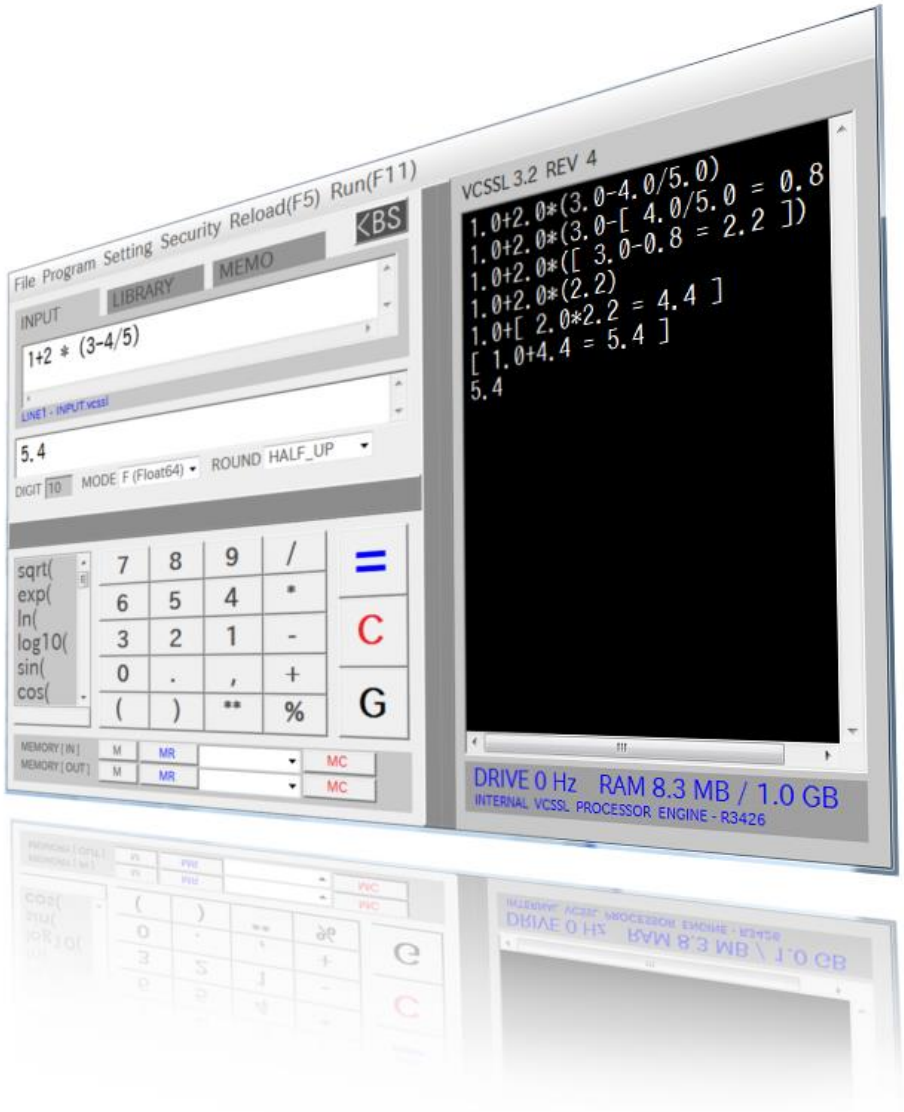
RINEARN
www.rinearn.com



RINEARN Processor 4.3

取扱説明書

第2版

はじめに

■ リニアンプロセッサーとは

リニアンプロセッサー（RINEARN Processor）は、数式の途中式表示機能やグラフプロット機能をはじめ、プログラミング機能やファイル入出力機能など、多彩な機能が搭載された関数電卓ソフトウェアです。通常の電卓としてのご利用はもちろん、ちょっとしたデータ解析や数値計算プログラムの作成と実行などにも便利です。

リニアンプロセッサーはクロスプラットフォーム設計で開発されており、オペレーティングシステムの種類を問わず、インストール不要で利用できます。

リニアンプロセッサー公式 Web サイト：

https://www.rinearn.com/processor/

■ ソフトウェアの利用ライセンスなどに関して

リニアンプロセッサーはどなたでも無償で利用できます。ただしライセンスにより、個人利用目的の範囲を超えた複製や再配布（ 特別に認可された場合を除く ）などの行為は禁止されています。

■ リニアンプロセッサーによる作成物の、権利と免責事項などに関して

リニアンプロセッサー開発者は、ユーザーがリニアンプロセッサーを使用して作成した数値、グラフ、データ、プログラムなどに関して一切の権利を主張しません。また、それら作成物に関する一切の責任も負担しません。それら作成物の著作権と責任は完全にユーザーに帰属します。

■ グラフ画像の利用に関して

リニアンプロセッサーを使用して作成したグラフ画像の著作権は、その作成者に完全に帰属し、作成者の判断でご自由にご使用頂けます。報告やクレジット表記などは一切不要です。

起動方法

まずは、リニアンプロセッサーを起動してみましょう。リニアンプロセッサーは、各種オペレーティングシステムにおいて、ダウンロードしてすぐに起動できます（ インストール不要 ）。

■ 【重要】なるべく新しい Java(R)実行環境をご利用ください

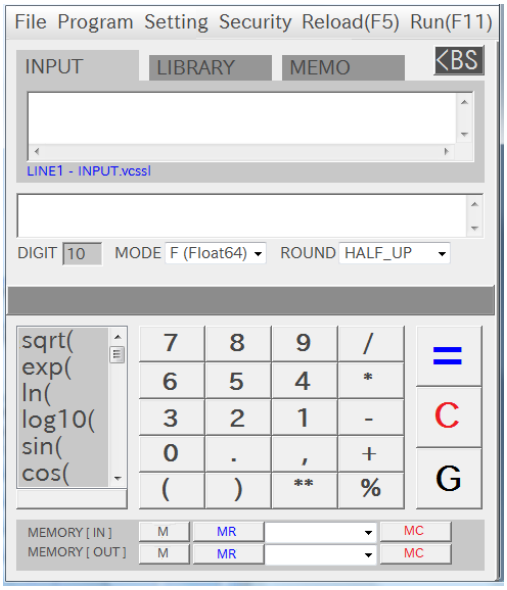
ご使用のコンピューターに導入されている Java(R) のバージョンが古い場合、リニアンプロセッサーが起動しないか、正しく動作しない（エラーで止まってしまう等）場合がございます。そのような場合は、下記ページへアクセスし、Java の実行環境を最新版へアップデートしてください。

https://www.java.com/ja/

■ 配布ファイルの解凍と設置、日本語化

まず、入手したリニアンプロセッサーの配布ファイル（ ZIP 形式 ）を解凍し、それを**フォルダごと、そのまま適当な場所に設置**してください。なお解凍方法は、配布ファイルを右クリックして「 ここに展開 」などを選択してください。続いて、画面を日本語化したい場合は**「日本語化.bat」をダブルクリックして実行**してください（※環境によっては日本語化すると文字化けする場合もあります）。

■ とりあえず起動してみる（ 各種オペレーティングシステム共通 ）



リニアンプロセッサーは、各種オペレーティングシステムにおいて、ダウンロードしてすぐに起動できます。それでは、とりあえず起動してみましょう。

先ほど解凍したフォルダ内にある、「 RinearnProcessor.jar（ JAR ファイル ） 」を**ダブルクリック**してください。するとリニアンプロセッサーが起動します。あとは、**デスクトップなどにショートカットを設置**しておくと便利です。

■ Windows(R)における高度な起動（拡張子関連付けやコマンド起動）

なお、メモリーを多く割り当てたい場合や、任意の拡張子のファイル（VCSSL プログラムなど）をダブルクリックするとリニアンプロセッサーで開きたい場合などは、解凍したフォルダ内にある「 **RinearnProcessor_4.*.*.bat** （ コマンドライン バッチファイル ） 」のほうも利用できます。特定の拡張子（ .vcssl など ）のファイルを、リニアンプロセッサーで開くように設定するには、まずそのファイルを右クリックし、**「 プログラムから開く 」**を選択してください。そして、上記の「 **RinearnProcessor_4.*.*.bat** 」を指定し、**「 この種類のファイルを開くときは、選択したプログラムをいつも使う 」**の項目を有効にして、「 OK 」を押すと完了します。

なお、コマンド入出力端末からリニアンプロセッサーを起動したい場合は、「 bin 」フォルダのパスを環境変数「 PATH 」に登録してください（詳細は「 Windows PATH 登録 」などで Web 検索してください）。すると、以下のようにコマンドでの起動が可能になります：

```
rinproc
```

起動と同時にファイル（ 例として test.vcssl ）を開きたい場合は、以下のようにしてください：

```
rinproc test.vcssl
```

これで、起動した時点で test.vcssl の内容が開かれた状態になりますので、必要に応じて編集し、「=」ボタンか F11 キーを押して実行してください。なお、開くファイルは、上の例のように VCSSL プログラムでなくても、数式をメモしたテキストファイルでも何でも構いません。

■ コマンドラインからの起動（Windows 以外の場合）

Windows 以外でも、bash 系のシェルを利用可能な環境では、コマンドでリニアンプロセッサーを起動できます。それにはまず、解凍したディレクトリ（フォルダ）を適当な場所に移動して、シェルを起動し、解凍フォルダ内にある「bin」フォルダまで cd コマンドで移動してください。ここでは例として、下記の通りの場所にあるとします。

```
/usr/local/bin/rinearn/rinearn_processor _4_*_*/bin/　（4_*_*の箇所はバージョン）
```

ディレクトリ内に移動した後に、以下のようにコマンド入力してください。

```
sudo chmod +x rinproc
```

最後に、ユーザーのホームディレクトリにある、「.bashrc（隠しファイル）」または「.bash_profile」もしくは「.profile」（どのファイルが有効かはオペレーティングシステムによって異なります）をテキストエディタで開き、最終行に下記の一行を追記してください。

```
export PATH=$PATH:/usr/local/bin/rinearn/ rinearn_processor _4_*_*/bin/
（4_*_*の箇所には正しいバージョンを記述してください）
```

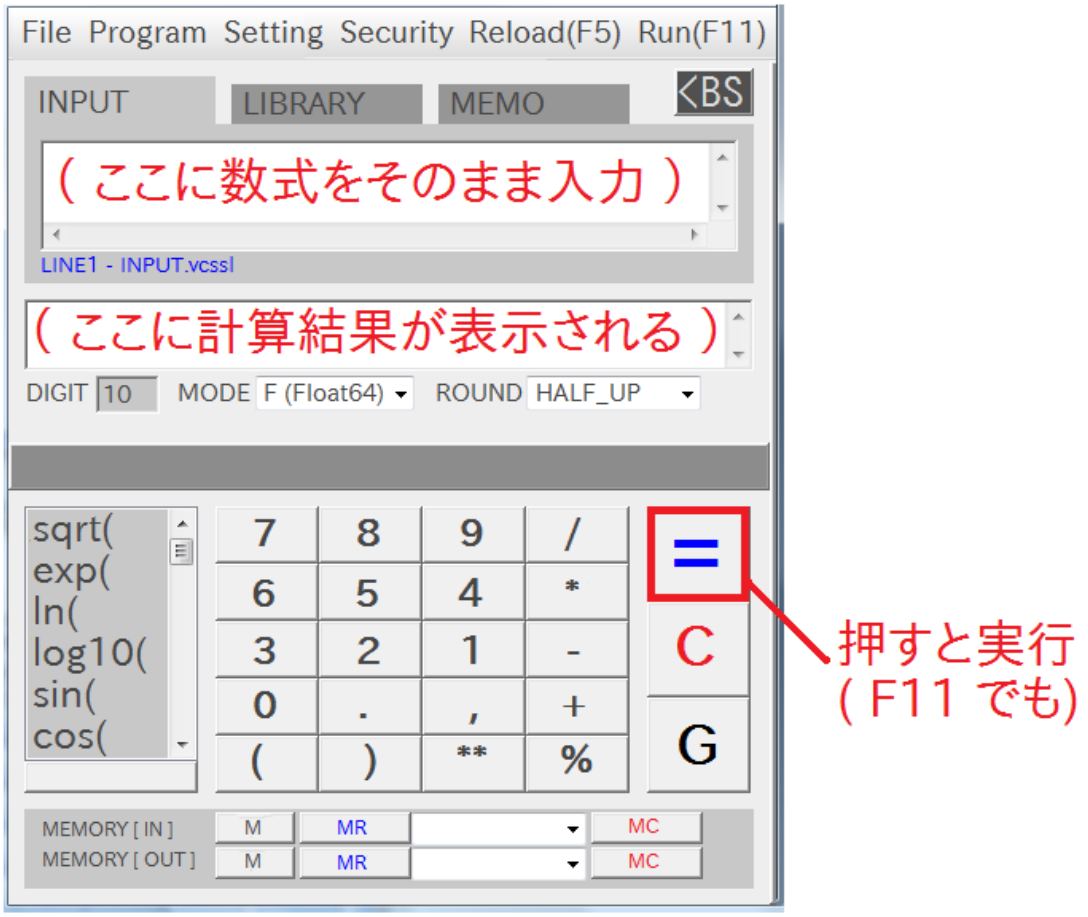
これで作業は完了です。再起動後、コマンド入力端末のどこからでも rinproc コマンドが使用可能となっています。コマンドの使用方法は先の Windows の場合と同様です。

数式を計算する

ここでは、電卓としての最も基本的な機能である、数式の計算を行ってみましょう。画面デザインや使い方は非常にシンプルで、基本的に普通の電卓と変わりません。

■ 起動時の画面

それではまず、リニアンプロセッサーを起動してください。すると、下図のような画面が表示されます（画面は英語のものです。日本語化した場合は、各項目が日本語になっています）。



上図のように、起動時のデザインは、スタンダードな普通の電卓と全く変わりません。実際、基本的な使い方も、普通の電卓と同じです。**なお、数式を入力する「 INPUT（入力） 」項目の文字サイズは、「Ctrl」+「U」キーで大きく、「Ctrl」+「D」キーで小さく調整できます。**

■ とりあえず計算してみる

それでは、実際に簡単な数式を計算してみましょう。上図で**「（ ここに数式をそのまま入力 ）」**と記載されている**「INPUT（入力）」項目**に、以下の数式を入力してください。

