



Guidelines for writing clean and fast code in MATLAB^{®*}


Nico Schlömer[†]

November 5, 2013

This document is aimed at MATLAB[®] beginners who already know the syntax but feel are not yet quite experienced with it. Its goal is to give a number of hints which enable the reader to write quality MATLAB[®] programs and to avoid commonly made mistakes.

There are three major independent chapters which may very well be read separately. Also, the individual chapters each split up into one or two handful of chunks of information. In that sense, this document is really a slightly extended list of dos and don'ts.

Chapter 1 describes some aspects of *clean* code. The impact of a subsection for the cleanliness of the code is indicated by one to five -symbols, where five 's want to say that following the given suggestion is of great importance for the comprehensibility of the code.

Chapter 2 describes how to speed up the code and is largely a list of mistakes that beginners may tend to make. This time, the -symbol represents the amount of speed that you could gain when sticking to the hints given in the respective section.

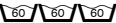














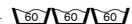

This guide is written as part of a basic course in numerical analysis, most examples and codes will hence tend to refer to numerical integration or differential equations. However, almost all aspects are of general nature and will also be of interest to anyone using MATLAB[®].

Contents

MATLAB[®] alternatives	3
Python	3

*MATLAB[®] Central “File Exchange Pick of the Week”, January 14, 2011 [Shoelson:2011:GMC]

[†]E-mail: nico.schloemer@gmail.com The author will be happy about comments and suggestions about the document.

GNU Octave	4
Scilab	4
Julia	5
1 Clean code	6
Multiple functions per file – 	7
Variable and function names – 	9
Indentation – 	11
Line length – 	11
Spaces and alignment – 	12
Magic numbers – 	12
Comments – 	13
Usage of brackets – 	14
Errors and warnings – 	14
Switch statements – 	18
2 Fast code	20
Using the profiler	20
The MATtrix LABoratory	21
Matrix pre-allocation – 	21
Loop vectorization – 	22
Solving a linear equation system – 	24
Dense and sparse matrices – 	25
Repeated solution of an equation system with the same matrix – 	26
3 Other tips & tricks	30
Functions as arguments – 	30
Implicit matrix–vector products – 	30

MATLAB[®] alternatives

When writing MATLAB[®] code, you need to realize that unlike C, Fortran, or Python code, you will always need the *commercial* MATLAB[®] environment to have it run. Right now, that might not be much of a problem to you as you are at a university or have some other free access to the software, but sometime in the future, this might change.

The current cost for the basic MATLAB[®] kit, which does not include *any* toolbox nor Simulink, is €500 for academic institutions; around €60 for students; *thousands* of Euros for commercial operations. Considering this, there is a not too small chance that you will not be able to use MATLAB[®] after you quit from university, and that would render all of your own code virtually useless to you.

Because of that, free and open source MATLAB[®] alternatives have emerged, three of which are shortly introduced here. Octave and Scilab try to stick to MATLAB[®] syntax as closely as possible, resulting in all of the code in this document being legal for the two packages as well. When it comes to the specialized toolboxes, however, neither of the alternatives may be able to provide the same capabilities that MATLAB[®] offers. However, these are mostly functions related to Simulink and the like which are hardly used by beginners anyway. Also note none of the alternatives ships with its own text editor (as MATLAB[®] does), so you are free to use the editor of your choice (see, for example, vim, emacs, Kate, gedit for Linux; Notepad++, Crimson Editor for Windows).

Python

Python is the most modern programming language as of 2013: Amongst the many award the language has received stands the TIOBE Programming Language Award of 2010. It is yearly given to the programming language that has gained the largest market market share during that year.



Python is used in all kinds of different contexts, and its versatility and ease of use has made it attractive to many. There are tons packages for all sorts of tasks, and the huge community and its open development help the enormous success of Python.

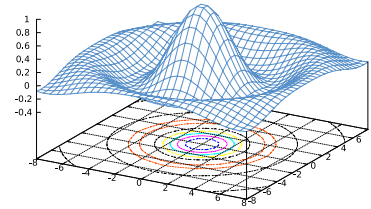
In the world of scientific computing, too, Python has already risen to be a major player. This is mostly due to the packages SciPy and Numpy which provide all data structures and algorithms that are used in numerical code. Plotting is most easily handled by matplotlib, a huge library which in many ways excels MATLAB[®]'s graphical engine.

Being a language rather than an application, Python is supported in virtually every operating system.

The author of this document highly recommends to take a look at Python for your own (scientific) programming projects.

GNU Octave

GNU Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB®. It may also be used as a batch-oriented language.



Internally, Octave relies on other independent and well-recognized packages such as gnuplot (for plotting) or UMFPACK (for calculating with sparse matrices). In that sense, Octave is extremely well integrated into the free and open source software (FOSS) landscape.

Octave has extensive tools for solving common numerical linear algebra problems, finding the roots of nonlinear equations, integrating ordinary functions, manipulating polynomials, and integrating ordinary differential and differential-algebraic equations. It is easily extensible and customizable via user-defined functions written in Octave's own language, or using dynamically loaded modules written in C++, C, Fortran, or other languages.

GNU Octave is also freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License (GPL) as published by the Free Software Foundation.

The project is originally GNU/Linux, but versions for MacOS, Windows, Sun Solaris, and OS/2 exist.

Scilab

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications.

Scilab is open source software and originates from the French International Institute for Research in Computer Science and Control (INRIA).



Since 1994 it has been distributed freely along with the source code via the Internet. It is currently used in educational and industrial environments around the world.

Scilab is now the responsibility of the Scilab Consortium, launched in May 2003. There are currently 18 members in Scilab Consortium (Phase II).

Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, C++, Fortran,...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter and a high level programming language.

Scilab works under Windows 9X/2000/XP/Vista, GNU/Linux, and most UNIX systems. Binary versions for these systems are freely available, along with the source code.

Julia

Quoting from julialang.org:

Julia is a high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. The library, largely written in Julia itself, also integrates mature, best-of-breed C and Fortran libraries for linear algebra, random number generation, signal processing, and string processing. In addition, the Julia developer community is contributing a number of external packages through Julia's built-in package manager at a rapid pace. IJulia, a collaboration between the IPython and Julia communities, provides a powerful browser-based graphical notebook interface to Julia.



