CNC 制御プログラム「Grbl」の解析

S. Tajima

Ver 1.1 2020 年 3 月

Contents

前書き

CNC フライス盤の制御には、いわゆる G コードが用いられている。このコード自体は簡単なキャラクターコードで、これをアプリケーションソフトによりパージングし、コードブロックごとにフライス盤に送信、インタープリターとして 3 次元切削動作を制御する。

G コードの構成自体は直接的で、それ程困難ではないが、設計自体が古く、コード体系としては、命令の直交性の悪さや、パラメータの解釈に対する統一がとれていないなどの欠点がある。しかし、これが CNC マシンの標準制御コードであり、これに従って実装するのは必然であろう。

しかも、非常に幸いな事に、この G コードによるフリー制御ソフトウエア一式が Grbl として公開されている。一方、3 次元制御を必要とする機器はフライス盤以外にも、3 次元プロッター、レーザーカッター等があり、かつその形式や規模など多種多様である。

Grbl はこの多種多様な機器に、できるだけ対応できるように設計されている。そのためコードの分量も膨大であり、ファイル数も非常に多く、これを読み解いてターゲット機器に応用するのは、実はそれほど容易ではない。

また、Grbl は、Arduino ベースのマイクロプロセッサに搭載する事を前提にし、G コードの解釈からステッピングモータの制御、スピンドルモータの回転制御から、装置の安全ドアの管理まで、極めて行き届いたソフトウエア群となっている。

この文書では、実際に製作したフライス盤に応用する事を前提として、Grbl のソフトウエア群の動作を読み解こうとしたものである。

ただし、必要最低限な機能を持つフライス盤に応用するため、ソフトウエア群から不要なものを省き、容易に理解し、実装するのを目的とした。

そのため、Grbl ソフトウエア群から以下のものは省いた。

＊ G コードエラーチェック部分：これは切削動作時にエラーを検出して動作が停止するのは望ましくなく、G コードのプリプロセスとして実装した。
＊プローブ関係：フライス盤のスピンドルにプローブを装着して、ワーク（材料）の位置計測が可能であるが、この機能は省いた。
＊フライス盤の各種機械的パラメータ設定関数等。これはターゲットにより決められていると仮定。
＊ Arduino ポート設定や、これらによる直接的なモータ制御関係。(Arduino は使わないため。）
＊ AMASS 関係。（ただし、機能については解説を加えてある。）
＊切削オイル、オイルミスト制御。
＊ COREXY 制御（X,Y ステッピングモータのやや特殊な、しかし凝った制御。）

一方、G コードそのものの解釈、あるいは解析無しには、CNC マシンに対して実装する事は不可能で、必要な限りの G コード解説も加えた。この過程で、G コードそのものの

性格もある程度見えてきたが、これについては主観的意見である。

以下の解説の中には思い違いや解釈間違いがあるかも知れない。情報源はあくまで Grbl ソースコードである [1]。

2019 年 7 月　　S. Tajima

[1] バグは付いている様である。

Chap1　概要

1.1　Grbl

Grbl はオープンソースの CNC マシンコントローラで、CNC フライス盤、レーザカッター等、3 次元の工作機器制御全般に対応したソフトウエアである。制御対象が多様で、ステッピングモータの制御についても速度の最適化等も含み、入力 G コードのパージング、エラーチェック、直線、円弧補間アルゴリズム（Bresenham アルゴリズム）および実際にモータを制御する部分、PC との通信制御等、必要と思われるものを網羅している。

逆に、この多量なファイル群から必要なソースコードを見つけ出し、読み解くのは容易では無い。特に、全体のソフトウエアアーキテクチャの解説がある訳でもなく、ソースコードの機能からどのような構造を持っているかを見出す必要がある。

Grbl が想定するハードウエアが用意でき、ソフトウエアを変更せずにコンパイルして動作させる場合でさえ、そうそう簡単には行きそうも無い。つまり、ソフトウエアの中身を理解せずに、「使用させていただく」という方針が成り立たないと考えていた方がよかろう。

この様な想定の元、まず本章では全体のソフトウエアアーキテクチャについて述べる。

1.2　Grbl のアーキテクチャ

Grbl は AVR Atmega328P(16MHz) ベースの Arduino で動作する様に設計されている。しかし、多機能でソースコード量も多く、Ver 1.1g(2018/11/12) では収納しきれないとの記述があり、ARM ベースプロセッサを計画している模様である。

従って、プロセッサの資源の有効活用がアーキテクチャの考え方を決定していると思われ、EEPROM の多用、ポート切り替え等のハードウエア対応の工夫、そしてソフトウエアアーキテクチャに反映されている。

特に、ソフトウエアアーキテクチャでは、種々の処理を非同期にし、それらが、プロセッサの空き時間を有効活用し、全体でのスループット向上を図っている。この関係を理解しないと Grbl 処理が分かり難く、以下、基本的非同期処理について記述する。

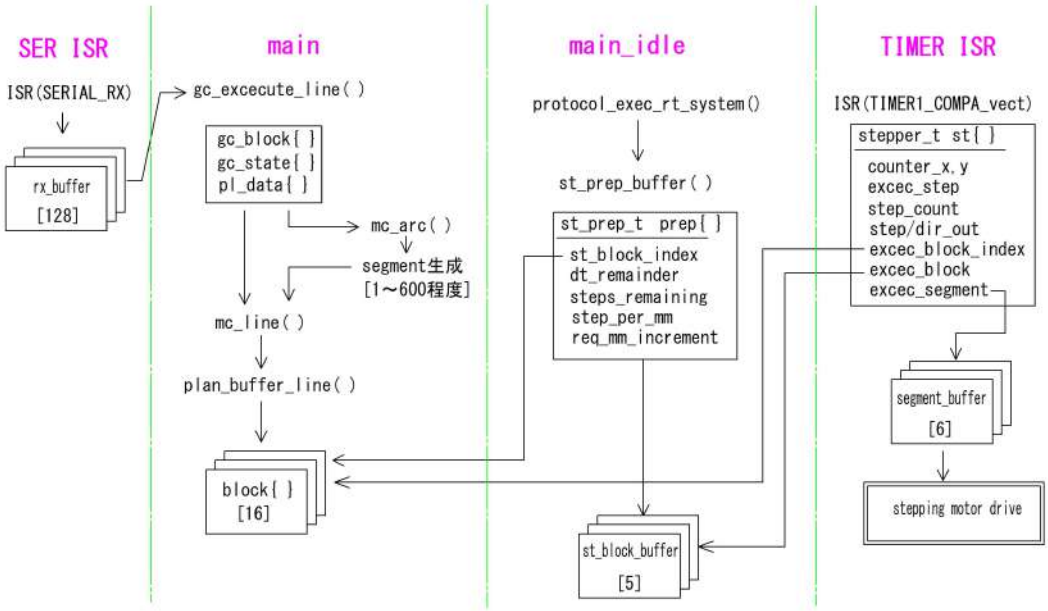

Figure 1.1: Grbl の全体構造

1.3 OS 無しのマルチスレッド的動作

元々のターゲッププロセッサである Arduino は OS 無しの 8 ビット AVR プロセッサを使用している [1]。

図 1.1 に全体のプログラム構造を示す。全体の処理としては、データ入力処理用シリアル通信割込み、ステッピングモータ駆動用タイマー割込み、main( ) プログラム、そしてプログラムの空き時間を使って処理する main_idle プログラムに分けられる。（主として）2 つの割込み処理プログラムと、その他のプログラムで、処理を非同期としている。この非同期プログラム間は構造体を用いたリングバッファにより情報を授受する。

この構造でマルチスレッド的処理を実現しているが、構造体やそのメンバーおよびリングバッファとプログラム処理ブロックの関係が分かり難い。概略、各非同期ブロックそれぞれに処理データのための構造体を用意し、非同期ブロック間のデータ授受用には必要データだけを収納する、小さな構造体を用意し、これのためのリングバッファを使う。

つまり、リングバッファをスレッド間のデータ引き渡しに利用する、マルチスレッドもどき処理を割込みルーチンを利用して実現、OS 無しマルチスレッド的動作となっている。

構造体やリングバッファの使用状況は概略次の通り。

* シリアルデータ通信：バイト単位のリングバッファ、rx_buffer[128]

[1] Arduino のプログラムは sketch と呼ばれて、入門者向けに容易にソフトウエアが実装できるようになっているが、Grbl は sketch とは関係なく、AVR のソフトである。

* G コード処理部：gc_block{}, gc_state{}, pl_data{}
* main プログラムと、ステッピングモータ駆動プログラムとのデータ授受：block{}[16]
* main_idle：st_prep_buffer{}、st_block_buffer{}[5]
* ステッピングモータ駆動：segment_buffer{}[6]

1.3.1 リングバッファ

【リングバッファの構造】

今後、リングバッファを多用するので、その構造と制御関数の役割をまとめて置く。

図 1.2 にリングバッファの構造を示す。リングバッファは書き込みと読み出しを非同期とするのに向いている。各バッファは配列番号は持つが、これはオーバーフロー時に 0 に戻る様に管理され、使用時に意識する必要は無い。

リングバッファの管理は、読み出し、書き込みの観点から考えれば良く、書き込みの先頭となるものが、_head であり、読み出す情報（あるいは読み出し中）は_tail に位置する。従って、情報として最後に書き込まれたものは next_buf_head にある。_head は次に書き込む情報をストアするスペースなので、その内容が上書き可能となる様にポインターが管理されている。実際の内容を気にせず、書き込み、読み出しが管理できる点がリングバッファの特徴である。

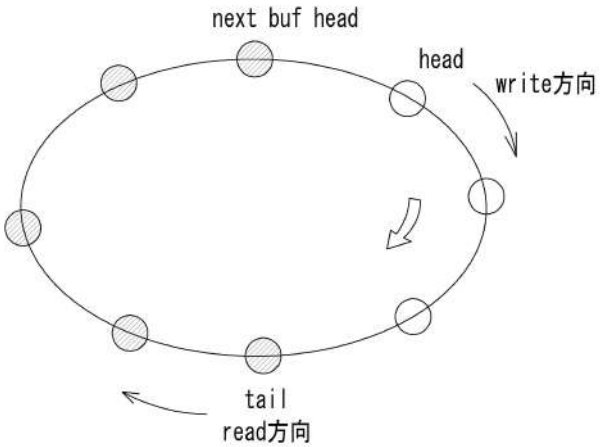

Figure 1.2: リングバッファ

リングバッファについては、_head、_tail に注意して読み書きする。
書き込み側：_tail の手前まで書き込み可能。
読み出し側：next_buf_head まで読み出し可能。

【リングバッファへの書き込み】

planner.c で用いられる plan_buffer[ ] を例として、実際の操作を示す。
主たる制御関数は plan_buffer_line( ) で

*target ; 絶対座標値 [mm 単位] データへのポインター
*pl_data ; 移動速度、行番号へのポインター

を引数として、プラン構造体 block{} を生成する。

```
plan_block_t *block = &block_buffer[block_buffer_head];
```

block_buffer_head は 0 に初期化されているので、リングバッファの [0] から始める事になる。リングバッファに収められる構造体をリセットし、

```
memset(block,0,sizeof(plan_block_t));
```

この中身を順次整備していく。

```
// ローカル変数に、行先までの各軸毎のステップ数データをコピー。
memcpy(position_steps, pl.position, sizeof(pl.position));
for (idx=0; idx<N_AXIS; idx++) {
    // ターゲット位置を絶対ステップ数で計算。-
   target_steps[idx] = lround(target[idx]*settings.steps_per_mm[idx]);
   // x,y 軸の大きい方の値選択。(閾値となる)
   block->step_event_count = max(block->step_event_count, block->steps[idx]);
    ......
}
.....
//buffer_head, next_buffer_head の値更新
block_buffer_head = next_buffer_head;
next_buffer_head = plan_next_block_index(block_buffer_head);
```

【リングバッファからの読み出し】

plan_buffer_line( ) はリングバッファへの書き込み操作を担当するが、これと非同期で読みだすのは stepper.c 中の st_prep_buffer() 関数である [2]。
各軸の移動動作は segment_buffer の中身により実行されるので、セグメントバッファーを読みつくす前に (segment_buffer_tail != degment_buffer_head)、次の block を読みだす。

[2] リングバッファの容量が、書き込みと読み出しの間の非同期時間余裕を与えるのは明確であろう。

```
// bloc_buffer の tail を読み出し。
pl_block = plan_get_current_block();
// 書き込みリングバッファのインデックスのインクリメント。
prep.st_block_index = st_next_block_index(prep.st_block_index);
// 書き込み側構造体リングバッファのアドレス設定。
st_prep_block = &st_block_buffer[prep.st_block_index];
// 以下、書き込み。

st_prep_block->direction_bits = pl_block->direction_bits;
.....
for (idx=0; idx<N_AXIS; idx++) {
   st_prep_block->steps[idx] = (pl_block->steps[idx] << 1);
}
st_prep_block->step_event_count = (pl_block->step_event_count << 1);
.......
prep.steps_remaining = (float)pl_block->step_event_count;
```

【リングバッファ操作関数】

(1) plan_reset_buffer()
リングバッファの初期化関数。

```
void plan_reset_buffer()
{
   block_buffer_tail = 0;  //tail インデックス＝0
   block_buffer_head = 0;  //head が空（ここに最初のデータ）なので = tail
   next_buffer_head = 1;   //tail の次=1
}
```

(2) plan_next_block_index(uint8_t index)
index で示された次のバッファのインデックスを返す。リングバッファなので、単純に index++とは出来ない。
(3) plan_prev_block_index(uint8_t index)
index で示された前のバッファのインデックスを返す。
(4) plan_get_block_buffer_available()
planner buffer 中の使用可能バッファ数を返す。

1.4 Grblのプログラム構成。

Grbl は膨大なプログラムにより構成されており、各ファイルの概要は以下のようである。

config.h :	コンパイル時のコンフィギュレーション用 define ファイル。
coolant_control.c/h:	切削オイル制御。
cpu_map.h:	CPU ポート設定
defaults.h:	Grbl デフォルト値 define ファイル。
eeprom.c/h:	EEPROM read/write 制御関係。
gcode.c/h:	G コードパージングおよび実行。
grbl.h:	Grbl 全体の取りまとめ (include) ファイル。
jog.c/h:	Jog 制御関係ファイル。
limits.c/h:	マシンリミッター制御関係。
main.c:	main ファイル。
motion_control.c/h:	直線補間、円補間のアルゴリズムファイル。
nuts_bolts.c/h:	雑ファイル。
planner.c/h:	切削ステップに分割した実動作制御。
print.c/h:	システム状態プリントアウト用。
probe.c/h:	プローブ動作、制御関係。
protocol.c/h:	実際の main プログラム。
report.c/h:	システム状態レポート関数関係。
serial.c/h:	シリアル通信制御関係。
settings.c/h:	EEPROM に設定する内容ファイル。
spindle_control.c/h:	スピンドルモータ制御ファイル。
stepper.c/h:	ステッピングモータ制御ファイル。
system.c/h:	G コードによらない直接システム制御。

ここで、解析の対象とする主たるファイルは、gcode.c/h、motion_control.c/h、planner.c/h、protocol.c/h、stepper.c/h、serial.c/h であり、これらをベースに G コード処理部分のファームウエアを調査する。

1.5 Grblによる制御

Grbl による制御について、次の 3 種類をサポートしている。

(1) (汎用)G コードによる制御。一般的な CNC 装置の制御と基本的に同等。
これについては第 2 章参照。

(2) $で開始される G コード類似制御。これは G コード実行中に割り込んで、CNC 装置を制御する時などに使用。

(3) リアルタイムコマンド。Gコードによる制御と同等の制御、機械の状態レポート、機械の現材状態保存や印刷、ホーミング、動作速度の変更等。

1.5.1 $ コマンド

ここでは、Grbl 制御コマンドの中で、システム動作に関係するものについて述べる。

【$#】

Grbl は G コードパラメータとして、G54-G59 に関するワーク座標, G28/G30 による既定位置、G92 による座標オフセットを保存する。これらの値の多くは、変更された時に EEPROM に書き込まれる (G92,G43.1 は EEPROM には書き込まれない)。

G54-G59 の値は G10 L2 Px または G10 L20 Px で書き換え可能であり、 G28,G30 の値はそれぞれ G28.1 および G30.1 で変更できる。

$#を用いると、Grbl は、次の様な情報を返す。(ツールオフセット、プローブ位置情報は省略)

```
[G54:4.000,0.000,0.000]
[G55:4.000,6.000,7.000]
[G56:0.000,0.000,0.000]
[G57:0.000,0.000,0.000]
[G58:0.000,0.000,0.000]
[G59:0.000,0.000,0.000]
[G28:1.000,2.000,0.000]
[G30:4.000,6.000,0.000]
[G92:0.000,0.000,0.000]
```

【$H】

このコマンドで機械のホーミングが可能である。

【$Jx=line】

Jog モーションコマンド。このフォーマットは、最初の 3 文字は常に「$Jx=」で Jog を示す。この部分がヘッダーとなる訳で、それに続き X、Y、Z および速度 F を記述する。F については常に必要で、モーダルパラメータとは扱われない。

また、この中で使用可能な G コードは G90/91、G53 のみである。

Jog モードの主な応用は、マニュアル動作を可能とする事であり、ジョイスティックやダイアルによる軸位置制御が可能となる。

【$G】

G コードのパージング状態の表示。
Grbl からの返答は、[GC:をヘッダーとする以下の様なフォーマットになる。

```
[GC:G0 G54 G17 G21 G90 G94 M0 M5 M9 T0 S0.0 F500.0]
```

1.5.2 リアルタイム コマンド

これらは 1 文字コマンドで、G コードで使用されない文字および拡張 ASCII 文字 (0x80 - 0xff) が割り当てられている。
【0x18 (ctrl-x)】: ソフトリセット。直ちに停止し、Grbl を（軸動作無しで）リセット。
【0x21 (!)】 : Grbl をサスペンド状態にする。
【0x7e (~)】 : サスペンド状態からの復帰。
【0x85】 : Jog キャンセル。

1.6 CNC ファイルの拡張子

Grbl 等のターゲットとなるファイルは、3 次元機械加工用ファイルとなるが、その拡張子は統一がとれておらず、メーカー毎に決められている状況である。それらの中でも、比較的よくつかわれる拡張子と、メーカ名を以下に記す。

```
*.nc        Acramatic, Axyz, Onsrud
*.tap       Fanuc, Heidenhein, Cincinnati, Datron
*.mpf       Siemens
*.cnc       Centroid, Vision, Apex
*.min       Okuma
```

特に、$*.min$、 $*.mpf$ はメーカ専用拡張子と思われる。

Chap2　Gcodeによる制御コマンド

2.1　G コードの基本構造

G コードは、

　大文字アルファベット + 数値（正負整数、正負浮動小数点）

が基本シンタックスであり、G コード命令の一番最初はアルファベットで開始される。これが大前提であり、この原則に基づいてパージングされる。CNC マシンの入力データはすべて 1byte 単位の String として読み込まれるので、この原則に基づいてアルファベットによる命令と数値の区別を実行する。

2.1.1　G コードの基本構造処理

Grbl では、実際の G コードのパージングの前に、プリプロセスとして gcode.c 中の gc_execute_line() で、次の様な処理を実行する。

最初に入力ストリングが'$' で開始されているかチェックする。これは CNC の（マニュアル）制御と G コードによる自動プロセスの切り分けのためである。

```
// Jog コマンドか G コードかの区別。
if (line[0] == '$') {
    // Set G1 and G94 enforced modes to ensure accurate error checks.
    gc_parser_flags |= GC_PARSER_JOG_MOTION;
    .....
}
```

以下のプロセスでは、大文字アルファベットを letter に読み込み、数値については read_float() 関数で処理する。

```
while (line[char_counter] != 0) {
    // 1 文字読み込み。最初は必ず大文字アルファベットの筈。
    letter = line[char_counter];
    // 大文字アルファベット以外はエラー。
    if((letter < 'A') || (letter > 'Z'))
        { FAIL(STATUS_EXPECTED_COMMAND_LETTER); }
        // 1 文字送る。次は数値であると仮定
```

```
            char_counter++;
            // 数値読み込み関数 read_floa() に渡す。フォーマット違反はエラー。
            // 浮動小数点は value に代入される。
            if (!read_float(line, &char_counter, &value))
                { FAIL(STATUS_BAD_NUMBER_FORMAT); }
            // 数値の整数部を int_value に代入。
            int_value = trunc(value);
            // 小数部を取り出し 100 倍したものを mantissa に代入。
            mantissa =  round(100*(value - int_value));
            .........
    }
```

2.1.2 read_float() の処理

read_float() 関数は nuts_bolts.c ファイルに纏められていて、数値の解読を実行する。この解読には、主として二つの目的がある。

(1) 'G','M' コードに続く小数を整数と小数点以下の数値に分解、それぞれ int_value および mantissa に代入する。これは G コードの（拡張）コマンドとして、例えば G28 に対して G28.1 が定義されているためにこれらの検出に用いられる。

(2) 'I','J'...'Y','Z' の後に続く数値（正負の整数または小数）を浮動小数点数として読み込む。これは座標データや速度データなどで、CNC マシンの具体的動作量を指定する。

【処理フロー】

図 2.1 に処理フローを示すが、まず 1 行の G コード中から 1 文字を読み込み、+/-の負号判定を行い、次の文字に進む。次の文字は数値が小数点であるので、(A) のブロックでは、数値または小数点以外の文字となるまで読み込みながら、数値を仮数部（整数）と指数を用いて表現する。

(B) のブロックでは浮動小数点に型変換を行い、最後に正負を確定する。

【数値検出】

ここでは MAX_INT_DIGITS=8 として、123.456789 という数値を例にとって動作を解析する。

(A) ブロックには数値と小数点のみが処理対象なので、ASCII'9'(0x39) から ASCII'0'(0x30) および小数点 ASCII'.'(0x2e) について考える。

まず、ASCII 文字を数値に変換するため、読み取った文字 c に対して

$c = c$ - '0'

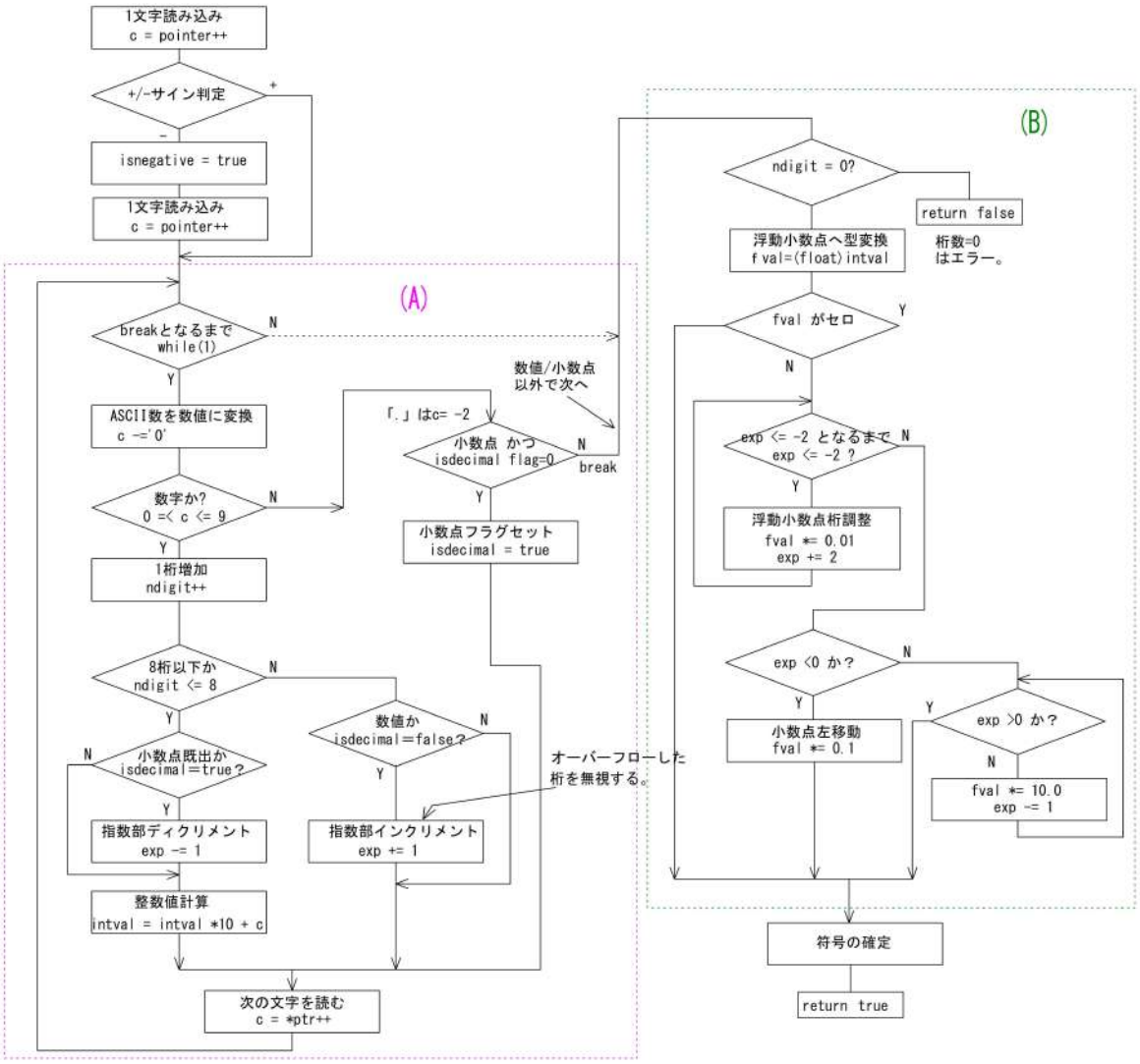

Figure 2.1: 数値処理フロー

を実行する。この計算により'.' は-2 となる。

最初に読み込まれる数字は c='1' なので、

ndigit =1, intval = intval(=0)*10 + 1 =1

次に、小数点が現れるまで同じ処理となるので、c='3' を読み込んだ時点では

ndigit =3, intval = intval(=12)*10 + 3 =123

となる。次は小数点なので、isdecimal フラグをセットするが、この時、このフラグがゼロを確認する。これにより、小数点が 2 個以上を排除する。

c='4' を読み込むと、小数点フラグ isdecimal=1 なので、exp=-1 として、

ndigit =4, intval = intval(=123)*10 + 4 =1234 exp = -1

を得る。

c='8' を読み込むと、

ndigit =8,　intval = 12345678　exp = -5

となる。この状態で次に数値 9... が存在してもそれらは無視される。

一方、小数点が全くない場合（整数）、例えば 123456789 を読み込むと、8 桁読み込んだ時点で、

ndigit =8,　intval = 12345678　exp = 0

である。このあとに 9 を読み込むと数値 9 は無視されるが、exp はインクリメントされ、

ndigit =8,　intval = 12345678　exp = 1

を得る。

【浮動小数点数への変換】

(B) ブロックでは、最終的に exp=0 となるような浮動小数点に変換している。これにより以降の計算はこの変換済みの値に対して、浮動小数点演算をすればよく、exp を忘れて構わない。