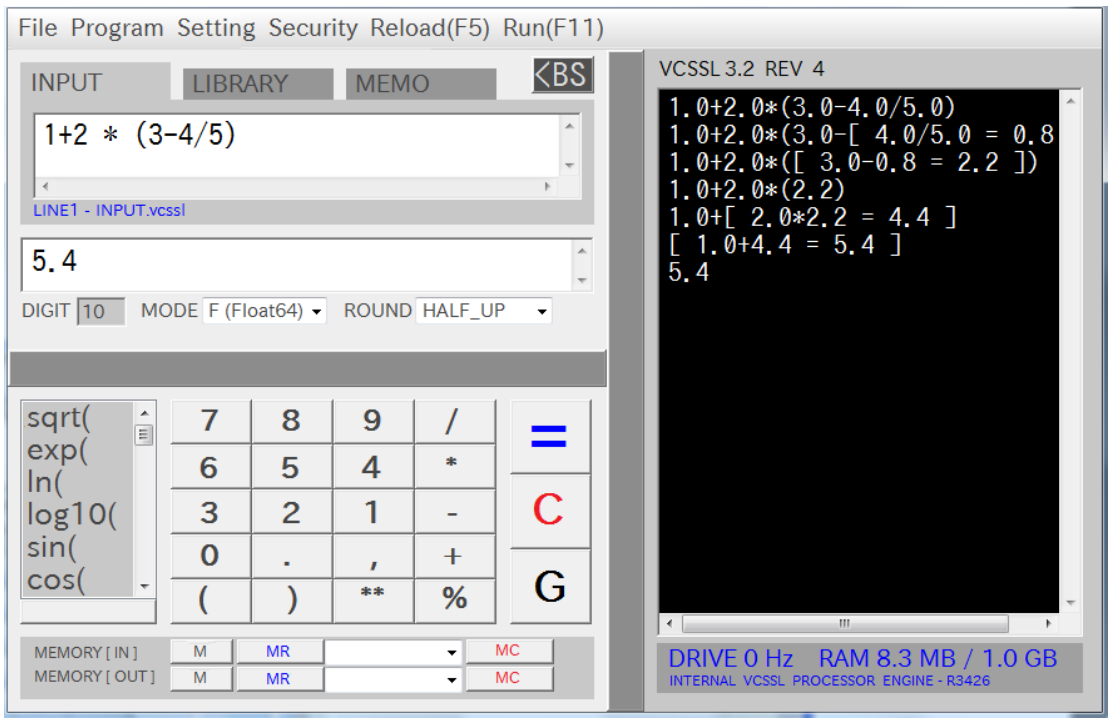
- INPUT（入力）-

```
1 + 2 * ( 3 - 4 / 5 )
```

数字や記号の入力は、もちろん画面に並んでいるボタンを押してもいいですし、キーボードから直接入力する事もできます。入力できたら、青色の**「 ＝ 」ボタン**を押してください。すると、上図で**「（ ここに計算結果が表示される）」**と記載されている**「OUTPUT（出力）」項目**に、下記の計算結果が表示されます（ 下図参照 ）。なお、キーボードの**「F11」キー**で実行する事もできます。

- OUTPUT（出力）-

```
5.4
```



ところで、「 ＝ 」ボタンを押すと同時に、画面が横に拡がり、黒いエリアが表示されました。これは**「 CONSOLE（ コンソール ）」**と言うもので、**計算の途中式や、エラーメッセージなどが表示される**領域です。実際、今回の計算では、以下のような途中式が表示されたはずです。

- CONSOLE（コンソール）-

```
1.0+2.0*(3.0-4.0/5.0)
1.0+2.0*(3.0-[ 4.0/5.0 = 0.8 ])
1.0+2.0*([ 3.0-0.8 = 2.2 ])
1.0+2.0*(2.2)
1.0+[ 2.0*2.2 = 4.4 ]
[ 1.0+4.4 = 5.4 ]
5.4
```

■ サポートされている演算と優先度

リニアンプロセッサーでは、下記の演算をサポートしています。

演算の記号（演算子）	演算の内容
+	加算（たし算）
-	減算（引き算）
*	乗算（かけ算）
/	除算（割り算）
%	剰余算（余り）
**	べき乗算（指数）

演算は、**優先度の高いものから先に処理**されます。優先度は、以下の順になっています。

べき乗算 ＞ 乗算 ＝ 除算 ＝ 剰余算 ＞ 加算 ＝ 減算

例えば、乗算（かけ算）は加算（たし算）よりも先に処理されます。なお、**同じ優先度の演算が並んでいる場合は、左から順に処理**されます。

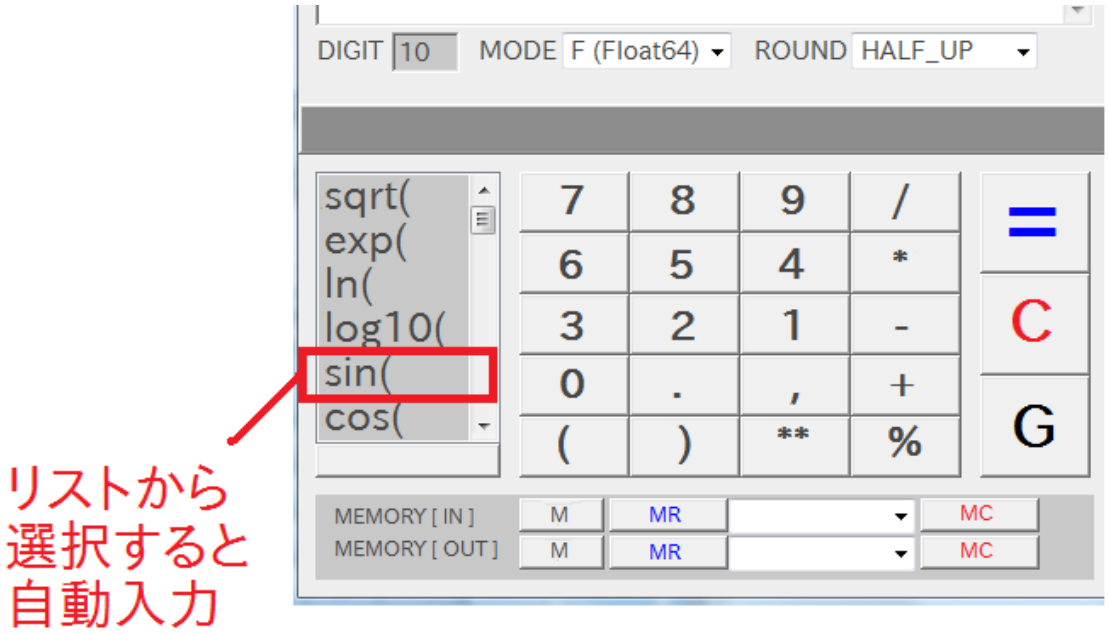
■ 関数を使う

今度は、sin や cos など、リニアンプロセッサーに最初から用意されている関数（一覧は後半の章参照）を使って計算してみましょう。先と同様に、以下の数式を計算してみてください。

- INPUT（入力）-

```
sin( 1 / 2 ) + cos( 3 / 4 )
```

この「sin」や「cos」という文字列は、キーボードから直接入力してもいいですが、画面左下にあるリストから選択すると、自動で入力されます。（下図参照）



この計算の結果は以下のようになります。

- OUTPUT（出力）-

```
1.211114407
```

そしてコンソールには、以下の途中式が表示されます。

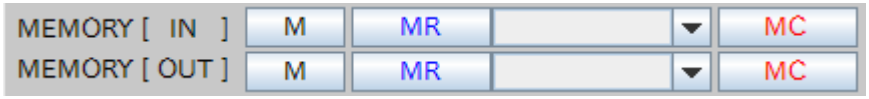
- CONSOLE（コンソール）-

```
sin(1.0/2.0)+cos(3.0/4.0)
sin([ 1.0/2.0 = 0.5 ])+cos([ 3.0/4.0 = 0.75 ])
[ sin(0.5) = 0.479425538604203 ]+[ cos(0.75) = 0.7316888688738209 ]
[ 0.479425538604203+0.7316888688738209 = 1.211114407478024 ]
1.211114407478024
```

■ 結果を記憶する ― メモリー機能

本章の最後に、計算結果や数式を記憶する、メモリー機能を使ってみましょう。リニアンプロセッサーのメモリー機能は、いくつでも無制限に記憶でき、さらに再起動後も内容が保持されるなど、とても強力です。

メモリー機能は、画面下部のパネルから使用します。（下図参照）



上下 2 つの段がありますが、上段が INPUT（入力）項目の内容を記憶するメモリー、下段がOUTPUT（出力）項目の内容を記憶するメモリーとなっています。各機能は以下の通りです。

「 M 」ボタン ― 現在の INPUT/OUTPUT エリアの内容を記憶します。記憶内容は「 MR 」ボタンの横にあるリストに追加されます。

「 MR 」ボタン ― を押すと、横のリストで選択されている値が、INPUT エリアに入力されます。

「 MC 」ボタン ― を押すと、リストに記憶されている値が全てクリアされます。

グラフを描画する

ここでは、リニアンプロセッサーの特徴の一つである、グラフ描画機能を使ってみましょう。

■ y (x) 形式の 2 次元グラフを描画する

リニアンプロセッサーには、数式から 2D/3D グラフを描画できる機能が用意されています。ここでは、実際にこの機能を使用してみましょう。

まず、画面右下にある「 **G** 」ボタンを押してください。するとファイルを選択するウィンドウが表示されます。その中には、以下のような複数のファイルが存在するはずです。これらは「 **グラフプログラム** 」と言い、描画するグラフの種類に応じて選択します。

グラフプログラム一覧

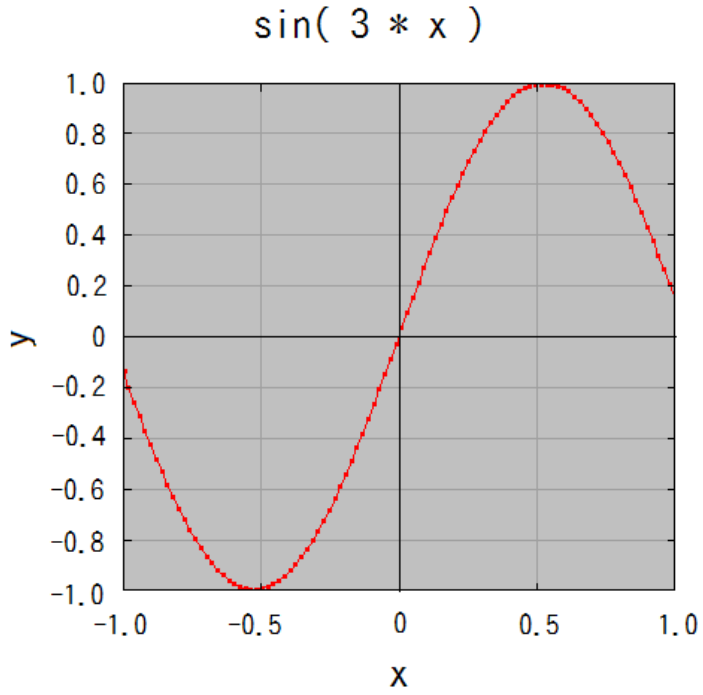
```
y( x )        … 2 次元グラフ
y( x, t )     … 2 次元グラフ （ アニメーション ）
z( x, y )     … 3 次元グラフ
z( x, y, t )  … 3 次元グラフ （ アニメーション ）
  …
```

ここでは「 **y(x)** 」を選択しましょう。選択するとシステムがグラフモードになり、「 SETTING 」というタイトルのウィンドウや、グラフ画面が出現します。SETTING ウィンドウでは、x-max 項目と x-min 項目でプロットする x 範囲、x-N 項目でプロット点数を指定できます。

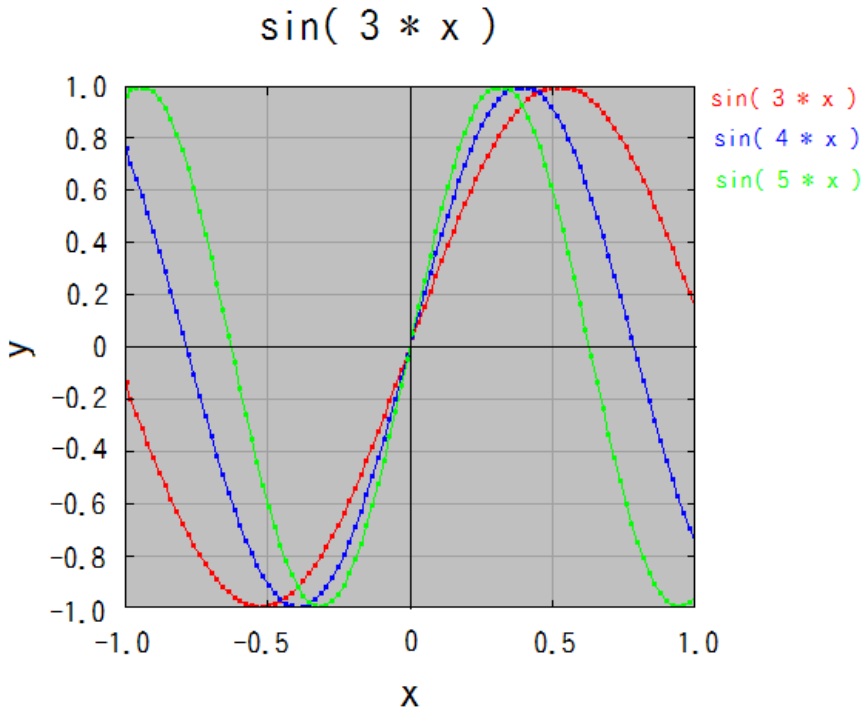
とりあえずここでは、「 y(x) = 」と書かれた項目に、以下のように式を記述し、「 **PLOT - プロット** 」ボタンを押してください。すると、グラフが描画されます。

- 「y(x) =」の項目 -

```
sin( 3 * x )
```



続いて、SETTING ウィンドウの「 y (x) = 」項目の内容を、「 sin(4 * x) 」と書き換えて、再度「 PLOT - プロット 」ボタンを押してください。すると、新しいグラフ線が、古いグラフ線の上に重ねて描画されます。さらに「 sin(5 * x) 」も重ねて描画させてみましょう。最終的に、下図のグラフが得られます。



■ z (x, y) 形式の 3 次元グラフを描画する

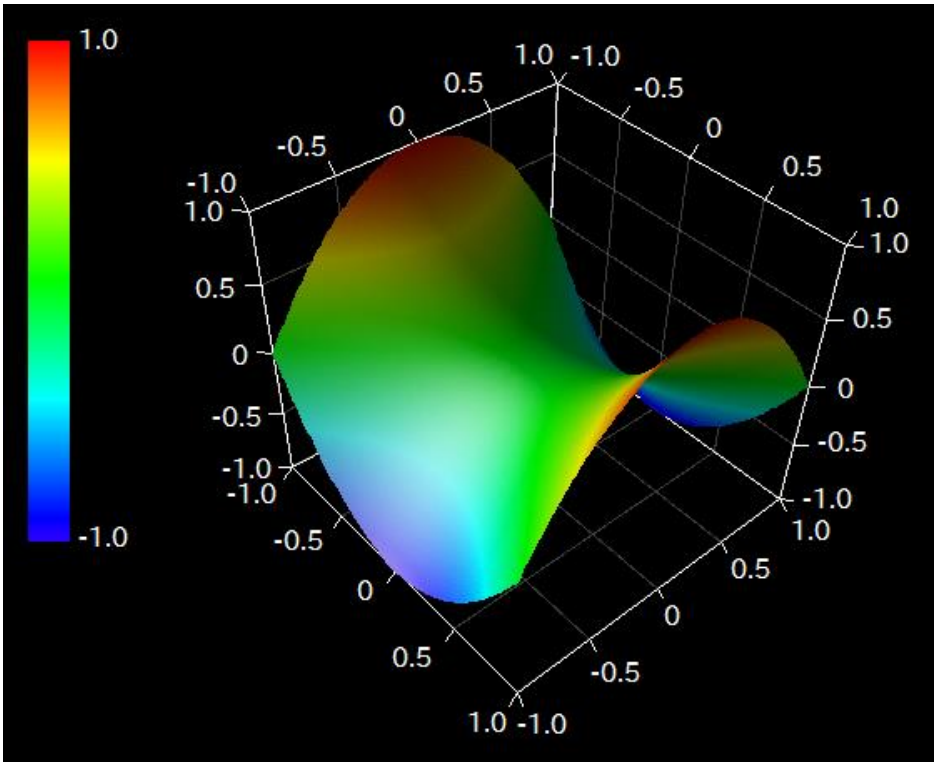
続いて、z(x, y)形式の 3 次元グラフを描かせてみましょう。

まず「 G 」ボタンを押し、今度は「 **z(x, y)** 」というファイルを選択してください。すると先ほどと同様、「 SETTING 」ウィンドウが表示されます。