1 Clean code

There is a plethora of reasons why code that *just works*TM is not good enough. Take a peek at listing 1 and admit:

- Fixing bugs, adding features, and working with the code in all other aspects get a lot easier when the code is not messy.
- Imagine someone else looking at your code, and trying to figure out what it does. In case you have you did not keep it clean, that will certainly be a huge waste of time.
- You might be planning to code for a particular purpose now, not planning on ever using it again, but experience tells that there is virtually no computational task that you come across only once in your programming life. Imagine yourself looking at your own code, a week, a month, or a year from now: Would you still be able to understand why the code works as it does? Clean code will make sure you do.

Examples of messy, unstructured, and generally ugly programs are plenty, but there are also places where you are almost guaranteed to find well-structured code. Take, for example the MATLAB[®] internals: Many of the functions that you might make use of when programming MATLAB[®] are implemented in MATLAB[®] syntax themselves – by professional MathWorks programmers. To look at such the contents of the `mean()` function (which calculates the average mean value of an array), type `edit mean` on the MATLAB[®] command line. You might not be able to understand what's going on, but the way the file looks like may give you hints on how to write clean code.

Multiple functions per file –

It is a common and false prejudice that MATLAB[®] cannot cope with several functions per file. The truth is: There *may* be more than one function in a file, but just the first one in the file will be *visible* to functions in other files or to the command line. In that

```
function l11(l11,l11,l11);if floor(l11/l11)<=1;...
l11(l11,l11+1,l11 );elseif mod(l11,l11)==0;l11(...
l11,l11+1,0);elseif mod(l11,l11)==floor(l11/...
l11)&&~l11;floor(l11/l11),l11(l11,l11+1,0);elseif...
mod(l11,l11)>1&&mod(l11,l11)<floor(l11/l11),...
l11(l11,l11+1,l11+~mod(floor(l11/l11),mod(l11,l11)) );
elseif l11<l11*l11;l11(l11,l11+1,l11);end;end
```

Listing 1: Perfectly legal MATLAB[®] code, with all rules of style ignored. Can you guess what this function does?

```

% =====
% callable from outside:
function topFun
    % [...]
    % calls helperFun1 and helperFun2
    % [...]
end
% =====

% =====
% only visible to all functions in this file:
function helperFun1
    % [...]
end
% =====


% =====
% only visible to all functions in this file:
function helperFun2
    % [...]
end
% =====

```

Listing 2: One source containing three functions: Useful when `helperFun1` and `helperFun2` are only needed by `topFun`.

sense, those functions in a file which do not take the first position can (only) act as a helper for the function on top (see listing 2).

When writing code, think about whether or not a particular function is really just a helper, or if needs to be allowed to be called from somewhere else. Doing so, you can avoid a cluttered mess of dozens of M-files in your program folder.

Subfunctions –  An issue that may come up if you have quite a lot of functions per file might be that you lose sight of which function actually requires which other function.

In case one of the functions is a helper function for not more than one other function, a clean place to put it would be *inside* the other function. This way, it will only be visible to the surrounding function and its name will not interfere with the name of any other

subfunction. The biggest advantage, however, is certainly that the subfunction is then syntactically *clearly associated* with its parent function. When looking at the code for the first time, the relations between the functions are immediately visible.

```
% [...]\n\n% =====\nfunction fun1\n    % call helpFun1 here\nend\n% =====\n\n% =====\nfunction fun2\n    % call helpFun2 here\nend\n% =====\n\n% =====\nfunction helpFun1\n    % [...]\nend\n% =====\n\n% =====\nfunction helpFun2\n    % [...]\nend\n% =====\n\n% [...]
```

The routines `fun1`, `fun2`, `helpFun1`, and `helpFun2` are sitting next to each other and no hierarchy is visible.

```
% [...]\n\n% =====\nfunction fun1\n    % call helpFun here\n\n    % -----\n    function helpFun\n        % [...]\n    end\n    % -----\nend\n% =====\n\n% =====\nfunction fun2\n    % call helpFun here\n\n    % -----\n    function helpFun\n        % [...]\n    end\n    % -----\nend\n% =====\n\n% [...]
```

Immediately obvious: The first `helpFun` helps `fun1`, the second `fun2` – and does nothing else.

Variable and function names –

One key ingredient for a consistent source code is a consistent naming scheme for the variables in use. From the dawn of programming languages in the 1950s, schemes have developed and decayed and are generally subject to evolution. There are, however, some general rules which have proven useful over the years in all kinds of various contexts.

Variable name	Usual purpose
<code>m, n</code>	integer sizes (,e.g., the dimension of a matrix)
<code>i, j, k (, l)</code>	integer numbers (mostly loop indices)
<code>x, y</code>	real values (x -, y -axis)
<code>z</code>	complex value or z -axis
<code>c</code>	complex value or constant (or both)
<code>t</code>	time value
<code>e</code>	the Eulerian number or ‘unit’ entities
<code>f, g (, h)</code>	generic function names
<code>h</code>	spatial discretization parameter (in numerical analysis)
<code>epsilon, delta</code>	small real entities
<code>alpha, beta</code>	angles or parameters
<code>theta, tau</code>	parameters, time discretization parameter (in n.a.)
<code>kappa, sigma, omega</code>	parameters
<code>u, v, w</code>	vectors
<code>A, M</code>	matrices
<code>b</code>	right-hand side of an equation system

Table 1: Variable names and their usual purposes in source codes. These guidelines are not particularly strict, but for example one would never use `i` to hold a float number, nor `x` for an integer.

In [Johnson:2002:MPS], a crisp and yet comprehensive overview on many aspects of variable naming is given; a few of the most useful ones are stated here.

Variable names tell what the variable does. Undoubtedly, this is the first and foremost rule in variable naming, and it implies several things.

- Of course, you would not name a variable `pi` when it really holds the value 2.718128, right?
- In mathematics and computer science, some names are connected to certain meanings. Table 1 lists a number of widely used conventions.

Short variable names. Short, non-descriptive variable names are quite common in mathematical computing as the variable names in the corresponding (pen and paper) calculations are hardly ever longer than one character either (see table 1). To be able to distinguish between vector and matrix entities, it is common practice in programming as well as mathematics to denote matrices by upper-case, vectors and scalars by lower-case characters.

```
K = 20;
a = zeros(K,K);
B = ones(K,1);

U = a*B;
```

```
k = 20;
A = zeros(k,k);
b = ones(k,1);

u = A*b;
```

Long variable names. A widely used convention mostly in the C++ development community to write long, descriptive variables in mixed case (camel case) starting with lower case, such as

```
linearity, distanceToCircle, figureLabel.
```

Alternatively, one could use the underscore to separate parts of a compound variable name:

```
linearity, distance_to_circle, figure_label.
```

This technique, although readable, is not commonly used for variable names in other languages. Also, some feel the underscore character is not quite easily typable on the keyboard.

