

LAFF-On

Programming for Correctness

Margaret E. Myers

Robert A. van de Geijn

Release Date August 29, 2016

This is a work in progress

Copyright © 2016 by Margaret E. Myers and Robert A. van de Geijn.

10 9 8 7 6 5 4 3 2 1

All rights reserved. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, contact any of the authors.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Library of Congress Cataloging-in-Publication Data not yet available

Draft Edition, Fall 2016

This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

Contents

0. Getting Started	1
0.1. Opening Remarks	1
0.1.1. Welcome to LAFF-On Programming for Correctness	1
0.1.2. Outline	2
0.1.3. What You Will Learn	3
0.2. How to LAFF-On	4
0.2.1. Setting Up to LAFF	4
0.3. Software to LAFF-On	4
0.3.1. Why MATLAB	4
0.3.2. Installing MATLAB	4
0.3.3. MATLAB Basics	4
0.4. Typesetting LAFF-On	6
0.4.1. Typesetting mathematics	6
0.4.2. Downloading TeXstudio	7
0.4.3. Testing TeXstudio	7
0.4.4. L ^A T _E X and TeXstudio Primer	7
0.5. Enrichments	8
0.5.1. The Origins of MATLAB	8
0.5.2. The Origins of L ^A T _E X	8
0.6. Wrap Up	8
0.6.1. Additional Homework	8
0.6.2. Summary	8
1. Introduction	9
1.1. Opening Remarks	9
1.1.1. Launch	9
1.1.2. Outline Week 1	16
1.1.3. What you should know	17
1.1.4. What you will learn	18
1.2. A Farewell to Indices	19
1.2.1. A motivating example: dot product	19
1.2.2. A new notation for presenting algorithms	20
1.2.3. Typesetting algorithms with FLAME notation and L ^A T _E X	22

1.2.4.	Representing (FLAME) algorithms in code	23
1.3.	Correctness of a Loop	24
1.3.1.	Predicates as assertions about the state	24
1.3.2.	Verifying loops	24
1.3.3.	Example	27
1.4.	Deriving a Correct Loop	27
1.4.1.	Goal-Oriented Development of Algorithms	27
1.4.2.	Typesetting a derivation	31
1.4.3.	Representing algorithms in code	33
1.5.	Enrichment	33
1.6.	Wrap Up	33
1.6.1.	Additional Exercises	33
1.6.2.	Summary	35

Index	39
--------------	-----------

Preface

Acknowledgments

Getting Started

0.1 Opening Remarks

0.1.1 Welcome to LAFF-On Programming for Correctness

Opening video.

Learn in Section [0.2.1](#) how and where to download videos onto your computer.

0.1.2 Outline

Following the “opener” we give the outline for the week:

0.1. Opening Remarks	1
0.1.1. Welcome to LAFF-On Programming for Correctness	1
0.1.2. Outline	2
0.1.3. What You Will Learn	3
0.2. How to LAFF-On	4
0.2.1. Setting Up to LAFF	4
0.3. Software to LAFF-On	4
0.3.1. Why MATLAB	4
0.3.2. Installing MATLAB	4
0.3.3. MATLAB Basics	4
0.4. Typesetting LAFF-On	6
0.4.1. Typesetting mathematics	6
0.4.2. Downloading TeXstudio	7
0.4.3. Testing TeXstudio	7
0.4.4. \LaTeX and TeXstudio Primer	7
0.5. Enrichments	8
0.5.1. The Origins of MATLAB	8
0.5.2. The Origins of \LaTeX	8
0.6. Wrap Up	8
0.6.1. Additional Homework	8
0.6.2. Summary	8

0.1.3 What You Will Learn

The third unit of the week informs you of what you will learn. This describes the knowledge and skills that you can expect to acquire. In addition, this provides an opportunity for you to self-assess upon completion of the week.


Upon completion of this week, you should be able to

- Navigate the different components of LAFF-On.
- Download and start MATLAB.
- Recognize the structure of a typical week.

You may want to print out Appendix ?? to track your progress throughout the course.

0.2 How to LAFF-On


0.2.1 Setting Up to LAFF

It helps if we all set up our environment in a consistent fashion. If you obtained this file from the github repository, it is already in a directory, LAFF-On, with a hierarchy similar to that illustrated in Figure 1. You should use  **Acrobat Reader** when viewing this document so that hyperlinks work properly.

0.3 Software to LAFF-On

0.3.1 Why MATLAB


We use **MATLAB** as a tool because it was invented to support learning about matrix computations. You will find that the syntax of the language used by MATLAB, called M-script, very closely resembles the mathematical expressions in linear algebra.