とりあえずここでは、「 **z (x, y) =** 」と書かれた項目に、以下のように式を記述し、「 **PLOT - プロット** 」ボタンを押してください。すると、グラフが描画されます。

- 「z(x,y)=」の項目-

x * x - y * y



なお、ここで表示された 2D/3D グラフ表示画面は、単体のグラフソフトウェアとしても公開されており、様々なオプションや機能が利用できます。詳細は下記公式サイトをご参照ください。

・リニアングラフ 2D 公式サイト / https://www.rinearn.com/graph2d/
・リニアングラフ 3D 公式サイト / https://www.rinearn.com/graph3d/

■ その他の形式のグラフを描画する（ アニメーションなど ）

ここまでに使った他にも、アニメーション可能な「 y(x, t) 」や「 z(x, y, t) 」など、各種グラフプログラムが存在します。また、グラフプログラムは新たに入手したり、自分で独自に開発したりする事も可能です。詳しくは後半の「 プログラミング 」の章をご参照ください。

ここでは例として、2 次元のアニメーショングラフを描画する「 y(x, t) 」の扱い方に触れておきます。まず「 G 」ボタンを押し、「 y(x, t) 」というファイルを選択してください。すると「 SETTING 」ウィンドウが出現します。

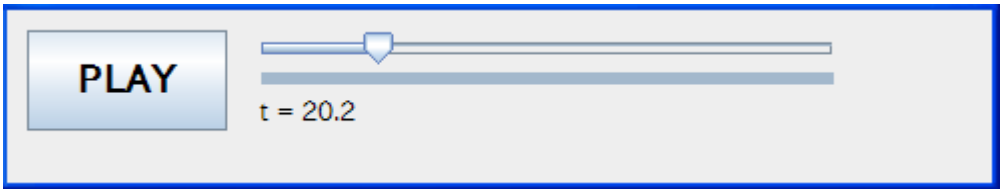
ここで「 y (x, t) = 」項目に以下のように入力してください。

- 「y (x,t) =」項目 -

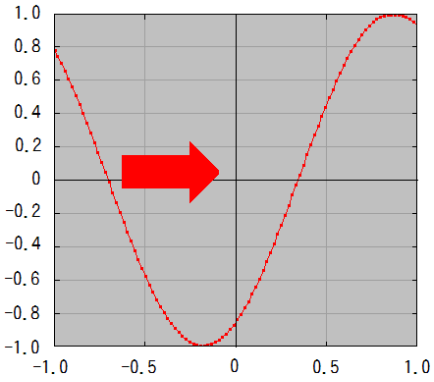
```
sin( 3 * x - t )
```

この「 t 」というのが時刻を表す文字です。アニメーション中は、この t の数値が時間経過に伴って変化していきます。

式を入力したら、続いて「 SET - セット 」ボタンを押すと、グラフ表示画面と、その上に「 ANIMATION 」と書かれたウィンドウが表示されます。



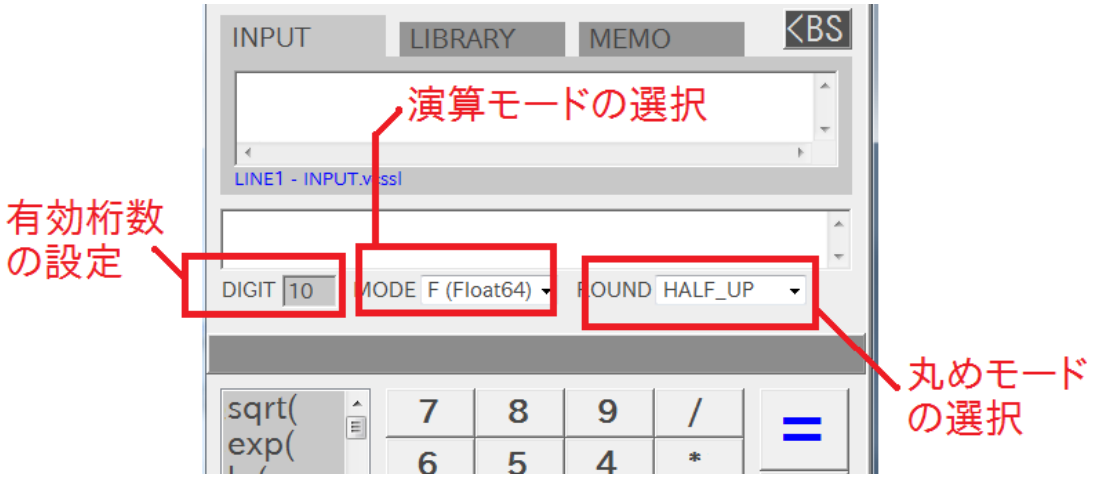
「 ANIMATION 」ウィンドウの「 PLAY 」ボタンを押すと、アニメーションの再生がスタートします。もう一度押すと再生がストップします。今回の例では、サイン波が右に進んでいきます。



桁数の設定と演算モード、演算精度

ここでは、表示する桁数の設定や、各演算モード時の精度について説明します。リニアンプロセッサーは、VF モードを使用する事で、大きな桁の演算も扱う事ができます。

OUTPUT（出力）項目の下で、桁数や演算モード、および丸めモードを選択できます。



■ 有効桁数の設定

有効桁数とは、簡単に言うならば **「 先頭から数えて何個有効な数字が続くか 」** という桁数です。厳密な解説はここでは割愛しますが、例えば 1.2345 も 123.45 も有効桁数は 5 桁です。

一般に、割り切れないような計算は何桁でも数字が続きますが、計算結果が表示される際には、OUTPUT エリアの左下にある**「 DIGIT 」**項目で設定した有効桁数に揃えられます。

なお、綺麗に割り切れる計算などで、計算結果の桁数が、設定された桁数よりも少なくなる場合もあります。そのような場合、標準設定では、少ないままの桁数で表示されます。もし、末尾に 0 の列を追加して、強制的に設定された桁数まで揃えたい場合（つまり有効桁数が何桁であるかを強調したい場合）は、**「 Setting（設定） 」** メニューの **「 Zeros - ゼロ揃え 」** 項目を有効にしてください。

■ 固定小数点表示

上で述べたように、DIGIT（有効桁）項目の設定値は有効桁数であって、いわゆる**「 小数点以下が何桁か 」** というような桁数とは全く無関係です。計算結果を、小数点以下の決まった桁数で切

り揃えたい場合（固定小数点表示）は、**「 Setting（設定） 」** メニューの **「 Fixed Digit - 固定小数点表示 」** 項目を有効にしてください。すると、DIGIT（有効桁）項目の下に **FIXED（固定桁）** 項目が出現するので、ここで小数点以下の桁数を指定してください。

なお、固定小数点表示は、結果を小数点以下の決まった桁で **“ 切り揃えて ”** 丸めるだけなので、もともとの有効桁数も足りるように設定しておく必要があります。例えば、DIGIT（有効桁）項目で設定された桁数では小数点以下 2 桁までしか数字が無いのに、FIXED（固定桁）項目で小数点以下 5 桁までに設定したとしても、もともと数字が無いので小数点以下 2 桁までしか表示されません。有効桁数が小数点以下何桁になるかは、計算内容によって異なるので、ご注意下さい。

■ 丸めモードの選択

計算結果を、設定された有効桁数に揃える際、**端数がどのように丸められるか**は、OUTPUT（出力）項目の右下にある**「 ROUND（丸めモード） 」**項目で選択します。例えば「 UP 」を指定すれば切り上げ、「 DOWN 」を指定すれば切り捨てを行います。

それ以外の 「 HALF_～ 」 のどれかを指定した場合、値によって切り上げか切り捨てのどちらかが行われ、**最も近い数値**に丸められます。例えば設定桁以降の端数が 500…1 と続くなら切り上げ、499…9 と続くなら切り捨てが行われます。ただし、設定桁以降の端数がちょうど 500…0 と続く場合、切り上げと切り捨てのどちらが近いというわけでもありません。このような場合をどのように扱うかは、「 HALF_～ 」のどれを指定するかで異なります。

「 HALF_UP 」を指定すれば、設定桁以降の端数が 500…0 と続く場合は切り上げられます。そして設定桁以降が 499…9 と続く場合は切り捨てられます。つまりこの場合、**設定桁の次の桁を（それ以降の桁は無視して）四捨五入したのと同様の結果**が得られます。

「 HALF_DOWN 」を指定すれば、設定桁以降の端数が 500…0 と続く場合は切り捨てられます。そして 500…1 と続く場合は切り上げられます。

「 HALF_EVEN 」を指定した場合は少し特別です。この場合では、設定桁以降の端数が 500…0 と続くとき、切り上げるか切り捨てるかは、その前の桁が偶数か奇数かによって反転します（偶数方向へ丸めます）。

なお、「 NONE 」を指定した場合は丸めを行わず、内部処理における演算結果をそのまま返します。演算結果を自分で丸めたい場合に使用して下さい。なお「 NONE 」の場合は、指定された桁数にはなりません。加えて、ゼロ揃え等の表示オプションも効きません。完全に内部処理における結果そのものを返します。

■ 演算モードの選択

少しコンピュータの仕組みに関わる内容になりますが、リニアンプロセッサーにおける数値の演算は、標準（F モード）では **「 2 進数 64bit 浮動小数点数 」** という形式で扱われます。これはコンピュータにおいて非常に高速に計算できる形式で、10 進数換算で約 16 桁前後の値を扱えます（ただし、2 進数特有の誤差を考慮する必要があります）。

場合によっては、16 桁を超える大きな桁数の小数を扱いたい場合や、2 進数誤差を嫌って 10 進数で演算してほしい場合もあります。また、小数ではなく整数として演算してほしい場合もあるでしょう。こういった様々な用途に対応するため、リニアンプロセッサーには、以下の 3 つの**「 演算モード 」** が存在します。

演算モード	**説明**
F (float64)	すべての入力値を約 16 桁の小数(2 進 64bit 浮動小数点数)として扱います。
VF (varfloat)	すべての入力値を任意桁の小数（10 進多倍長浮動小数点数）として扱います。
DIRECT	整数や小数など、異なる種類（型）の値を混在して扱います。型は C 言語系の**数値リテラル表記**で判断されます。例えば整数なら、「123」は 10 進数、「0b101」は 2 進数、「0o123」は 8 進数（※）、「0x123abc」は 16 進数とみなされます。「1.23」は 64bit 浮動小数点数、「1.23vf」は多倍長浮動小数点数となります。 ※ 8 進数リテラルについては、バージョン 4.2 までは 「 0（ゼロ）を頭に付ける 」という標準的なルールを採用していましたが、慣れない場合の混乱を防ぐため、バージョン 4.3 以降では「 0o（ゼロ・オー） 」を頭に付けるという記法に移行しました（VCSSL も同様です）。従来の記法も使用できますが、警告メッセージが表示されます。表示したくない場合は「設定」メニューから「互換性に関する警告を無視」を有効にしてください。

普通の電卓として特に重要なのは F モード（標準）と VF モードでしょう。これらは両者とも小数の演算を行うモードですが、**精度（有効桁数の限界など）と処理速度**が大きく異なります。特徴を次ページにまとめます。

■ F モード と VF モードの主な特徴

下の表は、一般的な用途向けである、F モードと VF モードの特徴について詳しくまとめたものです。

	F モード（標準）	VF モード
概要	内部処理に 64bit の 2 進数が使用されます。最大 16 桁程度まで扱えます。非常に高速で、数学関数も軽快で正確です。反面、末尾の数桁に、2 進数演算に特有の誤差などが生じます。	内部処理に任意桁数の 10 進数が使用されます。何桁でも扱えて、高精度です。反面、大きな桁数では、処理速度が低下します。特に数学関数は独自開発ライブラリのため、比較的重く、精度検証もまだ完全ではありません。
演算精度（桁）	限界で 16 桁程度です。信頼できるのは 14 桁程度で、現実的には、安全マージンを取ると 10 桁程度です。	任意桁数なので、必要に応じていくらでも桁数を増やせます。ただし桁数を増やすと、演算速度は低下します。
内部処理の型	64bit, 2 進数（ double ）	多倍長, 10 進数
内部処理における丸め(※)	開発言語における組み込み型（ double ）の丸め処理がそのまま使用されています。	Ver.4.2.48（VCSSL 3.3.12)以降から段階的に Half-even に移行中です。それ以前は単純な切り捨てです。
想定すべき誤差など	10 進/2 進変換に特有の誤差などに注意が必要です。**末尾の数桁には常に誤差が含まれると考えるべきです。**	10 進/2 進変換の誤差を気にする必要はありません。ただし、あくまで有限桁なので、割り切れない数などの丸め誤差は存在します。
演算速度	100 M FLOPS 程度 （ 数億回 / 秒 ）	最大 1 M FLOPS 程度 ※桁数が大きくなると遅くなり、除算の演算時間は特に低下します。
数学関数	開発言語における組み込み関数を使用しているため、非常に高速であり、かつ正確です。	歴史の浅い、独自開発のライブラリを使用しているため、比較的低速であり、精度も完全ではありません。 厳密性を要する場合は、適当な数値でテストする事が推奨されます。

※ここでの**「内部処理における丸め」**とは、先に述べた**「丸めモード」の設定項目とは無関係**であり、「内部で演算を行う際の丸め」の事を意味します（内部処理では、設定桁数よりも余裕を持った桁数で演算が行われます）。そうして演算を行った最終的な結果が、「丸めモード」で設定した方法で、設定桁数に丸められて表示されます。

■ 実際には設定桁数よりも大きな桁数で演算され、表示時に丸められる

さて、上の表では、精度や誤差など、細かい点が色々と記載されていますが、10 桁程度の計算で、小数を普通に扱うような場合であれば、このような点をあまり気にし過ぎる必要は無いでしょう。

なぜならリニアンプロセッサーは、**内部処理では設定桁数よりも大きな桁数で演算を行い、結果表示時に指定桁まで丸めて（丸めモードに依存、標準の HALF_UP 時は四捨五入）**、OUTPUT エリアに出力するからです。例えば、F モード時の 10 桁演算で、以下の数式を計算してみましょう。

- INPUT（入力）-

```
1.01 - 0.001
```

- OUTPUT（出力）-

```
1.009
```

- CONSOLE（コンソール）-

```
1.01-0.0010
[ 1.01-0.0010 = 1.0090000000000001 ]
1.0090000000000001
```

さて、コンソールに注目してください。この場合、内部処理では 17 桁の結果が得られた事が分かります。そして、その**最終桁は、本来「 0 」であるべきなのに「 1 」になっている**事が見て取れます。これは「桁落ち」という有名な演算誤差の一種です。こういった誤差は、内部処理における末尾桁の数桁まで影響する可能性があります。しかしながら、**結果は 10 桁に四捨五入（丸めモードに依存）されてから出力されたため、末尾の誤差「…001」の影響は切り捨てられた**事が分かります。なお、こういった例の他にも、例えば「 0.1 」という数値は 2 進数で表すと無限桁になってしまうため、0.1 を 10 回かけると 0.99999…となってしまうなどの誤差も有名です。