When still using the underscore notation, watch out for variable names in MATLAB®'s plots: its T_EX-interpreter will treat the underscore as a switch to subscript and a variable name such as `distance_to_circle` will read `distancetocircle` in the plot.

Remark. Using the hyphen ‘-’ as a separator cannot be considered: MATLAB® will immediately interpret ‘-’ as the minus sign, `distance-to-circle` is “*distance minus to minus circle*”. The same holds for function names.

Logical variable names. If a variable is supposed to only hold the values 0 or 1 to represent `true` or `false`, then the variable name should express that. A common technique is to prepend the variable name by `is` and, less common, by `flag`.

```
isPrime, isInside, flagCircle
```

Indentation –

If you ever dealt with nested `for`- and `if` constructs, then you probably noticed that it may sometimes be hard to distinguish those nested constructions from other code at first sight. Also, if the contents of a loop extend over more than just a few lines, a visual aid may be helpful for indicating what is inside and what is outside the loop – and this is where indentation comes into play.

Usually, one would indent everything within a loop, a function, a conditional, a `switch` statement and so on. Depending on who you ask, you will be told to indent by two, three, or four spaces, or one tab. As a general rule, the indentation should yield a clear visual distinction while not using up all your space on the line (see next paragraph).

```
for i=1:n
for j=1:n
if A(i,j)<0
A(i,j) = 0;
end
end
end
```

No visual distinction between the loop levels makes it hard to recognize where the first loop ends.¹

```
for i=1:n
    for j=1:n
        if A(i,j)<0
            A(i,j) = 0;
        end
    end
end
```

With indentation, the code looks a lot clearer.¹

Line length –

There is de facto no limit on how much you can write on a single line of MATLAB® code. In fact, you could condense every MATLAB® code to a “one-liner” by separating two commands by a ‘;’ or a ‘,’ and suppress the newline character between them. However, a single line with one million characters will potentially not be very readable.

But, how many characters can you fit onto a single line without obscuring its content? This is certainly debatable, but commonly this value sits somewhere between 70 and 80; MATLAB®’s own text editor suggests 75 characters per line. This way, one makes also sure that it is not necessary to have a 22" widescreen monitor to be able to display the code without artificial line breaks or horizontal scrolling in the editor.

Sometimes of course your lines need to stretch longer than this, but that’s why MATLAB® contains the ellipses ‘...’ which makes sure the line following the line with the ellipsis is read as if there was no line break at all.

```
a = sin( exp(x) ) ...
    - alpha* 4^6      ...
    + u'*v;
```

```
a = sin( exp(x) ) - alpha* 4^6 + u'*v;
```

¹What the code does is replacing all negative entries of an $n \times n$ -matrix by 0. There is, however, a much better (shorter, faster) way to achieve this: `A(A<0) = 0;` (see page 23).

Spaces and alignment –

In some situations, it makes sense to break a line although it has not up to the limit, yet. This may be the case when you are dealing with an expression that – because of its length – has to break anyway further to the right; then, one would like to choose the line break point such that it coincides with *semantic* or *syntactic* break in the syntax. For examples, see the code below.

```
A = [ 1, 0.5 , 5; 4, ...  
42.23, 33; 0.33, ...  
pi, 1];
```

```
a = alpha*(u+v)+beta*...  
sin(p'*q)-t...  
*circleArea(10);
```

Unsemantic line breaks decrease the readability. Neither the shape of the matrix, nor the number of summands in the second expression is clear.

```
A = [ 1      , 0.5   , 5 ; ...  
      4      , 42.23, 33; ...  
      0.33, pi     , 1  ];
```

```
a = alpha* (u+v)      ...  
+ beta*  sin(p'*q) ...  
- t*      circleArea(10);
```

The shape and contents of the matrix, as well as the elements of the second expression, are immediately visible to the programmer.

Spaces in expressions Closely related to this is the usage of spaces in expressions. The rule is, again: put spaces there where MATLAB[®]'s syntax would. Consider the following example.

```
aValue = 5+6 / 3*4;
```

This spacing suggests that the value of `aValue` will be 11/12, which is of course not the case.

```
aValue = 5 + 6/3*4;
```

Much better, as the the fact that the addition is executed last gets reflected by according spacing.

Magic numbers –

When coding, sometimes you consider a value constant because you do not intend to change it anytime soon. Take, for example, a program that determines whether or not a given point sits outside a circle of radius 1 with center (1,1) and at the same time inside a square of edge length 2, right enclosing the circle (see [Hull:2006:CCM]).

When finished, the code will contain a couple of 1s but it will not be clear if they are distinct or refer to the same abstract value (see below). Those hard coded numbers are frequently called *magic numbers*, as they do what they are supposed to do, but one cannot easily tell why. When you, after some time, change your mind and you do want

to change the value of the radius, it will be rather difficult to identify those 1s which actually refer to it.

```
x = 2; y = 0;

pointsDistance = ...
    norm( [x,y]-[1,1] );

isInCircle = ...
    (pointsDistance < 1);
isInSquare = ...
    ( abs(x-1)<1 ) && ...
    ( abs(y-1)<1 );

if ~isInCircle && isInSquare
% [...]
```

It is not immediately clear if the various 1s do in the code and whether or not they represent one entity. These numbers are called *magic numbers*.

```
x = 2; y = 0;

radius = 1;
xc = 1; yc = 1;

pointsDistance = ...
    norm( [x,y]-[xc,yc] );


isInCircle = ...
    (pointsDistance < radius);
isInSquare = ...
    ( abs(x-xc)<radius ) && ...
    ( abs(y-yc)<radius );

if ~isInCircle && isInSquare
% [...]
```

The meaning of the variable `radius` is can be instantly seen and its value easily altered.

Comments –

The most valuable character for clean MATLAB® code is ‘%’, the comment character. All tokens after it on the same line are ignored, and the space can be used to explain the source code in English (or your tribal language, if you prefer).