そして、最後に+/-を確定する。

2.2　G コードの考え方

G コードは設計が古く、一部の考え方が非常に分かり難い。ここでは、単に仕様を再録するのではなく、それらの解析を試みる。

2.2.1　モーダル、非モーダルコマンド

表 2.1 に示すコマンドのグループ分けで、g0 は非モーダルコマンド、それ以外はモーダルコマンドとも呼ばれる。非モーダルコマンドでは、コマンドの効力はその行のみ、モーダルコマンドでは、別のモーダルコマンドで書き換えられるまで有効と解釈される。

例えば、

```
G1 X10 Y20 F50
```

と記述すると、G1 の効力は、次に G0,G2,G3... コマンドが記述されるまで有効であるので、

```
G1 X10 Y20 F50
X5 Y15
X15 Y-10
Z -10
```

という一連の記述で、X,Y,Z 軸移動が実行される。

同じ事が座標選択コマンドについても言えて、例えば

```
（G54)
G1 X10 Y20 F50
G55
X5 Y15
X15 Y-10
Z -10
```

という記述では、最初にデフォルトの座標系 G54 が選択されていて、その中で G1 による軸移動が指定され、そのあと座標系を G55 に変更して X5 Y15... という軸移動を連続すると解釈される。しかし、この様な記述が実用的には大きな問題を抱えているのは明確である。従って、

```
（G54)
G1 X10 Y20 F50
G55
(ここに軸移動命令が必要であろう)
G1 X5 Y15
X15 Y-10
Z -10
```

の様に、座標変更をしたならば、新規に G コードを明示すべきである。

上述した事態は G10 コマンドではさらに明瞭で、

```
G10 L20 P1 X10 Y0
```

は座標系 G54(P のパラメータで指定される）の X 軸を 10mm 移動（原点の x 座標を-10mm 移動）という意味であるが、G10 は g0 のコマンドなので、この行でしか有効とならない。しかし、命令の結果の原点座標シフトは今後も有効となり、新たに G10 コマンドで変更されるまで保持される。

つまり、コマンドの結果では無く、コマンド自体が複数行で有効となるものをモーダルコマンドと定義しており、この点は非常に誤解し易い。

実用的観点からは、G54 の様な座標選択コマンドは、パラメータを伴わないので次行以降にパラメータだけを記述することはないが、G28,G30,G1,G2,G3 の様にパラメータを伴う場合は問題が多い。

例えば

```
G2 X20 Y0 R30
X40 Y10
```

ならば、（現在座標から）x=20mm, Y=0mm 離れた場所を終点として半径 30mm の CW 円弧を描画し、その終点から x=40mm,Y=10mm 離れた点に向かって（明示されていな

い）半径 r=20mm の円弧を描くと解釈されねばならない。しかし、これはコードが読みにくいという問題と、この様な描画の必要性[1] との複数の問題がある。

結局、モーダルコマンドは本来パラメータが不要なコマンドだけにとどめるべきであったので、G コードの設計の不備であろう。

この状況で、唯一実用性があるのは直線補間関係 G0,G1 のコマンドに関して、

```
G1 X20 Y0 F10
X40 Y10
X-20 Y0
Z -10
X50 Y30
....
```

のような場合であり、Grbl ではこの場合に限ってモーダルコマンドをオリジナルの定義の意味に実装している[2]。

以上のような混乱した状況で、どうしても 1 行だけ、機械座標に戻して（いわば、そこまでの混乱をリセットして、しかし次の行ではもとに戻る）動作させるコマンドが G53 であり、苦し紛れに用意したように見えるが、コマンドとしては有用である。

2.2.2 G コードのエラーチェック

モーダルコマンドはその内容によりグループに分類され、G コード 1 行（1 ブロック）の記法チェックに使われている。

表 2.1 は主要なモーダルコマンドのグループ分類である。例えば軸移動に関係する g1 について見ると、G1 は直線補間（直線加工）、G0 は高速移動なので、これらのモードは同時に存在できない。これは各モーダルグループについて言え、Grbl では同一モーダルグループのコマンドが 1 ライン中に複数存在する場合のエラーチェックを実行している。

一方、G1 が座標系 G55 に対して実行されるのは問題ではなく、モーダルグループが異なれば、コマンドは共存できる。しかし、実際の G コードプログラムでは

```
G55
G1 X40 Y10
```

の様に、行を変えて記述されるようである[3]。

表 2.2 に非モーダルグループの主要コマンドを示す。

---

[1] この様な事態が無いのではなく、この為には単に G2 を先頭に記述すれば済む。

[2] X,Y,Z パラメータが単独で存在する場合のみを、いわば特別扱いしている。

[3] Grbl の実装から判断すると、G55 G1 X40 Y10 も有効。

Groupe	Function	Commands
g1	Motion Mode	G0, G1, G2, G3, G38.2, G38.3, G38.4, G38.5, G80
g12	Coord Sys Sel	G54, G55, G56, G57, G58, G59
g2	Plane Sel	G17, G18, G19
g3	Distance Mode	G90, G91
g4	Arc Distance Mode	G91.1
g5	Feed Rate Mode	G93,G94
g6	Unit Mode	G20, G21
g7	Cutter Rudius Comp	G40
g8	Tool Length Offset	G43.1, G49
m4	Program Mode	M0, M1, M2, M30
m7	Spindle State	M3, M4, M5
m8	Cooland State	M7, M8, M9

Table 2.1: モーダルグループ

Groupe g0
G4, G10L2, G10L20, G28, G30, G28.1, G30.1, G53, G92, G92.1

Table 2.2: 非モーダルグループ

2.2.3 非モーダルコマンドの動作例

(1) 例 1

```
N1  G54
N10 G91
N11 G1 X10 Y20
N12 G53 X0
N13 X30
```

というコードの動きは、G53 が絶対座標動作コマンドなので、

`N1  G54` により座標 1 選択
`N10 G91` によりインクリメンタルモード
`N11 G1` により `X10`、`Y20mm` 移動
`N12 G53` により絶対座標の `X0` に移動
`N13 X30` は `X` 軸のみ `30mm` インクリメント

となる。さらに分かり難くしている点が、マシン依存のデフォルト値で、通常座標指定を省くと G54 であり、また絶対値モードをデフォルトとする機械ならば、G コードによる指示が何も無いならば G90 が指定される。

(2) 例 2

```
N1  G90            : 絶対座標モード
N2  G54            : 座標 P1 選択
N10 G0 X30 Y40     : G0 で座標 (30,40) に移動
N20 G4 P2          : G4 で 2 秒間現在位置
N30 X20 Y50        : (G0、G90 が有効で) 座標 (20,50) に移動
```

この例ではライン 10 で、座標 (30,40) に高速移動し、G4 でその移置に 2 秒とどまり、その後座標 (20,50) に高速移動する。つまり G0 はライン N30 も有効である。

(3) 例 3

```
N1  G54 G91               : G91 でインクリメンタル指定
N2  G10 L20 P1 X-10 Y-8   : G10 で座標を x=-10mm ,y=-8mm オフセット
N10 G0 X30 Y40            : G0 で X=30,Y=40mm 移動
N20 G4 P2                 : G4 で 2 秒間現在位置
N30 X20 Y50               : ( G91,G0 が有効で)X=20,Y=50mm 移動
```

この例では、座標 G54 に対して G91 でインクリメンタルモードを指定し、N2 で G54 に対してオフセットを与えている。N10 では N1 のインクリメンタルが有効なので、G0 はインクリメンタルとなる。

(4) 例 4

```
N1  G54 G91               : G91 でインクリメンタル指定
N2  G92 X-10 Y-8          : G92 で（全ての）座標を x=-10mm ,y=-8mm オフセット
N10 G0 X30 Y40            : G0 で X=30,Y=40mm 移動
N20 G92 X10 Y8            : G92 のリセット相当
```

G92 に関してはワンショットコマンドによる座標オフセット設定であるが、これもその効果は以降も有効となる。従って、リセットには N20 行の様に再度 G92 を用いて、オフセットをキャンセルできるが、Linux CNC ではリセットコマンド G92.1 という拡張コマンド（リセットコマンド）も用意されている [4]。

2.2.4 G コードの状況

第 2.2.3 節に示した様にわかりにくい記述ができるので、コマンドは 1 行に一つ、パラメータだけの記述は避ける [5] など、いわば常識（良識）をもった書き方をすべきである。逆に、G コードはこの様な常識をユーザに期待しており、さらに、G コードインタープリ

[4] 理由ははっきりしないが、G92 だけを見たのでは、設定/復帰の区別がつかない。
[5] Grbl では G0,G1 に対する X,Y,Z だけサポート。

タ（あるいはコンパイラ）はCNCマシンメーカが特定の機械に対して実装するものなので、コンピュータ言語の様な厳密な解釈を期待してはいけない。

このためGコードは機械およびメーカ依存の部分が多く、また現在では人間がGコードを記述するのが減少しているため、考え方に対する解説も少ないのだろう[6]。

2.2.5 絶対/インクリメンタル座標

切削動作が座標系に対して絶対座標を指定するか、現在からの相対移動量を指定するか、さらに座標がマシン座標なのか、ワーク座標なのかの違いにきを配る必要がる。

例えば、基本コマンドであるG1を考えても、これがどの座標(G54-G59等）に対してなのか、インクリメンタルか絶対値指定かが分からないと動作させられない。

特に問題なのがX,Y,Z指定で、これはGコードを抜き出した例などを見ると、単独で使用可能と見えてしまう。しかし、X,Y,Zだけで動きを指定するためには、このコマンドの前に、

(1) モードはG0かG1?
(2) 座標系が決定されている（G54-G59) か?
(3) 座標系指定後でも、絶対位置指定かインクリメンタルか?
が解決されている必要があり、さらに
(4) 上記がすべて解決されているのに、これらを無視するG53があるか?
という条件を見ないと分からない。

例えばつぎのコードの動作は以下の様になる。

```
N1  G54 G91             : G91でインクリメンタル
N2  G1 X-10 Y-8 F50     : G1でX=-10mm ,Y=-8mm移動
N10 G53 X0 Y0           : G53で座標(0,0)に移動
N20 Z10                 : Z軸のみ（上）方向に10mm移動
N30 X20 Y50             : ( G91,G1が有効で)X=20mm,Y=50mm移動
```

次のコードではG1コマンドの前に、G90で絶対座標が、G54で使用座標系が指定されているので、動作条件は揃っている。

```
N1  G90
N2  G54
N10 G1 X20.0 Y20.0 Z0 F10
N11 X20 Y0.0 Z0
N12 X0 Y0
N13 X0.0 Y20 Z0
N14 X20 Y20.0 Z0 F20
```

[6]言語としては消えゆく運命にあるのだろう。

```
N15 X10.0 Y25.0 Z0
N16 X0.0 Y20.0 Z0
```

上記コードによる結果をプロットして見ると、次の様な図形となる。

Figure 2.2: 描画例

2.2.6 G コードの実行順序

G コードはインタープリタにより解釈され、ラインの順に沿って実行される。一方 1 ライン中の実行順序は次の様に決められている（主要なものを示す）。ただし Grbl では 1 ライン中のコマンドの実行順序は実装されていない。

(1) 軸移動モード設定 (G94)。
(2) 軸移動速度設定 (F)。
(3) スピンドル速度設定 (S)。
(4) スピンドル ON/OFF (M3, M4, M5)。
(5) ドウェル (G4)。
(6) 加工平面設定 (G17)。
(7) 座標系設定 (G54, G55, G56, G57, G58, G59)。
(8) 距離モード設定 (G90, G91)。
(9) 基準位置移動 (G28, G30)、座標データ更新 (G10)、軸オフセット設定 (G92, G92.1)。
(10) 加工実行 (G0 to G3,G80 to G89)(G53 による変更含む)。
(11) ストップ (M0, M1, M2, M30)。

2.3 G コードの詳細解説

以下、代表的な G コードについての解説である。(原則として、単位は mm, 加工平面は X,Y を仮定。)

2.3.1 G0

高速移動コマンドで、各軸はその最高速で移動。従って G コードの仕様上は直線とは限らない。

{例 1} $G0\ \ X0\ \ Y0\ \ Z0$
座標系の初期値は G54 なので、指定なければ G54 系の原点 (0,0,0) に向けて移動。

{例 2} $G92\ \ G0\ \ X0\ \ Y0\ \ Z0$
ホームポジションに移動。

2.3.2 G1

直線補間。行先パラメータ X,Y,Z のいずれか一つは必須であり、移動速度指定も必要である。X,Y,Z が絶対座標か、相対移動量かの判断は、それぞれ G90、G91 によるモード設定による。

{例} $G1\ \ X20\ \ Y15\ \ F200$
現在の動作モードがインクリメンタルならば、現在位置から X20[mm]、Y15[mm] だけ直線移動。速度は 200[mm/min]。
現在の動作モードが絶対座標モードならば、座標 X20[mm]、Y15[mm] まで直線移動。速度は 200[mm/min]。

2.3.3 X,Y,Z 座標単独指定の解釈

モーダルコマンド G0,G1,G2,G3 の場合、一度 G コードを書けば、次の行からは X,Y,Z だけで軸移動を指定することができる[7]。この機能は直線補間の場合、比較的分かり易いが曲線補間では、かなり特殊な事態である。
即ち、曲線補間 (G2,G3) の場合にも現在座標が開始点となるので、円弧の連続を描画することになり、使用法が非常に特殊になる[8]。この点を考慮してか、Grbl では、X,Y,Z の単独指定しかサポートされておらず[9]、これは実質的に直線補間の場合のみ、X,Y,Z 単独指定を許している。

2.3.4 G2-G3

G2 は時計回り、G3 は反時計回りの円弧補間である。
円弧補間は直線の連続で実現する。この時、円弧データの与え方として、【半径および終点座標】および【中心および終点座標】の 2 方法が定義されている。以下の例では描画平面は G17 とする。
さらに円弧の角度については次のように定義される。
r が負： 円弧の回転角度が 180 度以上

[7]モーダルコマンドは、ある行に記述すれば、そのあとの行はパラメータだけ記述できる。
[8]これを実行したければ G2/3 を記載すれば可能。
[9]I,J,K,R,F なども円弧補間には必要となる。

r が正： 円弧の回転角度が 180 度以下
また開始点は常に現在座標である。

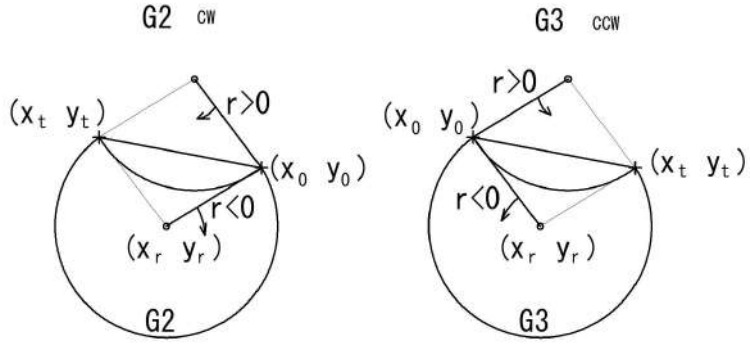

Figure 2.3: 円弧補間

(1) 半径および、終点座標が与えられる場合。開始座標は現在座標で、終点座標に向かい CW,CCW で円弧を描く。

{例 1} $G02\ \ X10\ \ Y10\ \ R10\ \ F50$

{例 2} $G02\ \ X10\ \ Y10\ \ R-10\ \ F50$

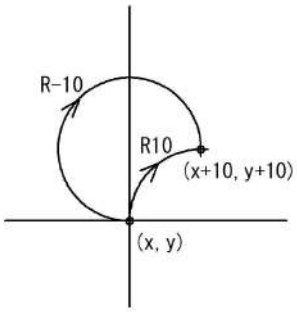

Figure 2.4: 円弧補間、半径モード

円弧の開始点は現在座標なので、現在のモードが絶対/インクリメンタルモードによる違いは無い。終点座標のどちらか一方は省略可能。終点位置は現在からの相対値で指定する。半径データは省略できない。また全周円を描く時は、終点座標が現在座標と同じになってしまうため、この方式は使用できない。

radius： 半径。
target_coord： 終点 xy 座標 (相対値)。
curr_pos ： 現在 xy 座標（暗黙の前提)。
一方、計算により求めるものは
xy_offset ： 現 xy 座標から円弧中心座標までのオフセット。

(2) 円弧の中心および、終点座標が与えられる場合。半径は現在座標と中心座標から計算されるので、終点座標は、回転角の決定に用いられ、必ずしも円弧上の存在しなくてもいい。

ターゲット座標は X,Y,Z により、現在位置からの相対値で与えられる。同様に中心座標は I,J,K により、現在座標からの相対値で与えられる。全周円はこの方法で指示する。
ijk ：円弧中心 xy 座標。 現在座標からの相対値。
target_coord： 終点 xy 座標 (相対値)。
curr_pos ： 現在 xy 座標（暗黙の前提)。
一方、計算により求めるものは
radius ： 半径。

{例 1} $G02\ X-30\ Y-40\ I-10\ J-15\ F50$
図 2.5 に示す様に、ターゲット座標を (-30,-40) とすると、これは円弧上には無い。そこで、円弧中心 (-10,-15) とターゲット座標を結んだ直線に達するまで、円弧が描画される。

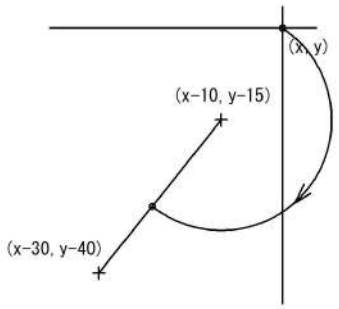

Figure 2.5: 円弧補間、中心モード

【G91.1】: 円弧補間 I、J、K オフセット絶対座標指定。I、J は G2/3 に関して XY 平面で設定。J、K は G2/3 に関して XZ 平面で設定等。

{例} $G91.1\ G0\ X5\ Y10$
これは、Linux 拡張コマンドであり、円弧補間のセンターモードに対し、中心座標を、現ワーク座標上での絶対値で指定する。しかし、一般的には、このサポートは少ない。

2.3.5 G4

dwell。現在位置に指定された時間だけ止まる。機械の実行時間に影響するので、時間は100msec 程度が通常の値である。P の単位は秒のみで、浮動小数点で指定する。

{例} $G4\ \ P1.0$
1 秒間現在位置にとどまる。

2.3.6 G10

オリジナル G コードでは G10 コマンドは座標設定やツール設定のためのレジスター編集機能として定義されている。従って、状況によりパラメータの意味が異なり扱いが面倒である。ここでは比較的一般的な例を述べるが、メーカやマシンにより解釈が異なり得るので注意。

【 G10L20Pn 座標設定（相対値指定)】

P に付随するパラメータ n は座標番号で整数であり、その座標に対してオフセットを指定する。L2 との違いは、X,Y,Z は相対値を与え、対象座標系に設定されている値に加算される点である。

{例 1} $G10\ \ L20\ \ P2\ \ X20$
座標系 2 の X 座標オフセット値を+20 加算。X 座標の原点を-20 の点にずらす事と同じ。
例 1 の場合、今まで加工していた原点が 20mm、左に移動する。これは例えば全く同じ加工をワークに対して 20mm 右側に実行する時などに便利である。この方法より明らかな様に、G10L20 による操作は、それ以前や以後が絶対座標動作をしていると辻褄が合わない[10]。そのため、このコマンドはインクリメンタルモード (G90) で使用するコマンドである。同じ操作は複数の座標系、例えば G54 と X 軸に対して 20mm シフトした G55 を用いても可能である。

{例 2} $G10\ \ L20\ \ P2\ \ X0\ \ Y0\ \ Z0$
G55 座標の原点を現在位置に設定。

例 2 の場合、座標のオフセット量をゼロとして設定する事になるので、**現在位置を原点に設定**するという意味である。

[10] CNC マシンとして不可能という意味では無いが、通常はやらない。

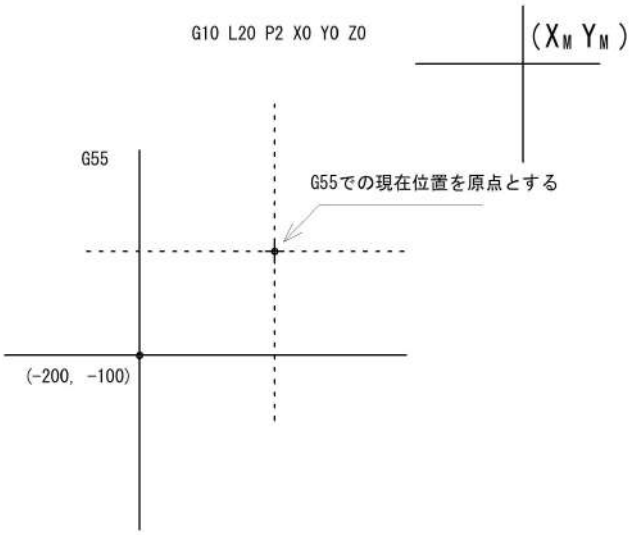

Figure 2.6: G10 L20 の効果

【G10L2Pn 座標系設定（絶対位置指定)】

座標系選択パラメータは n(座標番号、整数）であるが、その座標の原点を機械座標のどの点とするかを X,Y,Z により絶対値指定する。このオフセットは絶対値なので、CNC 動作の最初あるいはまとまった動作の区切りで使用される。つまり、インクリメンタルモード (G91) 中にこのコマンドを使用すると、予期せぬ動きとなる。

｛例 1｝ $G10\ L2\ P1\ X-200\ Y-100$
座標系 1 の x=-200,y=-100 を原点と設定。

この場合、マシン座標の x=-200、y=-100 を原点として加工することになる。従って CNC マシンのユーザは、実際のテーブル上のどこに対応するかをあらかじめ知っていなければならない。

｛例 2｝ $G10\ L2\ P1\ X0\ Y0\ Z0$
座標系 1 のオフセットクリア。

例 2 の場合、現在のスピンドルの位置に関わらずオフセットをゼロにするので、(P1 で指示される）G54 座標系をマシン座標と一致させる意味となる。

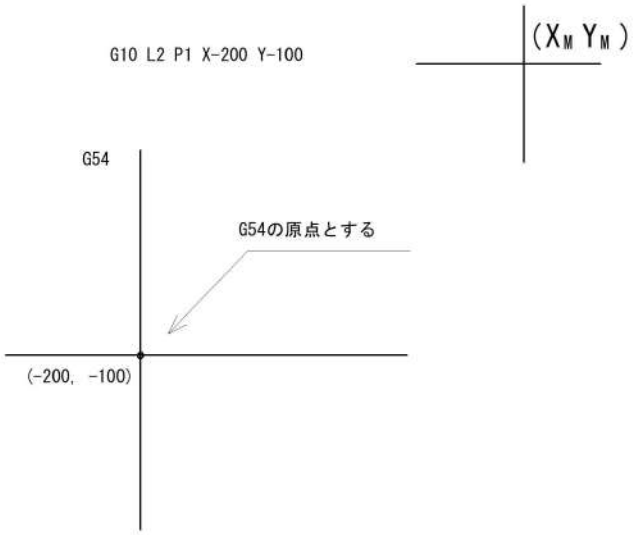

Figure 2.7: G10 L2 の効果

2.3.7 G17-G19

CNC による加工を x,y,z 空間のどの平面に対して実行するかを指定するコマンド。普通は G17 が使われる。

【G17】: XY 平面選択。

【G18】: ZX 平面選択。

【G19】: YZ 平面選択。

2.3.8 G20,G21

G20,G21 はそれぞれインチ単位、メートル単位選択コマンドである。
【G20】: 単位をインチに設定。

【G21】: 単位をミリメートルに設定。

2.3.9 G28,G38

CNC マシンによる作業が完了しワークを取り外すには、スピンドル軸を退避させねばならない。この時、いきなり機械原点 (0,0,0) に移動させると、その移動途中にワークに

衝突する可能性が高い。もし、X,Y 平面基準で加工しているならば、まず Z 軸を最高位置まで退避させたのち、X,Y 軸を原点復帰させるのが適当である。このような目的のために用意されているのが G28/G30 である。

G28/38 は機能は同一で、単に 2ヶ用意されているにすぎないが非常に注意せねばならない 2 ステップのコマンドである。まず、パラメターとして、プログラム上で G28 に付随する X,Y,Z 値と、機械に記憶される値 [11] があり、Linux CNC 文書では後者をパラメターと呼んでいる。

ここでは、前者を X,Y,Z 値と呼ぶことにするが、これらは、途中の経由座標を表し、現在がインクリメンタルモードか絶対値モードかで意味が異なる。後者は最終到達座標である。