上のように、多くの場合は丸めによって補正されますが、**しかしながら、F モードにはこういった類の誤差が常に存在するという事は、あらかじめ認識した上で使用して下さい。**そして、これが問題とな

る場合には、小数演算では VF モード、整数演算では DIRECT モードなどを使い分けて下さい。

■ 大きな桁数と精度が必要な場合は、VF モードを使う

F モード時には、内部処理でも最大で 16～18 桁までしか扱えないため、**10 桁よりも大きい桁数で計算させる際は注意が必要**です。極端な例では、F モード時に 16 桁などで計算させると、桁数がギリギリで丸める余地が無くなってしまい、誤差が結果に見える領域に現れる可能性があります。

どうしてもそのような大きな桁数を扱いたければ、VF モードを使う事が推奨されます。VF モードでは、内部処理で無制限に桁を使えるため、ちゃんと安全マージンを取って計算した上で、丸めて表示する事ができます。

実際、先ほどの例では、VF モード時の場合、16 桁に設定しても内部ではより大きな桁数で計算され、16 桁に丸められます（丸めモードに依存、標準の HALF_UP 時は四捨五入されます）。

また、VF モードでは数千桁の計算も扱う事ができます。例として、**VF モード**を使い、**桁数を 1000 桁に指定**して、以下の数式を計算してみてください。（ 少し時間がかかります ）

- INPUT（入力）-

```
sin(1)**2 + cos(1)**2
```

- OUTPUT（出力）-

```
1.0...（ 1000 桁続く ）
```

このように、1000 桁の正しい値を得る事ができました。

ただし、VF モード時における数学関数の処理には、まだ歴史の浅い、独自開発のライブラリが使用されています。そのため、あらゆる入力値と桁数について、精度が完全に確認されているわけではありません。

VF モードで数学関数を使う場合、厳密性を要求されるような場面では、上のように結果が分かっている数式を用いて、予め動作検証を十分に行う事が推奨されます。

2 進/8 進/10 進/16 進数の整数演算

ここでは、DIRECT モードを使用した、2 進/8 進/10 進/16 進数の整数演算を扱います。

■ 数値ごとに異なる種類（型）を混在させる「 DIRECT 」モード

ここまでは、入力された数値をすべて小数として扱う、F モードと VF モードのみに絞って解説してきました。その理由は、それらが電卓ソフトとして分かりやすい挙動をするからです。対して、ここで扱う「 DIRECT 」モードは、電卓というよりはプログラミング言語に近い挙動をするため、慣れないと少し分かり辛い挙動をします。しかし、2 進数や 16 進数などが扱えるなど、強力な特徴もあります。

DIRECT モードでは、一行の数式の中でも、数値ごとに異なる種類（型）のものとして扱われます。特にリニアンプロセッサーで常用するのは、**整数（int 型）**と**小数（float 型）**という 2 種類が挙げられます。例えば、以下の数式において、「21」は整数、「1.2」は小数として扱われます。

- INPUT -

```
21 + 1.2
```

このように、小数の場合は「 **小数点があれば少数として扱われる** 」という、分かりやすいルールがあります。これに対して、少しややこしいのは整数です。

■ 整数には、何進法で記述するかによって、複数の記述法がある

整数には、以下のように複数の記述ルールがあります。

基数	例 （括弧内は 10 進法表現）	記述ルールの詳細
10 進数	12345 (12345)	普通に整数を記述する（小数点は付けない）。
2 進数	0b1010010 (82)	先頭に「0b」を付け、それ以降を 2 進数で記述する。
8 進数	023532 (10074)	先頭に「0」を付け、それ以降を 8 進数で記述する。
16 進数	0x12ac14 (1223700)	先頭に「0x」を付け、それ以降を 16 進数で記述する。

このように整数では、10 進数（10 進法で記述した数）の場合を除いて、先頭に特殊な表記を付

加します。そうすると、その通りにリニアンプロセッサーに解釈され、扱われます。

例として、16 進数（16 進法で記述した数）の整数を使用してみましょう。INPUT（入力）項目に以下のように記述し、**MODE（演算モード）項目を「 DIRECT 」にした上で**実行してみてください。

- INPUT（入力）-

```
0xff
```

これは 16 進数で「ff」、つまり 10 進数の「255」と同じ数値です。なお、16 進法での一桁は
1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
というように表します。10 進法では 9 の次に 10 に繰り上がりますが、16 進法では 16 ではじめて次の桁に繰り上がります。つまり 16 進数の f は 15 を意味し、16 進数の 10 は 16 を意味します。

- OUTPUT（出力）-

```
255
```

結果は上の通り、入力値を 16 進数と解釈した上で、それを 10 進数に変換した「255」が得られました。このように、出力は標準では 10 進数に変換して表示されます。

この出力を 2 進数/8 進数/16 進数で得たければ、入力を以下のように記述します。

- INPUT（入力）… 2 進数として表示したい場合 -

```
bin( 0xff )
```

- INPUT（入力）… 8 進数として表示したい場合 -

```
oct( 0xff )
```

- INPUT（入力）… 16 進数として表示したい場合 -

```
hex( 0xff )
```

これで望む基数で表現した数値が得られます。結果には先頭に「0b」「0」「0x」が付加されます。
上の **hex( )** ように、括弧（かっこ）の前に特定のキーワードが付いたものは、数学と同様に関数

と呼びます。ここでの hex 関数は、sin や cos といった、通常の数学関数と同じようなものだと思っても問題ありません。つまり **hex は 括弧の中身を 16 進数に変換する関数**という事です。

■ 基数が異なる整数の混合演算

整数の四則演算では、異なる基数（16 進数と 10 進数など）の整数を混在させる事が可能です（内部処理ではすべて 2 進数に変換して扱われます）。

- INPUT（入力）… 16 進数と 10 進数の混合四則演算 -

```
0xf2 + 14
```

- OUTPUT（出力）-

```
256
```

このように、16 進数の 0xf2（10 進数で 242）と、10 進数の 14 で加算を行い、正しい答えとして 256 を得られました。

これを 16 進数で表示するには、**全体を hex( ) で囲みます**。

- INPUT（入力）… 16 進数と 10 進数の混合四則演算結果を 16 進数で表示 -

```
hex( 0xf2 + 14 )
```

- OUTPUT（出力）-

```
0x100
```

この通り、256 を 16 進法で表現した正しい答え「100」を得ました。先頭の 0x は、それが 16 進数である事を現すだけで、読む際は数に含めない事に注意してください。

■ 整数と小数の混合演算

整数と小数を混在させた演算も可能です。10 進数同士だけでなく、他のものも混在可能です。

- INPUT（入力）-

```
0x100 * 1.7
```

- OUTPUT（出力）-

```
435.2
```

0x100（10 進数の 256 と同じ）に 1.7（10 進数）を掛けた結果が、10 進数として表示されました。このように、**整数と小数の混合演算では、結果は小数として扱われる**というルールがあります。

なお、これをそのまま 16 進数にしようと hex( )で囲うとエラーとなります。なぜなら **hex( ) 関数は、括弧の中身が必ず整数でなければ受け付けない**という決まりになっているからです。つまり小数を 10 進数以外で表示する事はできません。

しかしどうしても必要な場合には、以下のようにして**小数点以下を切り捨て、無理やり整数に変換**して使う事もできます。これは**キャスト**と呼ばれる強引な演算で、切捨ての分だけ精度が落ちます。

- INPUT（入力）-

```
hex( (int)( 0x100 * 1.7 ) )
```

- OUTPUT（出力）-

```
0x1b3
```

このように、**(int)( )** で囲った中身の小数点以下が切り捨てられ、435 になったうえで、それを 16 進法で表現した結果が得られました。

■ ビットごとの論理演算（ ビット AND、ビット OR、ビット XOR ）

整数では、**「 ビットごとの論理演算 」**という、特殊な演算を扱う事ができます。これはやや専門的な分野で使われるものであり、一般には馴染みが無いため、あまり深くまで踏み込んだ解説は割愛します。

リニアンプロセッサーでは、ビットごとの論理演算は、演算子ではなく関数として提供されます。これは、C 言語系の言語でビット排他的論理和にマッピングされる「 ^ 」演算子が、一般の電卓ソフトではべき乗（指数）の演算子として普及しているため、誤用を避けるための仕様です。

and( )がビット論理積、or( )がビット論理和、xor( )がビット排他的論理和を返す関数です。これらはパラメータ（引数）を 2 つ要するので、コンマ記号「 , 」で区切って記述します。

※ これらの関数は、「Setting（設定） > Comma – カンマ付き数値 」項目を OFF にしないとエラーになります。

- INPUT（入力）-

```
bin( or( 0b101010 , 0b010101 ) )
```

- OUTPUT（出力）-

```
0b111111
```

このように、2 つの 2 進数「101010」と「010101」に対して、ビット（2 進数での桁）ごとに論理和（どちらか片方でも 1 なら 1 になる）をとった結果「111111」が得られました。なお、and を使うとビットごとの論理積（両方 1 の場合だけ 1 になる）となり、 0 （ 0b0 ） が得られます。

■ 整数同士の除算（割り算）には注意が必要

整数同士で除算（割り算）を行う場合には、注意が必要です。詳しい例はプログラミングの章で扱いますが、**整数同士の演算は整数になるというルールがある**ため、整数同士の除算では、結果の小数点以下が切り捨てられます。つまり 1/2 が 0 になります。このルールは、慣れるまで頻繁にミスを招きがちなため、特に注意が必要です。

変数や関数を使用する

ここでは、リニアンプロセッサーで標準サポートされている変数（定数）や関数をご紹介します。

■ 標準でサポートされている変数（定数）や関数

リニアンプロセッサーでは、いくつでも独自の変数や関数を定義する事が可能ですが、一般的なものは最初から用意されており、最初から使用できます。ここでは、それらをご紹介します。

■ 変数（定数）

・PI

使用例 ： PI

内容 ： 円周率の値（3.141592653589793）です。

・pi( )

使用例 ： pi( )

内容 ： こちらも円周率ですが、定数では無く、円周率を求める関数です。VF モードで使用すると、1 万桁程度まで円周率の値を求める事ができます。この関数は定数値を求めるため、引数を取りません。この関数が求める桁数は、VF モードの設定桁数がそのまま適用されます。なお、計算のアルゴリズムにはガウス=ルジャンドル法が使用されます。

・E

使用例 ： E

内容 ： 自然対数の底（自然対数の底）の値（2.718281828459045)です。

■ 基本的な関数

・ **sqrt(平方根を求める値)　※日本語で「 平方根(…) 」も使用可**
使用例 ： sqrt(2)　または　平均(2)
内容 ： 平方根を求めます。

・ **exp(指数を求める値)**
使用例 ： exp(2)
内容 ： 自然対数の底（ネイピア数）による指数関数を求めます。

・ **ln(自然対数を求める値)　※日本語で「 自然対数(…) 」も使用可**
使用例 ： ln(100)　または　自然対数(100)
内容 ： 自然対数を求めます。

・ **log10(常用対数を求める値)　※日本語で「 常用対数(…) 」も使用可**
使用例 ： log10(100)　または　常用対数(100)
内容 ： 常用対数（10 を底とした対数）を求めます。

・ **fac(階乗を求める値)　※日本語で「 階乗(…) 」も使用可。**
使用例 ： fac(5)　または　階乗(5)
内容 ： 階乗の値を求めます。

・ **abs(絶対値を求める値)　※日本語で「 絶対値(…) 」も使用可**
使用例 ： abs(-2)　または　絶対値(-2)
内容 ： 絶対値を求めます。

・ **rad(ラジアン値を求める度数)　※日本語で「 ラジアン(…) 」も使用可**
使用例 ： rad(45)　または　ラジアン(45)
内容 ： 角度の度数をラジアン値に変換します。

・ **deg(度数を求めるラジアン値)　※日本語で「 度(…) 」も使用可**
使用例 ： deg(PI / 4)　または　度(PI/4)
内容 ： 角度のラジアン値を度数に変換します。

■ 三角関数

・ sin(角度のラジアン値)

使用例 ： sin(PI/2)

内容 ： sin 関数の値を求めます。角度はラジアンで指定してください。

・ cos(角度のラジアン値)

使用例 ： cos(PI)

内容 ： cos 関数の値を求めます。角度はラジアンで指定してください。

・ tan(角度のラジアン値)

使用例 ： tan(PI/4)

内容 ： tan 関数の値を求めます。角度はラジアンで指定してください。

■ 逆三角関数

・ asin(角度を求めたい sin 値)

使用例 ： (asin(1) + acos(1))*2

内容 ： arcsin 関数の値を求めます。sin 値を指定すると、対応する角度をラジアンで返します。

・ acos(角度を求めたい cos 値)

使用例 ： (asin(1) + acos(1))*2

内容 ： arccos 関数の値を求めます。cos 値を指定すると、対応する角度をラジアンで返します。

・ atan(角度を求めたい tan 値)

使用例 ： atan(1) * 4

内容：arctan 関数の値を求めます。tan 値を指定すると、対応する角度がラジアンで返されます。

■ 双曲線関数

・ sinh(値)
使用例 ： sinh(0.1)
内容 ： ハイパボリックサイン関数の値を求めます。

・ cosh(値)
使用例 ： cosh(0.1)
内容 ： ハイパボリックコサイン関数の値を求めます。

・ tanh(値)
使用例 ： tanh(0.1)
内容 ： ハイパボリックタンジェント関数の値を求めます。

■ 逆双曲線関数

・ asinh(対応値を求めたい sinh 値)
使用例 ： asinh(5.0)
内容 ： arsinh 関数の値を求めます。

・ acosh(対応値を求めたい cosh 値)
使用例 ： acosh(5.0)
内容 ： arcosh 関数の値を求めます。

・ atanh(対応値を求めたい tanh 値)
使用例 ： atanh(0.1)
内容 ： artanh 関数の値を求めます。

■ 統計関数

・ **sum(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 和（…） 」も使用可**
使用例 ： sum(1, 2, 3, 4, 5)　または　和(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの和を返します。

・ **mean(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 平均（…） 」も使用可**
使用例 ： mean(1, 2, 3, 4, 5)　または　平均(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの平均値を返します。

・ **va(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 不偏分散（…） 」も使用可**
使用例 ： va(1, 2, 3, 4, 5)　または　不偏分散(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの不偏分散(分母は N-1)を返します。

・ **van(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 分散（…） 」も使用可**
使用例 ： van(1, 2, 3, 4, 5)　または　分散(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの分散(分母は N)を返します。

・ **sd(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 不偏標準偏差（…） 」も使用可**
使用例 ： sd(1, 2, 3, 4, 5)　または　不偏標準偏差(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの不偏標準偏差(平方根内の分母は N-1)を返します。

・ **sdn(値 1, 値 2, 値 3, … , 値 N)　※日本語で「 標準偏差（…） 」も使用可**
使用例 ： sdn(1, 2, 3, 4, 5)　または　標準偏差(1, 2, 3, 4, 5)
内容 ： 複数の値を受け取り、それらの標準偏差(平方根内の分母は N)を返します。

■ 微積分 / 微分方程式解析

※これらの関数は、リニアンプロセッサー 4.0 REV-8 から新たに追加された新しい関数であり、未発見の不具合を含む可能性があります。ご使用の前に、ある程度の検証を行ってください。

・ integ("積分する式", 区間下端 a, 区間上端 b)

使用例 ： integ("sin(x)", 0, PI)

