

# GAL-Solver User Guide

Filippo De Grandi  
Simone Vigasio



## Contents

<b>1</b>	<b>General</b>	<b>3</b>
<b>2</b>	<b>Commands</b>	<b>3</b>
<b>3</b>	<b>Commands list</b>	<b>5</b>
3.1	new . . . . .	5
3.2	calculate: . . . . .	10
3.2.1	matrix: . . . . .	10
3.2.2	vector: . . . . .	14
3.2.3	system: . . . . .	18
3.2.4	set: . . . . .	18
3.2.5	function: . . . . .	21
3.3	print: . . . . .	23
3.4	remove: . . . . .	25
3.5	view: . . . . .	26
3.6	cls: . . . . .	26
3.7	/help: . . . . .	26
3.8	END: . . . . .	26

## 1 General


The whole program functioning is centred around what the user-input is, recognizing high-level language commands and executing the corresponding function. The program has spelling mistake recognition, telling the user whenever something does not meet the required inputs.

Every element that is created is saved inside simply concatenated lists and the key-parameter that differentiates each one from the other is the name.

Two entities cannot have the same name, otherwise it would be impossible to find which entity saved in memory the user wants to use.

Two entities from different element-groups (i.e. matrices and vectors) can have the same name, as the research for a particular element is executed inside the element-list and not all the lists at once.

## 2 Commands



```
Function >> [base-command] [element type] [function]|
```

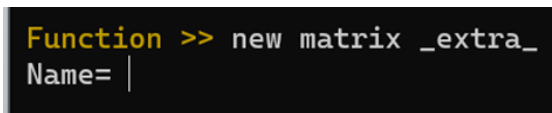
Figure 1: Template for every command

Each key-word has to be separated with a space (more than one does not make a difference).

Writing more than the required amount of input-words does not generate an error as the program recognizes the first useful commands and executes them, ignoring the extra ones.

Some commands do not need the function key-word.

Example:



```
Function >> new matrix _extra_  
Name= |
```

Figure 2: Only “new matrix” gets recognized and the extra part is ignored. More about commands syntax in the next paragraph.

After each function is executed, more input will be required from the user but the program uses high-level language words (as above, name refers to the new matrix name) to tell the user what they have to input in each field.

In some functions, if a spelling mistake is made the user will lose some of the progress done in that function scope, so a good advice would be to pay attention to the spelling of each command.

When required to input numeric values inside the entities (i.e. matrices, vectors, systems) the user can choose between:

- **Fractions:** values written as  $1/2$ ,  $3/7$ , and anything that is separated by a / (slash). The fractions will also be simplified as much as possible (i.e.  $4/8$  will be turned into  $1/2$ );
- **Float:** values with decimal value, written as 12.34, 1.03. These values will be turned into fractions, but the user can decide whether to see the values of every entity as Fraction or as Float values;
- **Integer:** normal integer values.

Obviously negative values are accepted.

### 3 Commands list

Below is the complete, detailed list of possible commands. The user can also see a brief list of the commands in the program if the command **/help** is executed. The following is the right way to write commands, lowercase and separated by spaces. The order is from left to right.

#### 3.1 new

- **matrix:** generates a new matrix based on the user input.  
Required input:  
**Name:** matrix name (has to be different from every matrix previously created);  
**Rows:** n° of rows in the matrix;  
**Columns:** n° of columns in the matrix; The user can now insert each value in the right position of the matrix (as seen below):

```
Function >> new matrix
Name= A
rows= 3
columns= 3

1 -1/2 0
2/4 9 -5
1.2 3.5 1/2
```

Figure 3: new matrix

- **vector:** generates a new vector based on user input.  
Required input:  
**Name:** vector name (has to be different from each vector previously created);  
**Dimension:** vector dimension (n° of values). The values need to be separated by spaces (more than one does not affect the program, nor does pressing enter to create a new line):

```
Function >> new vector
Name= b
dimension= 4

1 -1 1/2 0
```