＊インクリメンタルモード　; X,Y,Z 値が現在の座標系での相対値。

＊絶対値モード　　; X,Y,Z 値は現在の座標系での絶対値。

一方 CNC マシンに記憶されるパラメターはマシン絶対座標値である。従って、現在の座標系にも、動作モードにも影響されない値となる。

結局、G28,G30 は最初のステップでは X,Y,Z 値で示される座標に移動するが、この時は現在座標基準かつインクリメンタル/絶対値モードにより動作が異なるのに対し、2 ステップ目では現在の座標系も動作モードも関係しない。つまり、全く別の考えに基づく命令を [12] 無理やり一つにまとめてしまったものと考えられる。

{例 1} $G91\ \ G28\ \ Z0$

例 1 の場合、インクリメンタルモードなので、最初のステップでは Z 軸は移動せず、次のステップで Z 軸のホームポジション [13] に移動する。これはツール交換等に有用である。

{例 2} $G28\ \ Z0$

一方、例 2 の様に G91 を忘れると、デフォルトでは絶対値モード（G90) なので、Z 軸は 0 に向けて高速移動し、テーブル面が Z=0 と設定されているであろうから、ほぼクラッシュに至る。

{例 3} $G90\ \ G28\ \ Z-10$

例 3 は絶対値モードでの記述で、Z 軸は、現座標系の Z=-10 に移動した後、ホームポジションに戻る。ただし、Z=-10 が現座標系で問題無い位置である事をあらかじめ知っておかねばならない。

この様に見てくると、例 1 というのは、G91 を忘れない様にすれば、マシンのホームポジションを確認せずとも安全に動作する。(参考資料 [7]）

[11] X,Y,Z についてはそれぞれアドレスが 5161, 5162, 5163 と規定されている。
[12] 恐らく、便利だから。
[13] あるいは G28.1 で設定した Z 座標。

G28 および G30 のホームポジション座標を設定するのが G28.1/G30.1 コマンド [14] で Linux 拡張コマンドである。

{例 4} $G28.1\ X-10\ Y-25$
例 4 ではホームポジション位置を (-10, -25) と設定する。この値は現在の座標系ではなく機械座標系での値である点に注意。

G28/38 と同等の動きは例えば G0 を使って、2 行に書けば可能である。まず途中の点に移動し、そのあとホームポジションに移動させればいい。しかし、ホームポジションが原点以外の場合、あらかじめ調べておかねばならない。

{例 5} $G90\ G0\ Z-10$; 座標 $-10mm$ に移動
$X0\ Y0\ Z0$; 機械原点 $(0,0,0)$ に移動

以上の例から分かる様に、G0 により 2 ステップでホームポジション移動を実行するとなると、現在が絶対値かインクリメンタルか、またホームポジションが機械原点か否かなど、知らねばならない条件が多く面倒である。これが G28/G30 コマンドを用意した理由と思われるが、あまり巧い解ではない。

2.3.10 G53

G53 は現在使用中の座標系を無視し、絶対機械座標を使用するコマンドである。このコマンドは G0,G1 が有効な時のみ動作し、機械座標系で直線移動を実行する。

{例 1} $G53\ G0\ X-10\ Y-25\ Z-9$
スピンドルが機械座標 (-10、-25、-9) に高速移動。ツール交換時等に有用。次のブロックはプログラムに記載されていなくともワーク座標系で動作。

{例 2} $G53\ G0\ X-10\ Y-25\ Z-9$
$G53\ X-20\ Y-10\ Z-5$
スピンドルが機械座標 (-10、-25、-9) に高速移動後、(-20、-10、-5) に移動。2 行目は G0 が省かれていても、1 行目の G0 が有効となっている点に注意。

G28/G30 コマンドの代わりに G53 を使用することもできる。特に、ホームポジションのメモリは無く、単純に機械座標の絶対値を指定し、その座標に移動する。従って、G53 に続く G0 等のパラメターは基本的に負の値で (0,0,0) が機械原点である。

[14] アドレス 5161, 5162, 5163 に記憶すると規定されている。

2.3.11 G54-G59,G92,G92.1

G コードで座標系を設定あるいはモディファイするコマンドは G92 及び、G54-G59 である。G54-G59 は操作空間内にローカル設定可能な座標系であり、6 種類をサポートしている [15]。一方 G92 はすべてのローカル座標系に対して、その基準を設定するもので、機械座標系をオフセットしたものである。多くの工作機械では機械座標系は右奥の最高点を (0,0,0) とするので、すべての座標は負の値となる。

CAD 等を使用しての機械設計では左手前を X=0,Y=0 とし、加工テーブル面を Z=0 とする事が多いと想定されるが（座標値はすべて正）、これと CNC マシンの動作座標を合わせるには G92 を使えばいい [16]。

G92.1 は G92 のリセットコマンドである。

{例} $G92\ \ X-100\ \ Y-125\ \ Z-50$

X、Y、Z それぞれの可動範囲が 200、250、100mm ならば、その中心に G92 座標の原点を設定。

G コードの座標系とコマンドについて整理すると第 2.8 図の様になる [17]。

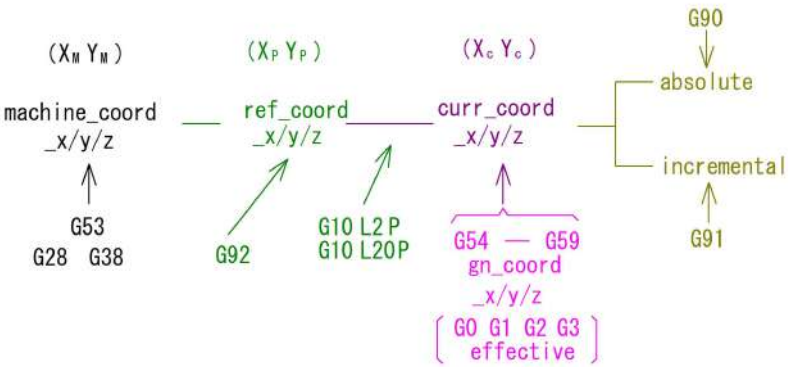

Figure 2.8: 座標系と対応 G コード

2.3.12 G80

複合サイクルキャンセル。

[15]高度な CNC マシンでは、このローカル座標数が不足するので、G コードも拡張されている。NC-15 では 6 種類限定。

[16]G コード体系で、G92 を用意したのはこの理由によるものであろう。

[17]G コードを解析した結果から想定したもので、すべての機械にあてはまるかは不明。

{例} $G80$

複合サイクルのキャンセルコマンドであるが、これは g1 グループの命令なので、他の g1 グループの命令 (G1 等）を記述すれば、自動的に G80 はキャンセルされる。このコマンドは加工用 G コードの先頭に、確認のため用いられる事が多い様である。

2.3.13 G90,G91

これらのコマンドは、軸移動動作のパラメータ指定がそれぞれ、絶対値指定か、インクリメンタル（相対値）指定であるかを支持する。

{例} $G90\ \ G0\ \ X-5\ \ Y-10$

現在位置から (-5, -10) の座標に最高速移動。

{例} $G91\ \ G0\ \ X10\ \ Y-10$

現在位置から X 方向に＋ 10mm、Y 方向に -10mm 最高速移動。

2.3.14 G92

現在位置からの座標オフセット設定。すべてのワーク座標について共通。

{例 1} $G92\ \ X10\ \ Y25\ \ Z0$

現在位置から X=10mm、Y=25mm の位置を原点とする。つまり、現在位置は (-10, -25) となる。G92 はすべてのワーク座標系 (G54-G59) に共通に作用するので、通常は切削用 G コードの最初に使用する事になろう。

{例 2} $G92\ \ X0\ \ Y0\ \ Z0$

現在位置から (0,0,0) 離れた点を原点とするので、これは現在位置を原点とするという意味になる。

2.3.15 G93,G94

G93 はフィードレートを時間の逆数基準で決める [18]。F ワードは、フィードが 1/F 分で完了する事を意味。

{例 1} $G93\ \ F2.0$

フィードは 30 秒で完了。もし単位が mm ならば、20mm/分のフィードレートで送る。

G94 は、単位を 1 分当たりとするコマンドである {例 2} $G94\ \ F20$

この例では、フィードレートは 20mm/分となる [19]。

[18] これはレーザーカッターなどの様に、切断速度一定を要求される場合に必要なもの。

[19] フィードレートの単位はインチ、mm あるいは角度。

2.4 M コード

2.4.1 M0,M1

M0 は機械停止、M1 はオプション機械停止ボタンが用意されている時有効となるオプション機能である。

2.4.2 M2,M30

G コードオリジナルの仕様では、M30 はプログラム完了後開始位置に戻るコマンドである。M2 もプログラム完了コマンドであるが、プログラムが開始位置に戻るかどうかは実装依存であり、現在は使用されておらず後方互換のため残されている。

2.4.3 M3,M5

M3 はスピンドルオン (CW)。速度は G97(default) または G96 で規定される。M5 はスピンドル停止である。

2.5 F,I,J,K,L,N,P,R,X,Y,Z コード

これらは、数値を設定するためのコードである。

F : 軸の移動速度を設定。単位は [mm/min]。
I,J,K : 円弧補間コマンドの中心座標設定。相対値で単位は [mm]。
L : G10 コマンドの補助パラメータ。整数値。
N : G コード行番号。整数値。NC-15 では 1-65536 行まで。
P : G54 のパラメータの場合、座標指定番号で 1-6 まで。
G4 のパラメータの場合、dwell 時間で浮動小数点、単位は秒。
R : 円弧補間の半径指定。単位は [mm]。
X,Y,Z : 単独で出現すると、G1 が設定されていると判断し、その座標パラメータ。
単位は [mm]。

2.6 複合 G コード

G コードには、幾つかの処理をまとめた、いわばマクロとなるものが定義されている。例えばドリリングなどは、スピンドルを特定の位置に移動、その移置で Z 方向に穴あけ、という動作の繰り返しが頻繁に発生する。

このような目的のため、複合動作が定義され、以下の様なコマンドが割り当てられる。

【G73】 高速深穴あけサイクル - 早送り

【G80】 固定サイクルキャンセル
【G81】 スポットドリリング - 早送り
【G82】 カウンターボーリング ドウェル 早送り
【G83】 深穴あけ - 早送り
【G85】 ボーリング - 切削送り
【G86】 ボーリング 主軸停止 早送り
【G88】 ボーリング ドウェル→主軸停止 手動
【G89】 ボーリング ドウェル 切削送り

2.7 ローカル座標系の計算

CNC 加工では、マシン座標系に対して、オフセットした座標系を考えるのが普通である。このオフセット座標は複数を用意可能であるが、ここでは 1 つのオフセットした座標系について考える。マシン座標系によるワークの目標座標を M_{pos} とし、オフセットを加

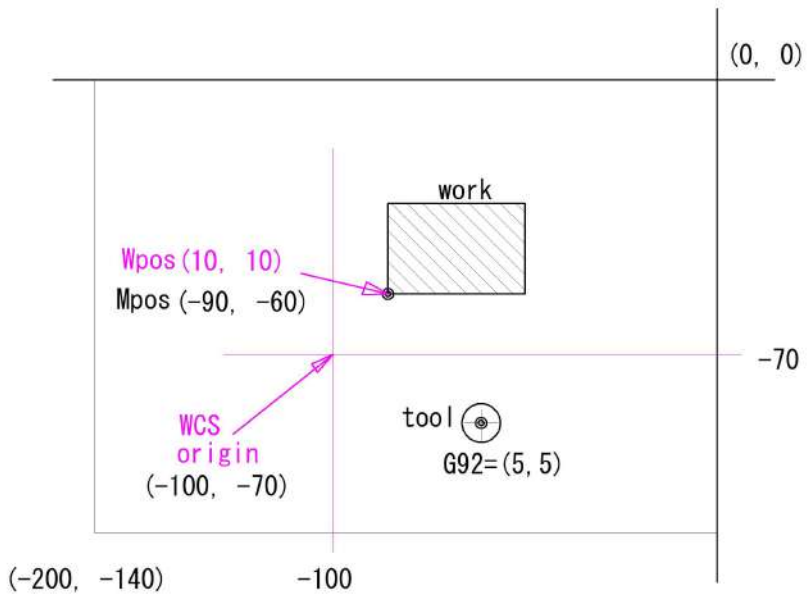

Figure 2.9: 座標のオフセット

えたワークの座標系を WCS とすると、ワークの（ワーク座標系の）位置座標は

$$W_{pos} = M_{pos} - WCS.$$

図 2.9 の場合

$$W_{pos} = (10, 10) = (-90, -60) - (-100, -70).$$

これにさらにローカルオフセット (例えばツール径によるオフセット) G92 を考慮すると、

$$W_{pos} = M_{pos} - WCS - G92$$

$$W_{pos} = (5, 5) = (-90, -60) - (-100, -70) - (5, 5)$$

が得られる。

【G92】

G92 は当初の動作と現在の動作が変化しているなど、誤解されやすいコマンドである。機械固有座標以外のすべての座標に対する（一時的な）オフセットコマンドと解釈され (第 2.9.2 節参照)。オフセットされた値は G92.1 によりリセットされるまで有効となる。G92 が与えるオフセット値は

$$G92 = M_{pos} - WCS - W_{pos}$$

となる。

【G10 L2】

G10 L2 コマンドでは P により指定された座標系の原点をオフセットするコマンドである。即ち、P で示されたローカル座標系にそれぞれ個別の原点を設定するものと考えればいい。X,Y,Z パラメータをゼロとすれば、マシン座標に戻せるので、リセットコマンドは無い。

$$WCS = W_{pos}$$

G10 L2 P1 X3.5 Y17.2：X=3.5,Y=17.2 のマシン座標を原点にする。
G10 L2 P1 X0 Y0 Z0 ：ゼロを原点とする（オフセットクリア)。

【G10 L20】

G10 L20 コマンドでは、P で指定された座標系に対して、X,Y,Z で指定されたオフセットを加えるが、

$$W_{pos} = M_{pos} - WCS - G92$$

より

$$WCS = M_{pos} - W_{pos} - G92$$

と言う計算を実行する。

例えば G10 L20 P1 X1.5 は G54 座標系に対して、X を+1.5 オフセットするコマンドである。即ち、ワーク座標 (例えば G54) がすでにローカル座標として、マシン座標系からオフセットを与えられた場合、このローカル座標に対するオフセットを与える。

G91 によりインクリメンタルモードが指定されている時は G10L2 は無効である。また P を指定しても、現在使用中の座標系が指定されたものに変化してしまう訳では無い。G92 によるオフセットが G10L2 の前に実行されていると、それによるオフセットも有効である。

2.8 ローカル座標系の考え方

以下の議論は Grbl による G コードのパージング処理の構造を理解するための参考情報である。

G コードでは座標系選択のモーダルコマンドとして、

G54、G55、G56、G57、G58、G59

が存在する。一方、座標にオフセットを与える非モーダルコマンドとして、

G10、G92、(G92.1)

が存在する。

これらの座標系制御コマンドを用意した理由を推定すると、以下の様な状況が考えられる。図 2.10(A) では、複数のワークに対して全く同一の加工を実行したい場合であり、この場合、各ワークに対して原点 (x1, y1)... を決め、これらに対して座標 G54,G55,... をアサインするのは自然な考えである。

一方 (B) では、一つのワークに対して、異なった点を原点として、異なった加工 [20] を実行したい場合を示す。この場合でも（x1,y1),(x2, y2)... に G54、G55... をアサインする事ができるが、ワーク 1 に対して、単に座標をオフセットしてもいい。まさにこの様な場合に対応する命令として、G10 が用意されている (例については第 2.10.1 節参照)。

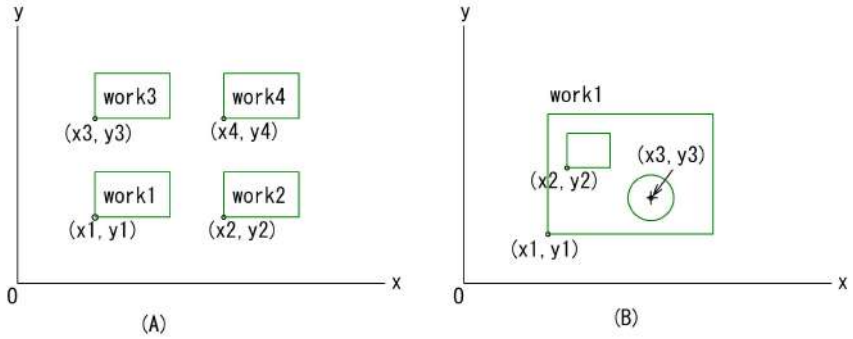

Figure 2.10: CNC の座標構造

G コードでは、使用可能座標数を限定しながら、それにオフセットを与えることで、等価的に座標系を制限を無くした実処理を可能とした。

[20] しかし、同じ加工を他にも使えるような場合。

例えばデフォルトの G54 のみを使用しながら、G10 L2 P1 X** Y**を用いれば、座標原点の数は無限に拡張できる。

なお、Grbl 中のソースファイル gcode.c 中で、value.p を次の様に切り捨てているのは、

```
coord_select = trunc(gc_block.values.p);
```

G54 系コマンドの拡張として G59.1... のような小数点付きのコマンドが定義されており、これらを排除する [21] ためである。

2.9 G コード補足解説

2.9.1 G90/91,G53

G53, G90, G91 はスピンドル軸移動をマシン座標絶対値とするか、現在使用中の座標に対して絶対値/相対値（インクリメンタル）とするかの選択を行う。

G90: モーダルコマンドで現座標系の絶対座標指定。

G91: モーダルコマンドで現座標系のインクリメンタル指定。

G53: 非モーダルコマンドでマシン一時的絶対座標選択

G90/91 による軸移動モードは、一度設定すると、G91/90 により再設定されるまで有効である。一方 G53 は非モーダルコマンドなので、絶対機械座標とするためには、その都度記述せねばならない。即ち、G91 によりインクリメンタルモードが有効であると、G53 のコマンド完了後には、再度インクリメンタルモードに戻る。

G90 が有効な場合でも、これは G54-G59 に対しての絶対座標モードである。ここで G53 が存在するとマシンの絶対座標モードとなるのに注意。

2.9.2 G92 補足

(https://smithy.com/cnc-reference-info/coordinate-system/g92-offsets/page/4)

Smithy.com の解説には、

「G92 is the most misunderstood and maligned part of EMC programming. The way that it works has changed just a bit from the early days to the current releases. This change has confused many users. It should be thought of as a temporary offset that is applied to all other offsets.」

とあり、機械により動作が異なる事を示唆する。さらに、Ray Henry が各種 G コードインタープリターを比較研究した処、

「G92 This command, when used with axis names, sets values to offset variables.」

の様に指定した軸に対してオフセットを与えるというのが、一般的な動作と想定される。

[21] Grbl では G59.1... を単純に G59 に読み替えてしまっているが、これは G59.1... をエラーとすべきであろう。

2.10 注

2.10.1 サンプル G コード

以下のサンプルコードは http://linuxcnc.org/docs/html/gcode/coordinates.html より引用としたものである。（ただし、注釈を付け、単位は mm にしてスケールも変更。）

(5 個の小円を十字形に配置、切削)
G10 L2 P1 X0 Y0 Z0 (座標 G54 の原点をマシン原点に設定)
G10 L2 P2 X10 (G55 を X=10mm オフセット)
G10 L2 P3 X-10 (G56 を X= -10mm オフセット)
G10 L2 P4 Y10 (G57 を Y=10mm オフセット)
G10 L2 P5 Y-10 (G58 を Y= -10mm オフセット)

G54 G0 X-2.0 Y0 Z0 (G54 による中心の円)
G1 F1 Z-1.00(Z 軸を下に 1mm 移動）
G3 X-10.0 Y0 I1.0 J0(座標 (-10, 0) の点に 1mm の深さで円を描画）
G0 Z0（Z 軸を Z=0 に移動)

G55 G0 X-2.0 Y0 Z0 (G55 による円)
G1 F1 Z-1.00(Z 軸を下に 1mm 移動）
G3 X-10.0 Y0 I1.0 J0(座標 (-10, 0) の点に 1mm の深さで円を描画）
G0 Z0（Z 軸を Z=0 に移動)

G56 G0 X-2.0 Y0 Z0 (G56 による円)
G1 F1 Z-1.00(Z 軸を下に 1mm 移動）
G3 X-10.0 Y0 I1.0 J0(座標 (-10, 0) の点に 1mm の深さで円を描画）
G0 Z0（Z 軸を Z=0 に移動)

G57 G0 X-2.0 Y0 Z0 (G57 による円)
G1 F1 Z-1.00(Z 軸を下に 1mm 移動）
G3 X-10.0 Y0 I1.0 J0(座標 (-10, 0) の点に 1mm の深さで円を描画）
G0 Z0（Z 軸を Z=0 に移動)

G58 G0 X-2.0 Y0 Z0 (G58 による円)
G1 F1 Z-1.00(Z 軸を下に 1mm 移動）
G3 X-10.0 Y0 I1.0 J0(座標 (-10, 0) の点に 1mm の深さで円を描画）
G0 Z0（Z 軸を Z=0 に移動
M2(プログラム終了）

上記プログラムで、非モーダルコマンド G10 がマシン座標に対して、G54-G58 座標オフセットを与えている。そしてこれらは、このプログラム最後まで有効で、G10 が記述され

ている行だけ有効という訳ではない。また、コード先頭の G10 L2 P1 X0 Y0 Z0 は、G54 座標の原点が （以前の作業によってオフセットされていても）想定する原点である事を保証するためである。

2.10.2 モーダル、非モーダルコマンドの解説例

モーダルコマンド、非モーダルコマンドについて混乱している事は、色々な情報源を参照すると想像できる。

https://en.wikipedia.org/wiki/G-code

によれば、モーダルコマンドは、

「Many codes are ”modal, meaning they remain (in) effect until cancelled or replaced by a contradictory code. For example, once variable speed cutting (CSS) had been selected (G96), it stays in effect until the end of the program. ... Similarly, once rapid feed is selected (G00), all tool movements are rapid until a feed rate code (G01, G02, G03) is selected.」

と定義される。

あるいは、

https://dynomotion.com/Help/GCodeScreen/EMC_Handbook/node65.html

には、

「”Non-modal” codes effect only the lines on which they occur. For example, G4 (dwell) is non-modal.」

とあり、さらに「There is some question about the reasons why some codes are included in the modal group that surrounds them. But most of the modal groupings make sense in that only one state can be active at a time.」という記述がある。

一方、http://linuxcnc.org/docs/html/gcode/g-code.html#gcode:g92 によると

「G92 offsets may be already be in effect when the G92 is called. If this is the case, the offset is replaced with a new offset that makes the current point become the specified value.」

とあり、G92 はモーダルコマンドの定義に当てはまる。しかし、G92 は非モーダルコマンドに分類されている。

Chap3　シリアル通信

Grbl では、RS232C 等による全二重シリアル通信を想定して設計してある。しかし、現在 PC から RS232C 端子は消滅しているのが実情で、このため USB-シリアル変換を用いたハードウエアが現実的である。そこで本章でも USB シリアル変換 IC(FT232) を使用したボード使用を前提とする。このような USB コネクタが実装済みボードは通信制御ソフトからは COM ポートに対する標準的通信となる。

なお、USB シリアル変換ボード使用では、PC 側にデバイスドライバインストールの必要があるが、これは Web 上で容易に入手できる。本章でも、このデバイスドライバは導入済みと仮定する。

3.1　シリアル通信

Grbl としては、通信速度は 115.2Kbps 程度以上を想定しており、ここでは 115.2Kbps による USART 通信とする。Atmega165P[1] の USART は、プロセッサの処理とは別に実行されるので、PC 側から 1 ブロックの G コードデータを送り込むと、プロセッサはそれを 1 バイトづつ受信、1 ブロック受信完了でパージング開始、切削プラン構造体を構成する。切削プラン構造体は、実際の x、y、z 軸の動き（速度含む）を時系列に記述したもので、これにより切削動作が実行される。1 ブロックは G コード規格により、最大 256 ワードとなり、かつ切削プラン構造体の全体容量が限られるので、1 ブロックが複数の動作に分割される場合もある。

Atmega165P ではシリアル通信に UDR レジスターを使う。このレジスターは送受信同名称であるが、ハード的に別のレジスターが存在する様に振る舞う [2]。

ここでは送受信とも割込み処理について述べる。

3.1.1　受信制御

ISR(USART0_RX_vect) 割込み関数を使用した非同期通信である。Atmega165P には USART 通信割込み操作のための設定レジスターとして、UCSR0B が用意されている。

[1] Grbl では Atmega328P(16MHz) を想定しているが、ここでは実際に使用した Atmega165P の場合として記述する。

[2] 送信のため UDR にデータをストア、それを読んでも、同じデータとはならない。

```
UCSR0B
 Bit   7      6      5      4     3     2      1      0
       RXCIE0 TXCIE0 UDRIE0 RXEN0 TXEN0 UCSZ02 RXB80  TXB80

UCSR0A
 Bit   7      6      5      4     3     2      1      0
       RXC0   TXC0   UDR0   FE0   DOR0  UPE0   U2X0   MPCM0
```

RXEN0=1 とすると、UART による受信を可能とし、RXCIE0=1 とすると、受信完了で UCSR0A のフラグ RXC0 がセット (=1) され、受信割込み処理の起動に使用できる。RXC0 は受信バッファ(UDR) が読みだされ、空になると自動的にリセットされる。

RXCIE0 は AVR ヘッダーファイルで、RXCIE0=7,RXEN0=4 と定義され、これらを設定するには、例えば次の記法が使える。

```
UCSR0B |= ((1 << RXCIE0)|(1 << RXEN0));
```

3.1.2 送信制御

割込みベースの送信制御のフラグとして UDRIE0 と TXC0 がある。どちらも割込みを発生させるが、以下の様な使い方となる。

(1) UDRIE

TXEN0=1 とすると、UART による送信を可能とする。最初の送信データを UDR にセットした後に、UDRIE0 をセットすると、UDR 空き割込みが有効となる。以後は、UDR が空くたびに、UDRIE0 ビットがセットされ割込みルーチンが呼ばれる。このルーチンの中で次のデータを UDR に送るが、割込みルーチンが新たにデータをセットしないと、割込みがクリアされないので、データの無い時はこの割込み処理内で、割込みをディスエーブルする。