Those not willing to invest in MATLAB will want to consider  **GNU Octave** instead.

0.3.2 Installing MATLAB

Instructions can be found at  <http://www.mathworks.org>.

0.3.3 MATLAB Basics

Below you find a few short videos that introduce you to MATLAB. For a more comprehensive tutorial, you may want to visit  **MATLAB Tutorials** at MathWorks and clicking “Launch Tutorial”.

HOWEVER, you need very little familiarity with MATLAB in order to learn what we want you to learn about how abstraction in mathematics is linked to abstraction in algorithms. So, you could just skip these tutorials altogether, and come back to them if you find you want to know more about MATLAB and its programming language (M-script).

What is MATLAB?



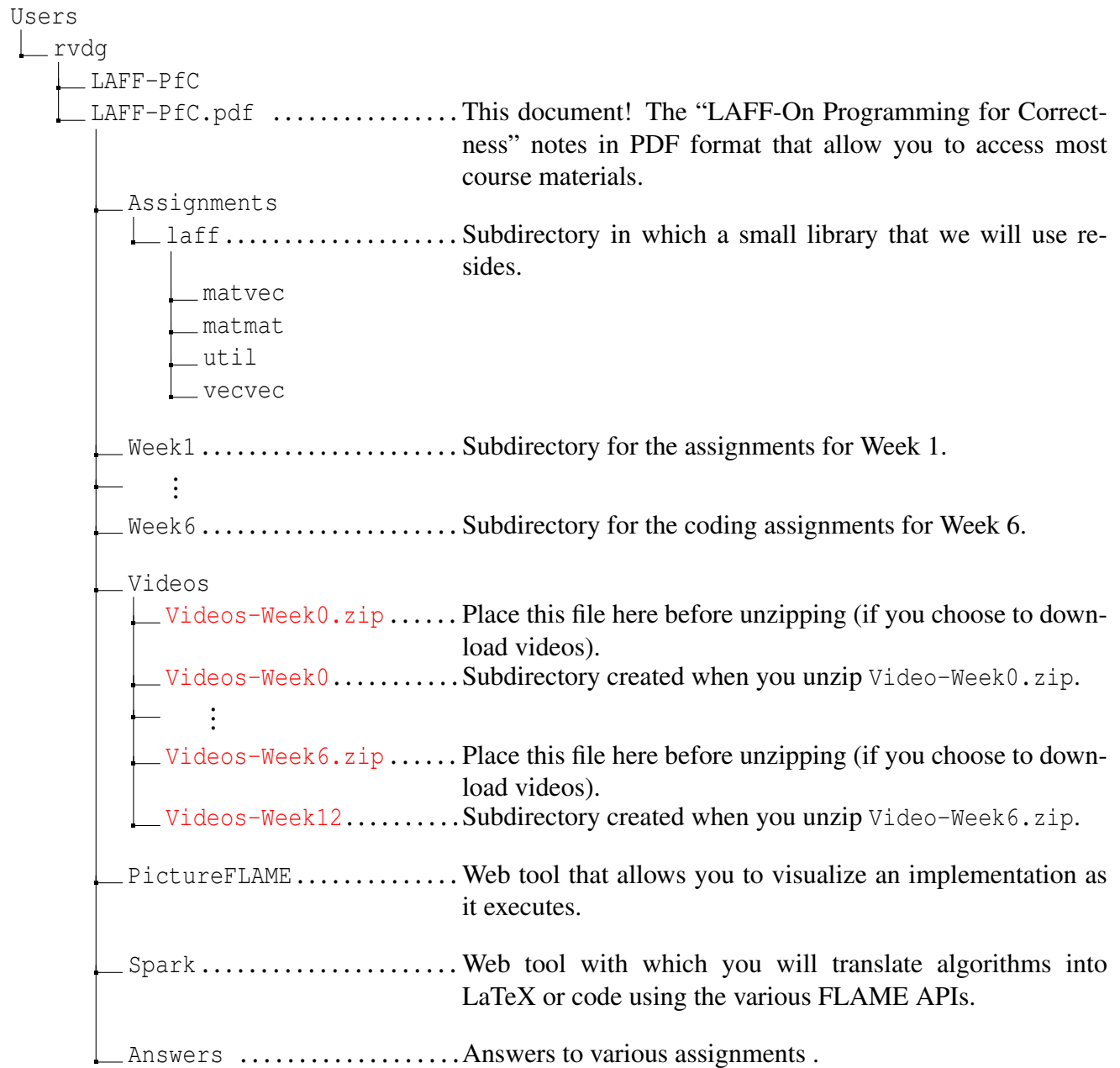


Figure 1: Directory structure for your LAFF-PfC materials. Items in **red** will be placed into the materials by you. In this example, we placed LAFF-PfC.zip in the home directory Users/rvdg before unzipping. You may want to place it on your account’s “Desktop” instead.