Documentation –  There should be a big fat neon-red blinking frame around this paragraph, as documentation is *the* single most important aspect about clean and readable code. Unfortunately, it also takes the longest to write which is why you will find undocumented source code everywhere you go.

Look at listing 3 for a suggestion for quick and clear documentation, and see if you can do it yourself!

Structuring elements –  It is always useful to have the beginning and the end of the function not only indicated by the respective keywords, but also by something more visible. Consider building ‘fences’ with commented ‘#’, ‘=’, or ‘-’ characters, to visually

separate distinct parts of the code. This comes in very handy when there are multiple functions in one source file, for example, or when there is a `for`-loop that stretches over that many lines that you cannot easily find the corresponding `end` anymore.

For a (slightly exaggerated) example, see listing 3.

Usage of brackets –

Of course, there is a clearly defined operator precedence list in MATLAB® (see table 2) that makes sure that for every MATLAB® expression, involving any unary or binary operator, there is a unique way of evaluation. It is quite natural to remember that MATLAB® treats multiplication (*) before addition (+), but things may get less intuitive when it comes to logical operators, or a mix of numerical and logical ones (although this case is admittedly very rare).

Of course one can always look those up (see table 2), but to save the work one could equally quick just insert a pair of bracket at the right spot, although they may be unnecessary – this will certainly help avoiding confusion.

```
isGood = a<0 ...
        && b>0 || k~=0;
```

Without knowing if MATLAB® first evaluates the short-circuit AND ‘&&’ or the short-circuit OR ‘||’, it is impossible to predict the value of `isGood`.

```
isGood = ( a<0 && b>0 ) ...
        || k~=0;
```

With the (unnecessary) brackets, the situation is clear.

Errors and warnings –

No matter how careful you design your code, there will probably be users who manage to crash it, maybe with bad input data. As a matter of fact, this is not really uncommon in numerical computation that things go fundamentally wrong.

Example. *You write a routine that defines an iterative process to find the solution $u^* = A^{-1}b$ of a linear equation system (think of conjugate gradients). For some input vector u , you hope to find u^* after a finite number of iterations. However, the iteration will only converge under certain conditions on A ; and if A happens not to fulfill those, the code will misbehave in some way.*

It would be bad practice to assume that the user (or, you) always provides input data to your routine fulfilling all necessary conditions, so you would certainly like to conditionally intercept. Notifying the user that something went wrong can certainly be done by `disp()` or `fprintf()` commands, but the clean way out is using `warning()` and `error()`. – The

```

% =====
% *** FUNCTION timeIteration
% ***
% *** Takes a starting vector u and performs n time steps.
% ***
% =====
function out = timeIteration( u, n )

% -----
% set the parameters
tau    = 1.0;
kappa = 1.0;
out    = u;
% -----

% -----
% do the iteration
for k = 1:n

    [out, flag] = proceedStep( out, tau, kappa );

    % - - - - -
    % warn if something went wrong
    if ~flag
        warning( [ 'timeIteration:errorFlag', ...
                    'proceedStep returns flag ', flag ] );
    end
    % - - - - -

end

% -----
end
% =====
% *** END FUNCTION timeIteration
% =====

```

Listing 3: Function in which ‘-’-fences are used to emphasize the functionally separate sections of the code.

1. Parentheses (`()`)
2. Transpose (`.'`), power (`.^`), complex conjugate transpose (`'`), matrix power (`^`)
3. Unary plus (`+`), unary minus (`-`), logical negation (`~`)
4. Multiplication (`.*`), right division (`./`), left division (`.\`), matrix multiplication (`*`), matrix right division (`/`), matrix left division (`\`)
5. Addition (`+`), subtraction (`-`)
6. Colon operator (`:`)
7. Less than (`<`), less than or equal to (`<=`), greater than (`>`), greater than or equal to (`>=`), equal to (`==`), not equal to (`~=`)
8. Element-wise AND (`&`)
9. Element-wise OR (`|`)
10. Short-circuit AND (`&&`)
11. Short-circuit OR (`||`)

Table 2: MATLAB[®] operator precedence list.

latter differs from the former only in that it terminates the execution of the program right after having issued its message.

```
tol = 1e-15;
rho = norm(r);

while abs(rho)>tol

    r = oneStep( r );
    rho = norm( r );
end

% process solution
```

Iteration over a variable `r` that is supposed to be smaller than `tol` after some iterations. If that fails, the loop will never exit and occupy the CPU forever.

```
tol = 1e-15;
rho = norm(r);
kmax = 1e4;

k = 0;
while abs(rho)>tol && k<kmax
    k = k+1;
    r = oneStep( r );
    rho = norm( r );
end

if k==kmax
    warning('myFun:noConv',...
            'Did not converge.')
else
    % process solution
end
```

Good practice: there is a maximum number of iterations. When it has been reached, the iteration failed. Throw a warning in that case.

Although you could just evoke `warning()` and `error()` with a single string as argument (such as `error('Something went wrong!')`), good style programs will leave the user with a clue *where* the error has occurred, and of what type the error is (as mnemonic). This information is contained in the so-called *message ID*.

The MATLAB[®] help page contain quite a bit about message IDs, for example:

The `msgID` argument is a unique message identifier string that MATLAB[®] attaches to the error message when it throws the error. A message identifier has the format `component:mnemonic`. Its purpose is to better identify the source of the error.

Switch statements –

`switch` statements are in use whenever one would otherwise have to write a conditional statement with several `elseif` statements. They are also particularly popular when the conditional is a string comparison (see example below).

```
switch pet
  case 'Bucky'
    feedCarrots();
  case 'Hector'
    feedSausages();
end
```

When none of the cases matches, the algorithm will just skip and continue.

```
switch pet
  case 'Bucky'
    feedCarrots();
  case 'Hector'
    feedSausages();
  otherwise
    error('petCare:feed',...
          'Unknown pet.')
end
```

The unexpected case is intercepted.

```

% =====
% *** FUNCTION prime
% ***
% *** Returns all prime numbers below or equal to N.
% ***
% =====
function p = prime( N )

    for i = 2:N
        % checks if a number is prime
        isPrime = 1;
        for j = 2:i-1
            if ~mod(i, j)
                isPrime = 0;
                break
            end
        end

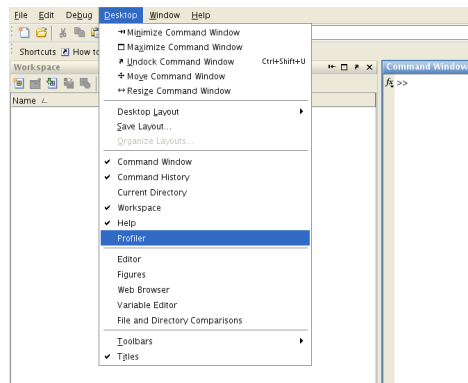
        % print to screen if true
        if isPrime
            fprintf( '%d is a prime number.\n', i );
        end
    end

end

% =====
% *** END FUNCTION prime
% =====

```

Listing 4: The same code as in listing 1, with rules of style applied. It should now be somewhat easier to maintain and improve the code. Do you have ideas how to speed it up?



