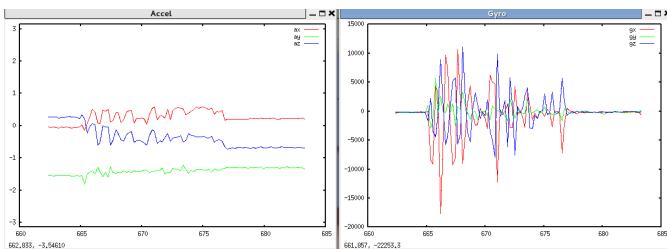


図 8.20: 簡易アニメーションの例

8.13 演算結果をプロットする

GNUPLOTでは、データファイルの中の値をそのままグラフにするのではなく、それらの値に演算をしたもの、たとえば定数を掛けたものをプロットすることができます。

```
gnuplot> a=0.123
gnuplot> b=3.210
gnuplot> plot "data.xy" using ($1*a):($2*b)
```

ここでは、定数a,bを定義しておいて、それを1列目と2列目のデータに掛け算してプロットした例を示した。

8.14 データを間引いてプロット

すべてのデータをプロットするのではなく、行やブロックを間引いてグラフを作成することができる。そのためには、`every` オプションを利用する。

8.14.1 ブロック

GNUPLOT は 1 行の空白行で区切られたデータのかたまりを一つのブロックとして認識する。つぎのようなデータファイル `gnuplot_sample1.xy` があるとする。

1	0.5	7.8	4.3
2	0.8	6.9	2.1
3	1.1	2.0	3.5
4	1.5	1.9	3.9
5	2.0	0.8	4.2
6			
7	2.2	0.9	5.0
8	2.4	1.0	5.1
9	2.6	1.3	6.0
10	2.8	1.8	6.5
11	3.0	1.5	4.8

6 行目の空白行でブロックが区切られており、1 ~ 5 行目がブロック 0、7 ~ 11 行目がブロック 1 となる。ここでは、ふたつのブロックの例を示したが、以下空白行で区切られたデータ行が現れるとブロック番号が増えたブロックとして認識される。

以下のような内容の `gnuplot_sample1.gp` というファイルを gnuplot で実行する。

```
1 set term post eps enh color "Helvetica,20"
2 set output "gnuplot_sample1.eps"
3 set xlabel "xlabel name"
4 set ylabel "ylabel name"
5 plot "gnuplot_sample1.xy" w lp
```

```
% gnuplot gnuplot_sample1.gp
```

データ間を線で結ぶオプションを用いたが、ブロック間は線で結ばれないで描画される。

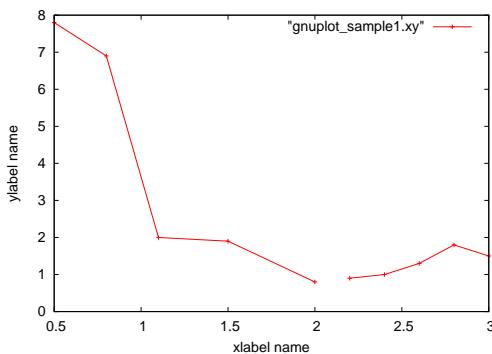


図 8.21: ふたつのブロックにわかれたデータのプロット例

8.14.2 every オプション

every オプションを用いて、行刻みを 2 に指定した

```

1 set term post eps enh color "Helvetica,20"
2 set output "gnuplot_sample2.eps"
3 set xlabel "xlabel name"
4 set ylabel "ylabel name"
5 plot "gnuplot_sample1.xy" every 2 w lp

```

を実行すると、データが2つごとに、すなわち1つ飛ばしで描画される。もちろん、`every 2` の部分を `every 3` とすれば、3つごとに描画さ

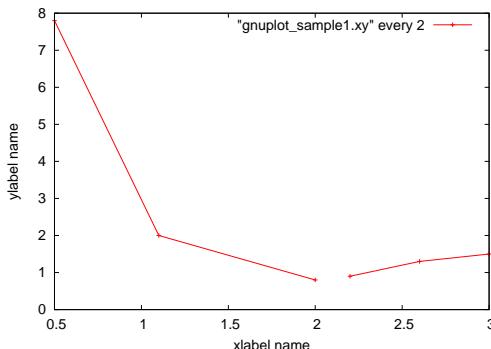


図 8.22: `every 2` オプションを用いて、データを2行ごとにプロット

れ、以下同様である。

`every` オプションでは、行刻み以外に以下のフォーマットで初期行や終了行などを指定できる。

every 行刻み: ブロック刻み: 初期行: 初期ブロック: 終了行: 終了ブロック

以下の `gnuplot_sample3.gp` というファイルを実行すると、

```
1 set term post eps enh color "Helvetica,20"
2 set output "gnuplot_sample3.eps"
3 set xlabel "xlabel name"
4 set ylabel "ylabel name"
5 set xrange [0.5:3.0]
6 set yrange [0:8]
7 plot "gnuplot_sample1.xy" every 1::1::3 w lp
```

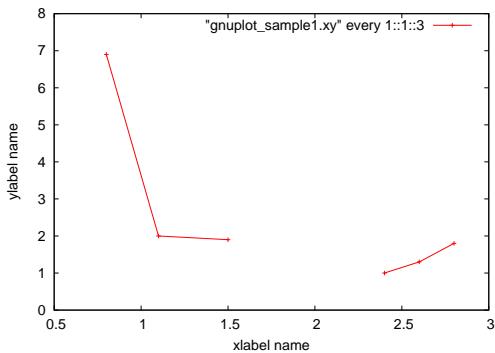


図 8.23: `every` オプションを用いて、開始行と終了行を指定

各ブロックのなかで、1 ~ 3 行目がプロットされる。行番号は、ブロック番号同様に、0 から始まることに注意する必要がある。

8.15 範囲指定してプロット

値の範囲が条件に合う場合だけ、プロットすることができる。usingオプションの中で、例えば

```
plot "gain140820a.dat" u 1:($4<0.1 ? ($4>0.01 ? $2:1/0):1/0)
```

のように値の範囲を指定する。この例では $0.01 < \$4 < 0.1$ の場合だけ y 軸に $\$2$ の値を用いてプロットする。

$1/0$ の部分には、本来条件に合わない場合にプロットされる値が入る。 $1/0$ は定義されない値なので、なにもプロットされない。

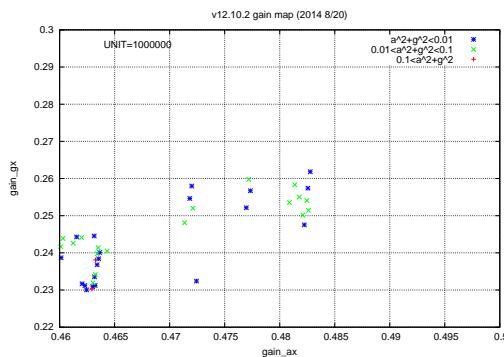


図 8.24: 値の範囲を指定してプロットした例

```
1 set term post eps enh color font 'Helvetica,20'
2 set output 'data1.eps'
3 set zeroaxis
4 set xlabel "x"
5 set ylabel "y"
6 set xrange [-20:30]
7 set yrange [-3:3]
8 f(x)=a0+a1*x+a2*x*x+a3*x*x*x
9 fit f(x) "data1.xy" u 1:2 via a0,a1,a2,a3
10 plot f(x) , "data1.xy" w p ps 3 pt 7
```

8.16 データを関数で fit する

数値データを関数で近似(fit)したい場合、たとえば下記のようにします。data1.xy の 1 列目を x 、2 列目を y として、3 次関数 $f(x)$ を fit しています。パラメータは 4 つあるので、9 行目の via で指定しています。