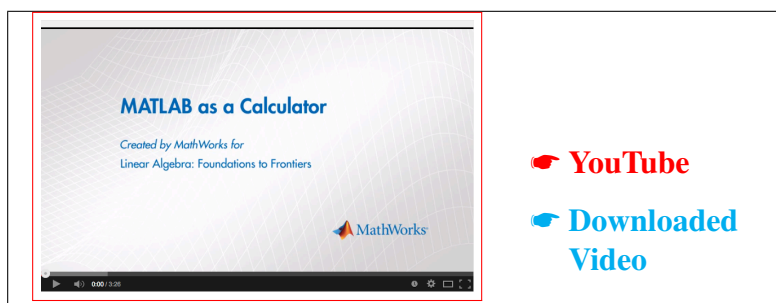
The MATLAB Environment



MATLAB Variables



MATLAB as a Calculator



0.4 Typesetting L^AT_EX-On

0.4.1 Typesetting mathematics

The tool of choice for typesetting mathematics is a document typesetting system called \LaTeX . We will use it to typeset many of our exercises in a way that captures how we want you to think about discovering algorithms hand-in-hand with their proofs of correctness.

We recommend that you use TeXstudio:

”TeXstudio an integrated writing environment for creating LaTeX documents.”

It is the environment I typically use in the videos for this course and that I used to create the notes and many of the activities. You are, of course, free to use whatever such environment you prefer.

0.4.2 Downloading TeXstudio

You can download TeXstudio from www.texstudio.org.

0.4.3 Testing TeXstudio

- Open the file `LAFF-PfC/assignments/Week0/HelloWorld.tex`.
 - I would recommend opening it with your "Finder" (or whatever it is called in Windows or Linux) so that you can make the default application for ".tex" files the TeXstudio application.
 - Alternatively, you can open the TeXstudio application and then click on

File -> Open

choosing the file.

- At this point, it is a good idea to click on


Options -> Root Document -> Set Current Document as Explicit Root

which will make the "HelloWorld.tex" file the root file for the "compilation" of the document. This is important when the root file itself includes other files in some hierarchical fashion.

- If you set up TeXstudio as the default application for ".tex" files, then clicking on

 [HelloWorld.tex](#)

should open TeXstudio with that file. This will allow you to similarly open future assignments that use TeXstudio by clicking on the appropriate link.

- Click on . Eventually you will see a "Process exited normally" in the message box in the lower left-hand corner as well as the formatted text to the right.

0.4.4 L^AT_EX and TeXstudio Primer

Insert video on L^AT_EX and TeXstudio.


The source for the L^AT_EX example in that video:

 [LaTeXPrimer.tex](#)

We will use L^AT_EX in a very limited way. But you may want to become more familiar with this tool by following some tutorials that you can find on the internet. (Search "LaTeX tutorial").

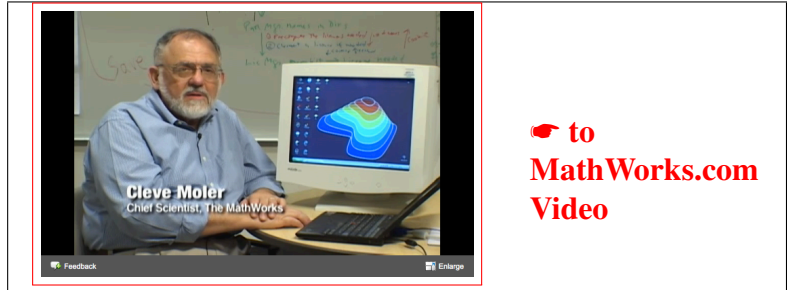
The following is probably a pretty good place to start:

[L^AT_EX Tutorials - a Primer.](#)

Whenever it talks about `latex myfile` and `xdvi myfile` instead just click on .

0.5 Enrichments

0.5.1 The Origins of MATLAB



0.5.2 The Origins of \LaTeX

You may enjoy this interview with Dr. Leslie Lamport, who created \LaTeX .