内容 ： 指定された式を、区間 a ～ b まで積分し、値を返します。

積分する式は、ダブルクォーテーション記号「 " 」でくくって指定してください（そのまま文字列として受け取られます）。また、式中の被積分変数は小文字の x を使用してください。

この関数の実装では、数値積分のアルゴリズムにシンプソン法を採用しています。標準では 50000 ステップの刻み区間に分割して計算されます。この数を設定したい場合は、LIBRARY エリアを開き、「 INTEG_DEFAULT_N 」という変数の値を変更してください。

なお、「 Program 」メニュー ＞「 DiffAndIntegration_微積分」フォルダから、グラフ描画機能付きの、より高度な機能を利用できます。

・ diff("微分する式", 微分地点の x, 微分幅 dx)

使用例 ： diff("sin(x)", PI, 0.00001)

内容 ： 指定された式を微分し、微分地点 x における微分値を求めます。

積分する式は、ダブルクォーテーション記号「 " 」でくくって指定してください（そのまま文字列として受け取られます）。また、微分対象の変数は小文字の x を使用してください。

dx は数値微分の微分幅です。基本的に小さく取るほど精度が向上しますが、あまり小さすぎても誤差が効いてきます。最適な値は、式と x によって異なります。

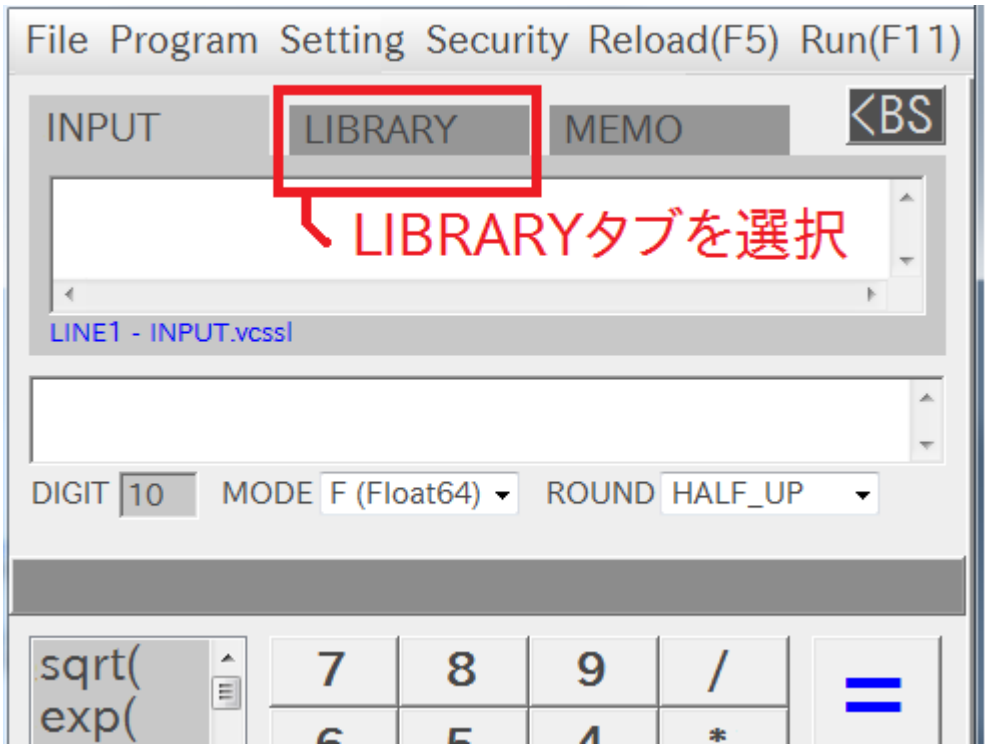
なお、「 Program 」メニュー ＞「 DiffAndIntegration_微積分」フォルダから、グラフ描画機能付きの、より高度な機能を利用できます。

変数や関数を定義する

ここでは、独自の変数や定数を定義し、数式中で使用する方法について解説します。

■「 LIBRARY（ライブラリ）」項目

リニアンプロセッサーでは、独自の変数や関数を、いくつでも自由に定義し、数式中で使用する事ができます。それらは**「 LIBRARY（ライブラリ）」**項目に定義します。LIBRARY エリアには、INPUT（入力）項目の上にある**「 LIBRARY（ライブラリ）」**と書かれたタブをクリックすると書き込めます。



なお、LIBRARY（ライブラリ）項目の編集内容は自動的に保存されます。そのため、特に保存したり、読み込んだりする必要はありません。**万一、色々と間違ってエラーだらけになってしまった場合は、etc フォルダ内にある「 LIBRARY.vcssl 」を削除すれば、初回起動時のものが再生成されます。**

■「 import Math ; 」は消してはいけない

LIBRARY（ライブラリ）項目の先頭付近には、以下の一行が記述されているはずです。

- LIBRARY（ライブラリ）-

```
import Math ;
```

この行は、**sin や cos など、リニアンプロセッサーで最初から使える関数を読み込む**ためのものです。
従って、**この行を消してはいけません。**消してしまうと sin や cos が使えなくなります。

■ 変数を定義する

それでは、変数を定義してみましょう。変数の定義は、以下のような 1 行を書き加えて行います。

```
float 変数名 = 値 ;
```

最初の「 **float** 」というのは、**小数を扱う変数**を定義するための用語だと思ってください（型と呼びます）。また、**行末の「 ; 」記号はついつい忘れがち**なので、注意が必要です。

実際に、**x という名前**で、**1.0 の値**を持つ変数を定義してみましょう。LIBRARY エリアに一行だけ追記し、以下のようにします。

- LIBRARY（ライブラリ）-

```
import Math ;

float x = 1.0 ; // x という名前で、1.0 という値の変数を定義
```

追記された最終行の、**// （ ダブルスラッシュ ）以降はコメントであり、無視されます。**これで、**x という名前**で、**1.0 の値**を持つ変数が定義できました。

定義した変数 x を使って、実際に計算を行ってみましょう。「 INPUT 」タブをクリックして INPUT エリアに戻り、以下の数式を計算してみてください。

- INPUT（入力）-

```
1 + x
```

この計算の結果は以下の通りです。

- OUTPUT（出力）-

```
2.0
```

このように、正しい値が得られました。

■ 関数を定義する

続いて、関数を定義してみましょう。関数は、以下のような形で定義します。

```
float 関数名( float 引数名 ) {
    return 計算内容 ;
}
```

ここでも「 float 」というキーワードが登場していますが、とりあえず今はあまり気にしないでください。小数を受け取って、小数を返すという意味なのですが、最初は全体を暗記した方が早いです。

「 引数（ひきすう） 」という耳慣れない言葉も登場していますが、これは関数を使う際、（ ）の中に書いた値が格納される、特別な変数です。パラメータと言った方が分かりやすいかもしれません。

それでは例として、「 half 」という名前で、受け取った値を２で割って返す関数を定義してみましょう。ライブラリに内容を追記し、以下のようにします。

- LIBRARY（ライブラリ）-

```
import Math ;

//  値を２で割って返す関数
float half( float a ) {
   return a / 2.0 ;
}
```

ここで、a/2 ではなく、a/2.0 としている事に注意が必要です。ここではとりあえず、**「 LIBRARY エリアでは、数値には必ず小数点を付けておく 」**と覚えておいてください。詳しい説明は割愛しますが、**小数点のない値同士で割り算を行った場合、余りが切り捨てられます。**

これは不具合ではなく、様々な事情により、そういうルールになっているのです。そして、余りが切り捨てられるのを防ぐために、数値には必ず小数点を付けるクセを付けておきましょう。

それでは、実際に half 関数を使用してみましょう。

- INPUT（入力）-

```
half( 1 )
```

ここで、括弧内に「1」と書いて、half 関数を使用しました。**この呼び出し時に書いた「1」という値が、half 関数の「a」に入ります。**この性質が、関数の最も重要な点です。「1」に対する計算、「2」に対する計算…というように、何度も同じような処理を書かなくても、**とりあえず「a」と置いて処理を書けばいい**わけです。そして、**使う時に a に具体的な数値が入る**というわけです。

さて、この計算の結果は以下の通りになります。

- OUTPUT（出力）-

```
0.5
```

このように正しい値が得られました。

■ 引数が複数ある関数

関数では、引数を複数使用する事もできます。それには、引数をカンマ記号で区切って定義します。例として、2 つの値の積を返す関数 mul を定義してみましょう。LIBRARY エリアに以下のように記述します。

- LIBRARY（ライブラリ）-

```
import Math ;

// 2 つの値の積を返す関数
float mul( float a, float b ) {
   return a * b ;
}
```

それでは INPUT エリアに戻り、以下の数式を計算してみましょう。

- INPUT（入力）-

```
mul( 2, 8 )
```

この計算の結果は以下の通りです。

- OUTPUT（出力）-

```
16
```

このように正しい値が得られました。

■ VF モード時に使用する変数と関数

ここまでで扱ってきた変数と関数の定義は、**F モードと DIRECT モードの時**にしか使用できません。使おうとしても、引数の型が一致しないという旨のエラーが表示されます。

それでは、**VF モードに使用する変数と関数の定義**は、どうすればよいのでしょうか。

それにはまず、これまで何度も登場した「 **float** 」というキーワードの代わりに、「 **varfloat** 」というキーワードを使用する必要があります（ ※ 実は VF モードの「 **VF** 」は、「 **V**ar**F**loat 」の略です ）。そして、1.0 や 2.0 などの数字の末尾に、**vf** の 2 文字を付けるようにします。2 進数や 16 進数の整数では先頭に 2 文字を付けましたが、それと同じ感覚で、今度は末尾に付けるだけです。

実際に、上で定義した変数 x と half 関数を、VF モード ON 時に使用するために書き直してみましょう。内容は以下のようになります。

- LIBRARY（ライブラリ）-

```
import Math ;

varfloat x = 1.0vf ; // x という名前で、1.0 という値の変数を定義（VF モード ON 用）

// 値を 2 で割って返す関数（VF モード ON 用）
varfloat half( varfloat a ) {
   return a / 2.0vf ;
}
```

これらの変数と関数は、VF モード時に使用できます。

プログラム

ここまでは、いわゆる電卓ソフトとしての、普通の機能を扱ってきました。しかし、リニアンプロセッサーでは、そういった普通の機能よりも高度な「 プログラム 」というものを利用する事ができます。

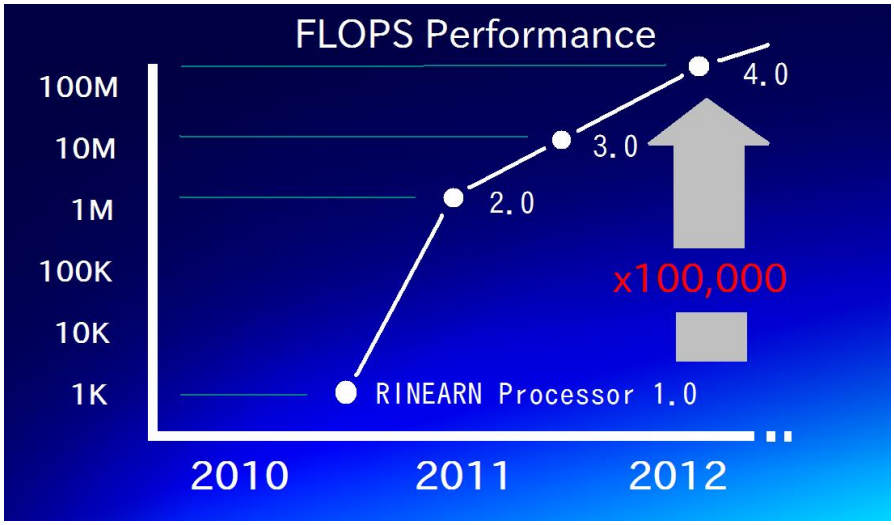
ここでは、プログラムの仕組みと利用方法を解説します。

■ リニアンプロセッサーの中核処理エンジンは、とても高性能 ！

リニアンプロセッサーにおけるほぼ全ての処理は、「 **VCSSL エンジン** 」という中核処理エンジンが担っています。このエンジンは、電卓ソフト用の処理エンジンとしては高性能で、**1 秒間に 1 億回（ 100 M FLOPS ）**を超える高速計算が可能です。

しかし、人間が電卓のボタンを叩けるのはせいぜい 1 秒に数回なので、普通に使っていても全く処理速度を活かしきれず、ただオーバースペックなだけになってしまいます。

もしも、リニアンプロセッサーの処理エンジンの性能を最大限に発揮できたなら、**複雑な処理を全自動化して一瞬で完了**させたり、**膨大なデータをまとめて処理**したり、**精密な科学技術計算**を行ったり、様々な事ができるでしょう。これを可能にしてくれるのが**「 プログラム 」**の仕組みです。



■「 プログラム 」とは「 する事のリスト 」

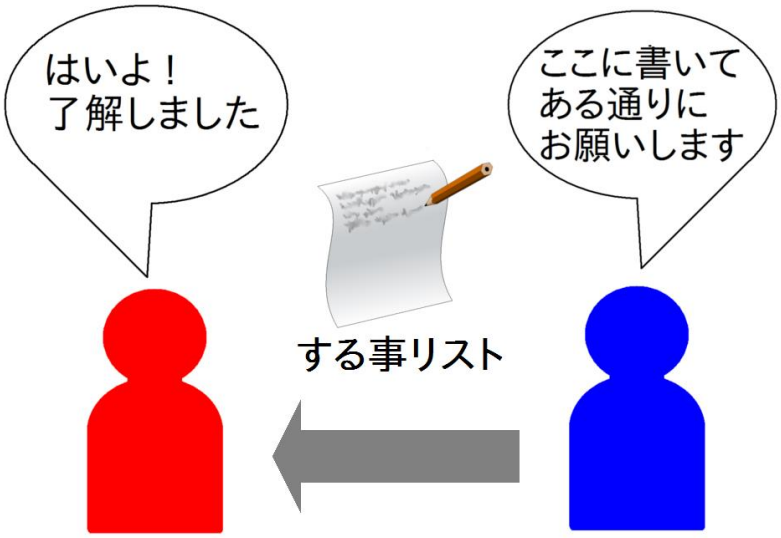
それでは、「 プログラム 」とは一体何なのでしょうか。それは、「 する事のリスト 」です。

例えばあなたが、自分で電卓のボタンを押して複雑な計算を行う代わりに、他の人にやってもらう場合を考えてみてください。その場合、どういった数値に、どういった内容の計算をして、結果をどのようにまとめるのか、「 する事のリスト 」を紙に書いて渡す事でしょう。例えば、「 1 から 100 までの素数の和を求めよ 」といった計算の場合は、下記のようなリストになるでしょう。

▼ 人間が人間に処理を頼む時の、「 する事のリスト 」

・ここからの内容を、2 から 100 まで繰り返してください。(繰り返し中の回数を i とします)
　・もし i を、2 から i-1 まで全ての数で割って、余りが一回も 0 にならないなら、それは素数です。
　・もし i が素数なら、i の値を合計に足してください。
・繰り返しはここまでです。
・最後に、合計の値を教えてください。

※ 実はこの場合には 2,3,5,7 で割った余りのみで素数か判定できるのですが、簡単のため上のようにしています



これを受け取った人は、そこに書かれている通りに電卓を叩いて、計算を行ってくれるわけです。実は、**この 「 する事のリスト 」 が、プログラム**なのです。

ただし、上のような普通の文章は、人間が人間に伝える場合には良いのですが、機械に伝える場合には、言い回しなどが不必要に複雑だったり、曖昧だったりして、理解するのが難しいという問題があります。よく 「 日本語は難しい 」 と言われますが、ただでさえ人間にとっても難しいのに、機械にとっては難し過ぎるのです。