(a) Evoking the profiler through the graphical user interface.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
14	pointsDistance = norm ([x, y...	1000000	3.912 s	51.7%	
15	isInCircle = (pointsDista...	1000000	0.607 s	8.0%	
16	isInSquare = (abs (x - xc...	1000000	0.475 s	6.3%	
12	xc = 1; yc = 1;	1000000	0.475 s	6.3%	
9	y = X(k);	1000000	0.445 s	5.9%	
All other lines			1.648 s	21.8%	
Totals			7.561 s	100%	

(b) Part of the profiler output when running the routine of section on magic numbers (see page 13) one million times. Clearly, the `norm` command takes longest to execute, so when trying to optimize one should start there.

Figure 1: Using the profiler.

2 Fast code

As a MATLAB[®] beginner, it is quite easy to use code that just works[™] but comparing to compiled programs of higher programming languages is very slow. The benefit of the relatively straightforward way of programming in MATLAB[®] (where no such things as explicit memory allocation, pointers, or data types “come in your way”) needs to be paid with the knowledge of how to avoid fundamental mistakes. Fortunately, there are only a few *big ones*, so when you browse through this section and stick to the given hints, you can certainly be quite confident about your code.

Using the profiler

The first step in optimizing the speed of your program is finding out where it is actually going slow. In traditional programming, bottlenecks are not quite easily found, and the humble coder would maybe insert timer commands around those chunks of code where he or she suspects the delay to actually measure its performance. You can do the same thing in MATLAB[®] (using `tic` and `toc` as timers) but there is a much more convenient way: *the profiler*.

The profiler is actually a wrapper around your whole program that measures the execution time of each and every single line of code and depicts the result graphically. This way, you can very quickly track down the lines that keep you from going fast. See figure 1 for an example output.

Remark. Besides the graphical interface, there is also a command line version of the profiler that can be used to integrate it into your scripts. The commands to invoke are `profile on` for starting the profiler and `profile off` for stopping it, followed by various commands to evaluate the gathered statistics. See the MATLAB[®] help page on `profile`.

The MATtrix LABoratory

Contrary to common belief, the MAT in MATLAB[®] does not stand for mathematics, but *matrix*. The reason for that is that proper MATLAB[®] code uses matrix and vector structures as often as possible, prominently at places where higher programming languages such as C or Fortran would rather use loops.

The reason for that lies in MATLAB[®]'s being an interpreted language. That means: There is no need for explicitly compiling the code, you just write it and have it run. The MATLAB[®] interpreter then scans your code line by line and executes the commands. As you might already suspect, this approach will never be able to compete with compiled source code.

However, MATLAB[®]'s internals contain certain precompiled functions which execute basic matrix-vector operations. Whenever the MATLAB[®] interpreter bumps into a matrix-vector expression, the contents of the matrices are forwarded to the underlying optimized and compiled code which, after execution, returns the result. This approach makes sure that matrix operations in MATLAB[®] are on par with matrix operations with compiled languages.

Remark. *Not only for matrix-vector operations, precompiled binaries are provided. Most standard tasks in numerical linear algebra are handled with a customized version of the ATLAS (BLAS) library. This concerns for example commands such as `eig()` (for finding the eigenvalues of a matrix), `\` (for solving a linear equation system with Gaussian² elimination), and so on.*

Matrix pre-allocation – !!!!!

When a matrix appears in MATLAB[®] code for the first time, its contents need to be stored in system memory (RAM). This process is called allocation. To do this, MATLAB[®] needs to find a place in memory (a range of *addresses*) which is large enough to hold the matrix and assign this place to the matrix. This process is called allocation. Note that typically, matrices are stored continuously in memory, and not split up to here and there. This way, the processor can quickly access its entries without having to look around in the system memory.

Now, what happens if the vector `v` gets allocated with 55 elements by, for example, `v=rand(55,1)`, and the user decides later in the code to make it a little bigger, say, `v=rand(1100,1)`? Well, obviously MATLAB[®] has to find a new slot in memory in case the old one is not wide enough to hold all the new entries. This is not so bad if it happens

²Johann Carl Friedrich Gauß (1777–1855), German mathematician and deemed one of the greatest mathematicians of all times. In the English-speaking world, the spelling with 'ss' instead of the original 'ß' has achieved wide acceptance – probably because the 'ß' is not included in the key set of any keyboard layout except the German one.

once or twice, but can slow down your code dramatically when a matrix is growing inside a loop, for example.

```
n = 1e5;
for i = 1:n
    u(i) = sqrt(i);
end
```

The vector u is growing n times and it probably must be re-allocated as often. The approximate execution time of this code snippet is **21.20 s**.

```
n = 1e5;
u = zeros(n,1);
for i = 1:n
    u(i) = sqrt(i);
end
```

As maximum size of the vector is known beforehand, one can easily tell MATLAB® to place u into memory with the appropriate size. The code here merely takes **3.8 ms** to execute!

Remark. *The previous code example is actually a little misleading as there is a much quicker way to fill u with the square roots of consecutive numbers. Can you find the one-liner? A look into the next section could help...*

Loop vectorization – ~~!!!!~~

Because of the reasons mentioned in the beginning of this section, you would like to avoid loops wherever you can and try to replace it by a vectorized operation.

When people commonly speak of ‘optimizing code for MATLAB®’, it will most often be this particular aspect. The topic is huge and this section can merely give the idea of it. If you are stuck with slow loop operations and you have no idea how to make it really quick, take a look at the excellent and comprehensive guide at [Mathworks:2009:CVG]. – There is almost always a way to vectorize.