割込み内で使用する変数には volatile 宣言をして、割込み内での変数の変更が確実に反映される様にする。

(2)TXC0

RS-485 による半二重通信の様に、送信と受信でハードウエアを切り替える必要のある時などを想定している。TXCIE0=1 とすると、送信完了で UCSR0A のフラグ TXC0 がクリア (=0) され、送信完了割込み処理の起動に使用できる。従って、送信完了割込みで、送受信ハードウエアを切り替える事ができる。TXC0 フラグは、送信完了割込み処理起動または”1” を書き込むとクリアされる。UDRIE0 割込みと同時に管理する必要がある。

Grbl では送信、受信ともリングバッファを用いる。この方法では、送信データはリングバッファにストア、UDR 空き割込みでリングバッファの内容を送信し続ける。リングバッ

ファが空になっても送信してしまうのを防ぐため、バッファが空になった時点でUDRIE0をクリアする。リングバッファの構造と操作については第1.3.1節参照。リングバッファの操作時は必ず割込み禁止とする事。

3.2 serial.c/h

ファイルserial.cがシリアル通信に使用される関数群を集めたファイルである。これと、gcode.c中のソースコードを参考にしてGrbl中のシリアル通信について述べる。Grblの中では分かり易いファイルで、Gコードがどこで処理されているかが分かれば問題は無いであろう。

UART0を使用した標準的なシリアル通信である。通信速度は115Kbpsで、128バイトの受信リングバッファを持つ。送信機能も用意してあり、送信リングバッファサイズは112バイト（ライン番号を使用しないならば104バイト）である。

使用する関数群と変数は以下の通りである。

```
ISR(SERIAL_RX)
uint8_t serial_rx_buffer[RX_RING_BUFFER];
uint8_t serial_rx_buffer_head = 0;
volatile uint8_t serial_rx_buffer_tail = 0;
uint8_t serial_tx_buffer[TX_RING_BUFFER];
uint8_t serial_tx_buffer_head = 0;
volatile uint8_t serial_tx_buffer_tail = 0;
void serial_write(uint8_t data);
uint8_t serial_read();
```

3.3 ISR(SERIAL_RX)

AVRに用意されたシリアル通信用インターラプトを使用した関数である[3]。USART0のRx Complete時に呼ばれるインターラプト処理関数で、Grblではここで、リアルタイム制御用コマンドとGコードの切り分けも実行している。

以下その構造を示す。

```
ISR(SERIAL_RX){
  switch (data) {
    case CMD_RESET:          システムリセット
    case CMD_CYCLE_START:    システム再スタート
    case CMD_FEED_HOLD:      システムポーズ
```

[3]この呼び方はAtmelStudioのバージョンによって異なり、AVR Studio 7ではUSART0_RX_vect_numである。

```
     -----------------
    default :
      if (data > 0x7F) {       リアルタイムコマンド
        switch(data) {
            ----------------
          case CMD_JOG_CANCEL:
            ----------------
          case CMD_FEED_OVR_RESET:
          case CMD_COOLANT_MIST_OVR_TOGGLE:
           -----------------
        }
     } else {       この部分がGコードで、serialリングバッファに書き込む
         ------------------
     }
  }
}
```

CMD_**で表されるのは、システムコマンドで、G コード内では使用されない文字（例えば CMD_RESET は ctrl-x）と ASCII コードで 0x7F 以上の値にアサインされている文字を使用している。これらのシステムコマンドを受信したならば、システム制御動作を実行する事になるが、この割込み処理ルーティンの中では、フラグを設定するだけにして、割込み処理時間を短くしている。

上記関数内では、「G」や「M」という文字が一切見えないがこれは else{ } 内で処理され、リングバッファ―に書き込まれる。リングバッファ―の長さ 128 バイトは G コードの規格による。

システムコマンドはソースコードを見ると、JOG モード関連、切削クーラントミスト（オイル）、デバッグ用状態送信機能等様々なものが用意されているが、これらは当然実ターゲット装置の状況を反映するものとなる。

Chap4　Gコード処理

この章は gcode.c の内容を解析したものである。

4.1　概要

パーソナルコンピュータから Grbl に送られる G コードは 128 バイトのリングバッファ serial_rx_buffer[ ] にストアされる。このシリアルデータがバッファにストアされると、直ちに G コード以外のコマンドやシステム制御用コマンドはパージング（単にキャラクターを読むだけ）され、（G コード以外による）システム直接制御に使われる。

オリジナルの Grbl では、リングバッファにストアされた G コードをパージング、エラー検出を行い、エラー出力を行う。しかし CNC 実行時にエラーが多発するようでは使用不能となってしまうため、ここではエラー検出部分については別プログラムに分割するとして述べる。

gcode.c の処理構造は、コンパイラーに例えると、2 パス方式を 1 パスとした構造と考えられる [1]。つまり、入力された G コードは一通り解釈しただけでは、コード実行情報が完成されず、再度解釈し、実行可能な状態に仕上げる。

一方、G コードコマンドには non-modal と modal コマンドが定義されていて (第**??**節参照)、modal コマンドはマシンのモードを変化させ、別のコマンドでモードが再設定やリセットされるまで保持せねばならない [2]。このために gc_state 構造体を用意する。non-modal コマンドについてはは 1 ブロック（1 ライン）中のみ有効で、これは gc_block 構造体にデータを保持する。

CNC 動作は gc_block 構造体をデータベースとして使用するので、1 ブロックのパージング収集を完了した時点で gc_state のデータをコピーし、内容を更新する。

これらの操作はおよそ、最初の部分でコマンドのパーズ、gc_block 構造体に必要情報をストアし、次の部分で gc_state 構造体に座標情報などのモーダル情報をストアするというプロセスとなる。

[1] このため、同じ変数による switch() 分が複数回出現する。一見、一まとめに出来そうに見えるが、2 パスを展開して 1 パスとした訳である。

[2] 保持されるのはコマンドであって、コマンドの結果の情報変化（例えば座標オフセット）は保持されるものも、保持されないものもある。ここが非常に誤解を招きやすい。

4.2　G コードパージング

serial_rx_buffer[ ] の内容は 1 ブロックのパージング完了で、gc_block 構造体にその結果がストアされる。この時点で G コードの情報は整理されたが、システムの現状との整合性は取れていない。例えば、軸移動命令の場合、移動ステップ数は示されても、現在位置は送られて来ないので、具体的位置を計算するには現在位置、そしてさらにその元となる座標系の情報の整理が必要である。

そのため、gc_block 構造体にデータストアが完了すると、gc_state 構造体に、座標データなどを更新しながら、必要データを揃える。この部分は一まとめにした処理となっており、インターラプトを除き、連続処理である。

G コードの処理が終了すると、motion_control.c 中の直線補間関数

```
mc_line(gc_state.position, gc_block.value.ijk, N_AXIS*sizeof(float));
```

を呼ぶが、mc_line() の中からさらに plan_buffer_line() が呼ばれる。

4.3　G コードパージングプロセス

G コードのエラーチェックは、2 つの基本的考えに基づいて実行する。

(1) G コード 1 行中には同一の動作モードグループ（modal group) コマンドは 1 つしか記述しない。

(2) 異なる動作モードグループコマンドでも軸移動に関するコマンドは排他である。

これらの構造は、1 行を実行中は、CNC 装置のスピンドルが実動作中で、その間に矛盾するコマンドがあっては困るという考えからの規制と考えられる [3]。

4.3.1　G コードエラーチェック

G コードは、以下のグループに属するコマンドは座標が付随し、お互いに排他的となる。

G10, G28, G30, G53, G92　(MODAL_G0)

G0, G1, G2, G3　(MODAL_G1)

G43.1, G49　(MODAL_G8)

以下に示す【例】は、コマンドの排他性の観点からは、この記述も可能と思われるものである。しかし、実用上は 2 行に分けて書かれる。（第 4.7 節参照）

【例】

G10 L2 P1 X3.5 Y17.2　G1 X2.0 Y 5.8

[3] G コード全体を 1 行と考えれば、そのシーケンス中にはすべてのコマンドが使用可能である。

G10 L2 P1 X3.5 Y17.2 により P1 の指定する座標 (G54) が X=3.5mm, Y=17.2mm オフセットされる。一方、G1 X2.0 Y 5.8 は軸を現在位置から (2.0, 5.8)mm 移動するコマンドとなるが、この現在位置が G10 L2 を実行する前なのか、実行後なのかはっきりしない[4]。

	word_bit	axis_command	axis_word	ijk_word
G10 L2	MODAL_G0	NON_MODAL(=1)	yes	-
G10 L20	MODAL_G0	NON_MODAL(=1)	yes	-
G30	MODAL_G0	NON_MODAL(=1)	yes	-
G92	MODAL_G0	NON_MODAL(=1)	yes	-
G28	MODAL_G0	NON_MODAL(=1)	yes	-
G53	MODAL_G0	NON_MODAL(=1)	yes	-
G28.1,G30.1	MODAL_G0	NON_MODAL(=1)	-	-
G92.1	MODAL_G0	NON_MODAL(=1)	-	-
G4	MODAL_G0	NON_MODAL(=1)	-	-
G0	MODAL_G1	MOTION_MODE(=2)	yes	-
G1	MODAL_G1	MOTION_MODE(=2)	yes	-
G2, G3	MODAL_G1	MOTION_MODE(=2)	yes	yes
G38	MODAL_G1	MOTION_MODE(=2)		
G80	MODAL_G1	-	-	-
G17,G18,G19	MODAL_G2	-	-	-
G90,G91	MODAL_G3	-	-	-
G91.1	MODAL_G4	-	-	-
G93,G94	MODAL_G5	-	-	-
G20,G21	MODAL_G6	-	-	-

[4]理論的に決定不能ではなく、どうするか定義すればいい。

```
G40                 MODAL_G7       -                -              -

G43.1,G49           MODAL_G8   TOOL_L_OFFSET(=3)    -              -

G54 - G59           MODAL_G12      -                -              -

G61                 MODAL_G13      -                -              -

-------------------------------------------------------------------------
                    word_bit    axis_command    axis_word     ijk_word
-------------------------------------------------------------------------
M0,M1,M2,M30        MODAL_M4       -                -              -

M3,M4,M5            MODAL_M7       -                -              -

I,J,K               WORD_I,J,K     -                -            set

L,N,P               WORD_L,N,P     -              -                -

R,S,T               WORD_R,S,T     -              -                -

X,Y,Z               WORD_X,Y,Z  MOTION_MODE(=2)   set              -
```

4.3.2 G コードエラーチェックアルゴリズム

Gcode.c の中で、G コードの排他チェックが実行されるが、そのアルゴリズムは次の様になっている。

コマンドはグループに分けられるが、このグループに属するものはGコード 1 行に一つしか許容できない。そこで、G コード 1 ラインをパーズする間に、どのグループのコマンドが存在するかを示す 16 ビットのビットマップ、command_words を用意し、各グループに属するコマンドを検出する毎に相当するビットを立てる。

word_bit は 0 から 14 どれかの値を取る。command_words は 16 ビットメモリで、そのどのビットが 1 かを bit(word_bit) で指定する。

```
if ( bit_istrue(command_words, bit(word_bit)) ) {
    printf(STATUS_GCODE_MODAL_GROUP_VIOLATION);
}
```

```
// ここで、word_bit が指示する command_words のビットを 1 にする。
command_words |= bit(word_bit);
```

最初は command_words は未設定である。G コード 1 ラインは各文字（とそれに続く数字）毎にパーズされ、コマンドを検出すると、command_words に対応するビットを立てる。従って、次の G コードコマンドを検出した時に、すでに command_word に記録されたビットを再度 word_bit により指定しようとすると、

bit_istrue(command_words, bit(word_bit)

が True となり、1 ブロック中に同じグループのコマンドが再出現した事が検出される。もちろん、別グループに属するコマンドが G コード 1 行中に存在するだけでは排他性は検出されない。

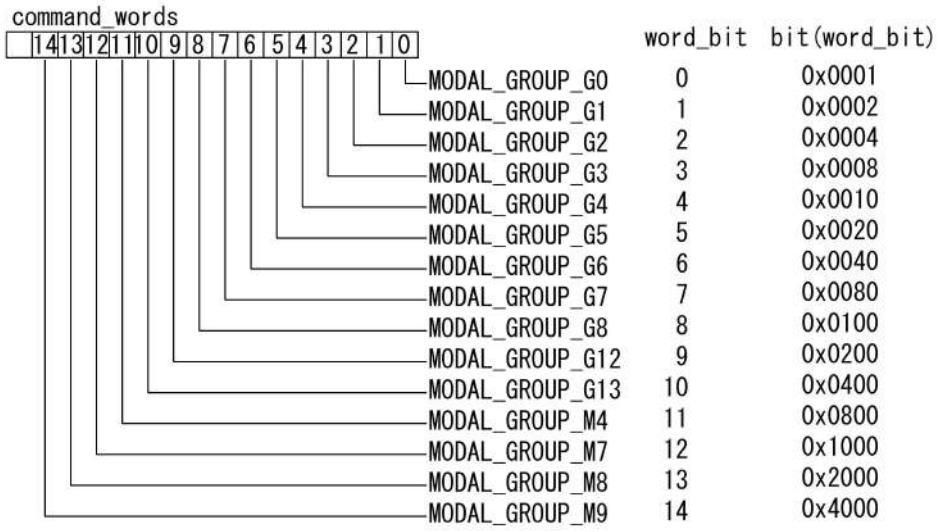

Figure 4.1: command_word と word_bit の関係

4.3.3 コマンド分類

G コードは設計が古く、コマンド体系が整理されていない。つまり、コマンドの直交性が悪く、同じ動作を複数のコマンドで可能であったり、コマンドに対するパラメータの意味がコマンドにより変化するなどパージングそして実行制御に工夫が必要である[5]。この典型的例は軸移動を、G コード＋ X,Y,Z パラメータで実行するだけでなく、X,Y,Z パラメータ単独でも実行できる点であろう。

この様に使い難いコマンドをパージングして、実行するために、まず G コードコマンド[6] をモーダルグループと非モーダルグループに分類する。モーダルグループは、コマン

[5] このため実質的に 2 パスパージングとなっている。
[6] 広い意味の G コードで M、L コードを含む。

ドが次の行以降でも有効なもの、非モーダルグループコマンドは、1 度だけ有効となるものである。モーダルグループの例として、G1 の様な直線補間コマンド、非モーダルグループとして、G28 のようなマシンホームポジション復帰を考えれば、違いが分かり易い。（第 2.2.1 節参照）

Grbl のサポートするコマンドに関しては次のように分類され、各グループ内のコマンドは、排他的である。

【Modal Group】	【Member コード】
Motion Mode:	G0, G1, G2, G3, G38, G38.x, G80
Coordinate System Select:	G54, G55, G56, G57, G58, G59
Plane Select:	G17, G18, G19
Distance Mode:	G90, G91
Arc IJK Distance Mode:	G91.1
Feed Rate Mode:	G93, G94
Units Mode:	G20, G21
(Cutter Radius Compensation):	G40
Tool Length Offset:	G43.1, G49
Program Mode:	M0, M1, M2,M30
Spindle State:	M3, M4, M5
(Coolant State):	M7, M8, M9

【Non Modal Group】

G4, G10 L2, G10 L20, G28, G30, G28.1, G30.1, G53, G92, G92.1

4.3.4 プログラム初期化

Grbl では G コードコマンドパージングの結果を gc_block 構造体に収納し、gc_state 構造体で、現在の動作状況を追跡する。例えば、gc_block 構造体には G コードパージング結果の座標移動量がストアされ、これはパージング完了時に、gc_state 構造体の座標値に反映される。そのため、データベースとして単一の構造体ではなく、2 つの構造体を持つ[7]。

以下、gc_block、gc_state 構造体のメンバーを示すが、どちらとも簡略化したものを示す。また分かり易くするため、構造体に入れ子となっている構造体も示す。

```
typedef struct {
 uint8_t              non_modal_command;         //{G10,G28,G30,G92,G4,G53}
 gc_modal_t  modal; --->typedef struct {
                            uint8_t   motion;      //{G0,G1,G2,G3,G80}
                            uint8_t   radius_mode; //{R}
                            uint8_t   distance;    //{G90,G91}
```

[7] gc_modal_t 構造体を共通に含み、重複している様に見えるが、これは分離した 1 個の構造体。

```
                        uint8_t   coord_select;//{G54 - G59}
                        uint8_t   program_flow;//{M0}
                        uint8_t   spindle;     //{M3,M4,M5}
                      } gc_modal_t;
 gc_values_t values; -->typedef struct {
                        float     f;           //実際の速度値
                        float     ijk[3];      //I,J,K 円弧オフセット
                        uint8_t   l;           // G10
                        int32_t   n;           // Line 番号
                        float    p;            // G10 or dwell 値
                        float    r;            // Arc radius
                        float   xyz[3];        // X,Y,Z 座標
                      } gc_values_t;
 } parser_block_t    gc_block
```

gc_state 構造体は次のパラメータより構成され、gc_modal_t 構造体をメンバーとして持つ。

```
typedef struct {
 gc_modal_t  modal; --->typedef struct {
                        uint8_t   motion;       //{G0,G1,G2,G3,G80}
                        uint8_t   radius_mode ; //{R}
                        uint8_t   distance;     //{G90,G91}
                        uint8_t   coord_select; //{G54 - G59}
                        uint8_t   program_flow; //{M0}
                        uint8_t   spindle;      //{M3,M4,M5}
                      } gc_modal_t;
  uint8_t              non_modal_command  //
  int32_t               line_number;
  float                 position[N_AXIS];        // machine position
  float                 coord_system[N_AXIS];    // work position
  float                 coord_offset[N_AXIS];    // G92 用オフセット
} parser_state_t   gc_state
```

プログラム初期化ではこれらの構造体の初期化と、ローカル変数の初期化を行う。特に次のローカル変数の使い方に気を付ける必要がある。

【axis_command】

変数であり、次の値ををを取る。

AXIS_COMMAND_NONE (=0)

AXIS_COMMAND_NON_MODAL　　(=1)
AXIS_COMMAND_MOTION_MODE　(=2)

G コードの排他性チェック（第 4.3.1 節）および、X,Y,Z のみが与えられた時に、軸移動を示すために用いられる。

【axis_words】

フラグであり、ビットとの対応は以下の通り。

```
bit  7   6   5   4   3   2   1   0
                         Z   Y   X
```

【ijk_words】

フラグであり、ビットとの対応は以下の通り。

```
bit  7   6   5   4   3   2   1   0
                         K   J   I
```

4.3.5　1 ブロック中の G-code 読み込み、解析

G コードに続く整数値をパーズして、非モーダルコマンドを以下の様に分類する。

gc_block.non_modal_command　　G10, G28, G28.1, G30, G30.1, G92, G92.1

この時、G28.1 の様に小数点を持つコマンド（拡張コマンドであり、もともとの G コードに定義してあったものと Grbl オリジナルのものがある）は、他のコマンドと重複しない適当な整数に変換する。具体的には小数部を 10 倍して整数部に加え、新しい整数を生成してコマンドのインデックスとする。例えば G28.1、G30.1、G92.1 に対しては、それぞれ 38、40、102 がアサインされる。

この部分のソースコードは以下の様である。

```
gc_block.non_modal_command = int_value;
     if ((int_value == 28) || (int_value == 30) || (int_value == 92)) {
         // mantissa=10 なので、G28.1=38,G30.1=40,G92.1=102 を生成。
         gc_block.non_modal_command += mantissa;
         // Set to zero to indicate valid non-integer G command.
         mantissa = 0;
     }
```

モーダルコマンドについては、さらに詳細に分類し、以下の様にストアする。

gc_block.modal.motion： G0, G1, G2, G3, G38, G80
gc_block.modal.distance： G90, G91
gc_block.modal.feed_rate： G93, G94
gc_block.modal.tool_length： G43, G49
gc_block.modal.coord_select： G54-G59
gc_block.modal.control： G61
gc_block.modal.program_flow： M0, M2, M30
gc_block.modal.spindle： M4, M5
gc_block.modal.override： M56

その他、F, I, J, K, L, N, P, R, X, Y, Z ワードについても値を gc_block 構造体にストアする。

ここまでで、1 ブロックの G コード全般のパージングと、構造体 gc_block のパラメータ設定が完了した。gc_block、gc_state 構造体のデータ構造を見ると、1 座標しかストアできない。これは、1 ブロックには複数の座標移動コマンドは記述できない事を意味する。また複数の共存可能 G コードはストア可能であるが、その記述順序は失われる。詳細は第 4.5 節の実例を参照。

4.3.6 gc_block 構造体のデータ更新

1 ブロック分の G コード及びパラメータが準備完了したが、これではまだ、G コードの「意味」が理解できたことにはならない [8]。つまり、G コードによって同じパラメータでも、別の意味に解釈しなければならない場合および座標解決が未了である（G コードは簡単なコンパイルを必要とするインタープリタ言語という趣を持つ）。

以下、この点について詳述するが、この部分のプログラムとしては、次の準備から開始する。

【X,Y,Z の処理準備】

(1) X,Y,Z コードはこれだけで軸移動コマンドになるので、これを反映。
具体的には、X,Y,Z コード検出では axis_word のフラグがセットされるが、
axis_command=0 となっているので、以下の様に設定。
axis_command = AXIS_COMMAND_MOTION_MODE
(2) ローカル変数 block_coord_system[ ] に gc_state.coord_sytem の座標情報を設定。
(3) gc_state.modal.coord_select に使用する座標系を設定。

[8] コンパイラならば、ソースコードの意味を解釈しただけで、具体的アドレス解決が未了状態。

つぎに、G10,G28,G30,G92 はそれぞれパラメータの扱いが異なるので、個別にパラメータを処理していく。

【Non Modal Commands (G10/G92 および X,Y,Z）の処理】

(1) G10; パラメータの解釈
サポートする G10 のシンタックスは、L2 または L20 と P を必ず伴い、P は 0 から 6 であり、次のように解釈される。
G10 L2 P ; 座標軸原点オフセット設定。P1 から P6 は G54 から G59 に対応する。
［例］G10 L2 P1 X3.5 Y17.2 ; 第一座標系 (G54) の原点を、 X3.5 Y17.2 に設定。
G10 L20 P　 ; 座標原点オフセット設定。
［例］G10 L20 P1 X1.5; 第一座標系 (G54) の原点の計算に X1.5 を代入
(2) G92; 座標オフセット解決
すべての座標系 (G54-G59) を (実際の軸移動はさせずに) オフセットさせる。
G92.1 によりリセットされるまで有効。
(3) X,Y,Z; 座標移動コマンドとして解釈
X,Y,Z コード検出で、フラグ axis_words が設定されている。
対応する X,Y,Z[9] の値を gc_block.value.xyz[ ] に反映。

【Non Modal Commands (G28,G30,G28.1、G30.1,G92.1,G53) の処理】

(1) G28,G30; 複合コマンド処理
G28,G30 は、X,Y,Z が指定されている時は、その値まで移動し、その後
それぞれ G28.1,G30.1 にストアされた位置に戻る。
一つのコマンドで、この様な複合動作をするので、これに対応 [10]。

【Modal Commands (G0,G1,G2,G3 の実行準備) 】

(1) G10,G28,G30,G92 を排除
(2) G0; 最高速軸移動。
(3) gc_block.modal.motion に設定されているコマンドのデータ準備
G1 については特に何もしない。
G2,G3 については時計回り、半時計まわりの設定および、次の準備計算を実行。

また G コードの最終的準備のためには、円弧に対する計算や arctangent が必要となる。arctangent は関数として用意されているが、その予備計算の意味を以下に記しておく。

[9] X,Y,Z は必要なものだけあればいい。
[10] これも G コードの良くない設計。

【円弧の計算】

円弧の描画には、二つの場合がある。

(1) 半径および現在座標、ターゲット座標が与えられる場合。

gc_block.values.r：　　半径

gc_state.position[ ]：　　現在座標

gc_block.values.xyz[ ]：　ターゲット座標

これらの情報から中心座標を計算する。この場合、中心座標は 2ヶ所ある。

x 軸の距離は、

x = gc_block.values.xyz[axis_0] - gc_state.position[axis_0]

y 軸の距離は、

y = gc_block.values.xyz[axis_1] - gc_state.position[axis_1] となる。

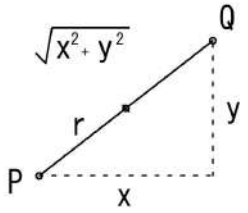

Figure 4.2: 円弧の条件

P を現在位置、Q をターゲット位置として、PQ を通過する円弧について、

$$\sqrt{x^2+y^2} < (2r)^2$$

とならないと円弧が描画できない。これをチェックするため、

$$C_1 = 4 \cdot r^2 - x^2 - y^2$$

を計算し C_1 が負の場合はエラーとする。

第 7.3 節の計算より、円弧の中心座標 (x_r, y_r) は、

$$x_r = \frac{1}{2}\left\{x \pm y\frac{\sqrt{4r^2-(x^2+y^2)}}{\sqrt{x^2+y^2}}\right\}$$

$$y_r = \frac{1}{2}\left\{y \mp x\frac{\sqrt{4r^2-(x^2+y^2)}}{\sqrt{x^2+y^2}}\right\}$$

となる。そこで、

$$C_2 = \frac{\sqrt{4r^2-(x^2+y^2)}}{\sqrt{x^2+y^2}}$$

を別途計算しておくと、

$$x_r = \frac{1}{2}\Big(x \pm y_d C_2\Big)$$

$$y_r = \frac{1}{2}\Big(y \mp x_d C_2\Big)$$

として、円弧中心座標が計算できる。

(2) 円弧の中心が与えられた場合。

gc_block.values.ijk[ ]： 円弧の中心座標
gc_state.position[ ]： 現在座標
gc_block.values.xyz[ ]： ターゲット座標

円の中心からターゲット座標までの距離 x は、

$$x = gc\_block.values.xyz[X] - gc\_state.position[X] - gc\_block.values.ijk[X]$$