Figure 4: new vector

- **system:** generates a new linear equations system based on user input.  
Required input:  
**Name:** system name (has to be different from each system previously created);  
**Coefficients matrix name:** this function finds the matrix with the given name and uses that as the coefficients matrix for the system being created. If the matrix with the given name is not found, a new one is created and is saved in memory. The required parameters are the same as the new matrix function;  
**Vector name:** refers to the known terms vector, has the same functioning as the coefficients matrix name, if the name is not found in the vector list a new one is created and saved in memory.  
If the selected vector already exists in memory, but is incompatible with the matrix (the vector dimension is bigger than the n° of rows of the matrix) the function will end, with the given error message.

The elements used in the example below are the ones created in the previous commands. Once the system is created, it will be printed like this:

```
Function >> new system
Name= S
Coefficients matrix name: A
Vector name: b
The selected vector is incompatible with the selected
matrix

Function >> new system
Name= S
Coefficients matrix name: A
Vector name: c

1 0 0

Name: S

1 -1/2 0 | 1
1/2 0 -5 | 0
6/5 7/2 1/2 | 0
```

Figure 5: new system

- **set:** creates a new set of vectors based on the user input.  
Required input:  
**Name:** set name (has to be different from each set previously created).  
If the user wants to create a Canonical base, they can give the name **Cn**, with **n** the dimension of the canonical base, to a new set and it will automatically save the set as the canonical base of  $\mathbf{R}^n$ ; **Vector** size: n° of elements of the vectors belonging to the set;  
**Dimension:** n° of vectors in the set;  
Then the user will be able to write each value in the corresponding vector (if more than the vector size values are inserted, only the required ones will be registered).

```
Function >> new set
Set name: Q
Vector size= 3
Dimension= 2

V1:  1 0 -2
V2:  0 1 0
```

Figure 6: new set

- **function:** creates a new linear function based on user input.  
Required input:  
**Function in expression form or matrix:** here the user is asked whether he already knows the representative matrix of the linear function or needs to find it based on the expression form.  
A typical expression form is the following:

$$f(x_1, x_2, x_3, x_4) = (3x_1 - 2x_3 + x_4, 4x_1 - 2x_2 + 2x_3 + 3x_4, x_1 + 2x_3 + 2x_4)$$

Figure 7: linear function in expression form

If the user chooses expression form (by writing **expression**), he will be required the following parameters:

**Name:** function name (has to be different from each function previously created);

**Base “from”:** refers to the Domain of the linear function. The user has to write the name of the base. If the base already exists (the base is saved in the set list) then that base will be used.

**Base “to”:** refers to the Range of the linear function. The user has to write the name of the base. If the base already exists (the base is saved in the set list) then that base will be used.

If the user writes the name of a new base, a new set of vectors will be created, following the procedure of the new set command.

If the user wants to use the canonical base of  $\mathbf{R}^n$ , all he is required to do is write the name **Cn**, where **n** is the dimension of the Domain.

**Coordinates:** refers to the variables of the expression form.

The coordinates are taken in input with the given coefficient and the variable name, that has to start from the letter "a" (in the Figure 7 it would be the equivalent of  $x_1$ ).

If a variable "in the middle" (i.e.  $x_2$  in the example above, in the first coordinate) is not present in the given coordinate, the user will be required to write 0 as the coefficient.

If the first variable of the coordinate is not  $x_1$ , then the user can start directly from the one in the expression.

The example (which refers to the Figure 7) shows the possible combinations.