Consider the following example a general scheme of how to remove loops from vectorizable operations.

```

n = 1e7;
a = 1;
b = 2;

x = zeros( n, 1 );
y = zeros( n, 1 );
for i=1:n
    x(i) = a + (b-a)/(n-1) ...
           * (i-1);
    y(i) = x(i) - sin(x(i))^2;
end

```

Computation of $f(x) = x - \sin^2(x)$ on n points between a and b . In this version, each and every single point is being treated explicitly. Execution time: approx. **0.91 s**.

```

n = 1e7;
a = 1;
b = 2;

h = 1/(n-1);

x = (a:h:b);
y = x - sin(x).^2;

```

Does the same thing using vector notation. Execution time: approx. **0.12 s**.

The `sin()` function in MATLAB[®] hence takes a vector as argument and acts as if it operated on each element of it. Almost all MATLAB[®] functions have this capability, so make use of it if you can!

Vector indexing and boolean indexing. When dealing with vectors or matrices, it may sometimes happen that one has to work only on certain entries of the object, e.g., those with odd index.

Consider the following three different possibilities of setting the *odd* entries of a vector v to 0.

```

% [...] create v

n = length(v);

for k = 1:2:n
    v(k) = 0;
end

```

Classical loop of the entries of interest (**1.04 s**).

```

% [...] create v

n = length(v);

v(1:2:n) = 0;

```

Vector indexing: Matrices take (positive) integer vectors as arguments (**1.14 s**).

```

% [...] create v

n = length(v);
mask = false(n,1);
mask(1:2:n) = true;
v( mask ) = 0;

```

Boolean indexing: Matrices take *boolean* arrays³ with the same shape as v as arguments (**1.41 s**).

³A mistake that beginners tend to make is to define `mask` as an array of integers, such as `mask = zeros(n,1);`.

In this case, where the indices to be worked on are known beforehand, the classical way of looping over the error is the fastest. Vector indexing makes the code shorter, but creates a slight overhead; boolean indexing, by having to create the boolean array `mask`, is significantly slower.

However, should the criteria upon which action is taken dynamically depend on the content of the vector itself, the situation is different. Consider again the three schemes, this time for setting the NaN entries of a vector `v` to 0.

```
% [...] create v

for k = 1:n
    if isnan(v(k))
        v(k) = 0;
    end
end
```

```
% [...] create v

ind = ...
    find(isnan(v));
v( ind ) = 0;
```

```
% [...] create v

mask = isnan(v);
v( mask ) = 0;
```

Classical loop: **1.19 s.** Vector indexing: **0.44 s.** Boolean indexing: **0.33 s.**

Iterating through the array `v` and checking each element individually means disregarding the “MAT” in MATLAB®. Making use of the `find()` function, it is possible to have `isnan()` work on the whole vector before setting the desired indices to 0 in one go. Even better than that, doing away with the overhead that `find()` creates, is to use the boolean array that `isnan()` returns to index `v` directly⁴.

See also [Mathworks:2001:MIM].

Solving a linear equation system –

When being confronted with a standard linear equation system of the form $Au = b$, the solution can be written down as $u = A^{-1}b$ if A is regular. It may now be quite seductive to translate this into `u = inv(A)*b` in MATLAB® notation. Though this step will certainly yield the correct solution (neglecting round-off errors, which admittedly can be quite large in certain cases), it would take quite a long time to execute. The reason for this is the fact that the computer actually does more work then required. What you tell MATLAB® to do here is to

1. explicitly calculate the inverse of `A`, store it in a temporary matrix, and then
2. multiply the this matrix with `u`.

⁴Remember: You can combine several `masks` with the logical operators `&` (“and”) and `|` (“or”). For example, `mask = isnan(v) | isinf(v);` is `true` wherever `v` has a NaN or an Inf.

However, one is most often not interested in the explicit form of A^{-1} , but only the final result $A^{-1}b$. The proper way out is MATLAB®'s `\` (backslash) operator (or equivalently `mldivide()`) which exactly serves the purpose of solving an equation system with Gaußian elimination.

```
n = 2e3;  
A = rand(n,n);  
b = rand(n,1);  
  
u = inv(A)*b;
```

Solving the equation system with an explicit inverse. Execution time: approx. **2.02 s.**

```
n = 2e3;  
A = rand(n,n);  
b = rand(n,1);  
  
u = A\b;
```

Solving the equation system with the `\` operator. Execution time: approx. **0.80 s.**

Dense and sparse matrices – ~~!!!!~~

Most discretizations of particular problems yield $N \times N$ -matrices which only have a small number of non-zero elements (proportional to N). These are called sparse matrices, and as they appear so very often, there is plenty of literature describing how to make use of that structure.

In particular, one can

- cut down the amount of memory used to store the matrix. Of course, instead of storing all the 0's, one would rather store the value and indices of the non-zero elements in the matrix. There are different ways of doing so. MATLAB® internally uses the condensed-column format, and exposes the matrix to the user in indexed format.
- optimize algorithms for the use with sparse matrices. As a matter of fact, most basic numerical operations (such as Gaußian elimination, eigenvalue methods and so forth) can be reformulated for sparse matrices and save an enormous amount of computational time.

Of course, operations which only involve sparse matrices will also return a sparse matrix (such as matrix–matrix multiplication `*`, `transpose`, `kron`, and so forth).

```

n = 1e4;
h = 1/(n+1);

A = zeros(n,n);
A(1,1) = 2;
A(1,2) = -1;
for i=2:n-1
    A(i,i-1) = -1;
    A(i,i) = 2;
    A(i,i+1) = -1;
end
A(n,n-1) = -1;
A(n,n) = 2;

A = A / h^2;

% continued below

```

Creating the tridiagonal matrix $1/h^2 \times \text{diag}[-1, 2, -1]$ in dense format. The code is bulky for what it does, and cannot use native matrix notation. Execution time: **0.67 s**.

```

% A in dense format
b = ones(n,1);
u = A\b;

```

Gaussian elimination with a tridiagonal matrix in dense format. Execution time: **55.06 s**.