円の中心からターゲット座標までの距離 y は、

$$y = gc\_block.values.xyz[Y] - gc\_state.position[Y] - gc\_block.values.ijk[Y]$$

半径 r は

$$r = gc\_block.values.r = \sqrt{gc\_block.values.ijk[X]^2 + gc\_block.values.ijk[Y]^2}$$

で計算される。

4.4 CNC 動作実行

gcode.c ソースファイルでは CNC 動作実行までが記述されているが、G コード処理と軸移動が別プロセスとなるような設計を反映し、gc_state 構造体のデータ更新を実行するものと、直線、曲線補間関数を起動するものとに分けられる。

(1) gc_state 構造体の情報更新。

これは、基本的に軸移動の前提条件を変更するので、$system\_flag\_wco\_change()$ により、現在実行中の軸移動が完了するまで待つ。

(2) 軸移動。

$mc\_line()$、$mc\_arc()$ 関数で実行される、これらは plan_buffer に線素として、動作条件を記入する。これは、さらに segment_buffer にコピーされ、軸移動となる。

CNC 動作は次の様にまとめられるが、これらの実行に当たって、上記の付帯処理が必要である。

(1) Non modal CNC 動作 (G10, G28, G30, G28.1, G30.1, G53, G92, G92.1)

(2) Modal CNC 動作 (G0, G1, G2, G3)

(3) プログラム一時停止、終了 (M0, M2)

CNC 動作による軸移動は、最終的にすべて直線移動に還元される。円弧も、（十分に細かい）直線補間の組み合わせで表現される。特に、円弧や、曲線などは細かい多数の直線区間の移動となるので、非常に多くの直線補間操作の集合となり、この部分は planner.c で記述される。

4.5 G コード例

以下は複合サイクル (Canned Cycle) を使用して、多数のドリリングを実行する G コードの例である。

この例から次の動作が読み取れる。

(1) 1 行に複数の G コードが有る場合、それらは実軸移動を伴わない。

(2) 最初にマシンの動作条件を設定。

(3) （安全な位置にセットした後）スピンドル ON

(4) 必要ならば、設定の念押し。(下の例では G90)

(5) 複合モード設定。

(6) 移動は x、y 座標設定によるが、これらはすべて別の行（ブロック）とする。

(7) 完了前にキャンセルすべきものは処理。(複合サイクル)

(8) スピンドル OFF, スピンドル位置を基に戻す。

(9) プログラム完了。

```
------------------------------------------------------------
G0 G49 G40  G17 G80 G50 G90
; RapidMove, ToolLengthCompCancel, CutterRCompOff, SetG54,
; CannedSycleCancel, SetOrigin, DistanceMode

M6 T0
; ToolChange, Tool_0

G20 (Inch)
; Unit _inch

M03 S500
; Spindle ON, Speed500rpm

M08
; CoolantON
```

```
G90
; DistanceMode

G00 G43 H0  Z0.1
; RapidMove,  ToolLengthOffset, Tool_0, Zaxis_0.1mm

G73 X-7 Y-4 Z-0.75 Q0.2 R0.1 F10
; DrillingCannedCycle, X_-7mm, Y_-4mm, Z_-0.75mm,
; Q(切り込み量)_0.2mm,  R(Z 軸戻り位置)_0.1mm, F(速度)_10mm/sec

(以下、X,Y の位置指定)
X-5.8889
X-4.7778
X-3.6667
X-2.5556
X-1.4444
(省略)
X-1.4444
X-0.3333
X0.7778
X1.8889
X3

G80
; CannedSycleCancel

M5 M9
; StopSpindle, CoolantOff

X-7  Y-4
; Move X_-7mm, Y_-4mm

M30
; ProgramEnd
-------------------------------------------------------------
```

Grbl による G コードパージングでは、1 ライン中に複数の G コードがある場合でも、その順序は管理していない。つまり、gc_block、gc_state 構造体に 1 ラインの G コードや、付随するデータは記述されるが、これは 1 ライン中に出現する順序は保持しない。また、

座標も 1 情報しか保持しないので、1 ライン中に複数回の軸移動は記述できない。これは、軸移動を伴う動作は 1 ライン中 1 コマンドしか記述できない事を意味する [11] が、この点をエラーチェックしている訳である。

4.6 ストリング数値の浮動小数点数値への変換

Grbl の G コード処理中に、uint8_t read_float(char *line, uint8_t *char_counter, float *float_ptr) 関数が使用される。

これは Grbl が実装される CNC 装置には、外部より制御用 G コードがすべてストリングとして送られてくるため、数値を浮動小数点数に解読しなおす必要がある事による。しかし、この関数の処理が難しい [12] ので、ここでその内容について述べる。

もとのソースコードは nuts_bolts.c 中に存在する。

4.6.1 数値の+/-判定

この関数は、引数として変換するデータの位置（ストリングで与えられた数字の先頭）を必要とする。それが*line, *char_counter である。

数字の先頭には-符号は必要であるが、+符号は通常省略される。しかし、+符号付きの場合も処理にいれ、逆に+符号無しの場合は、処理不要のため、明示的には何も書かれていない。

```
uint8_t read_float(char *line, uint8_t *char_counter,
    float *float_ptr){
    char *ptr = line + *char_counter;
    unsigned char c;

    // Grab first character and increment pointer. No spaces assumed
    // in line.
    // 数字を意味するストリングの最初を読み込む。+、-、0-9 の数値
    c = *ptr++;

    // Capture initial positive/minus character
    bool isnegative = false;
    if (c == '-') {
        isnegative = true;
        //ここは ptr のインクリメントが目的。*ptr++; だけで OK。
```

[11] G コードのルールであろう。

[12] 処理そのものの難しさに、マクロプロセッサ用に処理を軽くしている点が加わっている。

```
    c = *ptr++;
} else if (c == '+') {
    //ここは ptr のインクリメントが目的。*ptr++; だけで OK。
    c = *ptr++;
}
// 普通の正の数字はここで処理される。ptr のインクリメント無し。
// この部分省略されているので、全体の処理がつかみにくい。
// else NOP 相当;
```

4.6.2 数値変換

ここでは、ストリング数字を整数値 (intval) と、小数点位置を示すべき乗値 (exp) に変換する。

```
// Extract number into fast integer. Track decimal in terms of
//exponent value.
uint32_t intval = 0;
int8_t exp = 0;
uint8_t ndigit = 0;
bool isdecimal = false;
while(1) {
    c -= '0';
    if (c <= 9) {
        ndigit++;
        if (ndigit <= MAX_INT_DIGITS) {
             if (isdecimal) { exp--; }
                // x10 を μプロセッサ処理様にプログラム。
                // intval*10 + c
                intval = (((intval << 2) + intval) << 1) + c;
        } else {
            // Drop overflow digits
             if (!(isdecimal)) { exp++; }
        }
    // ここでは小数点'.' を検出。
    // '.' = c- '0' より、c = '.'-'0' となれば小数点。
    } else if (c == (('.'-'0') & 0xff)  &&  !(isdecimal)) {
        isdecimal = true;
    } else {
        break;
```

```
        }
       // ここでは、次のストリングを読み、かつポインターを進める。
       // 従って *ptr++だけではNG。
        c = *ptr++;
    }
    // Return if no digits have been read.
    if (!ndigit) { return(false); };
```

上記プログラム中の

```
   else if (c == (('.'-'0') & 0xff)  &&  !(isdecimal)) {
            isdecimal = true;
   }
```

の部分は、検出した文字が数字 (0-9) 以外に対する else if なので、この条件中に!(isdecimal) という余計な条件を入れるのは、論理的におかしい。
以下の様にするべきであろう。

```
   else if (c == (('.'-'0') & 0xff))) {
       if( !(isdecimal)){
            isdecimal = true;
       }
   }
```

0xff によるマスクは c が 1 バイトとするため、if(!(isdecimal)) は、小数点が 2 回出現してしまうのを排除する。しかし、この場合もエラー処理をすべきだろう。

4.6.3 浮動小数点への変換

浮動小数点ストリングを浮動小数点数値に変換。ストリングが整数でも、相当する浮動小数点値に変換。

```
    // Convert integer into floating point.
    float fval;
    fval = (float)intval;
    // Apply decimal. Should perform no more than two floating point
    // multiplications for the expected range of E0 to E-4.
    if (fval != 0) {
        while (exp <= -2) {
            fval *= 0.01;
            exp += 2;
```

```
    }
    if (exp < 0) {
        fval *= 0.1;
    } else if (exp > 0) {
        do {
            fval *= 10.0;
        } while (--exp > 0);
    }
  }

  // Assign floating point value with correct sign.
  if (isnegative) {
      *float_ptr = -fval;
  } else {
      *float_ptr = fval;
  }
  // Set char_counter to next statement
  *char_counter = ptr - line - 1;
  return(true);
}
```

4.6.4 数値変換の結果

ストリングから数値への変換結果は以下の様であり、+/-サイン、小数点を含まず、上位 8 桁までは正しく変換できている。CNC 機器用として問題なく使用できる。また浮動小数点の表示で非常に小さな桁に 1 が出現しているが、これは浮動小数点自体のコンピュータ処理によるものである [13]。

入力数字	出力数字
12345678	12345678.0
12345678901	12345678000.0
- 12345678901	-12345678000.0
-12345678	-12345678.0
123.45678	123.45678000000001
-123.45678	-123.45678000000001
0.12345678	0.1234567
-0.12345678	-0.1234567

[13] 何かのバグでは無い

4.7 注

【G コードコマンド排他性について】

G コードのコマンドは、通常 1 行に 1 コマンドの記述となっている。これは恐らく、経験から生まれた習慣であるが、この様な記述が徹底されれば 1 行の排他性チェックは不要である（それでも、単純な記述ミスには有効）。

以下は想像であるが、もともとの G コードは複数のコマンドを 1 行に記述する事を許容する仕様であり、そのために排他チェックを考慮したのであろう。この排他チェックの基準となるのが、コマンドのグループ分類である。

ところが、G コードの解釈に幅があり、これを少しでも改善するためには 1 行、1 コマンド記述が適していて現在の習慣が生まれたのであろう [14]。

[14] Grbl の実装では 1 行 80 文字であるが、1 行 1 コマンドを仮定すると、ここまでのバッファは不要であろう。

Chap5　CNC切削プラン生成

G コードのパージングが完了、1 ブロックの切削用データベース gc_state が完成すると、直線補間ならば mc_line()、曲線補間ならば mc_arc() を呼ぶ。mc_arc() は最後に mc_line() を呼ぶ。

mc_arc() は曲線を線素に分割し segment 生成、その線素毎に mc_line() を呼ぶ。

結局すべての動作は、mc_line() に帰着、これが plan_block 構造体を生成、リングバッファplan_buffer に収納する。plan_buffer は構造体のリングバッファなので、あまり大容量にはできず、Grbl ではライン番号をサポートする時で、容量は 16 である [1]。

5.1　軸移動プラン生成

plan_buffer を用意する意味は、円弧補間を直線補間に置き換えるためと思われる。これらの直線は連続して描画されるが、精度の高い切削には、直線と次の直線の継ぎ目部分は、直線部分より軸移動速度を落とす動作を行う。これは、原理的には、直線の継ぎ目部分 [2] は加速度が無限大となり、物理機械では実現不能という矛盾があるため、これの軽減措置である。このセグメント間にまたがる処理速度を最適化する。即ち、G コード生成と plan_buffer へストアされたデータの間には、

(1) 軸移動をすべて直線補間で実行するので、（細かい）直線による連続化処理。
(2) 連続化処理した一つのセグメント内では、軸移動速度の最適化を実施。従って、
　現セグメント実行後に次のセグメントに移動する時、移動速度調整が必要。
という処理が入る。

このため、オリジナルの Grbl 動作では plan_buffer の内容は、全体として軸移動速度を最適化し、切削は plan_buffer が空になるまでを 1 行程として実行する。何かの理由で plan_buffer 途中で処理を中断する時には、再開時、再度最適化する。

ただし、この部分は簡易的な CNC マシンでは軸移動速度は常に（選択された）一定値とするなど、簡略化されるであろう。

軸移動プランは planner.c 中の plan_buffer_line() 関数が主となって生成する。ここでは、

[1] ライン番号非サポート時は 15。ライン番号サポートでリングバッファ容量を 1 増やす理由がはっきりしないが、ライン番号を 1…とするからかも知れない。
[2] ここで折れ曲がるはずなので。

この関数のパラメータは以下の様に簡略化して説明する。
(1) 直線補間の移動先位置 ： target[N_AXIS]、単位は mm。
(2) 軸移動速度：単位は mm/sec
(3) 行番号。

上記パラメータを基に、plan_buffer_line() 関数では次の動作を実行する。
(1) 切削動作のシーケンスをまとめて planner buffer（リングバッファ）に収納。
(2) 切削動作の速度を最適化するため planner buffer 中のセグメント間の、軸移動速度の最適化。(これを実装しない場合も多いであろう。)
(3) 切削実行により、planner buffer が空になったならば、再度バッファを満杯にする。

ここで、planner buffer の情報は、軸移動速度について最適化されているので、空になるまで切削実行、空になったところで再度データで満たすのが原則である。（途中での最適化処理も用意されている。）
従って、オリジナルプログラムでは、planner buffer は、実行速度最適化および計算と実動作を非同期としている。

5.2 軸移動時振動軽減

以下は stepper.c のソースコードで記述されているが、軸移動プランの中の機能と考えた方が適切と思われ、ここで記述する。
Grbl は Bresenham アルゴリズムを用いて、線形補間を実装すると記述されているが、通常言われている方法とは異なる。使用されているアルゴリズムについては第 6.2.1 節で詳述するが、ここでは「Grbl 直線アルゴリズム」と呼ぶ事にする。このアルゴリズムでは、単に整数計算でいいので計算コストを低くできる。しかしながら、Grbl 直線アルゴリズムでは、例えば x 軸を連続ステップ送りすると、y 軸は適当にスキップされ、いわば従属動作をする。このため、x, y 軸に干渉（ビート）が発生し、機械的運動により、振動や音を発生する事がある [3]。
この振動を軽減した、改善 Grbl 直線アルゴリズムを実現するため、Grbl は Adaptive Multi-Axis Step Smoothing (AMASS) アルゴリズムを用いる。これは低いステップ周波数では AMASS は明示的に Grbl 直線アルゴリズム解像度を、元々の正確さを犠牲にする事なく上昇させる。AMASS はステップ周波数に応じて、自動的にステッピング速度（とそれに応じたステップ数）を上昇させる。これにより低周波でも、ステップ動作が静かになる（ステッピングモータは高速の方が静か)。
このため AMASS は Grbl 直線アルゴリズム step count をビットシフトする事で実現している。例えば, Level 1 ステップスムージングでは、1bit シフトを行い、各軸のステッ

[3] x、y 軸の送りの違いによる干渉の他に、NC フライス盤の機械的固有振動数と、ステッピングモータのステップ周波数が共振を起こし、大きな音を発生する周波数が存在する。これは各ハードウエアに固有で、AMASS を使用しすると、これにも効果が有るかもしれないが、やって見ないと分からない。

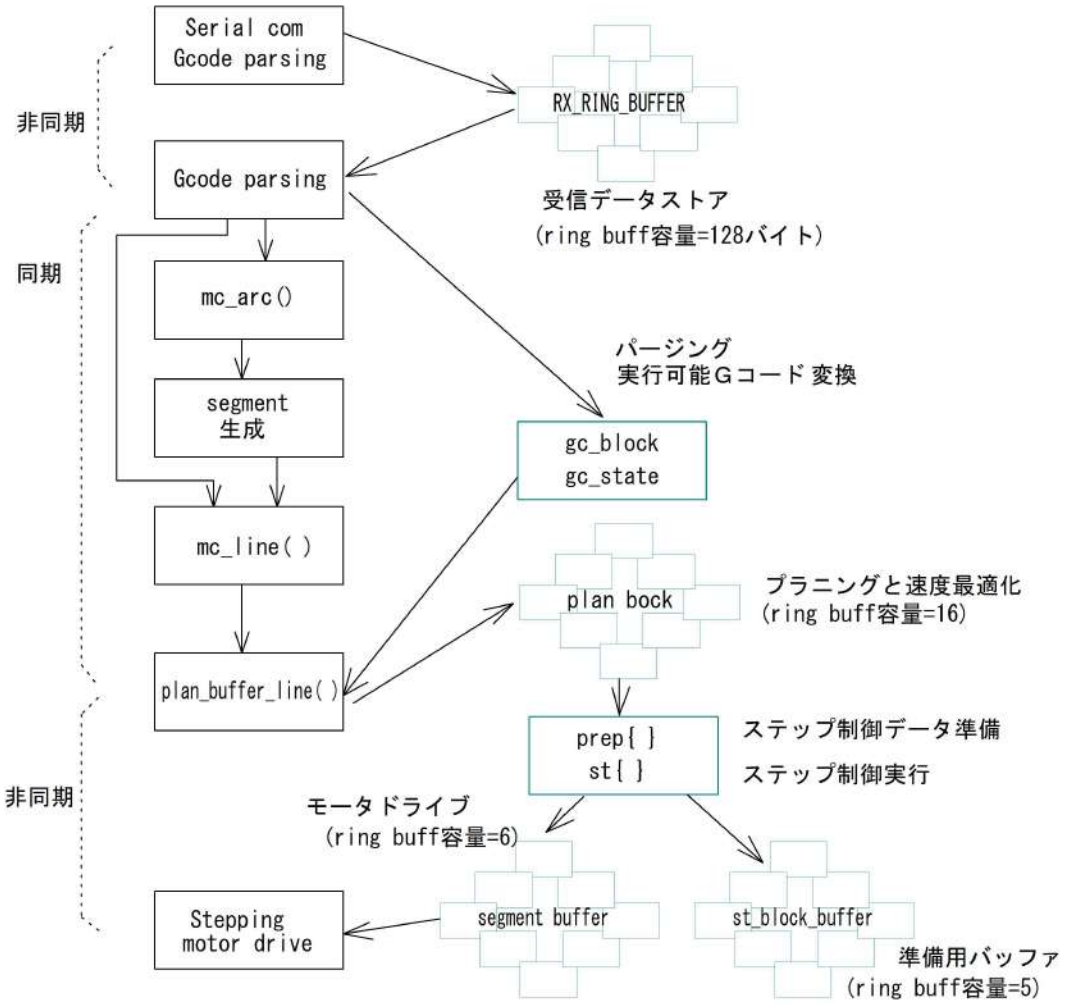

Figure 5.1: Grbl の処理プロセス

プカウントを変えず、実効的に 2 倍の細かさの動きとする。（ステッピング制御タイマー ISR 周波数を倍にする。）これにより、従属的 Grbl 直線アルゴリズムステップを ISR タイミングで、主要軸動作は 2 回の ISR タイミングに 1 回動作とする。AMASS Level 2 では、単純に再度 bit-shift を実行 4 倍の細かさでステッピングモータを動作させる。

これにより、Grbl の性能を落とさずに、Grbl 直線アルゴリズムの多軸制御時のエイリアシングが軽減される。しかも、CPU サイクルの使われていない時間をさらに有効活用する事になる。AMASS は Grbl 直線アルゴリズムの正確さを維持したまま、ステップ送りを細かくするが、これは Grbl 直線アルゴリズムがスケーラブルなので、計算コストが低くなるように、2、4、8 倍…としているわけである。

5.3 plan buffer と segment buffer

図 5.1 で、plan block と segment buffer にリングバッファが用意されているが、直線補間の場合、plan block の内容は 1segment で表現され、segment buffer は 1 ブロックしか使用しない。一方円弧補間では、各線素に対して一つの plan block が対応するので、円弧が非常に多数[4] の plan_block データを生成するがセグメントバッファ数は 16 に制限される。

従って、セグメントは順次追加要であるが、軸移動はタイマー割込みで実行、この割込み以外の空いた時間に plan buffer、segment_buffer にデータ追加が可能である。これがマルチスレッド的動作の利点でもある。

plan buffer のデータは segment buffer にコピーされるが、この時間は短い。一方、plan buffer と segment buffer は完全に非同期で、お互いの干渉が排除され、更新途中でデータを破壊するなどのバグも原理的に入り難い。plan buffer と segment buffer の内容が重複する部分もあり、やや冗長に見えるが、このような理由による構造と思われる。

5.4 planner.c/h

plan_buffer_line( ) 関数が主要な機能を担う。この関数の引数は以下の通り。

target[ ]； 移動先座標で、単位は mm。

plan_line_data_t 型構造体；pl_data。

このファイル内では構造体、pl_data および block を用いて処理を実行するが、それぞれは以下の様に定義される。最終的には plan_block_t 型構造体のリングバッファblock_buffer にデータがストアされる。

```
 typedef struct {
    float feed_rate;           // 軸動作速度。
    uint8_t  condition;
    int32_t line_number;       // 行番号（オプション）。
 } plan_line_data_t  pl_data;

typedef struct {
    uint32_t   steps[N_AXIS];     // X,Y 軸のステップ数。
    uint32_t   step_event_count;  // このブロック完了に必要なステップ数。
    uint8_t    direction_bits;    // 軸方向を示すフラグ。（オリジナルでは
                                      Arduino のポート指定）
    uint8_t  condition;
    int32_t line_number;          // 行番号（オプション）。
} plan_block_t  block;
```

[4] Grbl では最大でも 2000 以下と記述、簡易的マシンではこれより大分少ない

フラグ condition のビット割り当ては次の様になっている（主要なもののみ記述）。

PL_COND_FLAG_RAPID_MOTION	bit(0)
PL_COND_FLAG_SYSTEM_MOTION	bit(1)
PL_COND_FLAG_NO_FEED_OVERRIDE	bit(2)
PL_COND_FLAG_SPINDLE_CCW	bit(5)

plan_buffer_line( ) では、セグメント間速度最適化のための、各軸方向への変換係数を計算する。現在位置を基準として、移動先（ターゲット）の座標を (x, y) とすると、斜辺 L は

$$L = \sqrt{x^2 + y^2}$$

となる。従って、これを x, y 軸成分の移動距離換算すると

$$x \text{ 軸} := x \cdot \frac{1}{\sqrt{x^2 + y^2}}$$

$$y \text{ 軸} := y \cdot \frac{1}{\sqrt{x^2 + y^2}}$$

となる。従って、この係数を使うと、直線の傾きを、x、y 軸移動に反映できる。Grbl オリジナルでは、これを x,y,z, 3 軸について計算している。

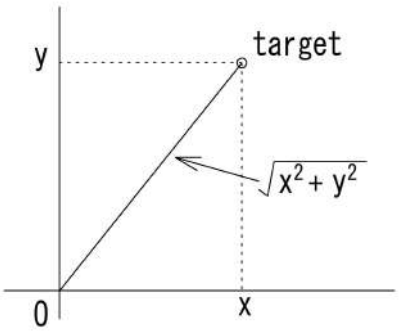

Figure 5.2: 変換係数

5.5 軸移動プラン実行

5.5.1 概要動作

motion_control.c 中の mc_line() の主要部分は以下の通り。

```
void mc_line(float *target, plan_line_data_t *pl_data)
```

```
{
   // バッファーが空になるまで、実行。
   do {
       if ( plan_check_full_buffer() ) {
          // 切削プランバッファがフルならばフラグ EXEC_CYCLE_START を立てる。
          protocol_auto_cycle_start();
       }
       else { break; }
   } while (1);
   // planner buffer に移動先座標、移動速度をロード。
   plan_buffer_line(target, pl_data);
}
```

上記ソースコードによると、バッファが空になるまでは、EXEC_CYCLE_START を立て main() をブロックし、その間にタイマー割込みによる軸動作が、リングバッファを消費する。各軸動作により、バッファが空になったところで、pl_buffer_line( ) が一つの pl_data を plan_buffer にストアする。

mc_line( ) を呼んだのが G コード中の直線補間であれば、この一つのデータは G コード 1 ブロックに相当する。一方、曲線補間は mc_arc( ) により処理されるが、この関数は曲線を多数のセグメントに分解し、その各セグメントが mc_line( ) を呼ぶ。従って、曲線補間の時は

曲線を直線セグメントに分割（この数は最大 2000 程度以下とすべき）。
セグメントバッファがフルとなるまで、直線セグメントデータをストア。
バッファフルで、タイマーによる軸加工が、バッファーが空になるまで続く。
次のセグメントをバッファーに満たす。

という動作を円弧動作が完了するまで繰り返す。

5.6 motion_control.c/h

直線補間 mc_line( ) と円弧補間 mc_arc( ) を実装したファイルであるが、mc_line( ) の主要機能である直線補間はステッピングモータ駆動プログラム stepper.c 中に実装され、この中では、中継的動作のみを実行する。

5.6.1 mc_line()

mc_line( ) に与える情報（パラメータ）は

*target ：target[N_AXIS] のポインターで、符号付きのターゲット絶対位置。単位は mm。

*pl_data： pl_data のポインターで、以下のメンバーを含む。
feed_rate：移動速度。単位は mm/second。
condition：planner コンディション　（1 バイトのフラグ）
line_number：行番号（オプション）

mc_line( ) は、関数 plan_buffer_line() を呼び出す。構造体 p_data のメンバーは以下の様であり、

```
typedef struct {
   float       feed_rate;
   int32_t     line_number;
 } plan_line_data_t   pl_data
```

plan_buffer_line() が、この構造体にパラメータ追加して block 構造体を生成、これをリングバッファに追加する。

```
typedef struct {
   uint32_t    steps[N_AXIS];
   uint32_t    step_event_count;
   uint8_t     direction_bits;
   int32_t     line_number;
} plan_block_t  block
```

G コードの 1 ブロック（1 行）のパージングにより、一つの構造体 pl_data が生成され、16ヶのリングバッファblock_buffer[ ] にストアする。

5.6.2 mc_arc()

【概要】

mc_arc() 関数は、円弧補間関数である。G コードでは、円弧指定に 2 方法が定義されているが、いずれの場合も gcode.c により、以下の情報にまとめる。