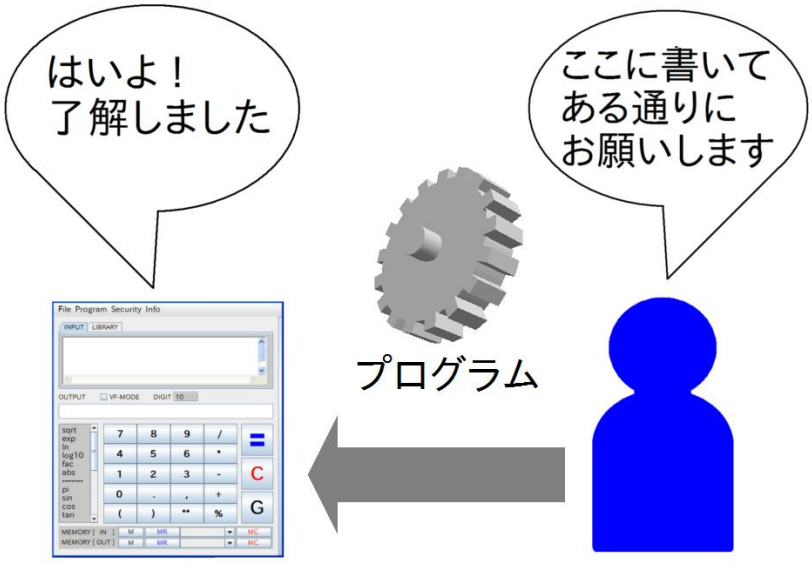
そこで一般に、機械やソフトウェアに「 する事のリスト 」を伝えたい場合は、**「 プログラミング言語 」と呼ばれる特別な言語で記述**します。プログラミング言語は、人間の話す言語（自然言語）に特有の曖昧さや複雑さを排除した、機械にとって都合の良い言語です※。

※ あくまで「自然言語に比べて」という意味であって、プログラミング言語の中でも機械寄りのもの（機械語、アセンブリ言語）から、人間寄りのもの（高級言語、スクリプト言語）まで、色々なものがあります。

先ほど例に挙げた、人間用の「 するの事リスト 」を、ちゃんとプログラミング言語で書き直すと下記のようになります。これが、一般に「 プログラム 」と呼ばれるものです。

▼ 人間が機械に処理を頼む時の、「 する事のリスト 」 = プログラム

```
int goukei = 0;
for( int i = 2; i <= 100; i++ ){
   bool sosuu = true;
   for( int j = 2; j <= i-1; j++ ){
      if( i % j == 0 ){
         sosuu = false;
      }
   }
   if( sosuu ){
      goukei += i;
   }
}
output( goukei );
```



このように、リニアンプロセッサーは、「 **する事のリスト** 」 = **プログラム** を読んで、複雑な計算を自動で行う事ができます。このように、処理内容をプログラムにまとめる事で、次のような利点があります。

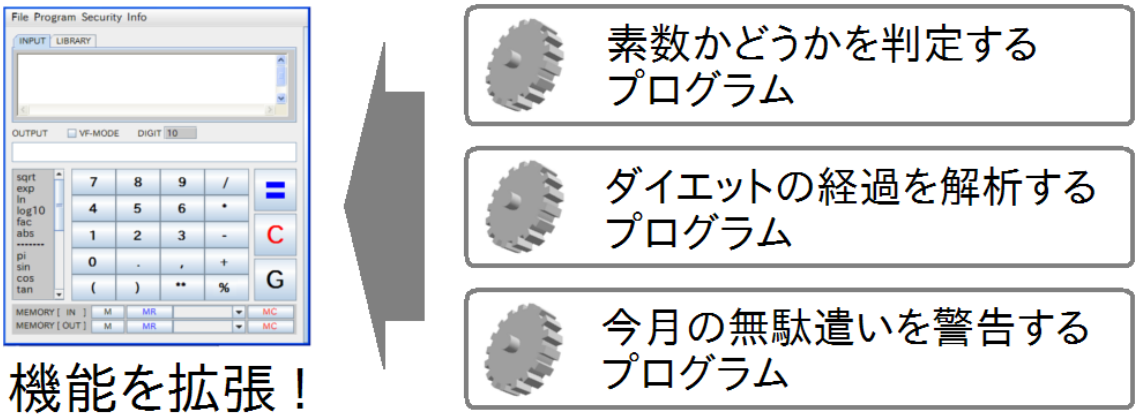
▼プログラムの利点

利点 1. 人間がボタンを押すのを待つ必要がないので、常に最大の処理速度を発揮できる
利点 2. いろいろな処理を複雑に組み合わせて、より高度な処理を実現できる
利点 3. 一度プログラムとして書いた処理は、何度でもすぐに使用できる(= 機能を拡張できる)

利点 1 に関してはこの通りで、最大 1 億回/秒に及ぶ処理速度を、フルに活用する事ができます。いちいち人間が数値を打ち込まなくても、ファイルに記述された膨大なデータを読み込んで、自動で次々と処理していく事などが可能です。

続いて利点 2 ですが、プログラムは文章として記述するという形態上、ボタンの直接打ち込みに比べて、はるかに長く複雑な処理内容を、正確に伝える事ができます。また、ボタンやウィンドウといった GUI 部品も使う事ができます。なので、頑張って数百行から数千行規模のプログラムを書けば、もはや一つのソフトウェアと呼べるような、高度なものでも作る事ができます。

最後に、最も大きな利点と呼べるのが、利点 3 です。リニアンプロセッサーでは、作ったプログラムを、「 Program（プログラム） 」メニューに追加してすぐに呼び出す事ができます。つまり、一度頑張ってプログラムに書いておいた処理は、それ以降は何度でもすぐに使えるようになるのです。これは、**リニアンプロセッサーの機能を自由自在に拡張**できる事に他なりません。



■ 実際にプログラムを使ってみる

ここまでは、プログラムの仕組みを説明してきました。ここからは、実際にプログラムを使用してみましょう。まず、メニューバーから**「 Program（プログラム） 」メニュー**をクリックしてください。すると、プログラムを選択する画面が表示されます。ここから**プログラムを選択すると、そのプログラムに記述された処理が実行**されます。

リニアンプロセッサーには、最初から便利なプログラムが付属しています。その一例としては、以下のようなものがあります。この標準プログラムは、今後も追加されていく予定です。

▼ 標準で付属しているプログラムの一例

・微積分

場所:「 Program（プログラム） 」メニュー ＞「 DiffAndIntegral_微積分 」フォルダ

数値微分と数値積分を求めるプログラム集です。グラフを描く事もできます。

・微分方程式

場所:「 Program（プログラム） 」メニュー ＞「 DiffEquation_微分方程式 」フォルダ

1～2 階/1～3 次元の常微分方程式を解析するプログラム集です。グラフを描く事もできます。

・雑用計算などの簡易アプリ集

場所:「 Program（プログラム） 」メニュー ＞「 Application_アプリ 」フォルダ

素数判定やカラーコード変換など、ちょっとした雑用計算を行うための簡易アプリ集です。

・いろいろな処理のサンプル

場所:「 Program（プログラム） 」メニュー ＞ 上に挙げた以外の各フォルダ

VCSSL プログラミングにおける基本的な処理を例示した、開発用サンプル集です。

■ 新しいプログラムは「 RinearnProcessorProgram 」フォルダへ追加

― たくさん集めて、自分の専門分野に特化したオリジナル電卓に ！

「 Program（プログラム） 」メニューをクリックした時に表示されるのは、**「 RinearnProcessorProgram 」フォルダ**の中身です。新しいプログラムを入手または自作したときは、ここに置いておきましょう。すると、「 Program（プログラム） 」メニューからすぐに使えて便利です。たくさん集めれば、リニアンプロセッサーを、自分の専門分野に特化したオリジナル電卓に育てることができます。

■ VCSSL コードアーカイブへアクセスして、新しいプログラムを入手する

VCSSL コードアーカイブは、VCSSL で記述されたプログラムを自由にダウンロードできるサービスです。VCSSL コードアーカイブのプログラムは、原則として無償公開である上に、著作権やライセンス上の制約が無いものがほとんどなので（ 大半がパブリックドメインで公開 ）、そのまま使用するのはもちろん、用途に合うように改造したり、自作プログラムの土台として流用したりする事もできます。

VCSSL コードアーカイブ:

https://ja.vcssl.org/code/

VCSSL コードアーカイブのプログラムは、基本的にリニアンプロセッサー上でも動作します。何か新しいプログラムが欲しいと思った時は、まずはアクセスして探してみてください。

プログラミング

ここからは、いよいよ独自のプログラムを開発する「 プログラミング 」を行って見ましょう。

■ プログラミング言語 VCSSL

リニアンプロセッサーで使用するプログラムは、
「 VCSSL (Visualization-Calculation-Simulation Script Language: 可視化・計算・シミュレーション用スクリプト言語) 」というプログラミング言語で記述します。

VCSSL はもともと、リニアンプロセッサー上で複雑な計算を行うための、専用言語として誕生したプログラミング言語です。しかしその後、単体のプログラミング言語として独立し、現在では**2D/3D コンピューターグラフィックス、GUI アプリケーション、テキスト処理、システムコマンド実行**など、数値計算以外にも使える便利な機能を獲得しています。下記公式サイトでは、無料の開発ガイドや、各種特集など、様々な VCSSL 関連情報が充実しています。

VCSSL

公式 WEB サイト　https://ja.vcssl.org/

VCSSL は C 言語系の文法を持つ言語ですが、一般の C 言語系言語が大規模開発をターゲットとしているのに対し、VCSSL は電卓上のちょっとした計算アプリや数値計算など、小規模なプログラムを即席で、簡単に作れる事を目指しているのが特徴です。また、習得の容易性も目指しています。

▼ VCSSL の主な特徴

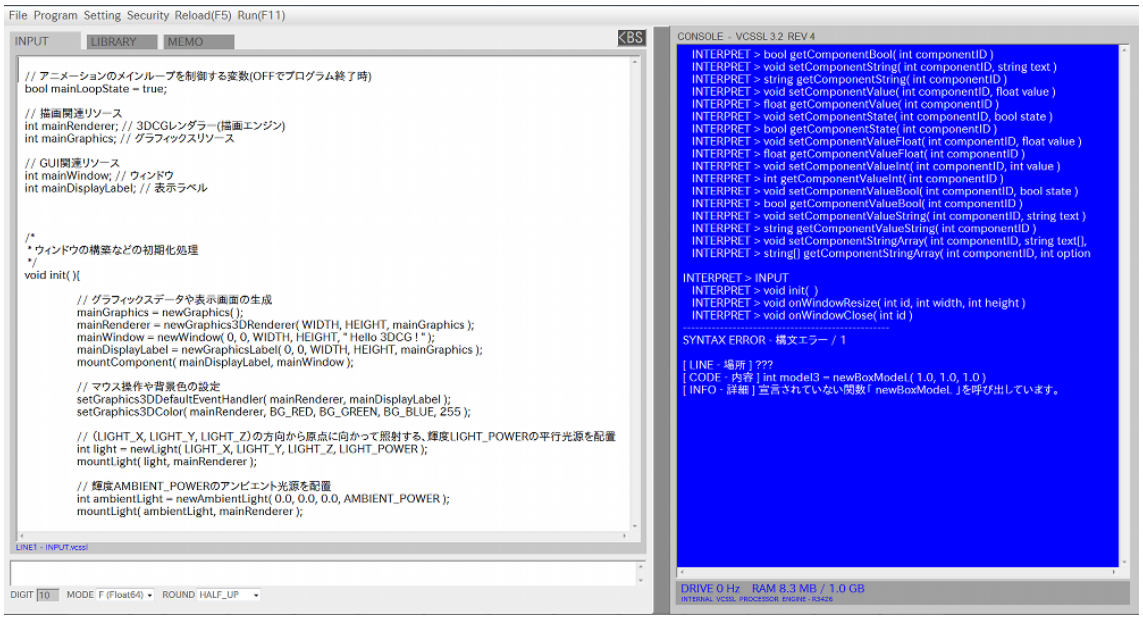
・C 言語をベースにシンプル化された文法（ 積極的な型変換、main 関数が必須でないなど ）
・非オブジェクト指向、変数型は静的で int/float/bool/string/varint/varfloat の 6 種類のみ
・配列の代入は、C 言語とは少し異なり、全要素値のコピーとなる
　（ 関数の引数へも要素値のコピー渡し、ただし引数宣言に&を付けて参照渡しも可能 ）
・簡単な GUI や 2D/3D グラフィックスを扱うライブラリが標準で揃っている
　（ 自由度や機能・性能といった点であまり高度では無いが、その代わりに習得が容易 ）

■ 画面を最大化する

それでは、いよいよリニアンプロセッサーでプログラムを開発していきましょう。

最初に、リニアンプロセッサーの**画面を最大化**してください。そして、**画面中央付近を横切る、灰色のバーを画面下部まで****スライド**させてください。

すると、画面内容がプログラミングに適したデザインに変化します。



このように、INPUT エリアがかなり大きくなりました。ここにプログラムを記述していきます。なお、**INPUT(入力)項目の文字サイズは、「Ctrl」キーを押しながら、マウスのホイール操作で大きくしたり小さくしたりできます。**また、キーボードの**「F11」キーでプログラムを実行し、「F12」キーで終了させる**事ができます。

ところで、リニアンプロセッサーは、INPUT エリアの記述内容が、数式か、それともプログラムなのか、一体どうやって判断しているのでしょうか？　その答えは、セミコロン記号「 ； 」の有無です。つまり、**INPUT エリアの記述内容がセミコロン記号「 ； 」を一個でも含む場合、それは数式ではなく、VCSSL によるプログラムと見なされます。**

なぜセミコロン記号のような特殊な記号で判別するのかと、不思議に思われるかもしれません。これは、**C 言語系のプログラミング言語では、行の区切りをセミコロン記号で行う**という規則があるためです。つまり、1 行以上のプログラムには必ずセミコロン記号が含まれるため、それでプログラムかどうかを判別する事ができるのです。

■ 値を OUTPUT（出力）項目に表示してみる

それでは、適当な値を OUTPUT（出力）項目に表示してみましょう。INPUT（入力）項目に以下のように記述し、「 ＝ 」ボタンを押してください（F11 キーでも OK）。プログラムが実行されます。

- INPUT（入力）-

```
output( 100.0 ) ;
```

最後のセミコロン記号「 ； 」は忘れがちなので、注意してください。必ず必要です。

実行した結果、OUTPUT（出力）項目に以下のように表示されます。

- OUTPUT（出力）-

```
100.0
```

なお、実行したプログラムを終了させるには、もう一度「 ＝ 」ボタンを押します。

さて、上の例では、output 関数を呼びました。この関数は、sin 関数や cos 関数のように値を返すものではなく、呼ぶ事自体に意味がある関数です。とにかく output 関数は、呼ばれると、引数の内容を OUTPUT（出力）項目に表示するという機能を持っているのです。