The name given to the representative matrix has the following syntax:

$M + (\text{name of domain base}) + (\text{name of the range base}) + (\text{name of function between parenthesis})$ .

```
Function >> new function
Function in expression form or matrix?: expression
Function name: T
Base "from": C4
Name: C4

{ (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1) }

Base "to": C3
Name: C3

{ (1, 0, 0), (0, 1, 0), (0, 0, 1) }

Coord.1: 3a +0b -2c +d
Coord.2: 4a -2b +2c +3d
Coord.3: a +0b +2c +2d

Name: T

Base "from": Name: C4

{ (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1) }

Base "to": Name: C3

{ (1, 0, 0), (0, 1, 0), (0, 0, 1) }

Name: MC4C3(T)

3 0 -2 1
4 -2 2 3
1 0 2 2
```

Figure 8: new function - expression form



If the user chooses matrix form (by writing **matrix**), he will be required the following parameters:

**Name:** function name (has to be different from each function previously created);

**Base “from”:** refers to the Domain of the linear function. The user has to write the name of the base. If the base already exists (the base is saved in the set list) then that base will be used.

If the user writes the name of a new base, a new set of vectors will be created, following the procedure of the new set command.

**Base “to”:** refers to the Range of the linear function. The user has to write the name of the base. If the base already exists (the base is saved in the set list) then that base will be used.

If the user writes the name of a new base, a new set of vectors will be created, following the procedure of the new set command.

**Representative matrix name:** name of the representative matrix.

If a matrix with the same name already exists it will be taken from memory. Otherwise, a new one will be created and saved in memory just like the new matrix command.

```
Function >> new function
Function in expression form or matrix?: matrix
Function name: T
Base "from": C3
Name: C3

{ (1, 0, 0), (0, 1, 0), (0, 0, 1) }

Base "to": C2
Name: C2

{ (1, 0), (0, 1) }

Representative matrix name: A
Rows= 2
Columns= 3

1 2 0
0 -1 0
```

Figure 9: new function matrix

## 3.2 calculate:

### 3.2.1 matrix:

- **product:** calculates the product between two matrices.  
**Matrix names:** insert the two names separated by a space (or a newline). The matrices need to have already been created with the new matrix command. The product is calculated only if the matrices are compatible. The new product matrix created is saved into memory with the following name syntax:  
 $(1^{st} \text{ matrix name}) + (x) + (2^{nd} \text{ matrix name});$

```
Function >> calculate matrix product
Matrix names: A B
Name: AxB

  2  -1/2
-2   1/2
```

Figure 10: matrix product

- **sum:** calculates the sum between two matrices.  
**Matrix names:** insert the two names separated by a space (or a newline). The matrices need to have already been created with the new matrix command. The sum is calculated only if the matrices are compatible. The new sum matrix created is saved into memory with the following name syntax:  
 $(1^{st} \text{ matrix name}) + (+) + (2^{nd} \text{ matrix name});$

```
Function >> calculate matrix sum
Matrix names: A B
Name: A+B

  3  -1/2
-1   3
```

Figure 11: matrix sum

- **difference:** calculates the difference between two matrices.  
**Matrix names:** insert the two names separated by a space (or a newline).  
 The matrices need to have already been created with the new matrix command. The difference is calculated only if the matrices are compatible.  
 The new sum matrix created is saved into memory with the name:  
*(1st matrix name) + (-) + (2nd matrix name);*

```
Function >> calculate matrix difference
Matrix names: A B
Name: A-B

-1  1/2
-1  -3
```

Figure 12: matrix difference

- **transpose:** calculates the transpose matrix from a already-created matrix.  
 The new transpose matrix calculated is saved into memory and its name has the following syntax: *(matrix name) + (T);*

```
Function >> calculate matrix transpose
Matrix name: A
Name: AT

1 -1
0 0
```

Figure 13: matrix transpose

- **scalar:** calculates the product between a already created matrix and a scalar.