この例では行っていませんが、パラメータの初期値を指定しないと、gnuplot が正常にパラメータを決定できず、fit が終了する場合があるので、パラメータ初期値を推測できる場合には、それを指定してから fit を実行した方が無難です。

付録A ボードコンピュータ

ロボットには、1枚のボードで構成されたボードコンピュータを用いる。多くのボードコンピュータは数センチ四方のサイズで、様々な入力および出力インターフェイスを備えている。ロボットの筐体に組み込んで、感覚運動写像などを行うことに適している。

ボードコンピュータには、Armadillo, BeagleBone Black(BBB), Edison, Raspberry Pi などがある。Armadillo, BBB および Raspberry Pi は、そのCPUにARMアーキテクチャを採用している。一方、EdisonはIntel i386アーキテクチャの流れを汲むAtom採用している。また、LEGO EV3はボードコンピュータではないが、ARM CPUを採用している。

これらのボードコンピュータにLinux OS、特にDebian GNU/Linuxを導入してロボット知能に用いることを考える。DebianはサーバOSあるいはデスクトップOSとして開発されてきた経緯を持つが、それ故に膨大なライブラリやアプリケーションの資産をもっている。それをボードコンピュータに導入することにより、それらの資産を活用すると共に、デスクトップOSとの親和性や運用の容易さが期待される。

A.1 BeagleBone Black

BeagleBone Black (以下 BBB と略記) は, AD 変換ポートと, PWM ポートを複数備えた, ボードコンピュータである¹. OS として, Debian をインストールすることも可能であり, 感覚行動写像を実現するハードウェアとして, PIC や H8 を必要としない点でも回路の簡略化や重量の軽量化が期待できる. また, Debian の中で C 言語を用いてセンサーやモーターを制御できるので, Armadillo と同様に, 複雑な制御プログラムもデスクトップ Linux と同様の環境の下で開発できることが期待される.

A.1.1 Debian のインストール

BBB 上に Debian をネットワークインストールする. ネットワークインストールというのは, 最初に起動するインストーラーのみをローカルなデバイス (microSD) から起動し, Debian の本体システムはネットワーク経由でサーバーからダウンロードしながらインストールする方法である. このような方法を採用することにより, 常に最新のヴァージョンがインストールできるというメリットがある.

用意するもの

以下のものを用意する.

¹<http://beagleboard.org/products/beaglebone%20black>

1. BBB
2. Debian PC と Internet 環境
3. USB-Serial ケーブル (6pin, 3.3V)
4. microSD (1GB 以上)
5. USB-LAN アダプター

BBB 自身は LAN インターフェイスを備えているが、インストーラの不都合上、自動的に認識されない場合があるので、USB インターフェイスに USB-LAN アダプタをつないで利用する。

手順

ネットワークインストール手順の流れは以下の通り。

1. Debian PC 上でプロクシ環境変数を設定する。

```
# export http_proxy="http://xxxx.yyyy.zzz:8080/"
```

2. microSD 上に、インストーラーを構築する。

まず、Debian PC 上に、インストーラなどをダウンロードする。

```
# git clone https://github.com/RobertCNelson/netinstall.git  
# cd netinstall
```

つぎに、必要な、ツール類をインストールする。

```
# apt-get install wget dosfstools parted u-boot-tools
```

もし、すでにインストールされていれば、自動的にスルーされる。
microSD を Debian PC に挿入し、dmesg コマンドで、microSD に対応するデバイス名を確認する。

```
# dmesg
...
[463528.914978] sd 6:0:0:0: [sdb] 15693824 512-byte logical blocks: (8.03 GB/7.48 GiB)
[463528.915109] sd 6:0:0:0: [sdb] Cache data unavailable
[463528.915113] sd 6:0:0:0: [sdb] Assuming drive cache: write through
[463528.915355] sd 6:0:0:0: [sdb] Cache data unavailable
[463528.915360] sd 6:0:0:0: [sdb] Assuming drive cache: write through
[463528.916454] sdb:
```

たとえば、上記のようなメッセージが末尾に表示されたとすると、
ブロックデバイスは/dev/sdb であることが分かる。そのブロック
デバイスへ、スクリプトを使ってインストーラを構築する。

```
# ./mk_mmc.sh --mmc /dev/sdb --dtb am335x-boneblack --distro \
wheezy-armhf --serial-mode linux-firmware
```

いくつか質問されるが、問題なければ”y”を答える。

3. BBB と Debian PC を USB-Serial ケーブルでつなぐ（図 A.1 参照）。
ピン配列の向き（リード線の色）に注意する。黒いリード線が
BBB 上の黒丸の方に位置するのが正しい向きである。Debian PC
側は USB ポートに接続する。
4. Debian PC 上で minicom （??参照）を実行する。

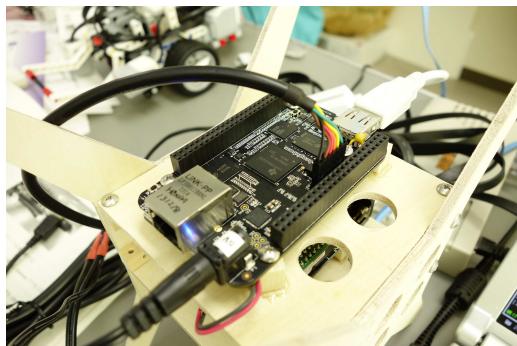


図 A.1: BBB にシリアルケーブルを接続する。

通信設定 : 115200bps, 8bit, Non Parity, ハードウェアフロー制御
OFF

5. BBB の USB ポートに, USB-LAN アダプターを接続し Internet アクセス可能な. ハブに接続する.

6. インストーラーが構築された microSD を BBB に挿入し, 電源を ON.

BBB は miniUSB ポートからの電源供給でも起動できるが, インストール中の電力不足が懸念されるので, 5V の AC アダプターからの電源供給が推奨される.

7. Debian インストーラーに従う.

Language は C を選択する. LAN インターフェイスは, eth0, eth1

のふたつ認識されるが、USB-LAN アダプタは eth1 に対応するのでそちらを選ぶ。DHCP サーバが無い場合には、手動で IP アドレスなど入力する。ディスクのパーティションはインストーラーの推奨する切り方で yes を選択する。

A.1.2 Debian のイメージを直接 microSD にコピー

新たに Debian をインストールしてシステムを構築するのもよいが、既存の Debian イメージファイルから、新たな microSD 上にそっくり元と同じ Debian をコピーすることもできる。パッケージはアップデートされないが、こちらの方が、新規構築の手間は省ける。

Debian のインストールされた microSD を、別の Debian PC に挿入する。dmesg などを使ってデバイスを確認して、例えば /dev/sdb と認識されているとする。dd コマンドを使ってイメージファイルを作る。

```
# dd if=/dev/sdb of=xxxxx.img
```

あらたな microSD カードを Debian PC に挿入し、

```
# dd if=xxxxx.img of=/dev/sdb
```

を実行する。

A.1.3 ネットワークデバイスに付けられる番号

この方法で作った microSD を新たな BBB に挿入して電源を入れると、当然ながら元の BBB の情報がそのまま残っている。特にネットワークデバイス eth0, wlan0 などは、ハードウェア情報に基づいて番号付けされるので、新たなネットワークデバイスをもつ BBB では eth1, wlan1 などと割り振られる。これらの情報は /etc/udev/rules.d/70-persistent-net.rules というファイルの中に自動的に記述されている。その中の eth1, wlan1 の部分を eth0, wlan0 に書き換えて、元の eth0, wlan0 の部分は削除すると、再起動後に、新たなハードウェアが eth0, wlan0 として認識される。

