

MATLAB Basics - Help Sheet

David Lindsay

November 7, 2002

Contents

1	Explanation of MATLAB screen layout	3
2	Essentials for creating basic programs	4
3	Using matrices in MATLAB	5
3.1	Basic matrix notation	5
3.2	MATLAB's built in matrix functions	5
3.3	Manipulating matrices in MATLAB	7
3.3.1	Using the colon to create ranges	7
3.3.2	Accessing and manipulating elements of matrices	8
3.3.3	Concatenating matrices	8
3.3.4	Deleting rows and columns	9
3.3.5	Filtering content of matrices	9
3.3.6	Formatting output	9
4	Plotting graphs	10
4.1	How to use the <code>plot</code> command	10
4.1.1	Setting the range	10
4.1.2	Selecting colour	10
4.1.3	Selecting line style	11
4.1.4	Selecting marker types	11
4.2	How to set axis titles	12
4.3	Building up multiple plots using <code>hold</code>	12
5	Basic programming	13
5.1	The <code>if</code> statement	13
5.2	The <code>switch</code> statement	14
5.3	The <code>while</code> and <code>for</code> loops	14
5.4	The <code>break</code> and <code>continue</code> statements	14
6	The <code>quadprog</code> optimiser	15

List of Figures

1	The MATLAB desktop layout	3
2	Example of a multiple plot	13
3	Plot of the function $f(w) = \frac{1}{2}w_1^2 + w_2^2 - w_1w_2 - 2w_1 - 6w_2$	17

Aim of this document

MATLAB is a very ‘weakly’ typed programming language, as compared to languages such as C++ and java. It does not require formal type or dimension declarations such as int, float, double, char etc. When MATLAB encounters a new variable it automatically allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and if necessary allocates new storage. All of this can lead to some initial confusion, as seemingly short code seems to magically perform wonderfully complex operations. The aim of this handout is to introduce you to some of the basic (and most important) features of MATLAB using simple examples.

1 Explanation of MATLAB screen layout

To start MATLAB type in `matlab` at the linux command prompt, you will then be presented with the MATLAB desktop environment as seen in Figure 1.