**Matrix name:** insert the name of the matrix

**lambda:** insert the value of the scalar

The new scalar product matrix calculated is saved into memory with this name syntax: *(scalar in fraction form) + (matrix name)*.

```
Function >> calculate matrix scalar
Matrix name: A
Lambda= 3
Name: 3A

  3  0
-3  0
```

Figure 14: matrix scalar product

- **stairs:** calculates the stairs form of an already saved matrix.  
The new matrix created is saved into memory with this name syntax: *(matrix name) + (sf)*;

```
Function >> calculate matrix stairs
Matrix name: B
Name: Bsf

  1 1/2  0
  0  0 -1/2
  0  0  0
```

Figure 15: matrix stairs form

- **rref:** calculates the stairs form of an already saved matrix.  
The new matrix created is saved into memory with this name syntax: *(matrix name) + (rf)*;

```
Function >> calculate matrix rref
Matrix name: B
Name: Brf

  1 1/2  0
  0  0  1
  0  0  0
```

Figure 16: matrix rref form

- **rank:** calculates the rank of a already saved matrix and returns its value.

```
Function >> calculate matrix rank  
Matrix name: B  
Rank= 2
```

Figure 17: matrix rank

- **base:** tells the user if the matrix is a base of  $R_n$ , where  $n$  is the number pivot in the matrix.

```
Function >> calculate matrix base  
Matrix name: B  
The matrix is not a base
```

Figure 18: matrix base

- **det:** calculates the determinant of an already saved matrix (only if the matrix is a square-matrix).  
The value is printed in fraction form, but also gets approximated into a float with two decimal positions.

```
Function >> calculate matrix det  
Matrix name: C  
Det= 547/42 ~ 13.02
```

Figure 19: matrix determinant

- **reverse:** calculates the reverse matrix from an already saved one.  
The new matrix created. The reverse matrix name has the following syntax: (matrix name) + (-1);

```
Function >> calculate matrix reverse
Matrix name: C
Name: C-1

111/223 224/223 126/223
-56/223 112/223 63/223
-28/223 56/223 -80/223
```

Figure 20: reverse matrix

### 3.2.2 vector:

- **scalar:** calculates the scalar product between two already saved vectors;  
The result is given in fraction form, but also approximated in float form with two decimal digits.

```
Function >> calculate vector scalar
1st vector name: a
2nd vector name: b
Scalar product: -7/4 ~ -1.75
```

Figure 21: vector scalar product

- **sum:** calculates the sum of two already saved vectors, only if the vectors are compatible.  
The new vector calculated is saved in memory with the following name syntax:  
(1<sup>st</sup> vector name) + (+) + (2<sup>nd</sup> vector name);

```
Function >> calculate vector sum
1st vector name: a
2nd vector name: b
Name: a+b = (0, -1/2, 1/4)
```

Figure 22: vector sum

- **difference:** calculates the difference of two already saved vectors, only if the vectors are compatible.

The new vector calculated is saved in memory with the following name syntax:

$(1^{st} \text{ vector name}) + (-) + (2^{nd} \text{ vector name});$

```
Function >> calculate vector difference
1st vector name: a
2nd vector name: b
Name: a-b = (2, 1/2, 7/4)
```

Figure 23: vector difference

- **norm:** calculates the norm of an already saved vector.  
The result is printed as a root value, with the following syntax:  
 $\text{sqrt}(\text{value});$   
A two-decimal-digits approximation is also printed.

```
Function >> calculate vector norm
Vector name: a
a norm = sqrt( 2 ) ~ 1.41
```

Figure 24: vector norm

- **angle:** calculates the angle between two already saved vectors, only if they are compatible;

```
Function >> calculate vector angle
1st vector name: a
2nd vector name: b
Angle= 2.7367
```

Figure 25: vector angle

- **CROSS:** calculates the cross-product between two already saved vector, only if they are compatible and only if the cross-product is possible; The new cross-product vector is saved in memory, with the following name syntax:  
 $(1^{st} \text{ vector name}) + (x) + (2^{nd} \text{ vector name});$

```
Function >> calculate vector cross
1st vector name: a
2nd vector name: b
Name: axb =      (-1/3, -1/3, 0)
```

Figure 26: vector cross product

- **cim:** calculates the counter image of a vector from a selected function. Required input:  
**Function name:** name of an existing function (created by the new function command)  
**Vector name:** name of a vector. If the vector is already in the vector list and its compatible with the requirements, it will be used to calculate the counter image, otherwise a new vector will be created.

```
Function >> calculate vector cim
Function name: T
Vector name: b
dimension= 2

    1 1

Name: Tcim

    { (-3, 1) }
```