A.1.4 PIC とのシリアル (UART) 通信

PIC と BBB はそれぞれ UART インターフェイスをもっているので、シリアル通信可能である。たとえば、PIC でセンサー値（電圧）を AD コンバートして、その結果を BBB におくるなどする場合に利用できる。BBB 自身も AD コンバートのためのインターフェイスを持っているが、最大電圧が 1.8V であるため、最大電圧が 5V の加速度センサー値などを直接読み取ることができない。一方 16F887 などの PIC は 5V の AD コンバータポートを持っている。なので、それを利用する。

PIC の電源電圧を 5V にすると、UART 出力パルスの最大電圧も 5V となる。ところが、BBB の UART パルス電圧は最大 3.3V であるので、このままだと、直接通信できない。そこで、PIC への入力電圧を 3.3V にすると、UART 出力パルスも 3.3V となる。

3.3V は、 BBB から電源用として出ているので、それを利用できる。また、 PIC 内の C 言語ソースで、電源が 5V を下回った時のリセット機能を無効にする措置が必要である。

BBB のシリアルポート有効化

ケープマネージャの機能を使って、シリアルポートのデバイスファイルを生成する。

```
# echo BB-UART1 > /sys/devices/bone_capemgr.*/slots  
# chmod 666 /dev/tty01
```

PIC(16F877) で実行するプログラム (C 言語)

ヘッダファイルのインクルードなど。

```
//通信速度 38.4kbps(10MHz)

#include<pic14/pic16f887.h>

///#define ADOSC 0x80 //ADC clock is 32TOSC
#define ACQUI 0x18
#define MAX_AN_PIN 14 //14個のAD 変換ポート

//BOR を不許可にする
int __at _CONFIG1 __config1 = _HS_OSC & _WDT_OFF & _PWRTE_ON
    & _LVP_OFF & _MCLRE_OFF & _DEBUG_OFF & _BOR_OFF;
int __at _CONFIG2 __config2 = _WRT_OFF;

//global variable
long int adc[MAX_AN_PIN]; //adc( AN0 ~ AN13 )

//prototype of functions
void set_config(void);
void get_send_adc(void);
void int_str(int send_num);
```

main 関数.

```
void main(void) {
    int i;
    set_config();

    while(1){
        get_send_adc();
        while(!TRMT);
        TXREG = 0x0A; //break line

        // Loop for decreasing sampling rate down to 50Hz
        i=0;
        while(i<2000){i++;}
    }
}
```

シリアルポートに文字列を書き出す関数(int_str())

```
void int_str(int value){
    int mod, c=0;
    char tmp[8];
    //num to ascii code
    do{
        mod = (value%10)+48;
        value /= 10;
        tmp[c] = mod;
        c++;
    }while( value );
    //send character
    do{
        c--;
        while(!TRMT);
        TXREG = tmp[c];
    }while( c>0 );
}
```

受け取った値 (value) を、アスキーコードに変換して、シリアルポートに割り当てられたアドレスに代入している。

AD 変換ポートの値を読み込んで、シリアルポートに送る関数(get_send_adc())は以下のようなものである。

```

void get_send_adc(void) {
    int t;

    //AN0 select
    ADCONO = 0xC3; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[0] = (ADRESH<<8) | ADRESL;
    //AN1 select
    ADCONO = 0x7C; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[1] = (ADRESH<<8) | ADRESL;
    //AN2 select
    ADCONO = 0xCB; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[2] = (ADRESH<<8) | ADRESL;
    //AN3 select
    ADCONO = 0xCF; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[3] = (ADRESH<<8) | ADRESL;
    //AN4 select
    ADCONO = 0xD3; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[4] = (ADRESH<<8) | ADRESL;
    //AN5 select
    ADCONO = 0xD7; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[5] = (ADRESH<<8) | ADRESL;
    //AN6 select
    ADCONO = 0xDB; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[6] = (ADRESH<<8) | ADRESL;
    //AN7 select
    ADCONO = 0xDF; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[7] = (ADRESH<<8) | ADRESL;
    //AN8 select
    ADCONO = 0xE3; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[8] = (ADRESH<<8) | ADRESL;
    //AN9 select
    ADCONO = 0xE7; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[9] = (ADRESH<<8) | ADRESL;
    //AN10 select
    ADCONO = 0xEB; for(t=0; t<ACQUI; t++); GO = 1; while(GO); adc[10] = (ADRESH<<8) | ADRESL;

    while(!TRMT); TXREG = 'a';
    //send an0 value
    while(!TRMT); intToStr(adc[0]); while(!TRMT); TXREG = ';';
    //send an1 value
    while(!TRMT); intToStr(adc[1]); while(!TRMT); TXREG = ';';
    //send an2 value
    while(!TRMT); intToStr(adc[2]); while(!TRMT); TXREG = ';';
    //send an3 value
    while(!TRMT); intToStr(adc[3]); while(!TRMT); TXREG = ';';
    //send an4 value
    while(!TRMT); intToStr(adc[4]); while(!TRMT); TXREG = ';';
    //send an5 value
    while(!TRMT); intToStr(adc[5]); while(!TRMT); TXREG = ';';
    //send an6 value
    while(!TRMT); intToStr(adc[6]); while(!TRMT); TXREG = ';';
    //send an7 value
    while(!TRMT); intToStr(adc[7]); while(!TRMT); TXREG = ';';
    //send an8 value
    while(!TRMT); intToStr(adc[8]); while(!TRMT); TXREG = ';';
    //send an9 value
    while(!TRMT); intToStr(adc[9]); while(!TRMT); TXREG = ';';
    //send an10 value
    while(!TRMT); intToStr(adc[10]); while(!TRMT); TXREG = ';';
}

```

この例では、11個のポートの値を読み込んで、”a”出始まる文字列として、それらの値を”;”で区切ってシリアルポートに書き出し(intToStr())ている。

シリアルポートの設定関数(setConfig)

```
void set_config(void){\n\n    //port setting //1:input, 2:output\n    TRISA = 0xff; //RA is input\n    TRISB = 0xff; //RB is input\n    TRISC6 = 0x00; //RC6 is output //serial port\n    TRISC7 = 0x01; //RC7 is input //serial port\n    TRISD = 0xff; //RD is input\n    TRISE = 0xff; //RE is input\n\n    PORTA = 0x00; //init portA\n    PORTB = 0x00; //init portB\n    PORTC = 0x00; //init portC\n    PORTD = 0x00; //init portD\n    PORTE = 0x00; //init portE\n\n    BRGH = 1;      //highspeed baudrate setting 1:high, 2:low\n    SPBRG = 15;    //38.4k 10MHz\n    //SPBRG = 10;   //115200 20MHz\n    //SPBRG = 129; //9.6k 20MHz\n    SYNC = 0;      //non sync\n    TXEN = 1;      //sending enable\n\n    //RCSTA register\n    SPEN = 1;      //serial port enable\n    RX9 = 0;        //no-parity 9bit receive\n    SREN = 0;       //1byte receive disable\n    CREN = 1;       //continuary receive enable\n    FERR = 0;       //no frame error\n    OERR = 0;       //no over run error\n    RX9D = 0;       //parity\n\n    //A/D setting\n    ADCON1 = 0x00; //AN0-AN7 is no reference\n    //ADCON1 = 0x30; //AN0-AN7 is analog AN2 is Vref- AN3 is Vref+\n    ADCON0 = 0xC0;  //use RC oscillator\n    ADON = 1;       //start adc module\n    ADFM = 1;       //A/D right-justified\n    ANSEL = 0xff;   //AN0-AN7 is analog input.\n    ANSELH = 0xff;  //AN8-AN13 is analog input\n}\n}
```