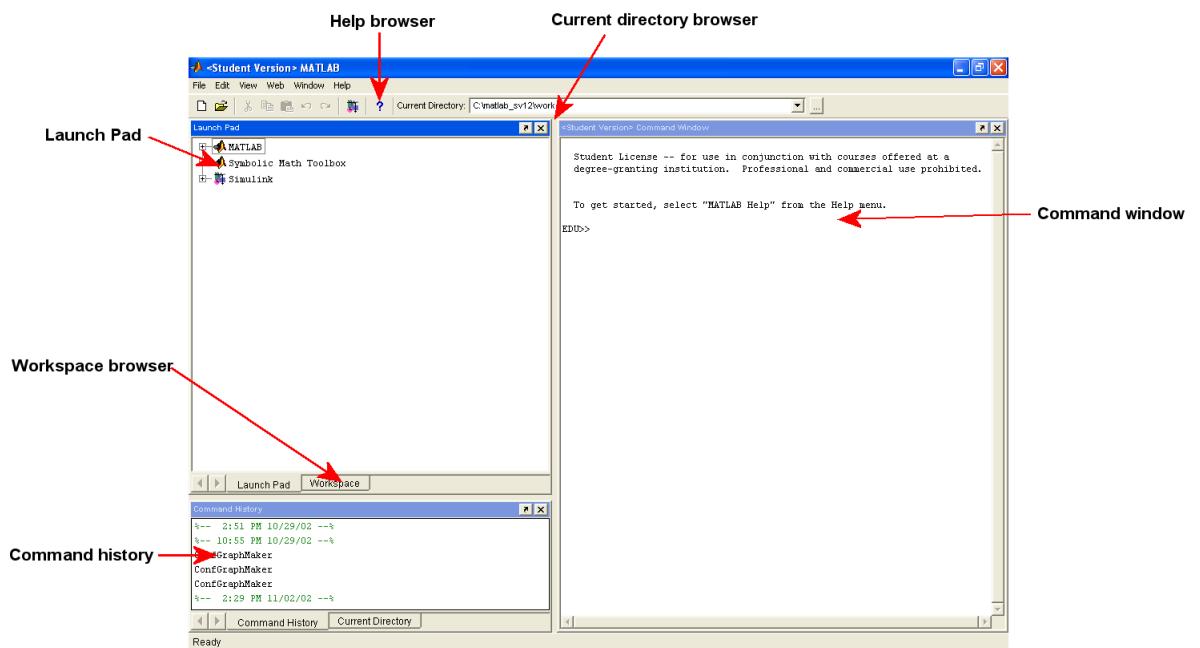


Figure 1 The MATLAB desktop layout: On starting up MATLAB you will be presented with a basic windows environment like so. Some of the most important features are highlighted above.

An explanation of these tools is as follows:

Command window This is where all the magic happens; this is where you will enter data, commands, run functions, see results etc.

4 LIST OF FIGURES

Command history Log of lines you enter in the **Command window**, which allows you to copy/select/re-execute lines easily.

Launch pad Provides easy access to other MATLAB tools (eg. Optimisation toolbox you will use for your coursework), as well as demos and documentation.

Help browser This allows you to search and view all documentation on MATLAB products.

Current directory browser This specifies which directory you are currently working in. Make sure this is pointing to the directory where your files are located.

Workspace browser This lists the names and sizes of arrays being used in your current session. This can be used as a basic visual debugging environment of your code.

2 Essentials for creating basic programs

You can either create your programs by typing in the commands directly into the MATLAB command window (one line at a time), or preferably structure the lines of code in a “.m” file, and then call the file within the **Command window**.

You could structure your MATLAB program in a file using MATLAB’s text editor by clicking on **File**→**New**→**M-file**. For example, try to create the file `helloworld.m` containing the code shown below:

```
x = (-10: 0.1 :10);  
y = x.^2;  
plot(x,y,'r-');  
title('My first MATLAB program');
```

Then (making sure that the current working directory is pointing to where the file is located) type `helloworld` in the command window to run the program.

To add comments to your code type the `%` character and whatever follows on that line will be ignored by MATLAB

If you terminate a line statement with the `;` character then MATLAB will not display the results of executing that particular line of MATLAB code. For example if a line of code specifying a matrix `A` does not end with a semi colon then the contents of matrix `A` will be output in the **Command window**.

This can be useful for debugging your code, and also cutting down the output of the program.

To find a formal specification about any of MATLAB’s built in functions type at the **Command window**, replacing `functionname` with the desired function:

```
help functionname
```

3 Using matrices in MATLAB

These are probably the most important core components of MATLAB and some basic understanding of matrix manipulations is required to use them effectively. Pretty much every operation you perform will involve manipulation of matrices and column/row vectors to derive solutions.

You can enter matrices with MATLAB in several ways:

1. Enter an explicit list of elements (using brackets []).
2. Load matrices from external data files (using the `load` command).
3. Generate matrices using MATLAB's built in functions such as `eye`, `diag` and `ones` etc.
4. Create matrices with your own functions in m-files.

3.1 Basic matrix notation

Matrices use the following format:

- ◇ Separate the elements of a row with a space or a comma.
- ◇ Use ; to indicate the end of a row of a matrix.
- ◇ Surround the entire list of elements with brackets [].
- ◇ Adding ; at the end of a statement triggers the 'don't display' option.