☞ Useful functions: `sparse()`, `spdiags()`, `speye()`, `(kron())`,...

```

n = 1e4;
h = 1/(n+1);

e = ones(n,1);
A = spdiags([-e 2*e -e],...
            [-1 0 1],...
            n, n);

A = A / h^2;

% continued below

```

The three-line equivalent using the sparse matrix format. The code is not only shorter, easier to read, but also saves gigantic amounts of memory. Execution time: **5.4 ms!**

```

% A in sparse format
b = ones(n,1);
u = A\b;

```

The same syntax, with A being sparse. Execution time: **0.36 ms!**

Repeated solution of an equation system with the same matrix – ~~!!!!~~

It might happen sometimes that you need to solve an equation system a number of times with the same matrix but different right-hand sides. When all the right hand sides are immediately available, this can be achieved with with one ordinary ‘\’ operation.

```

n = 1e3;
k = 50;
A = rand(n,n);
B = rand(n,k);

u = zeros(n,k);

for i=1:k
    u(:,k) = A \ B(:,k);
end

```

Consecutively solving with a couple of right-hand sides. Execution time: **5.64 s**.

```

n = 1e3;
k = 50;
A = rand(n,n);
B = rand(n,k);

u = A \ B;

```

Solving with a number of right hand sides in one go. Execution time: **0.13 s**.

If, on the other hand, you need to solve the system once to get the next right-hand side (which is often the case with time-dependent differential equations, for example), this approach will not work; you will indeed have to solve the system in a loop. However, one would still want to use the information from the previous steps; this can be done by first factoring A into a product of a lower triangular matrix L and an upper triangular matrix U , and then instead of computing $A^{-1}u^{(k)}$ in each step, computing $U^{-1}L^{-1}u^{(k)}$ (which is a lot cheaper).

```

n = 2e3;
k = 50;
A = rand(n,n);

u = ones(n,1);

for i = 1:k
    u = A \ u;
end

```

Computing $u = A^{-k}u_0$ by solving the equation systems in the ordinary way. Execution time: **38.94 s**.

```

n = 2e3;
k = 50;
A = rand(n,n);

u = ones(n,1);

[L,U] = lu(A);
for i = 1:k
    u = U \ (L \ u);
end

```

Computing $u = A^{-k}u_0$ by LU -factoring the matrix, then solving with the LU factors. Execution time: **5.35 s**. Of course, when increasing the number k of iterations, the speed gain compared to the ' $A \setminus$ ' will be more and more dramatic.

Remark. For many matrices A in the above example, the final result will be heavily corrupted with round-off errors such that after $k = 50$ steps, the norm of the residual

$\|u_0 - A^k u\|$, which ideally equals 0, can be pretty large.

Factorizing sparse matrices. When LU - or Cholesky-factorizing a *sparse matrix*, the factor(s) are in general not sparse anymore and can demand quite an amount of space in memory to the point where no computer can cope with that anymore. The phenomenon of having non-zero entries in the LU - or Cholesky-factors where the original matrix had zeros is called *fill-in* and has attracted a lot of attention in the past 50 years. As a matter of fact, the success of iterative methods for solving linear equation systems is largely thanks to this drawback.

Beyond using an iterative method to solve the system, the most popular way to cope with fill-in is to try to re-order the matrix elements in such a way that the new matrix induces less fill-in. Examples of re-ordering are *Reverse Cuthill-McKee* and *Approximate Minimum Degree*. Both are implemented in MATLAB[®] as `colrcm()` and `colamd()`, respectively (with versions `symrcm()` and `symamd()` for symmetric matrices).

One can also leave all the fine-tuning to MATLAB[®] by executing `lu()` for sparse matrices with more output arguments; this will return a factorization for the permuted and row-scaled matrix $PR^{-1}AQ = LU$ (see MATLAB[®]'s help pages and example below) to reduce fill-in and increase the stability of the algorithm.

```

n = 2e3;
k = 50;

% get a non-singular nxn
% sparse matrix A:
% [...]

u = ones(n,1);

[L,U] = lu( A );
for i = 1:k
    u = U\( L\u );
end

```

Ordinary LU -factorization for a sparse matrix A . The factors L and U are initialized as sparse matrices as well, but the fill-in phenomenon will undo this advantage. Execution time: **4.31 s**.

```

n = 2e3;
k = 50;

% get a non-singular nxn
% sparse matrix A:
% [...]

u = ones(n,1);

[L,U,P,Q,R] = lu(A);
for i = 1:k
    u = Q*( U\(L\(P*(R\u))) );
end

```

LU -factoring with permutation and row-scaling. This version can use less memory, execute faster, and provide more stability than the ordinary LU -factorization. Execution time: **0.07 s**.

This factorization is implicitly applied by MATLAB[®] when using the ‘\’-operator for solving a sparse system of equations *once*.

3 Other tips & tricks

Functions as arguments –

In numerical computation, there are set-ups which natively treat functions as the objects of interest, for example when numerically integrating them over a particular domain. For this example, imagine that you wrote a function that implements Simpson’s integration rule (see listing 5), and you would like to apply it to a number of functions without having to alter your source code (for example, replacing `sin()` by `cos()`, `exp()` or something else).

A clean way to deal with this in MATLAB® is using *function handles*. This may sound fancy, and describes nothing else then the capability of treating functions (such as `sin()`) as arguments to other functions (such as `simpson()`). The function call itself is written as easy as

```
a = 0;
b = pi/2;
h = 1e-2;
int_sin = simpson( @sin, a, b, h );
int_cos = simpson( @cos, a, b, h );
int_f    = simpson( @f, a, b, h );
```

where the function name need to be prepended by the ‘@’-character.

The function `f()` can be any function that you defined yourself and which is callable as `f(x)` with `x` being a vector of x values (like it is used in `simpson()`, listing 6).

Implicit matrix–vector products –

In numerical analysis, almost all methods for solving linear equation systems *quickly* are iterative methods, that is, methods which define how to iteratively approach a solution in small steps (starting with some initial guess) rather than directly solving them in one big step (such as Gaußian elimination). Two of the most prominent iterative methods are CG and GMRES.