この例では、10MHzのセラロックを利用して38.kbpsの通信速度でシリアル通信する設定となっている。通信速度などを変更する場合は、適宜SPBRGの値を変更しなければならない。

このプログラムをSDCCでコンパイルしたあと、HEX形式のファイルをPICライターでPIC(16F887)に書き込むと、そのPICは、14個のAD変換ポートに読み込んだ0~5Vの電圧値を数値（デジタル）に変換して、シリアルポートに出力する。

BBB 側の受信プログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <termios.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <math.h>

// **** serial setting *****
#define SERIAL "/dev/tty01" // For Acceleration sensor
#define SERIAL_SPEED B38400
// *****

// prototype of functions
void    init_serial(void);

//serial setting
struct termios oldtio;

//シリアルポートのファイルディスクリプタ
int serial_fd=0;
fd_set readfs;
```

シリアルポート初期化関数

```
void init_serial(void){

    struct termios newtio; // Set serial port

    serial_fd = open(SERIAL, O_RDWR | O_NOCTTY);
    if(serial_fd < 0){
        perror(SERIAL);
        exit(EXIT_FAILURE);
    }

    tcgetattr(serial_fd, &oldtio);
    bzero(&newtio, sizeof(newtio));

    // BAUDRATE: ポーレートの設定. cfsetispeed と cfsetospeed も使用できる.
    // CRTSCTS : 出力のハードウェアフロー制御 (必要な結線が全てされているケー
    // ブルを使う場合のみ. Serial-HOWTO の 7 章を参照のこと)
    // CS8      : 8n1 (8 ビット, ノンパリティ, ストップビット 1)
    // CLOCAL   : ローカル接続, モデム制御なし
    // CREAD    : 受信文字 (receiving characters) を有効にする.
    newtio.c_cflag = SERIAL_SPEED | CS8 | CLOCAL | CREAD;
    //newtio.c_cflag = SERIAL_SPEED | CRTSCTS | CS8 | CLOCAL | CREAD;

    // IGNPAR   : パリティエラーのデータは無視する
    // ICRNL    : CR を NL に対応させる (これを行わないと, 他のコンピュータで
    //             CR を入力しても, 入力が終りにならない)
    // それ以外の設定では, デバイスは raw モードである (他の入力処理は行わない)
    newtio.c_iflag = IGNPAR;

    //Raw mode output
    newtio.c_oflag = 0;

    // canonical input operation
    newtio.c_lflag = 0;

    // setting for waiting characters by read function
    newtio.c_cc[VTIME] = 0; // wait for a value time * 0.1 sec
    newtio.c_cc[VMIN]   = 1;
    // wait until input of character which number is determined by this

    // clear buffer
    tcflush(serial_fd, TCIFLUSH);

    //cfsetspeed(&newtio, SERIAL_SPEED);
    tcsetattr(serial_fd, TCSANOW, &newtio);
}
```

行の先頭から改行まで、1文字ずつシリアルポートを読み取るプログラム例。

```
int main(){
    int l;
    char data;
    char buff[256];

    // In order to start at head of line,
    // searching tail of line.
    read(serial_fd, &data, 1);
    while(data!='\n') {
        read(serial_fd, &data, 1);
    }

    l=0;
    read(serial_fd, &data, 1);
    buff[l]=data;
    while(data!='\n') {
        l++;
        buff[l]=data;
        read(serial_fd, &data, 1);
    }
    l++;
    buff[l]=data;
    l++;
    buff[l]='\0';

}
```

A.1.5 USB 無線 LAN

WNA1100

NETGEAR 社の WNA1100 を USB ポートに挿入する。このデバイス

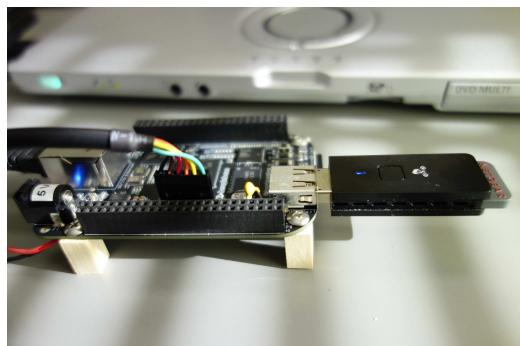


図 A.2: BBB に USB 無線 LAN アダプタとシリアルケーブルを接続する。

を認識するためのモジュールは `ath9k_htc` であるが、ファームウェアのバージョンを適切なものを選ばないと、`wlan0` が起動しない。

http://wireless.kernel.org/download/htc_fw/ から `htc_9271.fw` をダウンロードし `/lib/firmware` のなかにコピーする。その後、`WNA1100` を BBB の USB ポートに挿入すると、自動的にモジュールがロードされる。`lsmod` コマンドで、確認する。

```
root@bbb70:/home/honda# lsmod
Module           Size  Used by
nfsd            233548  2
exportfs          3650  1 nfsd
arc4             1659  2
ath9k_htc        76190  0
ath9k_common     3261  1 ath9k_htc
ath9k_hw         391711 2 ath9k_common,ath9k_htc
ath              16416  3 ath9k_common,ath9k_htc,ath9k_hw
mac80211        505859 1 ath9k_htc
cfg80211        424614 3 ath,mac80211,ath9k_htc
rfkill           18347  1 cfg80211
```

と表示されれば正常認識されている。アドレスの設定等は、付録6に従う。

WLI-UC-GNM

BUFFERLO の小型無線 LAN アダプタ、WLI-UC-GNM は ralink モジュールで自動認識される。ralink モジュールはデフォルトでインストールされているが、ファームウェアを追加でインストールする必要がある。

```
# aptitude install firmware-ralink
# reboot
```

以下に、設定例を示すが、無線 LAN に関する詳細は付録6を参照すること。