An example of explicitly creating a matrix is as follows:

```
A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

This will create the matrix **A**:

$$\mathbf{A} = \begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

Note: If we added the ; to the end it would not display the matrix in the command window.

3.2 MATLAB's built in matrix functions

As mentioned earlier, there are many built in MATLAB functions which are very useful. A basic example and description of these functions will be provided below. For further details about each function use MATLAB's help browser, or at the command window eg. `help eye`. You will find most of these functions can be used on both matrices and individual numbers. The best way to learn is to experiment with them.

6 LIST OF FIGURES

<code>B=sum(A)</code>	row matrix containing the column sums of A
<code>prod(A)</code> (when A is a list)	returns the product of the array elements in A
<code>prod(A)</code> (when A is a matrix)	returns row matrix whose elements are products of column elements.
<code>A'</code>	creates the transpose of matrix A (swap rows for columns)
<code>inv(A)</code>	create the inverse of matrix A (eg. $A^{-1}A = I$)
<code>det(A)</code>	calculates the determinant of a square matrix A
<code>A+B</code>	addition of matrices A and B
<code>A+2.1</code>	adds 2.1 element wise to matrix A
<code>A-B</code>	subtraction of B from A
<code>A*B</code>	multiplication of A and B
<code>A\B</code>	same as <code>inv(A)*B</code>
<code>A^3</code>	power of A, same as <code>A*A*A</code>
<code>B=sum(A')'</code>	stores rows of sums of A
<code>b=diag(A)</code>	stores the diagonal elements of a matrix A in the row vector b.
<code>B=diag(a)</code>	creates a matrix B which has row vector a's elements as its leading diagonal (all other elements = 0).
<code>sum(diag(A))</code>	sum of the diagonal elements of matrix A
<code>zeros(2,4)</code>	creates matrix $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$
<code>5*ones(3,2)</code>	creates matrix $\begin{bmatrix} 5 & 5 \\ 5 & 5 \\ 5 & 5 \end{bmatrix}$
<code>eye(3)</code>	creates a 3×3 identity matrix $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
<code>[n,m]=size(A)</code>	calculates the dimensions of the matrix A, the number of rows =n, the number of columns = m

<code>max(A)</code>	returns the largest value(s) within each column of a matrix
<code>min(A)</code>	returns the smallest value(s) within each column of a matrix
<code>mean(A)</code>	returns the average(s) of each column of a matrix
<code>std(A)</code>	returns the standard deviation(s) of each column of a matrix
<code>rand</code>	random number generator: creates uniformly between 0 and 1
<code>randn</code>	random number generator: creates using normal distribution
<code>fix</code>	rounds numbers downwards towards 0
<code>rand(4,5)</code>	4×5 matrix of random numbers
<code>pi</code>	constant for defining $\pi = 3.14159265\dots$
<code>Inf</code>	constant for defining infinity
<code>NaN</code>	constant for defining not a number
<code>A.*3</code>	performs element by element wise multiplication by 3 of matrix A
<code>A.^2</code>	performs element by element wise raising to the power 2 (squaring) of matrix A
<code>sqrtm(A)</code>	matrix of square roots of elements of A
<code>eig(A)</code>	gives the eigenvalues of A

3.3 Manipulating matrices in MATLAB

Along with basic functions there are some commonly used features of matrices in MATLAB which can save a lot of time when writing programs.

3.3.1 Using the colon to create ranges

Often in MATLAB you may want to create a range that you can pass to the `plot` command to specify which values to display the desired function for.

8 LIST OF FIGURES

This could be done explicitly by specifying a list of numbers, but this can become tedious if you wish to plot over a large range. This can be easily achieved using the `:` colon symbol like so:

`A=1:10` creates a row vector A of integer values ranging from 1 to 10 like so

```
A=[1 2 3 4 5 6 7 8 9 10]
```

`A=100:-7:50` creates a row vector A of integer values taking 7 away each time, starting from 100, but greater than 50

```
A=[100 93 86 79 72 65 58 51]
```

`A=0:pi/4:pi` creates a row vector A containing a range of values from 0 to π , using increments of $\frac{\pi}{4}$