(1) position; 初期 (現在）x,y 座標。mm 単位。
(2) target ; ターゲット xy 座標。mm 単位。
(3) offset ; 現在座標と円弧の中心座標の差。gc_block.values.ij[0],[1]。mm 単位。
(4) radius; 半径。mm 単位。
(5) clockwise; CW/CCW。

これらのデータを基に、直線セグメントに分割、直線補間関数 mc_line() に引き渡す。

直線セグメントの分割に際しては、最大誤差量を設定、円弧を直線で近似する誤差が設定値以下となる様にする。

セグメント分割のため円弧の角を知る必要があり、これを現在座標と目的座標を基に、arc tangent を計算して求める。

この計算のためには、上記 (1)、(2)、(3)、(5) を用い、回転角度をラジアン単位で求め、半径を使用して分割セグメント数を決める。

【π ラジアン以上の回転】

回転角が π ラジアンを超える図形については、半径 r を負とすることで、それ以下の図形と区別しており (図 5.3,(A))、別の工夫が必要である。

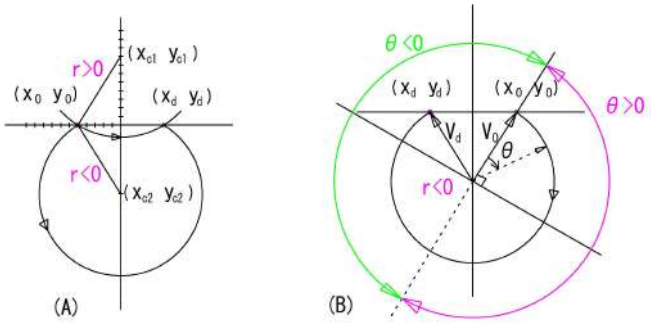

Figure 5.3: 円弧描画

図 5.3,(B) は、ベクトル V_0 を時計方向に回転し、ベクトル V_d とする場合を示すが、arctan2() による計算で $\theta > 0$ の場合は π ラジアンまでの回転、$\theta < 0$ の場合は、π ラジアン以上、2π ラジアンまでの回転である事が分かる。そこで、

```
if (is_clockwise_arc) {
    // #define ARC_ANGULAR_TRAVEL_EPSILON 5E-7 (5 times 10^(-7) radian )
    // 角度が-0.0000005 ラジアンより大きい時は
    if (angular_travel >= -ARC_ANGULAR_TRAVEL_EPSILON) {
         angular_travel -= 2*M_PI;
    }
}
// CCW の時の回転角。
else {
    if (angular_travel <= ARC_ANGULAR_TRAVEL_EPSILON) {
         angular_travel += 2*M_PI;
    }
}
```

という補正を入れる。

【円弧の精度設定】

Grbl では、通常の機械では位置精度が 0.001mm 以下とならない事と想定し
#define N_DECIMAL_COORDVALUE_MM 3
つまり、小数点以下 3 桁と定義している。
同様に円弧描画の精度 settings.arc_tolerance も浮動小数点 3 桁と設定されている [5]。これを基に、円弧から直線セグメントに分割する時のセグメント数を計算する。

```
segments = floor(fabs(0.5*angular_travel*radius)/
      sqrt(settings.arc_tolerance*(2*radius - settings.arc_tolerance)));
```

ここで、angular_travel は角度をラディアンを単位とした時の回転角であり、セグメント数は、単純に角度誤差量だけでは無く、半径も考慮に入れている [6]。即ち、大きな半径に対しては、同じ角度による移動量でも、周はより直線に近くなる事を計算にいれている。

【近似三角関数】

セグメントに分割すると、各セグメントの行先座標を、現座標から計算する。これは現在座標に対するベクトル（回転）計算となるが、この係数には三角関数が必要である（第 5.7(2)）。
Grbl では、計算速度を上げるため、これを近似計算で求める事にする。しかし、連続セグメントをこの様に計算し続けると誤差が大きくなるので、適当なところで、近似を用いない三角関数で計算、補正する。
近似計算による三角関数 $(\sin\theta, \cos\theta)$ は、以下のアルゴリズムによる。
$\cos x$ をテーラー展開すると、

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + ...$$

となるが、この 3 次近似までを使えば、一般的には十分な精度が得られる（この場合 3 次項は無いので 2 次）。
従って

$$\cos x = 1 - \frac{x^2}{2}$$

$$2\cos x = 2 - x^2$$

同様に $\sin x$ についてのテーラー展開により、

$$\sin x = x - \frac{x^3}{3!} = x(1 - \frac{x^2}{6})$$

[5] 線素の終端は円弧上にあるので、円弧直径は $2 \cdot$ settings.arc_tolerance となる。
[6] この式の根拠は不明。

$$= \frac{1}{6}x\{4 + (2 - x^2)\} = 0.167 \cdot x(4 + 2\cos x)$$

となる。プログラムでは

```
float cos_T = 2.0 - theta_per_segment*theta_per_segment;
float sin_T = theta_per_segment*0.16666667*(cos_T + 4.0);
cos_T *= 0.5;
```

である。

【mc_line() への引き渡し】

mc_arc() での作業は、与えられた円弧を適正な数に分割し、分割した各円弧の行先座標を求めることである。結果は（現在座標、行先座標）というペアになるが、現在座標は既知なので、各線素に対する行先座標を計算し、

mc_line(行先座標, pl_data 構造体)

の様に mc_line() に引き渡す。一方、pl_data 構造体の主要メンバーは、直線描画に必要なものと考えれば分かり易く

```
typedef struct {
   float feed_rate;
   uint8_t condition;
   int32_t line_number;
} plan_line_data_t   pl_data
```

が主たるものであるが、実質、軸移動速度のみである。

【mc_arc() の引数】

この関数の取るパラメータは以下の様になる。

(1) 半径および現在座標、ターゲット座標が与えられる場合。

target： ターゲット xyz 座標。
position ： 現在 xyz 座標。
pl_data： 付随データ収納構造体アドレス。（実質、速度のみ）
offset ： 現 xyz 座標からのオフセット。
axis_0,axis_1： 円弧面。
axis_linear： 円弧補間動作方向
radius： 半径。
isclockwise： CW/CCW。円弧補間は直線の連続で実現。

(2) 円弧の中心が与えられた場合。

target： ターゲット xyz 座標。
position ： 現在 xyz 座標。
ijk ： 中心 xyz 座標。
pl_data： 付随データ収納構造体アドレス。（実質、速度のみ）
offset ： 現 xyz 座標からのオフセット。
axis_0,axis_1： 円弧面。
axis_linear： 円弧補間動作方向
isclockwise： CW/CCW。円弧補間は直線の連続で実現。

【セグメント生成】

セグメント数 (segments-1) に達するまで直線補間セグメントを生成する。セグメント数は膨大になり得るので、ある数まで生成したら、軸移動動作によりデータが消費されるのを待つ。

セグメント用データはまず構造体 st_prep_block を生成、構造体アレー st_block_buffer 中に格納する。一方、実際の各軸ステップ動作は、st_exec_block 構造体の中にあるので、st_prep_block から適宜、情報をコピーする必要がある。これは stepper.c 中の st_prep_buffer() 関数により、軸移動の合間に実行される [7]。

5.7 注

5.7.1 angular_travel（角移動量）の計算

angular_travel はベクトル (x_0, y_0) が (x_t, y_t) まで、ある中心に対して角 θ 回転した時の角度移動量である。単位はラジアンで、半時計方向回転を基準とする。

これはソースコード中では、以下の様に定義される。

```
angular_travel = atan2(r_axis0 * rt_axis1 - r_axis1 * rt_axis0,
                       r_axis0 * rt_axis0 + r_axis1 * rt_axis1);
```

この関数を分かり易く書き直すと、

$$angular\_travel = atan2(x_0x_t + y_0y_t,\ \ x_0y_t - x_ty_0)$$

であり、arc tangent の計算の引数としてこの様なものを使う理由は以下のとおりである。

二つのベクトル (x_0, y_0)、(x_t, y_t) に対して内積および外積はそれぞれ次の様に定義される。

$$\boldsymbol{r}_1 \cdot \boldsymbol{r}_2 = |r_1||r_2| \cos\theta = x_0x_t + y_0y_t$$

$$|\boldsymbol{r}_1 \times \boldsymbol{r}_2| = |r_1||r_2| \sin\theta = x_0y_t - x_ty_0$$

[7] この部分の処理、冗長に見える。

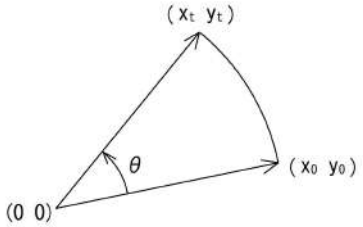

Figure 5.4: arctangent の計算

ただし、外積はベクトルとなるべきであるが、ここでは絶対値を考える。また

$$x_0 y_t - x_t y_0 = -(x_t y_0 - x_0 y_t)$$

より、CW（負）,CCW（正）方向が区別できる。

$$\tan\theta = \frac{|r_1||r_2|\sin\theta}{|r_1||r_2|\cos\theta} = \frac{\text{外積}}{\text{内積}}$$

なので、arctangent は

$$\theta = arctan\left(\frac{\text{外積}}{\text{内積}}\right)$$

となり、関数 atan2( ) の引数が外積と内積になる事がわかる。また、

$$-\pi < \theta < +\pi$$

である。

5.7.2 atan() と atan2() の違い

$atan(x)$ は $\tan\theta = x$ となる θ を与え、結果は $-\pi/2$〜$\pi/2$ となる。一方、$atan2(x, y)$ は (xy) 座標における偏角を与え、結果は $-\pi$〜π となる。Grbl では、平面上の回転を扱うため、atan2() が使われる。

5.7.3 Vector rotation

2 次元ベクトル (x_1, y_1) を (x_2, y_2) に回転移動する時の変換式は以下の様になる。

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x_1 \\ y_2 \end{pmatrix}$$

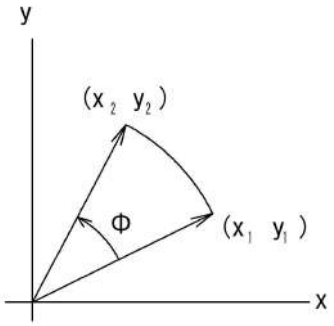

Figure 5.5: ベクトルの回転

これからも明瞭な様に、単純に計算しようとすると、三角関数の計算が沢山出現し、この計算コストは高くなってしまう（時間がかかる)。そこで、これを出来るだけ計算しない工夫が、(特にマイクロプロセッサの場合）必要となる。

Chap6　直線補間アルゴリズム

stepper.c に記述される部分はハードウエアに直結する部分であり、Arduino のポート操作がそのままステッピングモータ動作となる。一方、ステッピングモータの駆動方式そのものが、直線補間アルゴリズムとなっており、ここではステッピングモータについての特定の動作解析はせず、CNC のコアとなる直線補間がどの様に実装されているかについて調べる。

6.1　概要

stepper.c では、実際にステッピングモータを駆動する部分 (タイマー割込み関数) と、planner buffer から step segment buffer にステップ化されたデータをコピーする初期化部分 (step_prep_buffer( ) 関数) とから構成される。前者は特定のステッピングモータに対する処理と、「Grbl 直線アルゴリズム」が実装されている。

全体の処理フローは次の様である。
step segment buffer の用意。
planner —速度を最適化し、planner buffer にデータセット
↓
step segment buffer
↓
stepper – 実際の軸送り動作。 segment buffer 中の　ステップ実行。

ステップ実行で planner buffer 中の最初のブロックから追い出され、planner 動作と stepper 実行が干渉しない様になる。

planner buffer から追い出される数と　segment buffer 中のセグメント数は、それらの量が計算、調整されて、main program 中のすべての操作時間が、stepper がバッファーを空にする時間よりも長くならないようにしている。 現状では、segment buffer は、固くみつもって約 40-50 msec 分のステップを保つ。(単位は step 数、mm、minutes)

6.2　構造体、変数の準備

ソースコード中には Bresenham アルゴリズムと記述されているが、通常 Bresenham 直線補間アルゴリズムは第 7 章 7.1.2 節に述べる様に、

(1) x、y 軸のどちらを連続的にステップ動作させるか決定。
(2) x 軸を連続ステップ移動と判断したら、関数 $G = x_d - 2y_d$ により y 軸を
1 ステップ移動するか、そのままとするかを決める。
(3) 関数 G の値に、y 軸の誤差を蓄積
というプロセスをターゲット位置まで繰り返す。

一方 stepper.c に記述されている Grbl 直線アルゴリズムはこれとはかなり異なる処理となっている。

stepper.c 中の st_prep_buffer() 関数は、G コードパージング関数が計算した、pl_block 構造体中の座標移動値 steps を 2 倍して st_prep_block 構造体にコピーしている。また step_event_count 値は G コードパーザーが検出した、座標移動値の最大値で、これも 2 倍されている。

```
for (idx=0; idx<N_AXIS; idx++) {
    st_prep_block->steps[idx] = (pl_block->steps[idx] << 1);
}
st_prep_block->step_event_count = (pl_block->step_event_count << 1);
```

一方ステッピングモータを駆動するタイマー割込み関数では、

```
st.counter_x = st.counter_y = (st.exec_block->step_event_count >> 1);
```

構造体 st{} 中の counter_x、counter_y を 1/2 にして（もとに戻して）設定している。

6.2.1 Grbl 直線アルゴリズム

Grbl の直線アルゴリズムは、誤差関数を使用せず、各軸が共通変数値 step_event_count を使用するだけで実現しているという特徴がある [1]。

そのため、各軸ともプログラムは同一 (もちろん、操作する変数の違いはある）で、x 軸については以下の様である。

```
st.counter_x += st.exec_block->steps[X_AXIS];
if (st.counter_x > st.exec_block->step_event_count) {
     // モータ駆動フラグを step_outbits に設定
     st.step_outbits |= (1<<X_STEP_BIT);
     st.counter_x -= st.exec_block->step_event_count;
     // direction_bits の X 軸フラグが 1 ならば。
     if (st.exec_block->direction_bits & (1<<X_DIRECTION_BIT)) {
          sys_position[X_AXIS]--;
```

[1] この点について Grbl のソースコードには Bresenham の言及があちこちにあり、かつ実際のアルゴリズムについての解説は無く、理解し難い。

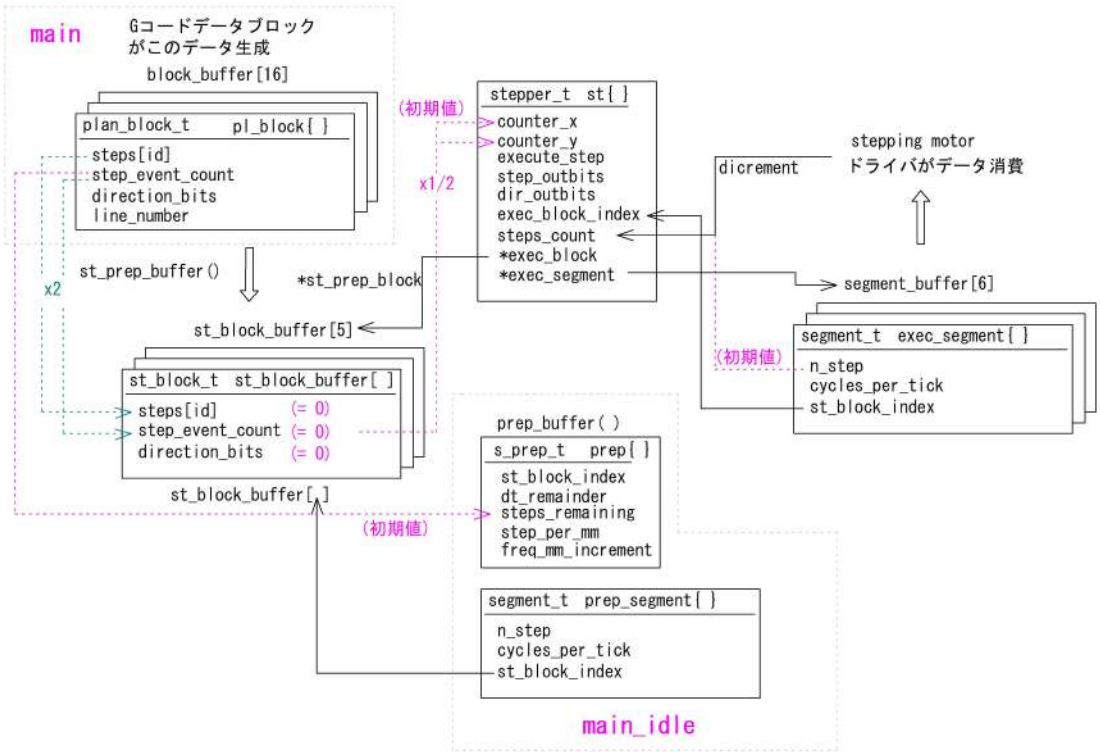

Figure 6.1: stepper で使用する構造体の関係

```
        }
        else { sys_position[X_AXIS]++; }
}
```

実際にステッピングモータを駆動する出力と方向を決める部分を除いてしまうと、x、y 軸に関するプログラムは以下の様になる。

```
st.counter_x += st.exec_block->steps[X_AXIS];
if (st.counter_x > st.exec_block->step_event_count) {
        st.counter_x -= st.exec_block->step_event_count;
        // x ステッピングモータを特定の方向に 1 ステップ駆動
}
st.counter_y += st.exec_block->steps[Y_AXIS];
if (st.counter_y > st.exec_block->step_event_count) {
        st.counter_y -= st.exec_block->step_event_count;
        // y ステッピングモータを特定の方向に 1 ステップ駆動
}
```

図 6.2 に $(x, y) = (10, 3)$ とした時の動作例を示す。x, y の最大値は 10 なので、$step\_event\_count = 2 \times 10 = 20$、$steps[X] = 20$、$steps[Y] = 6$ となる。$st.counter\_y$ の初期値は 10 であるの

で、x 軸については 1 回目から閾値である $step\_event\_count$ を超えて、x 軸のステップ動作が発生、これはすべてのステップについて続く。

一方 y 軸については初期値 $st.counter\_y = 10$ であるが、$steps[Y] = 6$ なので、y 軸にステップ動作が発生するのは、2、6、9 回目... となる。

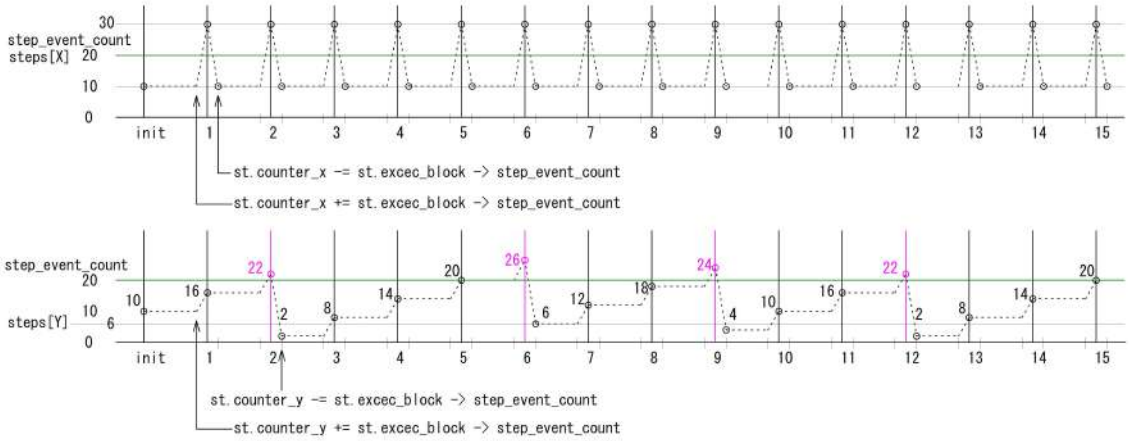

Figure 6.2: Grbl 直線アルゴリズムの動作例

ここで、何故閾値を x, y の最大値の 2 倍、計算するステップ数も、もとの値の 2 倍とするかははっきりしない。この状況は例えば、

st.counter_x += 5

としても x 軸については成立するが、同じ比で y 軸に反映させるためには計算は掛け算となる必要がある。しかし、閾値と x, y ステップ値は同じ倍率を掛けねばならない事を考えると、一番単純な 2 倍としたのかも知れない。また、

st.counter_x = st.counter_y = (st.exec_block → step_event_count >> 1);

という初期化は、ステッピング動作の、x, y の動作タイミングをやや平準化する効果がある（図 6.2 を見ると、y 軸の出力タイミングが $st.counter\_y = 0$ とするよりも早くなる）。

閾値や x, y を 2 倍としない場合を図 6.3 に示す。$(x, y) = (10, 3)$、$step\_event\_count = 10$、$steps[X] = 10$、$steps[Y] = 3$ とした場合である。ただし、

```
if (st.counter_x >= st.exec_block->step_event_count)
```

とする。この場合でも、x, y 軸に対する軸移動は Bresenham アルゴリズムと同等の効果となり、より原理に忠実、分かり易いと思われる。

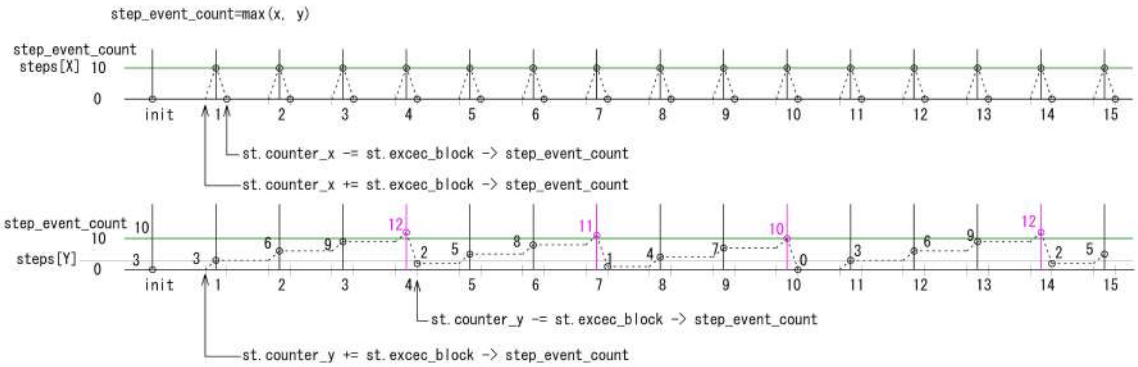

Figure 6.3: Grbl 直線アルゴリズムの動作例 2

6.2.2 Grbl 直線アルゴリズムの原理

「Grbl 直線アルゴリズム」の原理は次の様である。図 6.4 において上側が x 軸、下側が y 軸であり、ステップ数は $x > y$ とする。従って、x 軸は、各 ISR タイミング 1,2,3... 毎にステップが送られる。このステップ送りを決定するために、step_event_count を閾値と考えると、st_counter_x は ISR タイミング毎に計算し、閾値と比較する変数をとなる。st_counter_x が、ステップの間もアナログ的に増加、閾値を超えた所でリセットされる信号（点線）と考えると動作が分かり易い。

y 軸については、同じ閾値をつかうものの、y の値が小さいので st_counter_y が閾値に達するには、x 軸よりも時間（ステップ回数）がかかり、結局これが直線の傾きを与える。

このアルゴリズムでは、x、y の値の大きい方が毎回のステップ送りを生成せねばならないので

if (st.counter_x > st.exec_block → step_event_count)

が常に真になる必要がある。

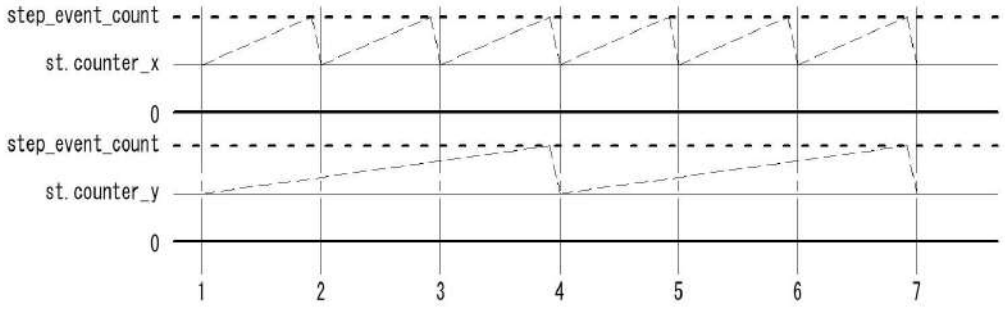

Figure 6.4: Grbl 直線アルゴリズムの原理

通常の Bresenham 直線補間アルゴリズムと比べても、洗練されていて、傾斜した直線を描画するのに、傾きも計算せず、かつ x、y 軸がお互いを参照しながら計算という手順も無い。

また x 軸を基準にして y 軸を移動させるか、あるいはその逆とするかの判断も、最初に閾値を決める時点でなされ、単純化されている。しかし、これらの考えが異なった操作に分散されて実装されているために、どの様な考えで、何が実装されているかの見通しは極めて悪い [2]。

「Grbl 直線アルゴリズム」は x、y 軸を同期したタイミングで異なった周波数でスイープすると、傾いた直線が得られるという、リサージュ図形を発生するものと同一の考え方である。この構造からも AMASS アルゴリズムが容易に適応できるのが分かる。（x、y の周波数に同じ倍率を掛けても、同じ形のリサージュ図形が得られる。ただし図形の描画周期が変化し、これがまさに AMASS の目的とする処である。）

[2]特に、閾値や x, y を 2 倍している理由が不明、かつソースコードの別の部分に分散しているので分かり難い。

Chap7　参考：Bresenham アルゴリズム

CNC 制御の根幹は、線形補間アルゴリズムにある。このアルゴリズムとして Bresenham[1] アルゴリズムが知られており、本章ではこのアルゴリズムについて詳述する。なお本章は参考資料であり、Grbl の実装自体には直接関係しない。