```
# aptitude install wireless-tools
# iwconfig wlan0 mode managed
# iwconfig wlan0 essid arm18
# iwconfig wlan0 channel 2
# ifconfig wlan0 172.16.10.77
```

A.1.6 i2c でセンサーの値を読み取る

i2c（アイ・ツー・シー）とは、周辺機器をCPUなどに接続するためのシリアルバス、あるいはその通信方式のことである。組み込みシステムや、携帯電話などでよく使われる。ここでは、BBBとセンサーなどのデバイスをi2cで接続してデータ通信する方法を説明する。

i2c通信においてデータ通信を行うのは、シリアルデータ（SDA）とシリアルクロック（SCL）と呼ばれる2本の通信線である。BBB側と、センサー側にあるSDAピンどうし、SCLピンどうしを、お互いに接続する。UARTシリアル通信の場合には、TXDとRXDをお互いに接続した。i2cの場合には、SDAとSDA、SCLとSCLを接続するので、注意が必要である。

その他に、電源（VDD）とアース（GND）が必要である。つまり、i2c通信を行うためには、BBBとセンサーを、最低4本のリード線で結ぶ必要がある。センサーなどには、それ以外のピンが存在するが、さらに他のi2cデバイスをディージーチェーン（数珠つなぎ）接続する際などに用いるものである。

A.1.7 PIN 配置

BBB は 3 つの i2c インターフェイスを持っているが、1 つは内部で利用されており、ピンが外部に開放されていない。2 つめの i2c インターフェイスとして、P9-19(SCL), P9-20(SDA) がある。この i2c インターフェイスをユーザーは利用できるが、内部で他の機器が接続されている。

3 つ目の i2c インターフェイスは、P9-17(SCL), P9-18(SDA) の 2 ピンが対応している。このインターフェイスには、内部でも他機器などが接続されておらず、すべてのアドレスが利用者に開放されている。これをセンサーなどの接続に利用する。

i2c ピンを含む、BBB の P8 ポートおよび P9 ポートのピン配置を図 A.3 に示した。SCL, SDA の他に、電源 (VDD) とグラウンド (DGND) を i2c センサーと BBB の間で配線する必要がある。VDD は 3.3V と 5V が BBB の P9 のピンから供給されているので、センサーに必要な電圧のピンから配線する。たとえば、MPU9150 というデバイスの電源電圧は 2.4V ~ 3.5V なので、P9-3(or 4) ピンから供給できる。

Device Tree

BBB に接続するデバイスなどの周辺機器は Device Tree とよばれる方法によって制御されている。BBB の起動時にその情報が kernel にわたされていればデバイスファイル (/dev/i2c-1 など) が利用可能になっている。

2 I2C ports

P9			P8		
DGND	1	2	DGND	1	2
VDD_3V3	3	4	VDD_3V3		
VDD_5V	5	6	VDD_5V		
SYS_5V	7	8	SYS_5V		
PWR_BUT	9	10	SYS_RESETN		
GPIO_30	11	12	GPIO_60		
GPIO_31	13	14	GPIO_40		
GPIO_48	15	16	GPIO_51		
I2C1_SCL	17	18	I2C1_SDA		
I2C2_SCL	19	20	I2C2_SDA		
I2C2_SCL	21	22	I2C2_SDA		
GPIO_49	23	24	I2C1_SCL		
GPIO_117	25	26	I2C1_SDA		
GPIO_125	27	28	GPIO_123		
GPIO_121	29	30	GPIO_122		
GPIO_120	31	32	VDD_ADC		
AIN4	33	34	GND_ADC		
AIN6	35	36	AIN5		
AIN2	37	38	AIN3		
AIN0	39	40	AIN1		
GPIO_20	41	42	GPIO_7		
DGND	43	44	DGND		
DGND	45	46	DGND		

図 A.3: BeagleBone Black の P8, P9 ポートピン配列. P9-17, P9-18 を i2c-2 として利用する.

起動時に /dev/i2c-* が有効になっていないときはどうすればいいだろうか? そのときには、次節で説明する Cape Manager をつかって Device Tree にデバイスを追加する方法をつかう.

Cape Manager

Cape Manager というのは、Debian の kernel が起動した後に、i2c ポートを有効化するツールである。kernel 起動後のデフォルト状態で、/dev/i2c-2 などがあるが有効化されていない場合にはこれを用いる。

Cape Manager をつかって /dev/i2c-2 を有効にするためには、

```
# echo BB-I2C1 > /sys/devices/bone_capemgr.*/slots
```

を実行する。つまり、所定のディレクトリにある slots というファイルに BB-I2C1 と書き込む。すると、Cape Manager はこのファイルを監視していて、自動的に i2c インターフェイスを有効化する。

i2cdetect というコマンドをつかうと、i2c デバイスの状態を確認することができる。これを利用するために i2c-tools パッケージをインストールする。

```
# apt-get install i2c-tools
```

一度インストールすれば、起動するたびにインストール必要はない。i2c インターフェイスが利用できる状態になっていれば、

```
# i2cdetect -l
```

を実行すると、

i2c-0 i2c	OMAP I2C adapter	I2C adapter
i2c-1 i2c	OMAP I2C adapter	I2C adapter
i2c-2 i2c	OMAP I2C adapter	I2C adapter

と表示される。3つの i2c インターフェイスが存在することがわかる。

さらに、i2cdetect コマンドをつかって、

```
# i2cdetect -r -y 2
```

を実行すると、i2c-2に接続されているデバイスのアドレスが表示される。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
10:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
20:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
30:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
40:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
50:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
60:	--	--	--	--	--	--	--	--	68	--	--	--	--	--	--	--
70:	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

この例では、0x68にデバイスが存在することがわかる。(MPU9150のi2cアドレスは0x68か0x69である)

デフォルトでは、管理者(root)だけにしかデバイスファイル/dev/i2c-*に読み書きの許可が与えられていない。そのため、i2dデバイスファイルをアクセスするためには、root権限でプログラムを実行する必要がある。

デバイスファイルをすべてのユーザーがアクセスできる許可を与えるためには、

```
# chmod a+rwx /dev/i2c-2
```

を実行する。これで、/dev/i2c-2に接続されたi2cデバイスをroot以外のユーザー権限プログラムから読み書きする準備が整った。

i2cデバイスへのデータの読み書き

MPU9150というセンサーから加速度値を読み取る場合を例に、i2cデバイスのアクセス(データ読み書き)の方法を説明する。

先述のように、 BBB の i2c 用の PIN にセンサーを配線すると、 Cape Manager はそれに対応してデバイスファイルを有効にする。i2c という通信規格は、ひとつの PIN セットに複数の i2c デバイスを接続できる仕様となっている。したがって、デバイスファイルを指定しても、ひとつのセンサーを指定したことにはならない。それぞれのデバイスを区別するのが、アドレスである。上の例では、0x68 であった。接続されている i2c デバイスが 1 つであっても、アドレスは指定しなくてはならない。

さらに、i2c デバイスは、レジスタというデータ格納領域をもっている。MPU9150 の例でいうと、加速度センサー値やジャイロセンサー値などが、それぞれ別々のレジスタに格納されている。つまり、データを読み書きするためには、そのレジスタも指定する必要がある。

まとめると、ひとつのデータを i2c デバイスから読み取るためには、デバイスファイル、アドレスおよびレジスタの 3 つが適切に指定される必要がある。

デバイスファイルを開くためには、ファイルディスクリプタでそれを開く。C 言語によってこれを行う例を以下に示す。

```
#include <fcntl.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>

...
#define I2CDEVICE "/dev/i2c-2"
...
int fd;
fd=open(I2CDEVICE, O_RDWR);
```

i2c-dev.h と ioctl.h は fd を開くためには必ずしも必要ないが、以下の i2c データのアクセスの中で必要となるので、まとめて示した。fd はア

クセスが終了したら close(fd) を実行して閉じなければならない。

つぎに、アクセスしたいデバイスのアドレスは、ioctl 関数を用いて指定する。

```
#define ADD 0x68  
...  
ioctl(fd, I2C_SLAVE, ADD);
```

MPU9150 はつねに i2c 通信において slave モードで用いる。これにより BBB 側が master モードとなる。これを一度だけ実行することにより、以下では fd に read, write する際に、アドレス 0x68 のデバイスがアクセスされる。逆に、1つの i2c インターフェイス（ファイルディスクリプタ）に接続された別のデバイスをアクセスしたい場合には、ioctl 関数でそのアドレスを再設定すればよいことになる。デバイスのアドレスは i2cdetect コマンドで確認することもできるが、デバイスの仕様書などにも記述されているので確認してみるとよい。

ファイルディスクリプタ fd で i2c インターフェイスの open, ioctl 関数で、そのアドレスの指定ができた。最後にデータの読み書きである。read および write 関数をつかう。MPU9150 の場合、レジスタ 0x75 に、MPU9150 自身のアドレス 0x68 が格納されているので、そのレジスタを読み込む例を以下に示す。レジスタを指定してそこからデータを読み込むには、いったんレジスタアドレスそのものを write する。その後 read 関数を実行すると、そのレジスタの値が読み込まれる。