<https://www.infoq.com/interviews/lamport-latex-paxos-tla>

0.6 Wrap Up

0.6.1 Additional Homework

For a typical week, additional assignments may be given in this unit.

0.6.2 Summary

You will see that we develop a lot of the theory behind the various topics in linear algebra via a sequence of homework exercises. At the end of each week, we summarize theorems and insights for easy reference.

Part 1

Introduction

1.1 Opening Remarks

1.1.1 Launch

Homework 1.1.1.1 Compute

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} =$$

[SEE ANSWER](#)

Given $m \times n$ matrix A , vector x of size n , and vector y of size m , we expose their elements as

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}, x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix}, \text{ and } y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix}$$

then the matrix-vector multiplication operation $y := Ax + y$ (y is overwritten with $Ax + y$ or “ y becomes A times x plus y ”) is computed as

$$\begin{aligned} \psi_0 &:= \alpha_{0,0} \chi_0 + \alpha_{0,1} \chi_1 + \cdots + \alpha_{0,n-1} \chi_{n-1} \\ \psi_1 &:= \alpha_{1,0} \chi_0 + \alpha_{1,1} \chi_1 + \cdots + \alpha_{1,n-1} \chi_{n-1} \\ &\vdots \\ \psi_{m-1} &:= \alpha_{m-1,0} \chi_0 + \alpha_{m-1,1} \chi_1 + \cdots + \alpha_{m-1,n-1} \chi_{n-1} \end{aligned}$$

This computation translates into the MATLAB function `MatVec1` given in Figure 1.1.

```

function [ y_out ] = MatVec1( A, x, y )
% Compute y := A x + y

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that x is a vector size n and y is a
% vector of size m...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:m
    for j=1:n
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
end
end

```

Assignments/Week1/matlab/MatVec1.m

Figure 1.1: Function that computes $y; = Ax + y$, returning the result in vector y_out .

Homework 1.1.1.2 Open the Live Script

 [Assignments/Week1/matlab/MatVec1LS.mlx](#)

and follow the directions in it.

 [SEE ANSWER](#)

Notice that we tend to start indexing of elements in arrays at zero:

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}$$

while MATLAB start at one: our $\alpha_{0,0}$ is stored in $A(1, 1)$. Better get used to this!

With few exceptions, we will use upper case Roman letters to denote matrices, lower case Roman letters to denote vectors, and lower case Greek letters to denote scalars.

Consider an $m \times n$ matrix A . We will often partition such a matrix into its columns

$$A = \left(a_0 \mid a_1 \mid \cdots \mid a_{n-1} \right).$$

Since each column is a vector, we use a lower case Roman letter to denote it. To further link the columns to the original matrix A , we use the lower case letter a .

Sometimes, we partition a matrix, A , into rows

$$A = \begin{pmatrix} \frac{a_0^T}{\hline} \\ \frac{a_1^T}{\hline} \\ \vdots \\ \frac{a_{m-1}^T}{\hline} \end{pmatrix}.$$

Each row is a row vector, which we denote with the lower case Roman letter a . But we view vectors as column vectors, which is why we add the T to indicate it is a vector that has been transposed to become the row.

When we expose individual entries of a matrix, for its scalar entries we typically use the lower case Greek letter that is similar to the letter used to denote the matrix.

Now, if $m = n$ then matrix A is square and if $\alpha_{i,j} = \alpha_{j,i}$ then it is said to be a symmetric matrix. In this case

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,1} & \cdots & \alpha_{n-1,n-1} \end{pmatrix} = \begin{pmatrix} \alpha_{0,0} & \alpha_{1,0} & \cdots & \alpha_{n-1,0} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{n-1,1} \\ \vdots & \vdots & & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,1} & \cdots & \alpha_{n-1,n-1} \end{pmatrix} = \begin{pmatrix} \alpha_{0,0} & \star & \cdots & \star \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \star \\ \vdots & \vdots & & \vdots \\ \alpha_{n-1,0} & \alpha_{n-1,1} & \cdots & \alpha_{n-1,n-1} \end{pmatrix}.$$

where the \star s are meant to indicate that the upper triangular part of the matrix needs not be stored.