■ 値を様々な形で表示する

ところで、OUTPUT（出力）項目以外にも、値を表示する事ができます。以下のプログラムを実行してみてください。

- INPUT（入力）-

```
output( 100.0 ) ;
println( 200.0 ) ;
alert( 300.0 ) ;
```

このプログラムを実行すると、OUTPUT（出力）項目に「 100.0 」と表示され、CONSOLE（コンソール）項目に「 200.0 」と表示され、さらに画面中央にメッセージウィンドウが出現し、「 300.0 」と表示されます。

なお、上の例のように**複数行の処理を記述すると、上の行から下の行へ、順番に実行されます。**この性質は全ての基本なので、よく把握しておいてください。

■ 計算を行う

続いて、簡単な計算を行ってみましょう。以下のように記述し、実行してみてください。

- INPUT（入力）-

```
output( 1.0 / 2.0 ) ;
```

実行すると、OUTPUT（出力）項目に以下のように表示されます。

- OUTPUT（出力）-

```
0.5
```

このように、正しく計算できました。

ただし、ここで一つ、注意が必要な点があります。それは、「 **数値には小数点を付けておく** 」という点です。もし、小数点を省略したらどうなるのでしょうか？

実際にやってみましょう。

- INPUT（入力）-

```
output( 1 / 2 ) ;
```

実行すると、OUTPUT（出力）項目 に以下のように表示されます。

- OUTPUT -

```
0
```

このように、0.5 ではなく、0 と表示されてしまいました。これはなぜでしょうか？

実は、VCSSL をはじめとする多くのプログラミング言語では、**整数同士の除算（ 割り算 ）の結果は、整数になる**というルールがあります。そのため、0.5 の小数点以下が切り捨てられて、0 になってしまったのです。

そこで、特に除算の際などは、数値の末尾に小数点をつけて、整数ではなく小数にしておく事で、この問題を防ぐ事ができるというわけです。

■ 変数を使う

計算結果は、変数に格納する事ができます。例として、「 value 」という名前の変数に計算結果を格納し、表示してみましょう。

- INPUT（入力）-

```
float value ;                   // 変数「value」を用意
value = ( 1.0 / 2.0 ) * 3.0 ;   // value に計算結果を格納
output( value ) ;               // value の中身を表示
```

※**ダブルスラッシュ「 // 」より右は、コメントとして無視**されます。

変数を用意する行で、先頭に「 float 」というキーワードが存在します。これは、**小数を扱う変数を用意する事を意味するキーワード**です。他にも、整数を扱う事を意味する「 int 」などがあります。

実行すると、OUTPUT エリアに以下のように表示されます。

- OUTPUT（出力）-

```
1.5
```

このように、変数に計算結果を格納する事ができました。

■ ユーザーに値を入力してもらう

変数を使う事で、プログラム実行中に、ユーザーに値を入力してもらう事ができます。それには、以下のように **input 関数**を使用します。

- INPUT（入力）-

```
float value = input( "数値を入力してください" ) ;
output( value ) ;
```

実行すると、ユーザーが入力した値がそのまま出力されます。

なお、上の input 関数のように、文字列を引数とする関数も存在します。**プログラム中で文字列を記述する際は、必ずダブルクォーテーション記号「 " 」で挟まなければいけない**というルールがあります。慣れない間は忘れがちなので、ご注意ください。

■ 条件分岐 ― if 制御構文（ if 文 ）

変数に格納されている数値に応じて、処理を分岐させたい場合があるかもしれません。それには、if 制御構文（ if 文 ）を使用します。以下のように記述し、実行してみてください。

- INPUT（入力）-

```
float value = input( "数値を入力してください" ) ;

if ( 3.0 <= value ){
    output( "3.0 以上です" ) ;
}
else{
    output( "3.0 未満です" ) ;
}
```

このプログラムを実行すると、入力した値が 3.0 以上か、未満かを答えてくれます。

このように、if 制御構文では、( ) の中に記述した条件が成立した際に、その直後の { } の中身の処理が実行されます。反対に、条件が成立しなかった場合は、その後の else{ } の中身が実行されます。

なお、条件には以下のような種類があります。

- 条件の種類 -

```
a <= b    … a が b 以下か?
a < b     … a が b 未満か?
a == b    … a と b が等しいか?
a != b    … a と b が異なるか?
```

■ 条件繰り返し ― while 制御構文（ while 文 ）

続いて、条件が成立している間、処理を繰り返させてみましょう。それには、while 制御構文（ while 文 ）を使用します。以下のように記述し、実行してみてください。

- INPUT（入力）-

```
int counter = 0 ; //  整数を扱う変数を用意（カウンタ）

while( counter <= 10 ){
   println( "繰り返し中 … " + counter ) ;
   counter = counter + 1 ; //  カウンタを 1 加算
}

println( "繰り返しから脱出しました" );
```

実行すると、CONSOLE（コンソール）項目に以下のように表示されます。

- CONSOLE（コンソール）-

```
繰り返し中 … 0
繰り返し中 … 1
繰り返し中 … 2
繰り返し中 … 3
繰り返し中 … 4
繰り返し中 … 5
繰り返し中 … 6
繰り返し中 … 7
繰り返し中 … 8
繰り返し中 … 9
繰り返し中 … 10
繰り返しから脱出しました
```

このように、while 制御構文では、( ) の中に記述した条件が成立している間、ずっと { } の中身の処理が繰り返されます。

■ 回数繰り返し ― for 制御構文（ for 文 ）

上のように、一定の回数だけ処理を繰り返すような場合は、while 制御構文よりも、for 制御構文（ for 文 ）を使用したほうが、シンプルに記述できます。実際に使ってみましょう。

- INPUT（入力）-

```
for ( int counter = 0 ; counter <= 10 ; counter++ ){
   println( "繰り返し中 … " + counter ) ;
}

println( "繰り返しから脱出しました" );
```

実行すると、CONSOLE（コンソール）項目に以下のように表示されます。

- CONSOLE（コンソール）-

```
繰り返し中 … 0
繰り返し中 … 1
繰り返し中 … 2
繰り返し中 … 3
繰り返し中 … 4
繰り返し中 … 5
繰り返し中 … 6
繰り返し中 … 7
繰り返し中 … 8
繰り返し中 … 9
繰り返し中 … 10
繰り返しから脱出しました
```

このように、先ほどと全く同じ結果が得られました。for 制御構文の（ ）の中身は、以下のような意味を持っています。

for(カウンタ変数の用意 ; 繰り返し条件 ; カウンタ変数++)

このように、一定の回数繰り返すのに必要な、カウンタ変数の用意や、加算していくような処理を、（ ）の中にまとめて記述する事ができます。

■ 配列 ― いくつもの値を格納できる、特殊な変数

変数は通常、1 つの値しか格納できません。しかし、多くの値を格納できる、「 配列 」というものも存在します。配列は、

float 配列名[要素の個数];

などと宣言します。そして、例えば 3 番目の要素に値を格納したければ、

配列名[3] = … ;

などとします。

ここで注意が必要なのは、**使用できる要素の番号は、[0]番目から、[要素の個数-1] 番目まで**であるという事です。[要素の個数]番目は、容量オーバーで使用できません。

例として、5 個の値を格納できる配列を使用してみましょう。以下のように記述し、実行してみてください。

- INPUT（入力）-

```
float values[ 5 ] ;   // 0 番～4 番まで、5 個の値を格納できる配列を用意

// 値の入力
for( int i=0; i<5; i++ ){
      values[ i ] = i * i ;      //   i 番目の要素に、i の 2 乗を格納
}

// 中身の表示
for( int i=0; i<5; i++ ){
      println( values[ i ] ) ;      //   i 番目の要素の中身を表示
}
```

実行すると、CONSOLE（コンソール）項目に以下のように表示されます。

- CONSOLE（コンソール）-

```
0.0
1.0
4.0
9.0
16.0
```

このように、配列 values に、5 つの値を格納できた事が分かります。

■ 数値ファイル入出力の前に…

本章の最後として、数値ファイルの入出力を扱います。これまでと比べて少し難しいですが、便利なのでぜひマスターしましょう。

ファイル入出力を行うには、まず、実行するプログラムを適当な場所に保存しておく必要があります。これは、その場所と同じフォルダ内で入出力が行われるためです。メニューバーから

「 File 」メニュー ＞「 Save File ― 保存 」メニュー

を選択し、適当な場所にプログラムを保存してください。ENCODE は AUTO でかまいません。

■ タブ区切り数値ファイル（TSV）にデータを書き出す

まずは、ファイル出力を行いましょう。最初に open 関数で、タブ区切り数値ファイル（ TSV ）の書き込みモード ”wtsv” を指定してファイルを開き、そのファイルの ID を整数の変数で取得します。そして、writeln 関数で内容を書き出し、最後に close 関数でファイルを閉じます。

- INPUT（入力）-

```
// ファイル「test.txt」を TSV 書き込みモードで開き、ファイル ID を整数変数に格納
int fileID = open( "test.txt", "wtsv" ) ;

for( int i=0; i<5; i++ ){

    // ファイルに、i と i*i の値を空白タブ区切りで書き込み、改行する
    writeln( fileID, i, i*i ) ;
}

close( fileID ) ;  // ファイルを閉じる（忘れると書き込みが完了しないので注意!）
println( "書き込み完了" ) ;
```

実行し、「 書き込み完了 」と表示されたら、プログラムと同じフォルダに生成されている、「 test.txt 」というファイルを、テキストエディタで開いてみてください。

- test.txt -

```
0	0
1	1
2	4
3	9
4	16
```

このように、ちゃんと書き込めました。

■ タブ区切り数値ファイル（TSV）からからデータを読み込む

続いて、今度はこのファイルを読み込んでみましょう。まず open 関数で、タブ区切り数値ファイル（TSV）の読み込みモード”rtsv”を指定してファイルを開き、そのファイルの ID を整数の変数で取得します。続いて、countln 関数でファイル行数を取得します。そして、readln 関数で一行の内容を読み込んで配列に格納し、最後に close 関数でファイルを閉じます。

- INPUT（入力）-

```
// ファイル「test.txt」を TSV 読み込みモードで開き、ファイル ID を整数変数に格納
int fileID = open( "test.txt", "rtsv" ) ;
int lineN = countln( fileID ) ;　　// ファイル行数を取得
int values[ 2 ] ;　　// 1 行の内容を、空白区切りで取得する配列

for( int i=0; i<5; i++ ){

　　// ファイルから 1 行を読み込み、空白区切りで配列に格納
　　values = readln( fileID ) ;
　　println( values[ 0 ] , values[ 1 ] );
}

close( fileID ) ;　// ファイルを閉じる
println( "読み込み完了" ) ;
```

実行すると、CONSOLE（コンソール）項目に以下のように表示されます。

- CONSOLE（コンソール）-

```
0       0
1       1
2       4
3       9
4       16
読み込み完了
```

このように、ちゃんと読み込めました。

■ カンマ区切り数値ファイル（CSV）にデータを書き出す

続いて、カンマ区切り数値ファイル(CSV)の入出力を扱ってみましょう。

まずは出力です。最初に open 関数で、カンマ数値ファイル（TSV）の書き込みモード ”wcsv” を指定してファイルを開き、そのファイルの ID を整数の変数で取得します。そして、writeln 関数で内容を書き出し、最後に close 関数でファイルを閉じます。

- INPUT（入力）-

```
// ファイル「test.txt」を CSV 書き込みモードで開き、ファイル ID を整数変数に格納
int fileID = open( "test.txt", "wcsv" ) ;

for( int i=0; i<5; i++ ){

    // ファイルに、i と i*i の値をカンマ区切りで書き込み、改行する
    writeln( fileID, i, i*i ) ;
}

close( fileID ) ;   // ファイルを閉じる（忘れると書き込みが完了しないので注意！）
println( "書き込み完了" ) ;
```

実行し、「 書き込み完了 」と表示されたら、プログラムと同じフォルダに生成されている、「 test.txt 」というファイルを、テキストエディタで開いてみてください。

- test.txt -

```
0,0
1,1
2,4
3,9
4,16
```

このように、ちゃんと書き込めました。

■ カンマ区切り数値ファイル（ CSV ）からからデータを読み込む

続いて、今度はこのファイルを読み込んでみましょう。まず open 関数で、カンマ区切り数値ファイル（ CSV ）の読み込みモード ”rcsv” を指定してファイルを開き、そのファイルの ID を整数の変数で取得します。続いて、countln 関数でファイル行数を取得します。そして、readln 関数で一行の内容を読み込んで配列に格納し、最後に close 関数でファイルを閉じます。

- INPUT（入力）-

```
// ファイル「test.txt」を CSV 読み込みモードで開き、ファイル ID を整数変数に格納
int fileID = open( "test.txt", "rcsv" ) ;
int lineN = countln( fileID ) ;      // ファイル行数を取得
int values[ 2 ] ;                    // 1 行の内容を、空白区切りで取得する配列

for( int i=0; i<5; i++ ){
    // ファイルから 1 行を読み込み、空白区切りで配列に格納
    values = readln( fileID ) ;
    println( values[ 0 ] + "," + values[ 1 ] );
}
close( fileID ) ;  // ファイルを閉じる
println( "読み込み完了" ) ;
```

実行すると、CONSOLE（コンソール）項目に以下のように表示されます。

- CONSOLE（コンソール）-

```
0,0
1,1
2,4
3,9
4,16
読み込み完了
```

このように、ちゃんと読み込めました。

ライブラリの使用 – グラフソフトウェア制御

続いて、応用編としてライブラリを使用し、グラフソフトウェアを制御してみましょう。

■ グラフソフトウェアの制御

ここでは応用編として、ライブラリの使用を扱います。具体的な題材としては、数あるライブラリの中から、特にリニアンプロセッサーのユーザー層に適切と思われる、グラフソフトウェアの制御を扱ってみましょう。

ただし、本章ではこれまでのような、完全な形式での解説はしません。仕様書を読みながら、なるべく手探りで進めてみましょう。

■ ライブラリのインポート

これまでに用いてきた output 関数や println 関数とは異なり、グラフソフトウェアの制御用関数は、そのままでは使用できません。まずは「 **ライブラリのインポート** 」を行う必要があります。

ライブラリとは、特定の目的を持つ関数がまとめて記載された、特殊なプログラムの事です。通常は、使用するライブラリのファイルを入手し、プログラムと同じ場所か、リニアンプロセッサーのフォルダ内にある「 lib 」フォルダの中に配置する必要があります。

しかし、リニアンプロセッサーには、グラフソフトウェアを制御するためのライブラリが内蔵されており、別途入手する必要はありません。インポートするだけで使用できます。

VCSSL におけるインポートとは、プログラム内において、使用するライブラリを指定する事を意味します。ライブラリをインポートすると、その中に記載された関数が使用可能になります。

それでは、実際にグラフソフトウェア制御用のライブラリをインポートしてみましょう。以下のように記述してください。

- INPUT（入力）-

```
// グラフソフトウェア制御用ライブラリのインポート
import tool.Graph2D ;
import tool.Graph3D ;
import Math ;
```

import で始まる行が 3 つ続いていますが、上の行が 2 次元グラフソフトウェア制御用ライブラリ、その次の行が 3 次元グラフソフトウェア制御用のライブラリ、最後の行が数学関数を使用するためのライブラリです。ここでは、これら 3 つをインポートし、その中の関数を使えるようにしています。

さて、ライブラリをインポートしたのはいいですが、一体どういった関数が使えるのか、また、それらはどんな機能を持っているのかを知らなければ、何も書けませんね。このような時は、ライブラリの仕様書を確認します。ここでは、下記 URL に仕様書がありますので、ご確認ください。

■ tool.Graph2D ライブラリの仕様書

https://ja.vcssl.org/lib/tool/Graph2D

■ tool.Graph3D ライブラリの仕様書

https://ja.vcssl.org/lib/tool/Graph3D

仕様書には、利用できる全関数のリストや、それらに関する詳細説明などが記載されています。

■ グラフソフトウェアの起動

まずすべき事は、恐らくグラフソフトウェアの起動だろうと予想が付くかと思います。それでは、一体どうすれば起動できるのでしょうか？ ここで仕様書を見てみましょう。まず、2 次元グラフソフトウェア用の仕様書を見ると、関数リストの先頭に、どうもそれらしい関数の説明が見つかります。

関数仕様	int newGraph2D( )
処理内容	2 次元グラフソフトウェアをデフォルトのサイズで起動し、それに固有の識別番号（ グラフ ID ）を割り振って返します。

関数仕様の「 **int newGraph2D( )** 」というのは、「 **newGraph2D( ) という関数を呼ぶと、処理結果を int（整数）として返します** 」という意味です。それでは、返される整数の値はどういった意味でしょうか?