```
unsigned char wbuf[2];
unsigned char rbuf[14];
...
wbuf[0]=0x75;
write(fd,wbuf,1); // 1バイト分書き込み（レジスタを指定）
read(fd,rbuf,1); // 1バイト分読み込み（レジスタから読み込み）
printf("WhoAmI=0x%02x\n",rbuf[0]);
```

WhoAmI=0x68 と標準出力されれば正常に接続されていることが確認できる。配列変数はサイズが 1 バイトの `unsigned char` 型を用いる。またそのサイズは、ここでは 1 以上であれば十分であるが、後の使用を考えて、2, 1 4 としている。

つぎにセンサーデバイスを初期化する。MPU9150 の場合、レジスタ 0x6b に 0x00 を、またレジスタ 0x37 に 0x02 を書き込むことによってリセットされセンサー値がレジスタにセットされ始めるので、以下を行う必要がある。

```
wbuf[0]=0x6b; wbuf[1]=0x00; write(fd,wbuf,2);
wbuf[0]=0x37; wbuf[1]=0x02; write(fd,wbuf,2);
```

ここで示したように、書き込みはレジスタの指定と、その値の 2 バイトをいちどに `write` できる。

加速度センサー値はレジスタ 0x3b から 2 バイトづつ x,y,z の値が格納されているので、それを読み出す例を以下に示す。

```
wbuf[0]=0x3b; write(fd,wbuf,1); read(fd,rbuf,6);
for(i=0;i<6;i++){ printf("0x%02x ",rbuf[i]); } printf("\n");
```

ジャイロセンサーの値は、この後ろの 6 バイトに格納されているので、

一気に読みだすのであれば、`read(bd,rbuf,12)`を実行すれば良い。

それぞれの値が上位 1 バイト、下位 1 バイトの順で格納されているので、

```
int ax;  
...  
ax=rbuf[0]*256+rbuf[1];  
if(ax>32768){ax=65536-ax;}else{ax=-ax;}
```

というように、角度に対応する値に変換する。

`0x3b` 以下の 14 バイトのレジスタは随時センサー値が更新格納されるので、再び読みこむことによって現在のセンサー値を知ることが出来る。

バッファの `fsync`

i2c デバイスのレジスタに値を書き込んで、その直後に同じレジスタを読みこめば書き込んだ値がそのまま読み取られるはずである。たとえば、加速度センサー値のノイズを平滑化する場合など、ローパスフィルターのレベルをレジスタに書き込んで指定することが出来る。指定したフィルターの強さを確認するためには、その値を再度読みこめよい。

しかし、それを実行しても異なる値が得られる場合がある。つまり、正常にレジスタに値が書き込まれていないか、あるいは書き込みが終了していないのである。

これは `write` 関数による書き込みが、その実行と同時に起こるのではなく、Linux kernel などのバッファに一旦蓄えられ、kernel の判断で実

際のレジスタに値が書き込まれることが原因である。この様にバッファを介する方が、即座に書き込みを行うよりも、より効率的に読み書きを行うためである。

即座に書き込みを反映させたい場合には `fsync`(ファイル記述子) を実行する。これを実行することにより、kernel のバッファ内のデータがデバイスのレジスタに実際に `write` される。

A.1.8 超音波センサー値を i2c で読み取る

`srf02` という超音波センサーを用いると、16cm ~ 600cm の範囲で対象物（壁など）からの距離を測定することができる。`srf02` からはシリアル通信と `i2c` 通信によって距離データを読み出すことができるが、ここでは `i2c` を使った方法を紹介する。

`i2c` では一本のバス上に 7bit のアドレスで区別されるセンサーデバイスを接続する。つまり、理論上 $2^7 = 128$ 個のデバイスが 1 個の `i2c` ポートに接続可能である。シリアル通信を使うばあい、接続できるセンサーの数はボードコンピュータのシリアルポートの数に制限されてしまう。`Beaglebone Black` の場合シリアルポートの数は 6 個である。多くのセンサーを接続したい場合には `i2c` 通信のほうが、大きな可能性を持っているといえる。

`i2c` プロトコルにおける、アドレスは 8bit で表現されているが、最下位のビットは書き込みか読み込みを区別するために利用されているので、実際のアドレスとして機能するのは、7bit である。`srf02` のデータシートを読むと、デフォルトのアドレスは $(E0)_{16} = (1110\ 0000)_2$ と記述され

ており 8bit 表記されていることが分かる。srf02 ではアドレスは $(E0)_{16}$, $(E2)_{16}, \dots, (FE)_{16}$ まで 16 個のアドレスを選択することができる、1 つの i2c ポートに 16 個まで srf02 を接続可能である。これらのアドレスはすべて偶数である。2 進数で表した場合、最下位のビットは常に 0 であることに注意が必要である。つまり、実際にアドレスとして機能しているのは上位の 7 bit ということである。

linux shell 上で, i2cdetect -r -y コマンドを実行すると、アドレス $(E0)_{16}$ ではなく、 $(70)_{16}$ が表示される。 $(70)_{16} = (1110\ 000)_2$ であるので、8bit アドレスの使われていない最下位ビットが省略されていることが分かる。linux 内では i2c デバイスのアドレスは 7bit で管理されているようである。このように、マニュアルやデータシートに記載されているアドレスと、linux 上で認識されるアドレスが異なるので、混乱を来さないように注意が必要である。たとえば、 $(FE)_{16}$ とアドレスを変更した場合、i2cdetect コマンドや C 言語プログラムからそのデバイスをアクセスする場合、それを 2 で割った $(7F)_{16}$ というアドレスを使う必要がある。

もうひとつ、linux shell 上で i2cdetect をする場合に注意する必要がある。デフォルトでは i2cdetect コマンドは $(03)_{16}, \dots, (77)_{16}$ までしか表示しない。 $(00)_{16}$ から $(FF)_{16}$ まですべてのアドレスを認識表示するために i2cdetect -r -y -a と -a オプションを付加する必要がある。

SRF02 のアドレス変更

SRF02 のアドレスを変更するためには、コマンドバッファ $(00)_{16}$ に $(A0)_{16}, (AA)_{16}, (A5)_{16}$ につづけて、変更後のアドレスを書き込む。

C 言語プログラムを用いて実際にアドレス変更した例を以下に示す。書き込む先のアドレスは Linux の ioctl を介して行うので、7 ビット表記を用いる。これに対して、書き込むアドレスは、srf02 が保持する値なので 8 ビット表記でなければならないことに注意する必要がある。