Figure 27: vector counter image



- **apply:** calculates the application of a function to a vector.  
**Function name:** name of an already existing function;  
**Vector name:** name of a vector. If a vector with the same name is already saved, it will be used to calculate the function application, otherwise a new one will be created and saved.  
The resulting vector from the function application is saved in memory with the following name syntax:  
 $(T) + (\text{vector name in parenthesis})$ ;

```
Function >> calculate vector apply
Function name: T
Vector name: b
Vector dimension: 3

1 2 -1

Name: T(b) = (-2, 3)
```

Figure 28: Function applied to a vector

### 3.2.3 system:

- **solution:** calculates the system solutions.

Required input:

**Name:** name of an already saved system

The solution-system is saved in memory with the following name syntax:

*(system name) + (S)*;

```
Name: S

1/2  0  -2 | 0
-3  1/3  0 | -2
1    1  -1 | 1

Function >> calculate system solution
System name: S
Name: SS

1 0 0 | 28/39
0 1 0 | 6/13
0 0 1 | 7/39
```

Figure 29: system solution

### 3.2.4 set:

- **base:** tells the user if the given set is a base of  $\mathbb{R}^n$ , where  $n$  is the dimension of each vector.

```
Function >> new set
Set name: C3
Name: C3

{ (1, 0, 0), (0, 1, 0), (0, 0, 1) }

Function >> calculate set base
Set name: C3
The set is a base of R3
```

Figure 30: set is base

- **li:** tells the user if the given set is a set of linearly independent vectors.

```
Function >> calculate set li
Set name: C3
The set is l.independent
```

Figure 31: set of linearly independent vectors

- **gen:** tells the user if the set is a set of generator vectors

```
Function >> calculate set gen
Set name: C3
The set is generator of R3
```

Figure 32: set of generators vectors

- **gs:** calculates an orthonormal base from a given set of vectors (only if the set is a base).  
The orthonormal-base calculated is printed and saved with **sqrt(value)** scalar values that multiply each vector of the base.  
The new orthonormal-base set is saved in memory with the following name syntax:  
 $(set\ name) + (gs)$

```
Function >> calculate set gs
Set name: S
Name: Sgs
{ 1/sqrt(14)(1, -2, 3), sqrt(7)/sqrt(12)(8/7, -2/7, -4/7), sqrt(6)/sqrt(25)(5/6, 5/3, 5/6) }
```

Figure 33: Gram Schmidt - Orthonormal base

- **ort**: calculates the Orthogonal-Complement of a set.  
The orthogonal complement calculated is printed **sqrt(value)** scalar values that multiply each vector of the new set.  
If the starting set is already a base, then the orthogonal complement will be composed of the null vector.  
The new Orthogonal-Complement set is saved in memory with the following name syntax:  
*(set name) + (ort)*;

```
Function >> new set
Set name: R
Vector size= 3
Dimension= 2

V1:  1 0 -2
V2:  1/2 1 0

Function >> calculate set ort
Set name: R
Name: Rort

{ 1/sqrt(6)(2, -1, 1) }
```

Figure 34: Orthogonal complement of a set

- **complete**: calculates the completion of the set to a base of  $\mathbf{R}^n$ . The new calculated set is saved in memory with the following name syntax:  
*(set name) + (\_C)*;

```
Function >> calculate set complete
Set name: R
Name: R_C

{ (1, 0, -2), (1/2, 1, 0), (1, 0, 0) }
```

Figure 35: set completion

### 3.2.5 function:

- **rm:** calculates the representative matrix of a function.  
The user can choose which bases the representative matrix involves and can both use already existing bases or create new ones.  
The new matrix created is saved in the matrix list with two different name syntax:  
If the **base from** and **base to** are the same, then the name of the representative matrix will be:  
 $(M) + (base\ name) + (function\ name\ in\ parenthesis)$   
If the **base from** and **base to** are different, then the name of the representative matrix will be:  
 $(M) + (1^{st}\ base\ name) + (2^{nd}\ base\ name) + (function\ name\ in\ parenthesis)$

```
Function >> calculate function rm
Function name: T
Base "from" name >> B
Vector dimension: 2
V1:  1 -1
V2:  1/2 0

Base "to" name >> B

Name: MB(T)

      4/3  1/2
     -17/3 -1/2
```

Figure 36: function representative matrix

- **ker:** calculates the kernel of the selected function (has to be already been saved in memory).