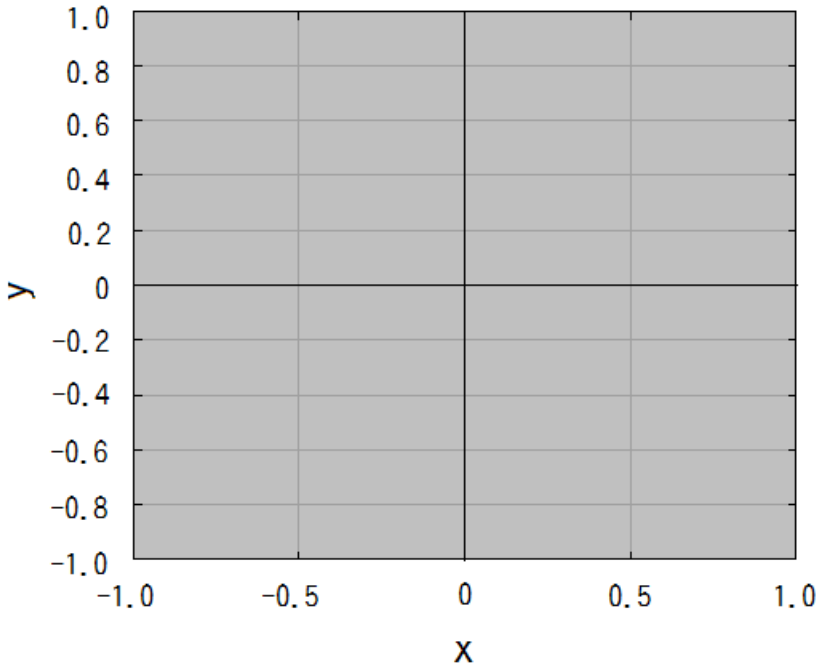
処理内容の説明を読むと、どうもそれは、起動したグラフソフトに割り振られる番号のようです。その番号をどう使うのかはまだ分かりませんが、とりあえず後で使いそうなので、変数に格納しておくのが良さそうです。

それでは、この関数を呼び出してみましょう。

- INPUT（入力）-

```
// グラフソフトウェア制御用ライブラリのインポート
import tool.Graph2D ;
import tool.Graph3D ;
import Math ;

// 2 次元グラフソフトウェアを起動してみる
int graph2D = newGraph2D( ) ;
```

実行してみましょう。すると、無事 2 次元グラフソフトウェアが起動します。



■ ファイルをプロットさせてみる

グラフソフトウェアの起動は無事できましたが、次にどうすればグラフをプロットさせる事ができるのでしょうか？再び、仕様書に目を通してみましょう。

すると、グラフをプロットするための関数がいくつか見つかります。ファイルをプロットさせたり、文字列や配列をプロットさせたりと、色々なプロット方法が用意されているようです。とりあえずは、ファイルを読み込んでプロットする以下の関数を試してみましょう。

関数仕様	**void setGraph2DFile (int graphID, string filePath)**
処理内容	引数 filePath に指定されたファイルの内容を、グラフにプロットします。

関数仕様の先頭にある「 **void** 」というキーワードは、「 この関数は何も値を返しません 」という事を意味します。そして、関数名の後の**(int graphID, string filePath)**の箇所は、**「 この関数を呼び出す際に、int（整数）型と string（文字列）型の 2 つの値を受け取ります 」**という意味です。処理内容を見ると、この string（文字列）型の値には、グラフ用のデータファイルのファイル名（または相対パス）を格納して渡してやればよさそうです。

なお、データファイルの書き方については、リニアンプロセサーに同梱されているグラフソフトは「リニアングラフ 2D/3D」なので、それらのガイドをご参照ください：

・リニアングラフ 2D の場合 / https://www.rinearn.com/graph2d/guide/
・リニアングラフ 3D の場合 / https://www.rinearn.com/graph3d/guide/

基本的には、各行ごとに 1 点ずつ座標を書いていき、2D の場合は「 x 値　y 値 」のように 2 列で、3D の場合は「 x 値　y 値　z 値 」のように 3 列で書けば OK です。

■ プロット

それでは、実際にデータを用意し、プロットさせてみましょう。具体的な手順としては、まず先の章で扱ったファイル入出力機能を使い、データを数値ファイルに書き出して、それを上の setGraph2DFile 関数でプロットさせてみます。

少し長いですが、以下のように記述し、実行してみてください。

- INPUT（入力）-

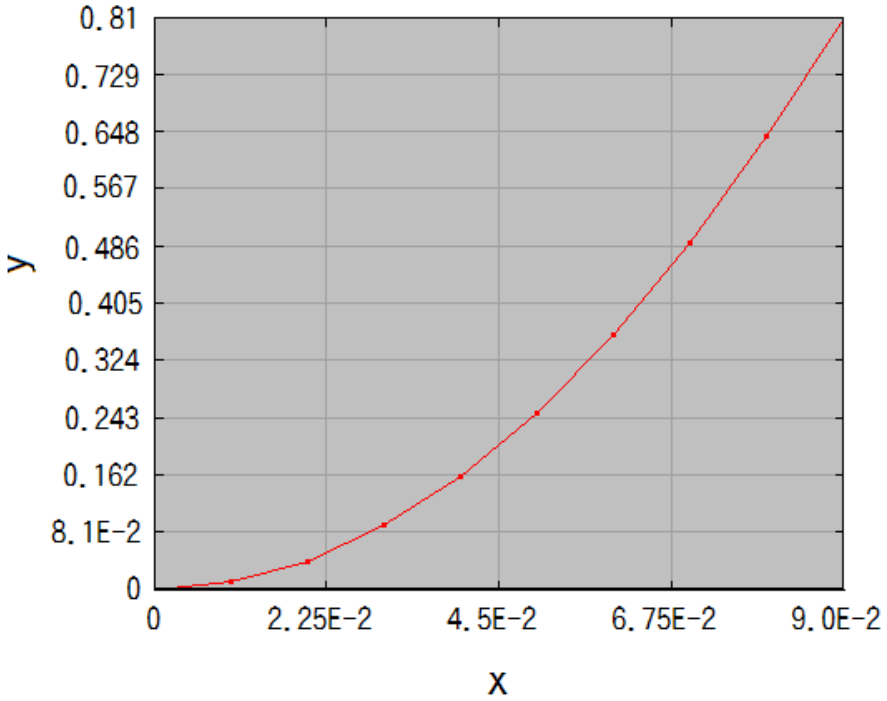
```
// グラフソフトウェア制御用ライブラリのインポート
import tool.Graph2D ;
import tool.Graph3D ;
import Math ;

// 2 次元グラフソフトウェアを起動してみる
int graph2D = newGraph2D( ) ;
// ファイルにデータを書き出す
int fileID = open( "data2d.tsv", "wtsv" ); // wtsv は TSV 形式の書き込みモード
float dx = 0.1 ;
float x ;
float y ;
for( int i=0; i<10; i++ ){
    x = i * dx ;
    y = x * x ;
    writeln( fileID, x, y ) ; // ファイル書き込み
}
close( fileID ); // ファイルを閉じる（忘れると書き込みが完了しないので注意!）

// CONSOLE エリアの内容を string（文字列）変数に格納
string data = load( ) ;

// グラフにプロット
setGraph2DFile( graph2D, "data2d.tsv" ) ;
```

実行すると、ファイル「 data.tsv 」に以下のように出力されます。

- data2d.tsv -

```
0.0     0.0
0.01    1.0E-4
0.02    4.0E-4
0.03    9.0E-4
0.04    0.0016
0.05    0.0025
0.06    0.0036
0.07    0.0049
0.08    0.0064
0.09    0.0081
```

そして、グラフは以下のようにプロットされます。



無事プロットできたようです。

■ 3 次元グラフソフトウェアでもやってみる

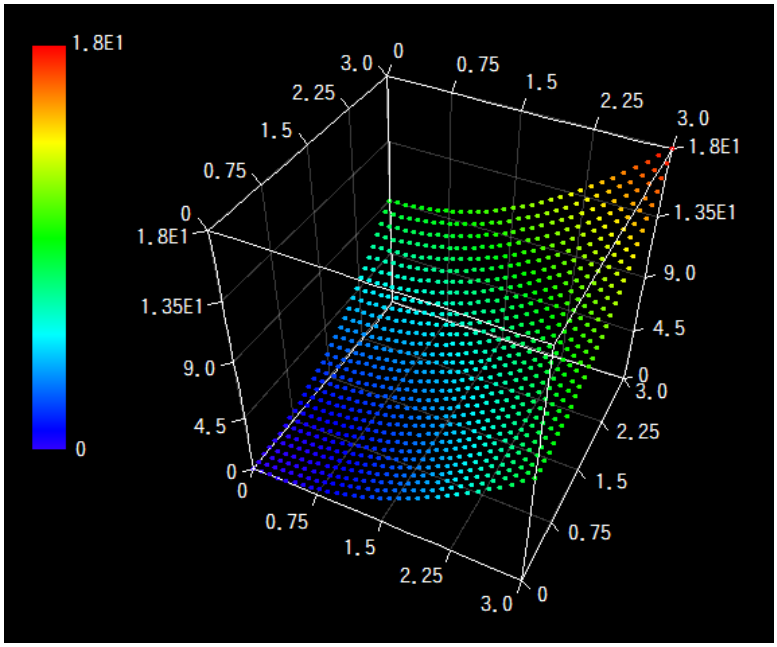
続いて、3 次元グラフのプロットも行ってみましょう。以下のように記述し、実行してみてください。

- INPUT（入力）-

```
// グラフソフトウェア制御用ライブラリのインポート
import tool.Graph2D ;
import tool.Graph3D ;
import Math ;

// 3 次元グラフソフトウェアを起動してみる
int graph3D = newGraph3D( ) ;

// ファイルにデータを書き出す
int fileID = open( "data3d.tsv", "wtsv" ); // wtsv は TSV 形式の書き込みモード
float dx = 0.1 ;
float dy = 0.1 ;
float x, y, z ;
for( int i=0; i<=30; i++ ){
    for( int j=0; j<=30; j++ ){
        x = i * dx ;
        y = j * dy ;
        z = x * x + y * y ;
        writeln( fileID, x, y, z ) ; // ファイル書き込み
    }
    writeln( fileID, "" ) ;
}
close( fileID ); // ファイルを閉じる（忘れると書き込みが完了しないので注意!）

// CONSOLE エリアの内容を string（文字列）変数に格納
string data = load( ) ;

// グラフにプロット
setGraph3DFile ( graph3D, "data3d.tsv" ) ;
```

このように、3 次元グラフソフトウェアの制御は、基本的には 2 次元グラフソフトウェアの制御と共通で、似た感覚で使用できます。

実行すると、グラフは以下のようにプロットされます。



■ その他のライブラリ

ここで扱ったライブラリの他にも、下記 URL から、リニアンプロセッサーにあらかじめ内蔵されているライブラリの仕様書を参照する事ができます。実に様々な機能がありますので、気になったライブラリは、今回と同じ要領で、ぜひ積極的に試してみてください。以下のページで、使用できる全てのライブラリの仕様書がご覧になれます。

https://ja.vcssl.org/lib/

■ より高度な制御へ

ここまで、プログラミングの土台から、ライブラリの扱い方まで、駆け足で解説してきました。ここで紹介したようなライブラリを活用しつつ、自作の処理を組み合わせる事で、自分の望む処理を行うプログラムを開発する事ができるようになります。そして、そのプログラムは、リニアンプロセッサーのメニューからいつでも使う事ができます。

これはつまり、リニアンプロセッサーにいくらでも、自分の望む機能を追加していけるという事です。VCSSL は、電卓付属のプログラミング言語としてはなかなか高機能なので、その気になればかなりのものを作れるはずです。その一つの例として、**リニアンプロセッサーのグラフ機能**が挙げられます。実は、あれらは**全て VCSSL で開発**されています。実際、

「 File 」メニュー ＞「 OpenFile ― ファイルを開く」メニュー

を選択し、そして「 **graph** 」フォルダ内にある「 **z(x, y, t)** 」というファイルを開いてみてください。リニアンプロセッサーのグラフ機能の「 中身 」を見ることができます。そしてそれは、紛れも無い VCSSL プログラムである事が確認できるはずです。その中身は、必要が無ければ、読めなくても全く問題ありません。しかし、必要ならば、このような機能を独自に作る事ができるという事は、ぜひ覚えておいてください。そして、必要になった際には、ぜひ機能の自作に挑戦してみてください！

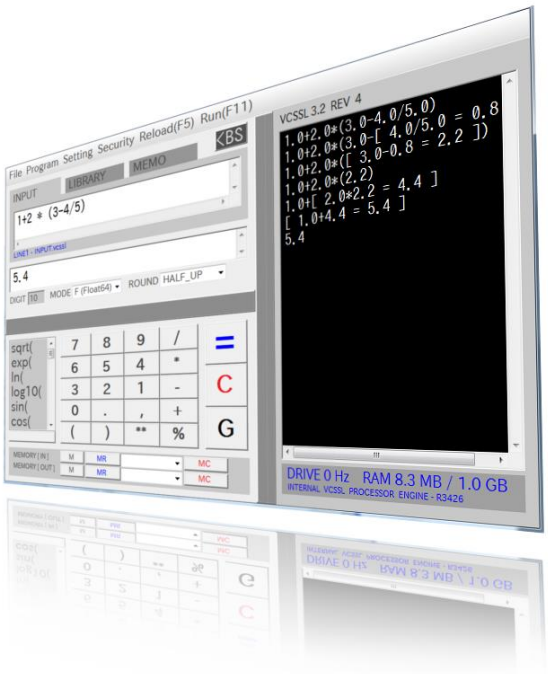
※この文書内では、VCSSL の機能の中から、電卓ソフトにおいて直接役立ちそうな、ごく一部のものだけを抜き出してまとめました。より幅広い、汎用的な機能も含めた詳細な解説は、同梱の「 **VCSSL リファレンスガイド** 」をご参照下さい。

商標などに関して

[１] Oracle と Java は、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。

[２] Windows は米国 Microsoft Corporation の米国およびその他の国における登録商標です。

[３] その他、文中に使用されている商標は、その商標を保持する各社の各国における商標または登録商標です。



RINEARN Processor 4.3　取扱説明書　第2版

著者　　松井文宏

この書籍に記載されている内容は、以下の Web サイトでも閲覧することができます。

https://www.rinearn.com/processor/