```
1 #include <stdio.h>
2 #include <linux/i2c-dev.h>
3 #include <fcntl.h>
4 #include <sys/ioctl.h>
5
6 #define SRF02_DEV "/dev/i2c-1"
7 #define SRF02_ADD 0x71
8 #define CHANGE_TO_ADD 0xfe
9
10
11 int main(){
12     int fd;
13     unsigned char wbuf[3];
14
15     fd=open(SRF02_DEV, O_RDWR);
16
17     // 0x00に0xA0, 0xAA, 0xA5をつづけて書き込む
18     ioctl(fd, I2C_SLAVE, SRF02_ADD);
19     wbuf[0]=0x00; wbuf[1]=0xA0; write(fd,wbuf,2);
20
21     ioctl(fd, I2C_SLAVE, SRF02_ADD);
22     wbuf[0]=0x00; wbuf[1]=0xAA; write(fd,wbuf,2);
23
24     ioctl(fd, I2C_SLAVE, SRF02_ADD);
25     wbuf[0]=0x00; wbuf[1]=0xA5; write(fd,wbuf,2);
26
27     // 0X00に新しいアドレス値を書き込む
28     ioctl(fd, I2C_SLAVE, SRF02_ADD);
29     wbuf[0]=0x00; wbuf[1]=CHANGE_TO_ADD; write(fd,wbuf,2);
30
31     close(fd);
32 }
```

この例では、アドレスが 7 ビット表記(71_{16})から 8 ビット表記(fe_{16})に変更されている。

A.1.9 BBB の電源

BBB の電源は 5V の AC アダプターから、あるいは、miniUSB ポートから供給できる。長時間、プログラムのデバッグなどを行う場合には、これらの電源をもちいるとよい。いっぽう、走行ロボットや飛行ロボットでは、これらの電源が利用できないので、バッテリーからレギュレータ (DC-DC コンバータ) を通して 5V 電源を確保する必要がある。



図 A.4: BeagleBone Black の 5V 電源をレギュレータなどからとるための配線

BBB 側の配線例を図 A.4 に示した。赤い線が正 (+) 側、黒い線が負 (-) 側のリード線である。BBB も含めた直流電源を用いる電子機器は、電源の正負を間違うと破損するので、注意が必要である。

A.1.10 PWM でモーターを制御する

感覚運動写像などにおいて、運動を調節するためにはモーターの回転数を出力値に応じて制御しなければならない。モーターの制御は BBB の PWM 機能を用いて行うことが可能である。PWM(Pulse Width Modulation)とは、電圧パルスの電圧の高さを変化させるのではなく、パルスの幅を大きくしたり小さくしたりすることによってモーターの回転数を変化させる方法一般を指す。

ブラシレスモーターの場合、モーターは専用のアンプによって回転数が制御される。アンプに対して PWM パルスを BBB から出力することによって、ブラシレスモーターの回転数を制御できる。

ここでは BBB の PWM 機能を使って、ブラシレスモーターの回転数を制御する方法を説明する [15]。BBB の PWM 機能は、Cape Manager に実装されたデバイスドライバを通じて実現される。具体的には、PWM 制御用の仮想ファイルに値を書き込むことによってパルス幅などを変化させることができる。

仮想ファイルへの値の書き込みは、Linux の echo コマンドを使って行うことも可能であるが、ここでは感覚運動写像などのプログラムから PWM 制御を行うことを念頭に、C 言語でその仮想ファイルに値を書き込む方法を説明する。

BBB による PWM の概略

仮想デバイスファイルへの書き込みなどの機能は参考文献 [15] に示されたライブラリ libBBB を用いておこなう。そのため、まず libBBB をインストールしなければならない。この作業は BBB 上の Debian で一度だけ行えば良い。Debian の起動のたびに行う必要はない。

つぎに、Cape Manager を通じて PWM ドライバを有効にする必要がある。これは Debian が再起動されると無効になるので、起動のたびに行う必要がある。

さいごに仮想ファイルへの値の書き込みによってパルスを制御する。ここでは前述のように libBBB を利用して C 言語で PWM 制御を行う例を示す。

libBBB のインストール

libBBB ライブラリをインストールするためには、make コマンドと gcc をもちいるので、まだインストールされていない場合には、インストールする。

```
# aptitude install make  
# aptitude install gcc-4.7  
# ln -s /usr/bin/gcc-4.7 /usr/bin/gcc
```

次に libBBB をインストールする。libBBB は参考文献 [15] にあるものを自分で入力してもよいが、<http://www.rutles.net/download/393/index.html> からダウンロードすることも出来る。make は libBBB というディレクト

り内でおこなう。ルートディレクトリ (/) 下にそれを作ると、Debian のディレクトリ構造が煩雑化するので、作業ディレクトリを作成し、その下で make を行う。

```
# mkdir /home/hogehoge/work
# cd /home/hogehoge/work
... libBBB.tar.gz のダウンロード
# tar xzpvf libBBB.tar.gz
# cd libBBB
# make clean
# make
# make install
```

Cape Manager による仮想デバイスファイルの生成

以下のコマンドを実行する。

```
# echo am33xx_pwm > /sys/devices/bone_capemgr.*/slots
```

```
# gcc pwm_test.c -lBBB -o pwm_test
```


付録B シリアルコンソール

ボードコンピュータのシリアルポートと、Debian PC の USB ポートを接続して、コンソールを利用する方法をここに説明する。ネットワークなど、コンソール以外でログインできない場合や、起動ログを詳しく見たい場合など、シリアルコンソールを利用する必要がある。

B.1 シリアルポートの読み書き許可

Debian PC の USB ポートに USB-シリアル変換ケーブルを接続する。シリアルポートがブロックデバイスファイル/dev/ttUSB0として認識されている場合、

```
# chmod 666 /dev/ttUSB0
```

を実行することによって、一般ユーザー権限で、シリアルポートを利用可能になる。

B.2 minicom

利用するシリアルポートや通信設定を最初1度だけ行う。この設定は、後ほど変更したり、minicomを起動するたびに行なうこともできるが、よく使う設定は、最初に行って、保存しておくと便利である。

```
# minicom -s
```



図B.1: minicom -s を実行した画面

矢印キーを使って、「シリアルポート」を選択するとシリアルポートの通信速度などが表示されるので、通信相手のボードコンピュータにおける設定と一致していることを確認する。

ハードウェアフロー制御をしていない（多くの場合）場合には”F”を入力して「いいえ」とする。リターンを入力し、最初の選択画面にもどる。これらの設定が後のminicom起動時にも有効になるように、保存



図 B.2: シリアルポートを選択した画面



図 B.3: "F"を入力し、ハードウェアフロー制御「いいえ」を選択

を選択する。

設定が正常に行われると、ボードコンピュータのログインプロンプトが表示される。

minicom は CTRL-A X と入力することによっても終了することができる。



図 B.4: 「”dfu”に設定を保存」を選択し、設定を保存する。

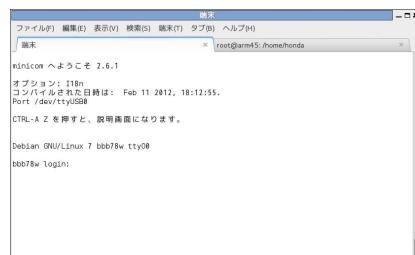


図 B.5: ボードコンピュータのログインプロンプトが表示される。

付録C 起動時に実行される シェルスクリプト

自分で作成したスクリプトを、 Debian マシンの電源投入時や、 reboot 時に自動的に実行されるようにする方法をここに示します。 まず、 シェルスクリプトをつくり、 insserv あるいは update-rc.d コマンドを実行する必要があります。

C.1 insserv

Debian 6 以降のバージョンでは、 insserv コマンドを用います。 Debian 6 以前のバージョンについては、 update-rc.d を用います。

実行したスクリプト (hoge.sh) を作り /etc/init.d に置いて、 insserv コマンドを実行します。

```
# mv hoge.sh /etc/init.d  
# insserv hoge.sh
```

find コマンドを使って、 登録されたスクリプトを表示してみると。

```
# find /etc/ -name "*hoge.sh"
```

ランレベルに合わせて登録された結果が表示されます。

```
/etc/rc0.d/K01hoge.sh  
/etc/rc5.d/S16hoge.sh  
/etc/rc6.d/K01hoge.sh  
/etc/rc2.d/S16hoge.sh  
/etc/init.d/hoge.sh
```

ランレベルに応じて、複数のディレクトリにリンクが作成されます。

S ではじまるリンクが起動時に実行され、K ではじまるものが Debian 終了時に実行されます。終了時に特に実行する必要ない場合は、K ではじまるリンクを削除すれば、実行されません。

リンク名の先頭に”S16”などと記号と数字が付加されます。この数字が小さいものから自動的に実行されます。最後に実行したいスクリプトは、この数字をいちばん大きいものに変更しておきます。

たとえば、無線LANの iwconfig などのスクリプトは、起動の一番最後に実行したほうが確実性が高いようです。

C.2 update-rc.d

Debian 6 以前のバージョンについては、update-rc.d を用います。実行したスクリプト (hoge.sh) を作り /etc/init.d に置きます。その後、update-rc.d コマンドを実行します。