線形補間アルゴリズムは、基本的に x、y、2 方向の移動 [2] メカニズムを用いて、任意の角度を持つ直線を表現するアルゴリズムである。円弧補間も、それを細かい直線（線素）に分割して実行するので、線形補間がその基礎となる。

本章では、実数座標や実数変数と整数座標、整数変数を頻繁に扱い、かつそれがどちらかが重要なので、前者を x、y、z の様に表し、後者を $\mathcal{X}$、$\mathcal{Y}$、$\mathcal{Z}$ のように記述する。

7.1　直線補間

G コードで、直線補間と呼ばれる処理 (G01) で、現在の座標から、指示された座標まで、直線で移動する。以下のフォーマットで表現され、

$$G01\ X\_\ Y\_\ Z\_\ F\_$$

$X\_, Y\_, Z\_$ は行先の座標、$F\_$ は速度を表す。

ここでは、x、y 座標を考え、速度も考慮しない。現在位置を $(\mathcal{X}_0, \mathcal{Y}_0)$、目的座標を $(\mathcal{X}_d, \mathcal{Y}_d)$ とすると、x 軸に対する y 軸の送り量 k は、それぞれの送り速度は等しいとして、傾きは

$$k = \frac{\mathcal{Y}_d - \mathcal{Y}_0}{\mathcal{X}_d - \mathcal{X}_0}$$

となる。傾き k は整数の比なので、有理数とはなるが、整数となるとは限らない。

各 x の値に対して、y 座標を

$$y = kx,$$

により決定するのが、数学的原理であるが、実際の CNC フライス盤では、座標はすべて整数で表される。

従って、計算入力値は整数であるものの、k は小数となり、プロセッサの処理として重くなる。そこで、これをできるだけ軽くするような工夫がなされる。

[1] Jack Bresenham。1962 年に IBM 1401 に接続された、Calcomp 製プロッター 駆動のアルゴリズムとして考案された。

[2] x, y, z の内から設定した任意の 2 次元平面で定義できるが、ここでは x, y 平面とする。

なお CNC フライス盤の $(\mathcal{X}, \mathcal{Y})$ 座標は、右上が原点で、左に向かって $\mathcal{X}$ 座標が、手前に向かって $\mathcal{Y}$ 座標が増加するように設定する。これは原点に工具がある時に作業物が手前に配置され、作業し易くなるためである。

原点を (0, 0) とすると、機械座標では、右手前が $(-\mathcal{X}_{max}, -\mathcal{Y}_{max})$ となる。

7.1.1 直接法

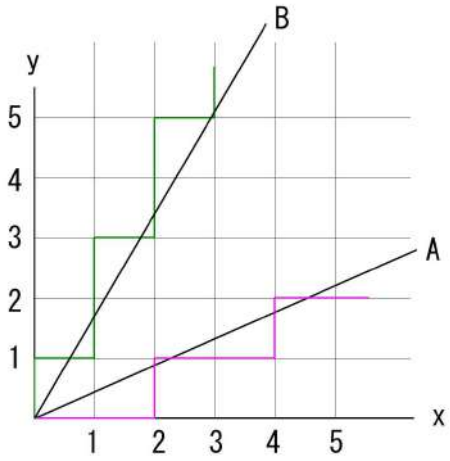

Figure 7.1: G01 のアルゴリズム

まずは、原理的な方法について考える。図 7.1 より、明確であるが、直線の傾きを考えると、$\mathcal{X}$ の単位送りに対して $\mathcal{Y}$ 軸の送りを計算する $k < 0.5$ の場合、$\mathcal{X}, \mathcal{Y}$ 同ステップ送りの $k = 0.5$ の場合、そして $\mathcal{Y}$ 軸送り 1 ステップに対する $\mathcal{X}$ 軸送り量を計算する $k > 0.5$ の場合に分けて実装するのが適当であろう。

(1) $k < 0.5$

$\mathcal{X}$ 軸送りを基準にして、$\mathcal{X}$ 軸送り量を計算する。つまり、$\mathcal{X}$ 軸は 1 ステップ毎の送りであるが、何ステップ後に $\mathcal{Y}$ 軸を 1 ステップ送るかを計算する。

(2)$k = 0.5$

一番単純で、$\mathcal{X}$ 軸、$\mathcal{Y}$ 軸を交互に 1 ステップづつ送る。

(3)$k > 0.5$

$\mathcal{Y}$ 軸は 1 ステップ毎の送り、$\mathcal{X}$ 軸の送り量を計算する。

x、y 軸の送りの計算式は、

$$y = kx, \quad x = \frac{1}{k}y$$

である。しかし、$\mathcal{X}$、$\mathcal{Y}$ 軸とも送りは 1 ステップ単位なので、誤差が蓄積しない様に、計

算式による余りを適切に処理する必要がある。そのため、誤差を四捨五入して、送りに反映させるが、誤差成分を加算して次のステップに反映させる。

これらのアルゴリズムは直線や曲線描画に際して、ディジタイズされたコンピュータ画面やプロッタ画面のピクセルを適切に選択するというのが考え方である。モニター画面の場合、例えば円の描画は、対称性を利用して 8 分円を同時に描画し、高速化する。しかしプロッタや CNC では、これは不可能で、かつ描画開始位置も通常は指定されるので、指定位置からの逐次描画となる。

7.1.2 Bresenham アルゴリズム

ここでは Bresenham アルゴリズム [3] について述べる。

【原理】

第 7.1.1 節の方法では、直線の方程式は実数を取ると考え、計算の後に整数化する。この場合計算に、小数の掛け算、割り算が含まれ、処理速度が遅くなる [4]。

そこで、実際の直線の描画では、整数化された現在位置 $(\mathcal{X}_0, \mathcal{Y}_0)$ から、次の整数化された座標に移動（しかも $\mathcal{X}_1$、$\mathcal{Y}_1$ は 1 ステップまたは 0）するとして、直線の方程式を構成する。これにより実数の整数化（量子化）と処理速度の高速化の両方を同時に実行する。

直線の傾きは、

$$k = \frac{\mathcal{Y}_d - \mathcal{Y}_0}{\mathcal{X}_d - \mathcal{X}_0},$$

となる。ここで $(\mathcal{X}_d, \mathcal{Y}_d)$ は行先座標であり、傾きは有理数となる（整数化されていない）。

直線の方程式は、

$$y = \frac{\mathcal{Y}_d - \mathcal{Y}_0}{\mathcal{X}_d - \mathcal{X}_0} x$$

あるいは 2 変数の関数 [5] とすると、

$$f(x, y) = (\mathcal{X}_d - \mathcal{X}_0) \cdot y - (\mathcal{Y}_d - \mathcal{Y}_0) \cdot x = 0.$$

となる。直線補間を考える時、原点からの移動と出来るので、$(\mathcal{X}_0, \mathcal{Y}_0) = (0, 0)$ として簡単化すると、

$$f(x, y) = \mathcal{X}_d \cdot y - \mathcal{Y}_d \cdot x = 0$$

が得られる。

---

[3] Bresenham アルゴリズムを円の描画に拡張したものとして、Bresenham の円弧アルゴリズムと Midpoint アルゴルズムがある。この直線アルゴリズムは円弧の Midpoint アルゴリズムと同一である。

[4] ただし、プロセッサの高速化、並列化が進んでいるので、常に遅いとは限らない。

[5] $\mathcal{Z}$ 軸まで考えると 3 変数の関数。

一般の直線は、その座標は実数であるが、CNC 装置ではこれを整数で近似、つまり実数値を最寄りの整数に置き換えて行く。同時に、通常は小数となる k（元々は整数の比）に関して、小数計算しないような処理を考えて行く。

これを次の考えで実行する。

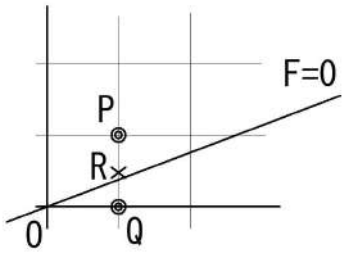

Figure 7.2: 直線による平面の分割

任意の傾きを持った直線と格子点の関係を考える。図 7.2 を参照すると、点 P では $F < 0$、点 Q では $F > 0$ となり、格子点が関数 $f(x, y) = 0$ により分割された 2 平面のどちらに属するかはその点の関数値を計算すれば分かる。

ステッピング動作を考えると、x、y 軸のどちらかは 1 ステップ必ず送るが、ここではまず x 軸を必ず送る軸とする。これは直線で言えば、傾きが 45 度以下の直線の場合である。

ここで、$\mathcal{X}_0(=0) \to \mathcal{X}_1$ のステップ送りに対して y 軸をどうするか (1 ステップ送って、点 P とするか、そのまま点 Q に止まるか）について考えると、$\mathcal{X}_1$ に対応する y_1 の値を四捨五入して決める事になろう。即ち $y_1 < 1/2$ あるいは $y_1 \geq 1/2$ により判断する。

点 $(\mathcal{X}_0 + 1, \mathcal{Y}_0 + 1/2)$ においての関数の値は、

$$F = f(\mathcal{X}_0 + 1, \mathcal{Y}_0 + 1/2) - f(\mathcal{X}_0, \mathcal{Y}_0)$$

$$= \mathcal{X}_d \cdot (y_0 + 1/2) - \mathcal{Y}_d \cdot (x_0 + 1) - \mathcal{X}_d \cdot y_0 + \mathcal{Y}_d \cdot x_0$$

$$= \frac{1}{2}\mathcal{X}_d - \mathcal{Y}_d.$$

従って、x 軸を原点から 1 ステップ送り $\mathcal{X}_1$ とする時、y 軸をどうすべきかは上記関数値で判断できる。図 7.2 の場合では直線 F は中点 R の下側を通過するので、x 軸を 1 ステップ送り $\mathcal{X}_1$ とする時、y 軸はそのまま点 Q となる。

【誤差関数】

しかし、点 Q を選択すると、直線上から外れた格子点（それでも最寄り）を選択せざるを得なかったため、直線と $\mathcal{X} = 1$ の交点の y 座標を y_1 とすると、

$$\text{誤差} = f(\mathcal{X}_1, y_1) - f(\mathcal{X}_1, \mathcal{Y}_1)$$

$$= (\mathcal{X}_d - \mathcal{X}_0)y_1 - (\mathcal{Y}_d - \mathcal{Y}_0)\mathcal{X}_1 - \{(\mathcal{X}_d - \mathcal{X}_0)\mathcal{Y}_1 - (\mathcal{Y}_d - \mathcal{Y}_0)\mathcal{X}_1\}$$

$$= \mathcal{X}_d y_1 - \mathcal{Y}_d\mathcal{X}_1 - \{\mathcal{X}_d\mathcal{Y}_1 - \mathcal{Y}_d\mathcal{X}_1\} \quad \because \mathcal{X}_0 = \mathcal{Y}_0 = 0$$

$$= \mathcal{X}_d y_1 \quad \because \mathcal{X}_1 = \mathcal{X}_0 + 1 = 1,\ \mathcal{Y}_1 = 0$$

$$= \mathcal{Y}_d\mathcal{X}_1 = \mathcal{Y}_d \quad \because y_1 = \frac{\mathcal{Y}_d}{\mathcal{X}_d}\mathcal{X}_1$$

が発生する（この場合、過剰)。

一方、点 P を選択した場合の誤差は $f(\mathcal{X}_1, \mathcal{Y}_0) \neq 0$ に注意して、

$$\text{誤差} = f(\mathcal{X}_1, \mathcal{Y}_1) - f(\mathcal{X}_1, \mathcal{Y}_0) - \mathcal{Y}_d$$

$$= (\mathcal{X}_d - \mathcal{X}_0)\mathcal{Y}_1 - (\mathcal{Y}_d - \mathcal{Y}_0)\mathcal{X}_1 - (\mathcal{X}_d - \mathcal{X}_0)\mathcal{Y}_0 + (\mathcal{Y}_d - \mathcal{Y}_0)\mathcal{X}_1 - \mathcal{Y}_d$$

$$= \mathcal{X}_d - \mathcal{Y}_d. \quad \because \mathcal{Y}_0 = 0,\ \mathcal{X}_1 = \mathcal{Y}_1 = 1$$

が得られ、この場合、不足である。

【整数化】

関数 F の計算では、0.5 という小数値が出現する。そこで、処理を容易で高速にするため、$2F$ で直線のどちらかにあるかを判定かつ、誤差関数も 2 倍した $D(x, y)$

$$D_{(step)} = 2(\mathcal{X}_d - \mathcal{Y}_d);\quad y \text{ 軸 1 ステップ送り}$$

$$D_{(stay)} = -2\mathcal{Y}_d;\qquad y \text{ 軸送り無し}$$

を用いる。

【アルゴリズム】

現在座標 $(\mathcal{X}_0, \mathcal{Y}_0)$ に対して、$\mathcal{X}$ 座標をインクリメントした $\mathcal{X}_1$ に対応する $\mathcal{Y}_1$ 座標は $\mathcal{Y}_0$ または $\mathcal{Y}_0 + 1$ となる。このどちらかを決定するために以下の直線を表す関数を用いて判断する。

$$G = \mathcal{X}_d - 2\mathcal{Y}_d.$$

ただし、2 ステップ以降では誤差の値を計算に入れる必要がある。

(1) $G > 0$ の時は y 軸はそのまま。この時の誤差は、

$$D_{(stay)} = -2\mathcal{Y}_d.$$

(2) $G < 0$ の時は y 軸をインクリメント ($\mathcal{Y}_0 + 1$ とする)。この時の補正誤差量は

$$D_{(step)} = 2(\mathcal{X}_d - \mathcal{Y}_d).$$

【具体例】

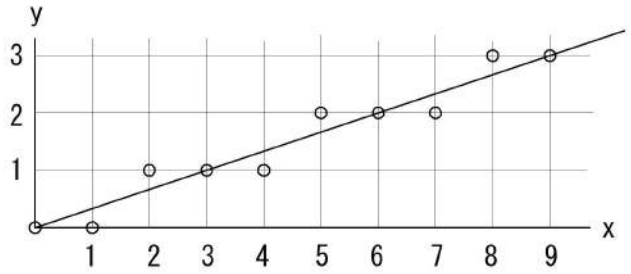

Figure 7.3: Bresenham のアルゴリズム

$(\mathcal{X}_0, \mathcal{Y}_0) = (0, 0)$、$(\mathcal{X}_d, \mathcal{Y}_d) = (9, 3)$ とすると、$\mathcal{Y}_d = 3, \mathcal{X}_d = 9$ である。

(1) $\mathcal{X}_1$

判定関数の初期値は、

$$G_1 = \mathcal{X}_d - 2\mathcal{Y}_d = 3 > 0.$$

従って、$\mathcal{Y}$ 軸は送らず、次に繰り越す誤差は以下となる。

$$D_1 = D_{stay} = -2\mathcal{Y}_d = -6$$

(2) $\mathcal{X}_2$

ステップ 1 の誤差を入れた関数値 G_2 は

$$G_2 = \mathcal{X}_d - 2\mathcal{Y}_d + D_1 = 3 - 6 = -3 < 0.$$

従って、$\mathcal{Y}$ 軸は 1 ステップ送り、次に繰り越す誤差は以下となる。

$$D_2 = D_{(step)} = 2(\mathcal{X}_d - \mathcal{Y}_d) = 12$$

(3) $\mathcal{X}_3$

ステップ 2 の誤差を入れた関数値 G_3 は

$$G_3 = \mathcal{X}_d - 2\mathcal{Y}_d + D_2 = 3 + 12 = 15 > 0.$$

従って、$\mathcal{Y}$ 軸は送らず、次に繰り越す誤差は以下となる。

$$D_1 = D_{stay} = -2\mathcal{Y}_d = -6$$

以上より、それぞれの $\mathcal{X}_n$ に対応する $\mathcal{Y}_n$ は以下の様になる。

$\mathcal{X}_1 : D_1 > 0, \quad \mathcal{Y}_1 = \mathcal{Y}_0, \qquad D_2 = D_1 + D_n = 3$

$\mathcal{X}_2 : D_2 > 0, \quad \mathcal{Y}_2 = \mathcal{Y}_1 + 1, \quad D_3 = D_2 + D_p = -9$

$\mathcal{X}_3 : D_3 < 0, \quad \mathcal{Y}_3 = \mathcal{Y}_2, \qquad D_4 = D_3 + D_n = -3$
$\mathcal{X}_4 : D_4 < 0, \quad \mathcal{Y}_4 = \mathcal{Y}_3, \qquad D_5 = D_4 + D_n = 3$
$\mathcal{X}_5 : D_5 > 0, \quad \mathcal{Y}_5 = \mathcal{Y}_4 + 1, \quad D_6 = D_5 + D_p = -9$
.....

【プログラム　例１】

プログラムにすると以下を得る。

```
void move(int x,int y){
   int x,y, dx, dy, D;
   dx=x1-x0;
   dy=y1-y0;

   D = 2*dy - dx;
   y=y0;

   for( x = x0+1, x1){
      if (D > 0){
          y = y+1;
          D = D + (2*dy-2*dx);
    }
    else{
          D = D + (2*dy);
    }
     return y;
}
```

【プログラム　例２】

```
def Bresenham3D(x1, y1, x2, y2 ):
    ListOfPoints = []
    ListOfPoints.append((x1, y1))
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    if (x2 > x1):
        xs = 1
    else:
```

```
        xs = -1
    if (y2 > y1):
        ys = 1
    else:
        ys = -1

   # Driving axis is X-axis"
    if (dx >= dy ):
        D = 2 * dy - dx

        while (x1 != x2):
            x1 += xs
            if (D >= 0):
                y1 += ys
                p1 -= 2 * dx
            D += 2 * dy

            ListOfPoints.append((x1, y1))

    # Driving axis is Y-axis"
    elif (dy >= dx ):
        p1 = 2 * dx - dy

        while (y1 != y2):
            y1 += ys
            if (D >= 0):
                x1 += xs
                D -= 2 * dy
            D += 2 * dx
            ListOfPoints.append((x1, y1))

    return ListOfPoints
```

この例では、すべての直線の傾きに対応している。入力は開始及び終点の x、y 座標であり、まずそれぞれの距離を計算して、x 軸送りに対して y 軸を決めるか、その逆とするかを判断する。誤差 D は

$$D = 2 \cdot dy - dx; \quad x \text{ 軸送り}$$

$$D = 2 \cdot dx - dy; \quad x \text{ 軸送り}$$

と定義する。そして送り軸 1 ステップごとに D の値の正負をチェックし、x 軸送りなら

$$D > 0; y_1 = y_1 + y_s,\ \ D = D - 2dx$$

$$D < 0;\ \ D = D + 2dy$$

とする。これを終点まで繰り返す。

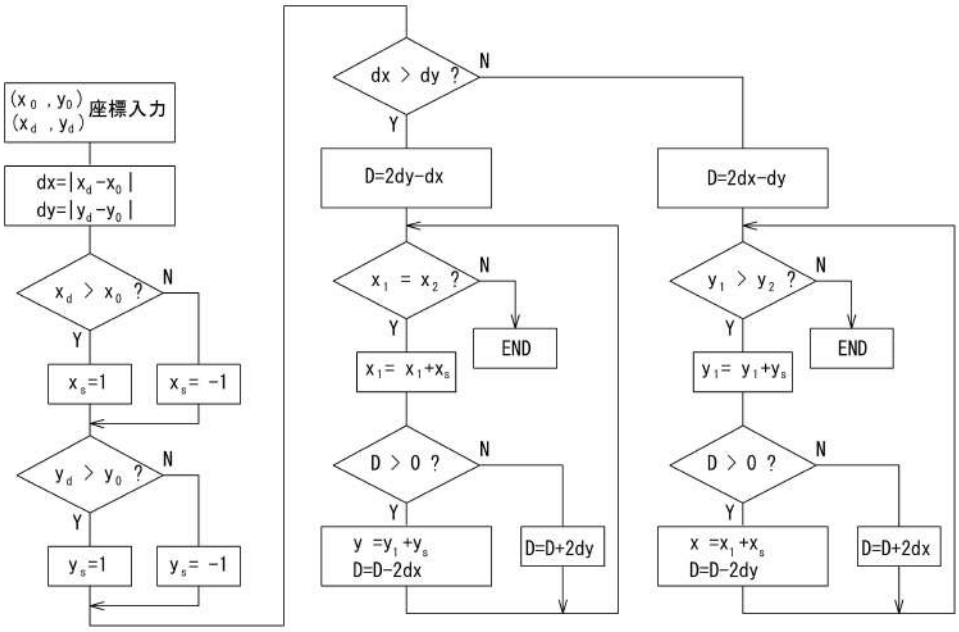

Figure 7.4: Bresenham アルゴリズムフロー

7.2 円弧補間アルゴリズム; G02,G03

G02,G03 とも、円弧の描画コードであり、手動で送る事が不可能な、自動フライス盤動作の基本である。G コードとしては、二つの方式、

(1) 円弧の中心を、現座標からの移動量で示し、行先座標を指定する方式、

(2) 円弧の半径 r と、行先座標を指定する方式、

が定義されている。これらのそれぞれについて、円を描画する方向（時計回り；G02、反時計回り；G03）が定義される。

7.2.1　円弧中心、行先指定

図 7.5 は、円弧の中心を現在位置 $(\mathcal{X}_0, \mathcal{Y}_0)$ からの変位 (I, J) で表現する状況を示したもので、変位 I, J はそれぞれ $\mathcal{X}$、 $\mathcal{Y}$ 座標の相対的移動量である。フォーマットは

$$G03\ X\_\ Y\_\ I\_\ J\_\ F\_$$

となる。座標 $(X\_, Y\_)$ は、行先の座標である。

円弧は決まった平面にしか作成できないので、対象となる平面は $X-Y$、$Z-X$、$Y-Z$ の 3 平面である。従って、実際のプログラムでは、平面指定コード $G17, G18, G19$ のいずれかで、平面の指定も行う。

$$G17G03\ X\_\ Y\_\ I\_\ J\_\ F\_$$

半径 r は

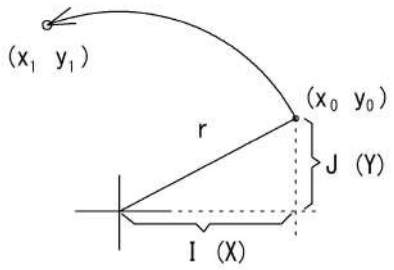

Figure 7.5: G03 のアルゴリズム

$$r = \sqrt{I^2 + J^2}$$

で計算されるので、指定しない。

7.2.2　半径 r、行先指定

【コマンドの基本動作】

G コードは G02 および G03 であり、フォーマットは

$$G02\ X\_\ Y\_\ Z\_\ R\_\ F\_$$

である。ここに、$X\_\ Y\_\ Z\_$ は、行先の座標であり、この点まで $R\_$、速度 $F\_$ で描画という意味である。現在位置 $(\mathcal{X}_0, \mathcal{Y}_0)$ は既知で、半径 $\mathcal{R}$ および、行先 $(\mathcal{X}_d, \mathcal{Y}_d)$ が与えられるので、円弧の中心は 2 点となる。$(\mathcal{X}_0, \mathcal{Y}_0)$ から出発して、時計まわりに描くと C, D が、

半時計まわりに描くと A, B が得られる。従って、時計回り、反時計回りの他に、さらに一つの条件をつけて A, B, C, D がすべて表現出来る様にする。（図 7.6 参照。）

このため、円弧の移動角が 180 度以上の場合は、R に $-$ を付けて区別する。従って、

(1) 円弧 A：　$G03\ X\mathcal{X}_d\ Y\mathcal{Y}_d\ R\mathcal{R}\ F\mathcal{V}$

(2) 円弧 B：　$G03\ X\mathcal{X}_d\ Y\mathcal{Y}_d\ R(-\mathcal{R})\ F\mathcal{V}$

(3) 円弧 C：　$G02\ X\mathcal{X}_d\ Y\mathcal{Y}_d\ R\mathcal{R}\ F\mathcal{V}$

(4) 円弧 D：　$G02\ X\mathcal{X}_d\ Y\mathcal{Y}_d\ R(-\mathcal{R})\ F\mathcal{V}$

となる（(x, y) 平面の円弧とする）。

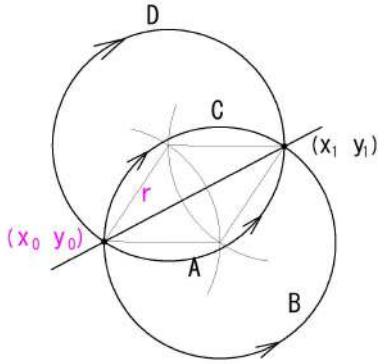

Figure 7.6: G02,03 の基本アルゴリズム

図 7.7 は、y、z 座標に変化がない単純化した場合で、(A) は $(\mathcal{X}_0, \mathcal{Y}_0)$ から $(\mathcal{X}_d, \mathcal{Y}_d)$ まで時計回りに半径 r で描画する場合、(B) は半時計まわりに描画する場合である。現座標からの連続動作を原則として考えているので、例えば (B) で、現座標が $(\mathcal{X}_0, \mathcal{Y}_0)$ にあるのに、$(\mathcal{X}_d, \mathcal{Y}_d)$ から時計回りに描画するということはやらない。

7.3　円弧補間のための基礎解析

円弧補間のためには、いずれにしても、円弧を描く平面 $x-y$、$y-z$、$z-x$ と円弧の中心座標 (x, y)、(y, z)、(z, x) および半径 r が必要となる。