In particular, those methods *do not require the explicit availability of the matrix* as in each step of the iteration they merely form a matrix-vector product with A (or variations of it). Hence, they technically only need a function to tell them how to carry out a matrix-vector multiplication. In some cases, providing such a function may be easier than explicitly constructing the matrix itself, as the latter usually requires one to pay close attention to indices (which can get extremely messy).

```

% =====
% *** FUNCTION simpson
% ***
% *** Implements Simpson's rule for integrating
% *** the sine function over [a,b] with granularity
% *** h.
% ***
% =====
function int = simpson( a, b, h )

    x = a:h:b;

    int = 0;
    n = length(x);
    mid = (x(1:n-1) + x(2:n)) / 2;
    int = sum( h/6 * (      sin(x(1:n-1)) ...
                        + 4*sin(mid      ) ...
                        +      sin(x(2:n  )) ) );

end
% =====
% *** END FUNCTION simpson
% =====

```

Listing 5: Implementation of Simpson's rule for numerically integrating a function (here: `sin`) between `a` and `b`. Note the usage of the vector notation to speed up the function. Also note that `sin` is hardcoded into the routine, and needs to be changed each time we want to change the function. In case one is interested in calculating the integral of $f(x) = \exp(\sin(\frac{1}{x}))/\tan(\sqrt{1-x^4})$, this could get quite messy.

```

% =====
% *** FUNCTION simpson
% ***
% *** Implements Simpson's rule for integrating
% *** a function f over [a,b] with granularity h.
% ***
% =====
function int = simpson( f, a, b, h )

    x = a:h:b;
    mid = (x(1:n-1) + x(2:n)) / 2;

    n = length(x);

    int = sum( h/6 * ( f(x(1:n-1)) ...
                      + 4*f(mid) ...
                      + f(x(2:n)) ) );

end
% =====
% *** END FUNCTION simpson
% =====

```

Listing 6: Simpson's rule with function handles. Note that the syntax for function arguments is no different from that of ordinary ones.


```

% =====
% *** FUNCTION A_multiply
% ***
% *** Implements matrix--vector multiplication with
% *** diag[-1,2,-1]/h^2 .
% ***
% =====
function out = A_multiply( u )

    n = length( u );
    u = [0; u; 0];

    out = -u(1:n) + 2*u(2:n+1) - u(3:n+2);
    out = out * (n+1)^2;

end
% =====
% *** END FUNCTION A_multiply
% =====

```

Listing 7: Function that implements matrix–vector multiplication with $1/h^2 \times \text{diag}(-1, 2, -1)$. Note that the function consumes (almost) no more memory than u already required.

Beyond that, there may also a mild advantage in memory consumption as the indices of the matrix do no longer need to sit in memory, but can be hard coded into the matrix-vector-multiplication function itself. Considering the fact that we are mostly working with sparse matrices however, this might not be quite important.

The example below illustrates the typical benefits and drawbacks of the approach.

```
n = 1e3;
k = 500;
```

```
u = ones(n,1);
for i=1:k
    u = A_multiply( u );
end
```

Computing $u = A^k u_0$ with the function `A_multiply` (listing 7). The memory consumption of this routine is (almost) no greater than storing n real numbers. Execution time: **21 s**.

```
n = 1e3;
k = 500;
```

```
e = ones(n,1);
A = spdiags([-e,2*e,-e],...
            [-1, 0,-1],...
            n, n );
A = A * (n+1)^2;

u = ones(n,1);
for i=1:k
    u = A*u;
end
```

Computing $u = A^k u_0$ with a regular sparse format matrix `A`, with the need to store it in memory. Execution time: **7 s**.

All in all, these considerations shall not lead you to rewrite all you matrix-vector multiplications as function calls. Mind, however, that there are situations where one would *never* use matrices in their explicit form, although mathematically written down like that:

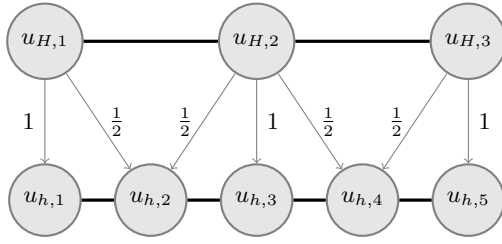
Example (Multigrid). *In geometric multigrid methods, a domain is discretized with a certain parameter h (“grid width”) and the operator A_h written down for that discretization (see the examples above, where $A_h = h^{-2} \text{diag}(-1, 2, 1)$ is really the discretization of the Δ -operator in one dimension). In a second step, another, somewhat coarser grid is considered with $H = 2h$, for example. The operator A_H on the coarser grid is written down as*

$$A_H = I_h^H A_h I_H^h,$$

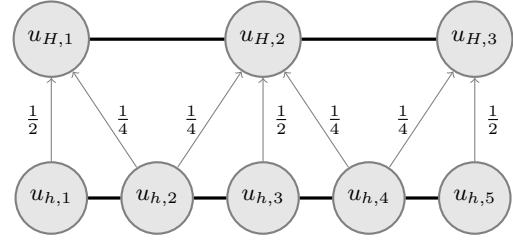
where the I_*^* operators define the transition from the coarse to the fine grid, or the other way around. When applying it to a vector on the coarse grid u_H ($A_H u_H = I_h^H A_h I_H^h u_H$), the above definition reads:

1. $I_H^h u_H$: Map u_H to the fine grid.
2. A_h : Apply the fine grid operator to the transformation.
3. I_h^H : Transform the result back to the coarse grid.

How the transformations are executed needs to be defined. One could, for example, demand that I_H^h maps all points that are part of the fine grid and the coarse grid to itself; all points on the fine grid, that lie right in between two coarse variables get half of the value of each of the two (see figure 2a).



(a) Possible transformation rule when translating values from the coarse to the fine grid. See listing 8.



(b) Mapping back from fine to coarse.

Figure 2

```
% =====
% *** FUNCTION coarse2fine
% ***
% *** Transforms values from a coarse grid to a fine grid.
% ***
% =====
function uFine = coarse2fine( uCoarse )

N = length(uCoarse);
n = 2*N - 1;

uFine(1:2:n) = uCoarse;

midValues      = 0.5 * ( uCoarse(1:N-1) + uCoarse(2:N) );
uFine(2:2:n) = midValues;

end
% =====
% *** END FUNCTION coarse2fine
% =====
```

Listing 8: Function that implements the operator I_H^h from the example (see figure 2a). Writing down the structure of the corresponding matrix would be somewhat complicated, and even more so when moving to two- or three-dimensional grids. Note also how matrix notation has been exploited.

In the analysis of the method, I_H^h and I_h^H will always be treated as matrices, but when implementing, one would certainly not try to figure out the structure of the matrix. It is a lot simpler to implement a function that executes the rule suggested above, for example.