```
# update-rc.d hostname.sh defaults
```


付録D 分散バージョン管理システム gitlab

gitlab とは、ソースコードなどを開発する際のバージョン管理を WEB 上 (<https://gitlab.com/>) で行うサービスである。従来の中央管理型を始めとした様々ななかたちのワークフローを実現できる。それ故に、理解のしやすい側面があるが、ここでは基本的な利用の仕方を述べる。

gitlab では、基本的な機能は無料で自由に使える。大まかな流れは、

1. ユーザーの登録
2. プロジェクトの作成
3. git add, commit, push
4. git pull

である。この他にも、もちろん多くの機能があるが、ここでは割愛する。ユーザー登録は、最初の sign in の際だけ必要である。ここでは、ローカルな Debian 上にあるプログラムなどを gitlab.com のプロジェクトとして管理する方法を示す。

ユーザーの登録

<https://gitlab.com> をアクセスすると右上に Sign in ボタンが表示されるので、クリックする。初めての場合は Sign up の項目に必要事項を入力し、gitlab への登録をおこなう。以降は同じユーザー名で Sign in できる。

セキュア通信を行うために鍵の生成と登録を行う。ローカル Debian で以下を実行し公開鍵を表示する。

```
$ ssh-keygen -t rsa -C "$your_email"  
$ cat ~/.ssh/id_rsa.pub
```

gitlab の Dashboard 表示の状態で、右上にある Profile setting のアイコン  をクリックする。左端の列に  が現れるので、クリックする。右上の  をクリックすると、title と key の入力画面になるので、key の枠に先ほど cat したキーをコピペーストする。ssh- ではじまり、ユーザー名で終わるすべての部分をペーストする必要がある。title も入力する。以上で gitlab とファイルをやり取りするための準備が完了した。

プロジェクトの作成

Dashboard 右上の  をクリックしてプロジェクトをつくる。あたらしいプロジェクトが作られると以後に必要な作業が gitlab 上に表示されるので、それにしたがう。

```

Git global setup

git config --global user.name "ユーザー名"
git config --global user.email "メールアドレス"

Create a new repository

mkdir プロジェクト名
cd プロジェクト名
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add origin https://gitlab.com:ユーザー名/プロジェクト名.git

```

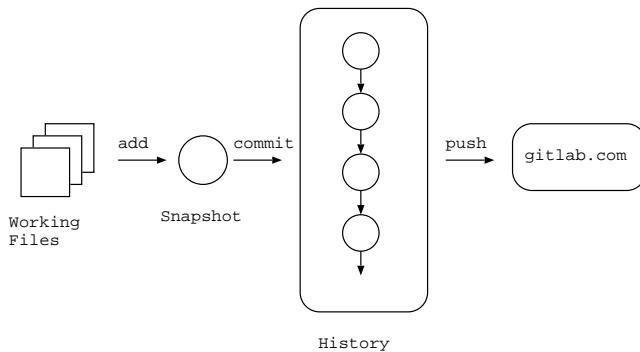


図 D.1: gitlab ワークフローの概略

一連の作業の概略を図 D.1 に示した。gitlab では Working files をいきなり gitlab.com に送ることはしない。作業履歴を History の中に保存し

てから push を行う。git commit では、かならずコメントをつける必要がある。つまり、編集作業の内容を記録する必要がある。さらに、gitlab では History に commit するまえの段階に snapshot が存在する。snapshot が存在することにより、作業内容ごとに add をして、その作業に対してコメントを付して履歴を残すことが可能となる。逆に言えば、作業内容を気にせずに working file の編集を行うことが出来る。

git add, commit, push

ローカル Debian 上でファイルを編集などした後、gitlab にプロジェクトを登録するには以下を行う。

```
$ cd プロジェクト名  
$ git add -A  
$ git commit -m "コメント"  
$ git push origin master
```

git pull

gitlab.com 上で行った編集をローカル Debian に反映するには、git pull を実行する。

```
$ cd プロジェクト名  
$ git pull origin master
```

gitlab.com 上で編集が行われた際に、git pull を行わずに git push を行う

と、履歴に齟齬が生じたとみなされて push が拒否される。その際には、一旦 git pull を行なってから git push を行う。

関連図書

- [1] 「岩波講座ロボット学4 ロボットインテリジェンス」浅田 稔, 國吉 康夫 著, 岩波書店
- [2] 「オイラーの贈物（人類の至宝 $e^{i\pi} = -1$ を学ぶ）」吉田 武 著, 東海大学出版会
- [3] 「系統的プログラミング／入門」N. Wirth 著, 野下浩平, 篠 捷彦, 武市正人 共訳, 近代科学社
- [4] 「プログラム書法」B.W. Kernighan and P.J. Plauger 著, 木村 泉訳, 共立出版
- [5] 「行列の固有値」F. シャトラン著, 伊理正夫, 伊理由美訳, シュプリンガー・フェアラーク東京
- [6] 「強化学習」Richard S. Sutton and Andrew G. Barto 著, 三上 貞芳, 皆川 雅章 共訳, 森北出版
- [7] 「力学」ランダウ＝リフシツ 理論物理学教程, 広重徹, 水戸巖訳, 東京図書

- [8] 「回転翼飛行ロボットの時間遅れ運動制御」橋本理寛, 本田泰, 第18回交通流のシミュレーションシンポジウム(2012)論文集, 33–36
- [9] 「回転翼飛行ロボットの時間遅れ運動制御シミュレーション」佐藤宏樹, 橋本理寛, 本田泰. 第19回交通流のシミュレーションシンポジウム(2013)論文集, 45–48
- [10] 「時間遅れの繰り込みによる感覚行動系の安定性」本田泰, 第19回交通流のシミュレーションシンポジウム(2013)論文集, 53–56
- [11] 「数値計算」川上一郎著, 岩波書店
- [12] 「物理のための数学」和達三樹著, 岩波書店
- [13] 「UNIX ワークステーションによる科学技術計算ハンドブック」戸川隼人著, サイエンス社
- [14] 「ニューメリカルレシピ・イン・シー(C言語による数値計算のレシピ)」ウィリアム・H・プレス／丹慶勝市著, 技術評論社
- [15] 「BeagleBone Blackで遊ぼう！」米田聰, Rutles

索引

- .exrc, 130
- .gvimrc, 130
- /etc/hosts, 136
- /etc/network/interfaces, 135
- /etc/resolv.conf, 138
- apt-cache, 127
- apt-cdrom, 126
- apt-get, 126
- BeagleBone Black, 208
- capemanager, 228
- dd, 38, 212
- dmesg, 210
- dpkg, 125
- EPS 出力, 191
- ev3dev, 37
- exports, 139
- fit, 205
- fsync, 235
- gnuplot, 179
- gvim, 133
- i2c, 226
- ifconfig, 135, 145
- insserv, 247
- iwconfig, 144
- iwlist, 146
- kernel, 141
- load, 190
- logscale, 190
- make-kpkg, 141

- minicom, 244
- mount, 139
- NFS, 139
- PID 制御, 94
- ping, 137
- plot, 183
- proxy, 138
- replot, 191
- route, 147, 150
- SRF02, 236
- terminal, 188
- title, 188
- unzip, 38
- update-rc.d, 248
- vi, 133
- xlabel, 185
- xrange, 186
- ylabel, 185
- yrange, 186
- 感覚運動写像, 94
- 感覚行動写像, 26
- 行列の指數関数, 69
- ケーブマネージャ, 214
- 興奮性結合, 28
- 身体性, 20, 26
- 遷移行列, 85
- 相空間, 80
- 走性, 28
- 対数スケール, 190
- 単位行列, 85
- フィボナッチ遷移行列, 107
- ブロックデバイス, 210

Y. Honda
2018 1/24