```
A=[0 0.7854 1.5708 2.3562 3.1416]
```

3.3.2 Accessing and manipulating elements of matrices

Another useful function of the colon symbol is to add/remove rows or columns, and access individual elements of a matrix like so:

`A(1,4)=39.7` This will set the element row 1, column 4 to be value 39.7

`B=A(1:k,j)` stores the first k elements of the j th column of A in B

`B=A(:,end)` stores all the elements of the last column of A in B

`B=A(:,[permutation])` stores A in B interchanging columns according to the given permutation

`B=A(:,7)` stores the 7th column of A in B

`B=A(5,:)` stores the 5th row of A in B

3.3.3 Concatenating matrices

To create even more complex matrices, it is often useful to concatenate two (or more) simpler matrices. This is achieved by expressing the new matrix as a list (using the `[]` brackets and `;` semi colon) of matrices like so:

10 LIST OF FIGURES

`A=[4/3 0.0000123456 log(2)]` explicitly creates the matrix A

`A=[1.3333 1.2346e-005 0.69315]`

`format short` this rounds numbers to 4 decimal places

`A=[1.3333 0.000 0.6931]`

`format long` this rounds numbers to 14 decimal places

`A=[1.3333333333333333 0.00001234560000 0.69314718055995]`

`format bank` this rounds numbers to 2 decimal places

`A=[1.33 0.00 0.69]`

`format rat` approximates numbers as rational fractions

`A=[4/3 2/162001 1588/2291]`

4 Plotting graphs

4.1 How to use the plot command

The plot command has the following simple layout:

`plot(x,y,'colour style marker')`

The x and y is the range of values to be plotted, these are specified as row vectors. The x values are plotted horizontally, and the y values are plotted vertically.

These (x, y) coordinate pairs are then plotted on a 2 dimensional plot.

Further settings such as connecting the points with a line, the type of markers used for each point and the colour of these lines and markers can be specified as mentioned below.

4.1.1 Setting the range

It is very important that the range of values is specified correctly. You must make sure that the row vectors x and y are the same dimension, so that it can properly group each pair of coordinates.

4.1.2 Selecting colour

The colour of the line and markers can both be set using the following codes:

◇ c = cyan

◇ m = magenta

- ◇ `y` = yellow
- ◇ `r` = red
- ◇ `g` = green
- ◇ `b` = blue
- ◇ `w` = white
- ◇ `k` = black

4.1.3 Selecting line style

If no line is specified to connect the points then the resulting plot will look like a scatter plot of individual points. The type of the line used to connect the points can be specified using the following codes:

- ◇ `-` = solid
- ◇ `--` = dashed
- ◇ `:` = dotted
- ◇ `-.` = dash-dot
- ◇ `none` = no line

4.1.4 Selecting marker types

The type of marker that is used to draw each point can be specified using the following codes:

- ◇ `+` = plus sign
- ◇ `o` = circle
- ◇ `*` = star
- ◇ `x` = cross
- ◇ `s` = square
- ◇ `d` = diamond
- ◇ `^` = up-triangle
- ◇ `v` = down-triangle
- ◇ `>` = right-triangle
- ◇ `<` = left-triangle
- ◇ `p` = pentagram
- ◇ `h` = hexagram

4.2 How to set axis titles

To make the graphs that you produce easier to interpret you can introduce titles to your plot. Remember that you can put latex into these strings, to introduce mathematical notation. Some of the most useful title commands are displayed below.

```
title('Nice graph of  $y=x^2+\alpha$ ', 'FontSize', 12, 'FontWeight', 'bold')
```

this will create the plot title to be “Nice graph of $y = x^2 + \alpha$ ”, with font size = 12pt and font wieght = bold

```
xlabel('w_{1}', 'FontSize', 11)
```

this will place the text w_1 on the horizontal axis

```
ylabel('w_{2}', 'FontSize', 11)
```

this will place the text w_2 on the vertical axis

4.3 Building up multiple plots using hold

In many applications you will probably want to superimpose many plots on top of each other to accurately compare results. If you were to type multiple plot commands one after another you would normally just be left with the last plotted graph. To superimpose you must therefore place the text `hold on` before all of your plot commands, and then type `hold off` at the end.

This following example demonstrates all of the previous sections in constructing the simple graph shown in Figure 4.3.

```
x=-4:0.5:4;
```

sets the range for the x values from -4 to 4 using increments of 0.5

```
y1=(x.^2)-8;
```

specifies the first function $y = x^2 - 8$

```
y2=x;
```

specifies the second function $y = x$

```
hold on;
```

suppress any plot commands

```
plot(x,y1,'r-+');
```

plot the first function

```
plot(x,y2,'b:s');
```

plot the second function

```
title('Graph comparing  $y=x$  and  $y=x^2-8$ ', 'FontSize', 12, 'FontWeight', 'bold')
```