以下、$x-y$ 平面に描画する場合について述べるが、他の平面でも、座標を読み替えるだけである。

7.3.1　円弧中心の計算

円弧の中心座標は与えられず、現在位置、行先、半径から計算する必要がある。

まず円弧の中心を数学的に計算する（別の方法については第 7.4.3 節）。簡単のために、現

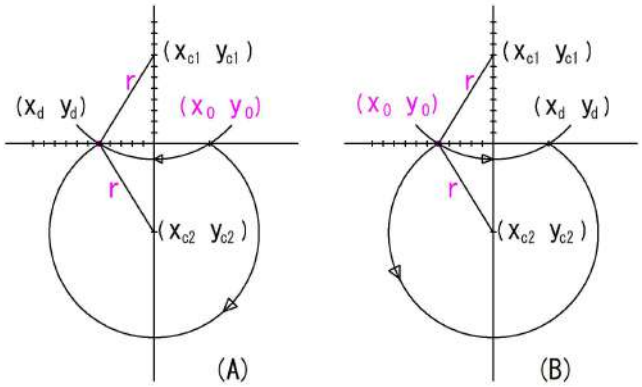

Figure 7.7: G02 のアルゴリズム

在位置を $O(0,0)$ とし、行先 (x_d, y_d) が与えらた時の円弧の中心座標 $P(x_r, y_r)$ を計算する。

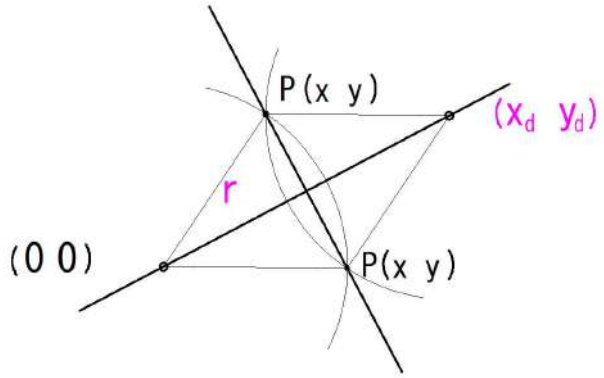

Figure 7.8: 中心座標の計算

図 7.8 より、現在位置と行先を結ぶ直線は、

$$y = \frac{y_d}{x_d}x$$

で与えられる。この直線の中点にに直交する直線は

$$y = kx + b = -\frac{x_d}{y_d}x + b,$$

であり、座標 $(1/2x_d, 1/2y_d)$ を通過するので、

$$b = \frac{1}{2}\Big(\frac{x_d^2 + y_d^2}{y_d}\Big),$$

が得られる。従って、

$$y = -\frac{x_d}{y_d}x + \frac{1}{2}\Big(\frac{x_d^2 + y_d^2}{y_d}\Big),$$

が得られる。この直線は (x_r, y_r) も通過するので、

$$y_r = -\frac{x_d}{y_d}x_r + \frac{1}{2}\Big(\frac{x_d^2 + y_d^2}{y_d}\Big) = kx_r + b,$$

円弧の中心座標を (x_r, y_r) とすると、

$$x_r^2 + y_r^2 = r^2.$$

従って、

$$x_r^2 + (kx_r + b)^2 - r^2 = 0.$$

$$x_r^2 + k^2x_r^2 + 2kbx_r + b^2 - r^2 = 0$$

$$(1 + k^2)x_r^2 + 2kbx_r + b^2 - r^2 = 0$$

$$x_r^2 + \frac{2kb}{1 + k^2}x_r + \frac{b^2 - r^2}{1 + k^2} = 0$$

係数を計算すると、

$$\frac{2kb}{1 + k^2} = \frac{2(-\frac{x_d}{y_d})}{1 + \frac{x_d^2}{y_d^2}}\frac{1}{2}\Big(\frac{x_d^2 + y_d^2}{y_d}\Big) = -x_d$$

$$\frac{b^2 - r^2}{1 + k^2} = \frac{\Big\{\frac{1}{2}\Big(\frac{x_d^2 + y_d^2}{y_d}\Big)\Big\}^2 - r^2}{1 + \frac{x_d^2}{y_d^2}}$$

$$= \frac{1}{4}(x_d^2 + y_d^2) - \frac{r^2y_d^2}{x_d^2 + y_d^2}$$

これらにより、

$$x_r^2 - x_dx_r + \frac{1}{4}(x_d^2 + y_d^2) - \frac{r^2y_d^2}{x_d^2 + y_d^2} = 0$$

$$x_r = \frac{1}{2}\Big\{x_d \pm y_d\frac{\sqrt{4r^2 - (x_d^2 + y_d^2)}}{\sqrt{x_d^2 + y_d^2}}\Big\}$$

$$y_r = \frac{1}{2}\Big\{y_d \mp x_d\frac{\sqrt{4r^2 - (x_d^2 + y_d^2)}}{\sqrt{x_d^2 + y_d^2}}\Big\}$$

が得られ、円弧の中心座標 (x_r, y_r) が計算できる。

7.3.2　円弧中心の計算アルゴリズム

図 7.9 において P 点が求める、半径 r の円弧の中心である。この点に至る、漸近アルゴリズムを考える。P 点は OS と直交する直線上の点であるから、OS の中点 $(1/2\mathcal{X}_d, 1/2\mathcal{Y}_d)$ となる $\mathcal{X}$ から始め、$\mathcal{X}$ をインクリメントするたびに対応する $\mathcal{Y}$ を求め、半径 r と比較すると、常に次式が成立する。

$$\mathcal{X}^2 + \mathcal{Y}^2 < r^2$$

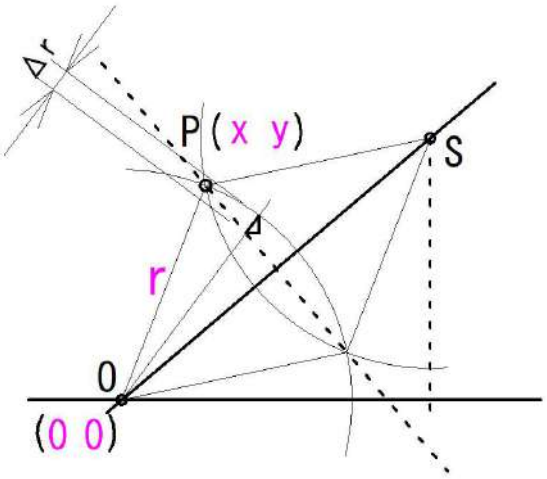

Figure 7.9: 中心座標の計算

ここで、

$$\alpha = r - \overline{OP},$$

と定義すると、$\alpha \geqq 0$ であり、かつ α は単調減少である。

従って、$\alpha \leqq 0$ まで $\mathcal{X}$ を計算すれば、$(\mathcal{X}, \mathcal{Y})$ が求める座標となる。この時、比較対象 r は与えられ、計算不要であるが、$\mathcal{X}^2 + \mathcal{Y}^2$ は、$(\mathcal{X}, \mathcal{Y})$ を求めるための基本変数値であるので、この計算はどうしても避けられない。

円弧中心座標を解析的に計算する場合に比べ、各漸近計算時に $\mathcal{X}^2 + \mathcal{Y}^2$ という単純計算だけでいいという事で満足すべきであろう。

7.3.3　オクテット判定

CNC の場合、円弧の開始座標 $(\mathcal{X}_0, \mathcal{Y}_0)$ は自由に選べず、加工ソフトウエアから与えられる。この時半径 r は必ず指定されるが、中心 $(\mathcal{X}_c, \mathcal{Y}_c)$ は与えられる場合と計算で求めね

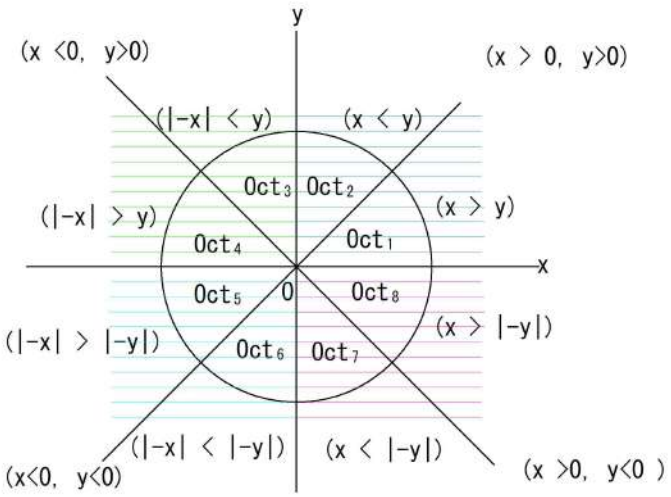

Figure 7.10: 8 分円

ばならない場合がある。いずれにしても、ここでは $(\mathcal{X}_c, \mathcal{Y}_c)$ が既知であるとし、$(\mathcal{X}_0, \mathcal{Y}_0)$ がどのオクテットに存在するかを判定する方法を考える。

しかし、これは容易で、

$$\mathcal{X} = \mathcal{X}_0 - \mathcal{X}_c, \quad \mathcal{Y} = \mathcal{Y}_0 - \mathcal{Y}_c$$

により、$(\mathcal{X}, \mathcal{Y})$ を計算し、これらの要素の値を比較すればいい。

図 7.10 を参照すると、次の様になる。

(1) 第 1 オクテット； $\mathcal{X} \geqq 0, \mathcal{Y} \geqq 0 \quad \mathcal{X} \geqq \mathcal{Y}$
(2) 第 2 オクテット； $\mathcal{X} \geqq 0, \mathcal{Y} \geqq 0 \quad \mathcal{X} < \mathcal{Y}$
(3) 第 3 オクテット； $\mathcal{X} \leqq 0, \mathcal{Y} \leqq 0 \quad |\mathcal{X}| < \mathcal{Y}$
(4) 第 4 オクテット； $\mathcal{X} \leqq 0, \mathcal{Y} \leqq 0 \quad |X| \geqq \mathcal{Y}$
(5) 第 5 オクテット； $\mathcal{X} < 0, \mathcal{Y} < 0 \quad |\mathcal{X}| \geqq |\mathcal{Y}|$
(6) 第 6 オクテット； $\mathcal{X} < 0, \mathcal{Y} < 0 \quad |\mathcal{X}| < |\mathcal{Y}|$
(7) 第 7 オクテット； $\mathcal{X} < 0, \mathcal{Y} < 0 \quad \mathcal{X} < |\mathcal{Y}|$
(8) 第 8 オクテット； $\mathcal{X} < 0, \mathcal{Y} < 0 \quad \mathcal{X} \geqq |\mathcal{Y}|$

7.4 円弧補間のアルゴリズム

直線補間と同様に、円弧も、半径の角度が 0-45 度、45-90 度までに分けて考え、さらに円全体はこれらの符号を変化させたアルゴリズムを組み合わせで実現する。従って、45 度までの円弧の描画が基本となる。

円の方程式は、

$$f(x,y) = x^2 + y^2 - r^2 = 0.$$

45 度までの 8 分円を考えているので、y 座標の増加に対して、x 座標は必ず減少する。従って、$\mathcal{Y}_0$ の増加に対して $\mathcal{X}_0$ はもとのままか、$\mathcal{X}_0 - 1$ となるが、この判断をするために、誤差関数を使う。(直線補間の時、直線が判別関数のもととなったように、円の方程式から誤差関数を作る。)

モニターに表示するグラフィックスの場合、8 分円の一つの値を計算、あとは対称性を利用して同時に描画する。しかし CNC では、これは不可能で、(円の場合) 8 分円を逐次連結していく必要がある。また、現在位置が指定されているため、現在位置が円の中心に対して、どの 8 分円内にあるかの判定も必要となる。

7.4.1 midpoint アルゴリズム

これ以降第 1 オクテット (Oct_1) について考える。開始点 $(\mathcal{X}_0, \mathcal{Y}_0)$ は格子点上ににある。

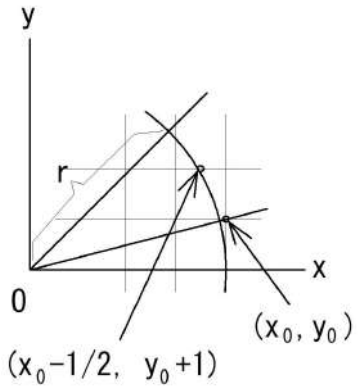

Figure 7.11: 初期値

直線に関する Bresenham アルゴリズム同様、$\mathcal{Y}$ 座標を 1 ステップ送った $\mathcal{Y}+1$ に対し、$\mathcal{X} - 1/2$ の点の誤差関数値を計算する。$(\mathcal{X}_0, \mathcal{Y}_0)$ 座標が始点なので、

$$D_0 = f(\mathcal{X}_0 \frac{1}{2}, \mathcal{Y}_0 + 1) - f(\mathcal{X}_0, \mathcal{Y}_0)$$

$$= (\mathcal{X}_0 - \frac{1}{2})^2 + (\mathcal{Y}_0 + 1)^2 - r^2 - ({\mathcal{X}_0}^2 + {\mathcal{Y}_0}^2 - r^2)$$

$$= \frac{5}{4} - \mathcal{X}_0 + 2\mathcal{Y}_0.$$

(グラフィックスの場合、y 軸のゼロから開始できるので、$\mathcal{Y}_0 = 0, \mathcal{X}_0 = r$ とする事ができる。)

ここで 5/4 という小数が出現するが、直線補間の時と同様、分数計算が出現しない様に、誤差関数を定義する。

$$g(\mathcal{X},\mathcal{Y}) = 4\{f(\mathcal{X}-1/2,\mathcal{Y}+1) - f(\mathcal{X},\mathcal{Y})\} = 4\Big\{D_n + (\mathcal{X}_0 - \mathcal{X}) + 2(\mathcal{Y}-\mathcal{Y}_0) + \frac{5}{4}\Big\}$$

$$= 4D_n - 4(\mathcal{X}-\mathcal{X}_0) + 8(\mathcal{Y}-\mathcal{Y}_0) + 5.$$

これを用いる事で、円弧の場合も整数演算のみで、次の座標が計算される。

【Step 1】

半径 r、中心 $(\mathcal{X}_c,\mathcal{Y}_c)$ を記憶しておく。中心が与えられない場合、計算で求める。

開始点（現在位置）を $(\mathcal{X}_0,\mathcal{Y}_0)$ とすると、この点は格子点上にあり、整数である。半径 r として与えられた数値は整数でも、計算上の半径 $\mathcal{R}$ は整数とは限らない。中心 $(\mathcal{X}_c,\mathcal{Y}_c)$ に関して、

$$\mathcal{X}_c = \mathcal{X}_0, \quad or \quad \mathcal{Y}_c = \mathcal{Y}_0$$

の場合は $r = \mathcal{R}$ である。

【Step 2】

開始点 $(\mathcal{X}_0,\mathcal{Y}_0)$ の誤差関数の初期値を計算しておく。

$$g(\mathcal{X}_i,\mathcal{Y}_i) = 4D_{i-1} - 4(\mathcal{X}_i - \mathcal{X}_{i-1}) + 8(\mathcal{Y}_i - \mathcal{Y}_{i-1}) + 5.$$

$D_i > 0$ ならば、点 $S = (\mathcal{X}_i - 1, \mathcal{Y}_i + 1)$ を選択。繰り越し誤差量を計算。

$$D_{i+1} = 4D_{i-1} - 4(\mathcal{X}_i - \mathcal{X}_{i-1}) + 8(\mathcal{Y}_i - \mathcal{Y}_{i-1}) + 5.$$

$D_i \leqq 0$ ならば、点 $T = (\mathcal{X}_i, \mathcal{Y}_i + 1)$ を選択。この時の繰り越し誤差は、

$$D_{i+1} = 4D_{i-1} + 8(\mathcal{Y}_i - \mathcal{Y}_{i-1}) + 5.$$

【Step 3】

計算された座標を中心 $(\mathcal{X}_c,\mathcal{Y}_c)$ に移動し、目的座標を得る。

$$\mathcal{X} = \mathcal{X} + \mathcal{X}cC, \mathcal{Y} = \mathcal{Y} + \mathcal{Y}_c$$

【Step 4】

step-3 から 5 まで、$\mathcal{X}_i >= \mathcal{Y}_i$ となるまで繰り返す。

7.4.2 Bresenham アルゴリズム理論解析

【解析】

円の方程式、$x^2 + y^2 = r^2$ より、

$$D = \mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2,$$

を考えると、点 $(\mathcal{X}_i, \mathcal{Y}_i)$ が円の外側にある時は正、内側で負、円上でゼロとなる。そこで、この式を点が円内外にあるかの判別に使用する。

円の描画は、(最初のオクテットを想定しているので)、$\mathcal{Y}_i$ をインクリメントした時、$\mathcal{X}_i$ をディクリメントするかそのままとするかを判断しながら実行する。つまり $(\mathcal{X}_i - 1, \mathcal{Y}_i + 1)$、あるいは $(\mathcal{X}_i, \mathcal{Y}_i + 1)$ を次の座標とするかを決めて行く。

$(\mathcal{X}_i - 1, \mathcal{Y}_i + 1)$ とする場合、誤差関数は

$$f(\mathcal{X}_i, \mathcal{Y}_i) = (\mathcal{X}_i - 1)^2 + (\mathcal{Y}_i + 1)^2 - r^2$$

$$= \mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2 + (1 - 2\mathcal{X}_i) + (2\mathcal{Y}_i + 1).$$

$(\mathcal{X}_i, \mathcal{Y}_i + 1)$ とする場合は、

$$f(\mathcal{X}_i, \mathcal{Y}_i + 1) = \mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2 + (2\mathcal{Y}_i + 1).$$

円の描画条件は

$$f(\mathcal{X}_i - 1, \mathcal{Y}_i + 1) < f(\mathcal{X}_i, \mathcal{Y}_i + 1)$$

なので、

$$\sqrt{\{\big(\mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2 + (2\mathcal{Y}_i + 1)\big) + (1 - 2\mathcal{X}_i)\}^2}$$

$$< \sqrt{\{\mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2 + (2\mathcal{Y}_i + 1)\}^2}$$

結局

$$g(\mathcal{X}_i, \mathcal{Y}_i) = 2\{\big(\mathcal{X}_i^2 + \mathcal{Y}_i^2 - r^2 + (2\mathcal{Y}_i + 1)\} + (1 - 2\mathcal{X}_i),$$

を判断する事になり、$g(\mathcal{X}_i, \mathcal{Y}_i) > 0$ ならば、 $(\mathcal{X}_i - 1, \mathcal{Y}_i + 1)$ を次の点とする。

モニター上の描画のように、$\mathcal{Y}_i = 0$ から開始可能な場合、$\mathcal{X}_i = r$ となり、初期値 $(3 - 2r)$ が得られる。

Bresenham アルゴリズムでは、比較対象が円の方程式なので、r を与えられた時の計算は 1 回ですむが、$(\mathcal{X}_i, \mathcal{Y}_i)$ 座標については各回ごとに、2 乗和計算が必要となる。

【処理手順】

以下は、モニター上に描画する場合の処理手順である。CNC の場合初期値は必ずしも $(3-2r)$ とはならない。しかし、考え方が参考になり、CNC との比較としても、円描画に対する理解が深まる。

Step 1 − 円弧の中心座標および半径を $x, y,$ および r に保存。P=0、Q=r と設定。
Step 2 − 誤差関数設定。 D = 3-2r。
Step 3 − P ≤ Q の間、step-8 まで繰り返し。
Step 4 − 行き先座標 (P, Q) が得られる。
Step 5 − P をインクリメント。
Step 6 − もし D < 0 ならば D = D + 4P + 6 とする。
Step 7 − それ以外 r = r - 1, D = D + 4(P-Q) + 10　とする。
Step 8 − 行き先座標決定 (P, Q)。
Draw Circle Method(X, Y, P, Q).
（モニター上の描画なので、8 オクテット同時に描く）
Call Putpixel (X + P, Y + Q).
Call Putpixel (X - P, Y + Q).
Call Putpixel (X + P, Y - Q).
Call Putpixel (X - P, Y - Q).
Call Putpixel (X + Q, Y + P).
Call Putpixel (X - Q, Y + P).
Call Putpixel (X + Q, Y - P).
Call Putpixel (X - Q, Y - P).

7.4.3　注記；円弧の中心計算

ここでは、円弧の中心座標計算の別例を記す。

図 7.12 を参照し、簡単のために、原点 $O(0,0)$ から点 $S(x,y)$ に対して半径 r の円弧を描く場合を考える。円弧の中心である P 点の座標が求めるものである。

$$\overline{OS} = \sqrt{x^2+y^2}$$

$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}}$$

$$\sin\theta = \frac{y}{\sqrt{x^2+y^2}}$$

$$\cos\phi = \frac{\sqrt{x^2+y^2}}{2r}$$

また、

$$(\overline{PT})^2 + \frac{x^2+y^2}{4} = r^2$$

より

$$\sin\phi = \frac{\overline{PT}}{r} = \frac{\sqrt{4r^2 - (x^2+y^2)}}{2r}$$

$$r\sin\phi = \frac{\sqrt{4r^2 - (x^2+y^2)}}{2}$$

一方

$$\cos(\theta+\phi) = \cos\theta cos\phi - \sin\theta\sin\phi$$

$$\sin(\theta+\phi) = \sin\theta cos\phi + \cos\theta\sin\phi$$

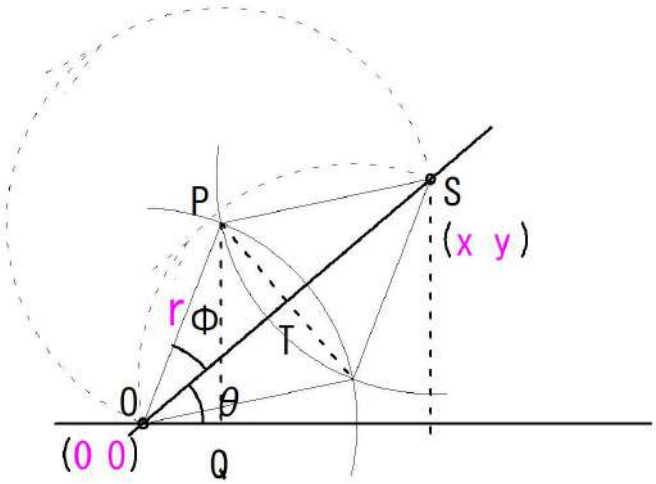

Figure 7.12: 幾何学的解析

$$P_x = \overline{OQ} = r\cos(\theta+\phi)$$

$$P_y = \overline{PQ} = r\sin(\theta+\phi)$$

従って、円の中心の座標は、

$$P_x = r\cos(\theta+\phi) = \frac{x}{\sqrt{x^2+y^2}}\frac{\sqrt{x^2+y^2}}{2r} - \frac{y}{\sqrt{x^2+y^2}}\sin\phi$$

$$= \frac{x}{2} - \frac{y}{\sqrt{x^2+y^2}} r\sin\phi$$

$$= \frac{x}{2} - \frac{y}{\sqrt{x^2+y^2}}\frac{\sqrt{4r^2-(x^2+y^2)}}{2}$$

$$= \frac{1}{2}\left\{x - y\sqrt{\frac{4r^2 - x^2 - y^2}{x^2 + y^2}}\right\},$$

$$P_y = r\sin(\theta + \phi) = \frac{1}{2}\left\{y + x\sqrt{\frac{4r^2 - x^2 - y^2}{x^2 + y^2}}\right\}.$$

これも第 7.2.1 節に述べた方法同様、計算量が多い。

参考資料

[1] FlatCAM: PCB prototyping
http://flatcam.org/

[2] CNC Made Easy
http://diymachining.com/g-code-example/

[3] Milling PCBs with cheap Chinese "desktop" CNC-router
https://forum.electricunicycle.org/topic/11205-milling-pcbs-with-cheap-chinese-desktop-cnc-router/

[4] UGS(Universal Gcode Sender)
https://winder.github.io/ugs_website/#platform

[5] Interfacing with Grbl
https://github.com/grbl/grbl/wiki/Interfacing-with-Grbl

[6] Grbl ソースコード
https://github.com/gnea/grbl

[7] G28 Versus G53
https://www.mmsonline.com/columns/g28-versus-g53

[8] Grbl v1.1 Commands
https://github.com/gnea/grbl/wiki/Grbl-v1.1-Commands

[9] G Codes
http://linuxcnc.org/docs/html/gcode/g-code.html#gcode:g90.1-g91.1

[10] Lesson 1: Bresenham's Line Drawing Algorithm
Dmitry V. Sokolov
https://github.com/ssloy/tinyrenderer/wiki/Lesson-1:
-Bresenham%E2%80%99s-Line-Drawing-Algorithm

[11] LinuxCNC ”G-Code” Quick Reference
http://linuxcnc.org/docs/html/gcode.html

[12] ブレゼンハムのアルゴリズム
Wikipedia
https://ja.wikipedia.org/wiki/%E3%83%96%E3%83%AC%E3%82%BC%E3%83%B3%E3%83%8F%E3%83%A0%E3%81%AE%E3%82%A2%E3%83%AB%E3%82%B4%E3%83%AA%E3%82%BA%E3%83%A0

[13] Wikipedia: Midpoint circle algorithm
https://en.wikipedia.org/wiki/Midpoint_circle_algorithm

[14] HOWTO draw circles, arcs and vector graphics in SDL?
https://stackoverflow.com/questions/38334081/howto-draw-circles-arcs-and-vector-graphics-in-sdl

[15] Circle Generation Algorithm
https://www.tutorialspoint.com/computer_graphics/circle_generation_algorithm.htm

[16] Mid-Point Circle Drawing Algorithm
https://www.geeksforgeeks.org/mid-point-circle-drawing-algorithm/

[17] A Fast Bresenham Type Algorithm For Drawing Circles - Oregon State University.
https://web.engr.oregonstate.edu/~sllu/bcircle.pdf

[18] Intuitive Wizard: G-code generator for milling a round contour
http://www.intuwiz.com/circle.html#.XRlCXuj7RaR

[19] Bresenham ’ s circle drawing algorithm
https://www.geeksforgeeks.org/bresenhams-circle-drawing-algorithm/

[20] HOW TO IMPROVE THE 2-AXIS CNC GCODE INTERPRETER TO UNDERSTAND ARCS
https://www.marginallyclever.com/2014/03/how-to-improve-the-2-axis-cnc-gcode-interpreter-to-understand-arcs/

[21] KiCad EDA
http://kicad-pcb.org/

[22] Managed Solutions

http://managedsolutions.com/2017/03/reference-common-g-code-extensions/

[23] Machinekit

http://www.machinekit.io/docs/gcode/gcode/#sec:G92_1-G92_2