The kernel is a base of vectors that have the **Null** vector as their image.

The name of the kernel-set created has the following syntax:

*(ker) + (function name in parenthesis);*

```
Function >> calculate function ker
Function name: F
Name: ker(F)

{ (-3/35, -9/35, 1) }
```

Figure 37: function ker

- **im:** calculates the image of the selected function (has to be already been saved in memory).

The image is a base of vectors that represent the image-subset of the function range .

The name of the image-set created has the following syntax:

*(im) + (function name in parenthesis);*

```
Function >> calculate function im
Function name: T
Name: im(T)

{ (1, -2), (-1/2, 2/3) }
```

Figure 38: function image

### 3.3 print:

- **matrix:** prints a matrix previously saved in the list.  
The user is only required to remember and input the matrix name.  
**Print format:** can be **fraction** or **float**, as explained before (2).

```
Function >> print matrix
Matrix name: A
Print format: fraction
Name: A

  1  -1/2  0
-2  2/3  1
```

Figure 39: print matrix

- **vector:** prints a vector previously saved in the list.  
The user is only required to remember and input the vector name.

```
Function >> print vector
Vector name: b
Name: b = (1, -1, 1/3)
```

Figure 40: print vector

- **system:** prints a system previously saved in the list.  
The user is only required to remember and input the system name.  
**Print format:** can be **fraction** or **float**, as explained before (2).

```
Function >> print system
System name: S
Print format: fraction
Name: S

  1/2  -3  0  |  1
   1  2/5  2  | -1
   1   1  0  |  0
```

Figure 41: print system

- **set:** prints a set previously saved in the list.  
The user is only required to remember and input the set name.

```
Function >> print set
Set name: R

{ (1, -1, 1/3), (0, -2, 1/2) }
```

Figure 42: print set

- **function:** prints a function previously saved in the list.  
The user is only required to remember and input the set name.

```
Function >> print function
Function name: T
Name: T

Base "from": Name: C3

{ (1, 0, 0), (0, 1, 0), (0, 0, 1) }

Base "to": Name: C2

{ (1, 0), (0, 1) }

Name: A

1 -2 1/3
0 7/6 2
```

Figure 43: print function



No screenshots will be provided with the view command as it has a straightforward output.

### 3.4 remove:

- **matrix:** remove a matrix previously saved in the list.  
The user is only required to remember and input the matrix name.  
**Print format:** can be **fraction** or **float**, as explained before
- **vector:** remove a vector previously saved in the list.  
The user is only required to remember and input the vector name.
- **system:** remove a system previously saved in the list.  
The user is only required to remember and input the system name.  
**Print format:** can be **fraction** or **float**, as explained before (2).
- **set:** remove a set previously saved in the list.  
The user is only required to remember and input the set name.
- **function:** remove a function previously saved in the list.  
The user is only required to remember and input the set name.

No screenshots will be provided with the view command as it has a straightforward output.

### 3.5 **view:**

- **matrices:** prints all matrices previously saved in the matrix list.
- **vectors:** prints all vectors previously saved in the vector list.
- **systems:** prints all systems previously saved in the system list.
- **sets:** prints all sets previously saved in the set list.
- **functions:** prints all functions previously saved in the function list.

### 3.6 **cls:**

Clears the console buffer.

### 3.7 **/help:**

Shows a brief view of all the available commands

### 3.8 **END:**

End the program.