Homework 1.1.1.3 Knowing that the matrix is symmetric, compute

$$\begin{pmatrix} 1 & \star & \star \\ -2 & 2 & \star \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} =$$

[SEE ANSWER](#)

Homework 1.1.1.4 Copy the file `MatVec1.m` to file `SymMatVec1.m`, change the name of the function to `SymMatVec1` change the implementation so that it only accesses entries of array `A` that are on or below the diagonal. Test your implementation with the Live Script in

[Assignments/Week1/matlab/SymMatVec1LS.mlx](#)

[SEE ANSWER](#)

```

function [ y_out ] = SymMatVec1( A, x, y )
% Compute y := A x + y, assuming A is symmetric and stored in lower
% triangular part of array A.

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that m = n, x is a vector size n and y is a
% vector of size n...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for i = 1:n
    for j=1:i
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
    for j=i+1:n
        y_out( i ) = A( j,i ) * x( j ) + y_out( i );
    end
end
end

```

Assignments/Week1/matlab/SymMatVec1.m

Figure 1.2: Function that computes $y := Ax + y$, returning the result in vector `y_out`. Matrix A is assumed to be symmetric and only stored in the lower triangular part of array `A`.

.

Now, MATLAB stores matrices in *column-major* order, which means that a matrix

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix}$$

```

function [ y_out ] = MatVec2( A, x, y )
% Compute y := A x + y

% Extract the row and column size of A
[ m, n ] = size( A );

% (Strictly speaking you should check that x is a vector size n and y is a
% vector of size m...)

% Copy y into y_out
y_out = y;

% Compute y_out = A * x + y_out
for j = 1:n
    for i=1:m
        y_out( i ) = A( i,j ) * x( j ) + y_out( i );
    end
end
end

```

Assignments/Week1/matlab/MatVec2.m

Figure 1.3: Function that computes $y; = Ax + y$, returning the result in vector y_out .

is stored in memory by stacking columns:

1
-2
-1
-1
2
1
2
0
-2

Computation tends to be more efficient if one accesses memory contiguously. This means that an algorithm that accesses A by columns often computes the answer faster than one that accesses A by columns.

In a linear algebra course you should have learned that,

$$\begin{pmatrix} 1 & -1 & 2 \\ -2 & 2 & 0 \\ -1 & 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} = (2) \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} + (-1) \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix} + (1) \begin{pmatrix} 2 \\ 0 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}$$

$$= \left((1) \begin{pmatrix} 2 \\ 0 \\ -2 \end{pmatrix} + (-1) \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix} + (2) \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} + \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix} \right),$$

which means that the implementation in Figure 1.1 can be rewritten as the one in Figure 1.3, which accesses the matrix by columns. Notice that the two implementations differ only in the order of the loops indexed by i and j .

Homework 1.1.1.5 Copy the file `MatVec2.m` to file `SymMatVec2.m`, change the name of the function to `SymMatVec2` change the implementation so that it only accesses entries of array `A` that are on or below the diagonal. Test your implementation with the Live Script in

 [Assignments/Week1/matlab/SymMatVec2LS.mlx](#)

 [SEE ANSWER](#)

Now we get to two exercises that help motivate the course. They are surprisingly hard, even for an expert:

Homework 1.1.1.6 Modify the code in Figure 1.2 so that it only accesses the lower triangular part of array `A` by columns, resulting in routine `SymMatVec3`. Test your implementation with the Live Script in

 [Assignments/Week1/matlab/SymMatVec3LS.mlx](#)

 [SEE ANSWER](#)

Homework 1.1.1.7 Find someone who knows a little (or a lot) about linear algebra and convince this person that the answer to the last exercise is correct. Alternatively, if you did not managed to come up with an answer for the last exercise, look at the answer to that exercise and convince yourself it is correct.

 [SEE ANSWER](#)

The point of these last two exercises is:

- It is difficult to find algorithms with specific (performance) properties even for relatively simple operations.
- It is difficult to give a convincing argument that even a relatively simple algorithm is correct.

One could ask "But isn't having any algorithm to compute the result good enough?" The graph in Figure 1.4 illustrates the difference in performance of the different implementations (coded in C). The implementation that corresponds to `SymMatVec3` is roughly five times faster than the other implementations. It demonstrates there is a definite performance gain that results from picking the right algorithm.

In this course, we hope to equip you with the tools to systematically develop whole families of algorithms that are correct by construction. From this family of algorithms the one that has a desirable property (e.g., that it accesses memory contiguously so that better performance can be achieved) can then be chosen.