```
xlabel('x', 'FontSize', 11)
```

```
ylabel('y', 'FontSize', 11)
```

set appropriate titles

```
hold off;
```

allow graphs to be plotted

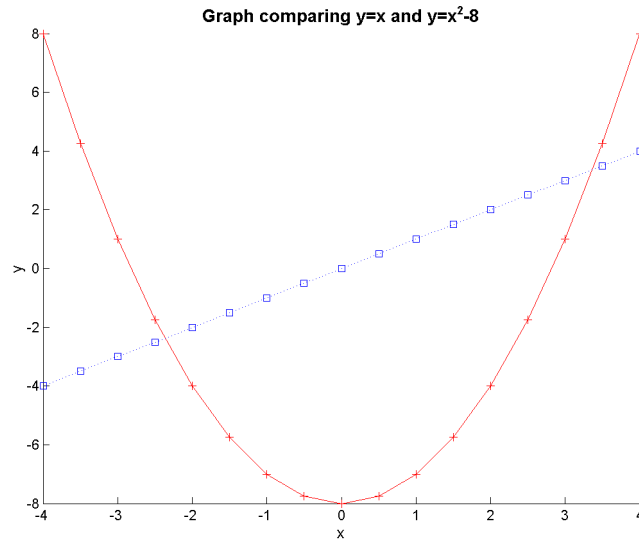


Figure 2 Example of a multiple plot : This is a very basic plot comparing two functions.

5 Basic programming

For completeness MATLAB offers many of the basic programming statements and routines commonly used in languages such as C++ and java. A brief example of each of these is given below.

5.1 The if statement

Much like if statements in many programming languages MATLAB requires the following format:

```
if logical expression
statements
elseif logical expression
statements
else
statements
end
```

The following comparison operators can be used:

```
== = equals
~= = not equals
> = greater than
>= = greater than or equal to
<= = less than or equal to
< = less than
~< = not less than
```

To compare the equality of two matrices A and B use:

```
if isequal(A,B)
```

5.2 The switch statement

To avoid messy nested if statements use the switch statement like so:

```
switch expression
case value1
statements
case value2
statements
.
.
otherwise
statements
end
```

5.3 The while and for loops

The for loop must be provided a range like so:

```
for i = 3:2:77
statements
end
```

The while loop requires a logical expression like so:

```
while logical expression
statements
end
```

5.4 The break and continue statements

The continue statement passes control to the next iteration of a for or while loop in which it appears.

The break statement lets you exit early from a for or while loop.

6 The quadprog optimiser

The quadprog MATLAB routine solves quadratic optimisation problems. The

function try's to solve for a vector of n variables $\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$, that minimises

the function:

$$\begin{aligned} f(\mathbf{w}) &= \frac{1}{2}\mathbf{w}'Q\mathbf{w} + u'\mathbf{w} \\ &\text{subject to the constraints} \\ &A'\mathbf{w} \leq a \\ &B'\mathbf{w} = b \\ &c \leq \mathbf{w} \leq d \end{aligned}$$

Note that in general solving n variables, with m constraints:

Q is an $n \times n$ matrix, u is a $n \times 1$ matrix used to specify the main term in the optimisation problem

A is an $m \times n$ matrix, a is a $m \times 1$ matrix used to specify the \leq side constraints (note that these can be simply reversed to \geq side constraints by multiplying by -1).

B is an $m \times n$ matrix, b is a $m \times 1$ matrix used to specify the = side constraints.

c is an $n \times 1$ matrix, d is a $n \times 1$ matrix used to specify the lower and upper bounds of the side constraints.

This can then be solved using the following MATLAB code:

```
w=quadprog(Q,u,A,a,B,b,c,d)
```

This will solve the optimisation problem and store the solution in the $n \times 1$ matrix \mathbf{w} .

Please note that you do not have to use all of these inputs to quadprog. For example if you do not require the additional constraints such as $B'\mathbf{w} = b$ and $c \leq \mathbf{w} \leq d$ then you could either enter them as matrices or don't include them at all eg. `quadprog(Q,u,A,a)` or `quadprog(Q,u,A,a,[],[],[],[])`.

16 LIST OF FIGURES

Let us consider a practical example of solving the following quadratic optimisation problem:

$$\begin{aligned} f(\mathbf{w}) &= \frac{1}{2}w_1^2 + w_2^2 - w_1w_2 - 2w_1 - 6w_2 \\ &\text{subject to the constraints} \\ w_1 + w_2 &\leq 2 \\ -w_1 + 2w_2 &\leq 2 \\ 2w_1 + w_2 &\leq 3 \\ w_1 &\geq 0 \\ w_2 &\geq 0 \end{aligned}$$

Using the same matrix notation as used earlier for the `quadprog` function we can formulate the problem using the following matrices:

$$Q = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}, \quad u = \begin{pmatrix} -2 \\ -6 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \\ -1 & 2 \\ 2 & 1 \end{pmatrix}, \quad a = \begin{pmatrix} 2 \\ 2 \\ 3 \end{pmatrix}, \quad c = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This can therefore be solved using the following MATLAB code:

```
Q=[1 -1; -1 2] % defines the quadratic part of the main optimisation term
u=[-2; -6] % defines the linear part of the main optimisation term
A=[1 1; -1 2; 2 1] % defines the less than or equal constraints
a=[2; 2; 3]
c=[0; 0] % defines the greater than or equal constraints
w=quadprog(Q,u,A,a,[],[],c,[]) % solves for the optimal w vector
```

The corresponding output of this code gives the solution for $\mathbf{w} = \begin{pmatrix} 0.6667 \\ 1.3333 \end{pmatrix}$.

If we look at a plot of this function as a 3-dimensional surface shown in Figure 3 we can see that these coordinates do indeed correspond to a minima. This function was plotted in MATLAB using the code:

```
ezmeshc('0.5*x.^2)+(y.^2)-(x*y)-(2*x)-(6*y)', [-3,3,-3,3])
```

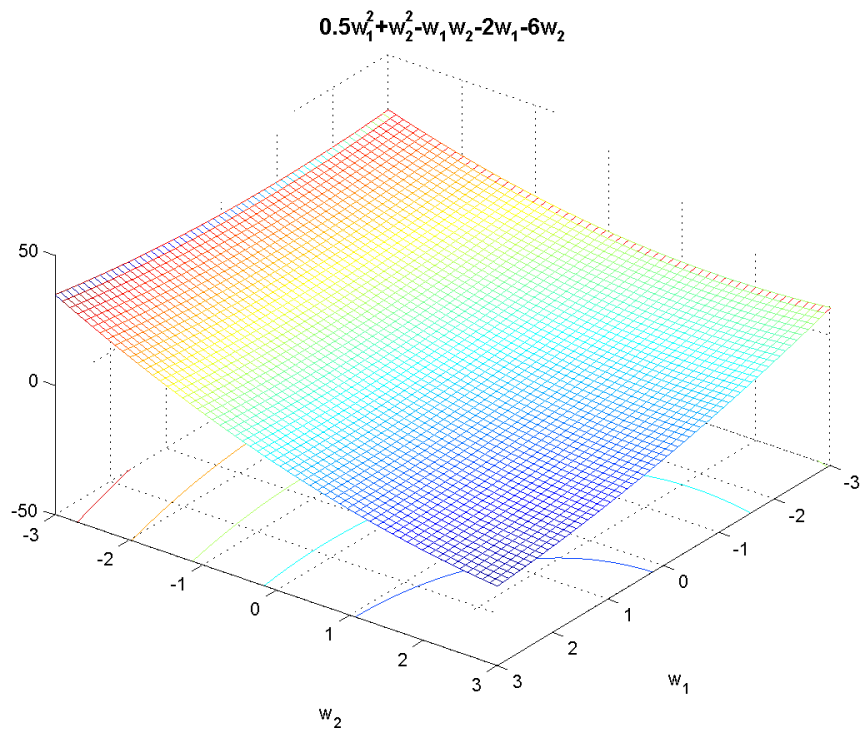


Figure 3 Plot of the function $f(w) = \frac{1}{2}w_1^2 + w_2^2 - w_1w_2 - 2w_1 - 6w_2$: This plot clearly demonstrates a local minima in the surface of the function $f(w)$. Notice that the solution given by quadprog is located in the low blue area of this plot.