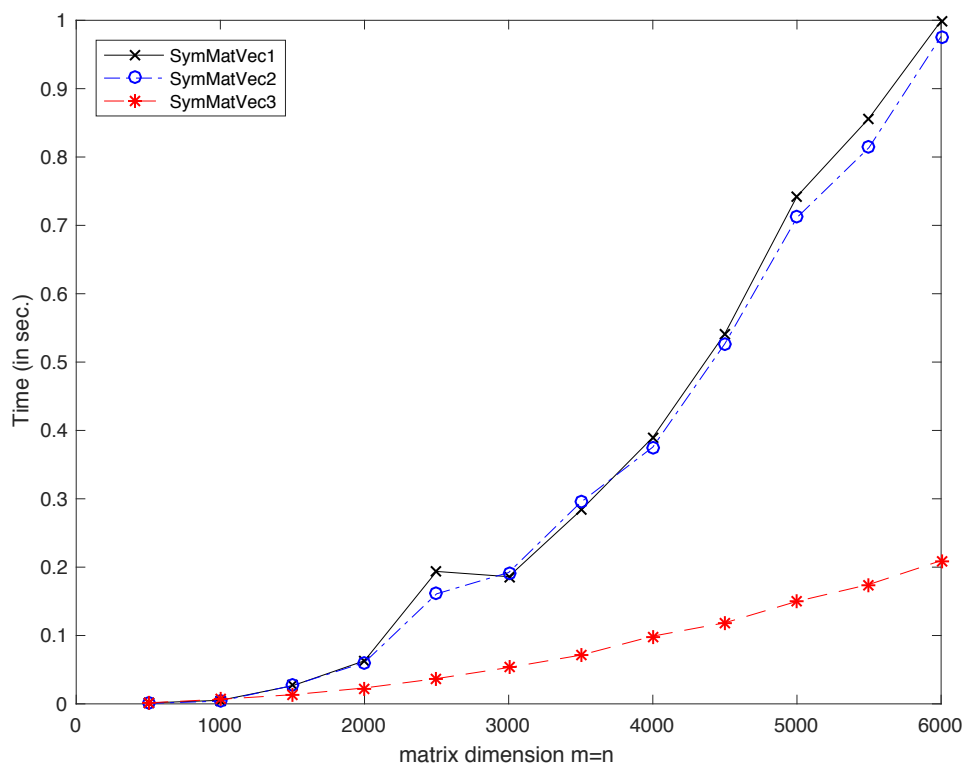


Figure 1.4: Execution time as a function of matrix size for the different implementations of symmetric matrix-vector multiplication.

1.1.2 Outline Week 1

1.1. Opening Remarks	9
1.1.1. Launch	9
1.1.2. Outline Week 1	16
1.1.3. What you should know	17
1.1.4. What you will learn	18
1.2. A Farewell to Indices	19
1.2.1. A motivating example: dot product	19
1.2.2. A new notation for presenting algorithms	20
1.2.3. Typesetting algorithms with FLAME notation and \LaTeX	22
1.2.4. Representing (FLAME) algorithms in code	23
1.3. Correctness of a Loop	24
1.3.1. Predicates as assertions about the state	24
1.3.2. Verifying loops	24
1.3.3. Example	27
1.4. Deriving a Correct Loop	27
1.4.1. Goal-Oriented Development of Algorithms	27
1.4.2. Typesetting a derivation	31
1.4.3. Representing algorithms in code	33
1.5. Enrichment	33
1.6. Wrap Up	33
1.6.1. Additional Exercises	33
1.6.2. Summary	35

1.1.3 What you should know

1.1.4 What you will learn

This week introduces the reader to the systematic derivation of algorithms for linear algebra operations. Through a very simple example we illustrate the core ideas: We describe the notation we will use to express algorithms; we show how assertions can be used to establish correctness; and we propose a goal-oriented methodology for the derivation of algorithms. We also discuss how to incorporate an analysis of the cost into the algorithm. Finally, we show how to translate algorithms to code so that the correctness of the algorithm implies the correctness of the implementation.

Upon completion of this week, you should be able to

-

Track your progress in Appendix ??.

1.2 A Farewell to Indices

1.2.1 A motivating example: dot product

In this section, we introduce a notation for expressing algorithms that avoids the pitfalls of intricate indexing and will allow us later to more easily derive, express, and implement algorithms. We present the notation through a simple example, the *inner product* of two vectors, an operation that will be used throughout this chapter for illustration.

Given two vectors, x and y , of length m , the *inner product* (or *dot product*) of these vectors is given by

$$\alpha := x^T y = \sum_{i=0}^{m-1} \chi_i \psi_i$$

where χ_i and ψ_i equal the i th elements of x and y , respectively:

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix}.$$

We will use the symbol “:=” (“*becomes*”) to denote *assignment* while the symbol “=” is reserved for *equality*.

Example 1.1 *Let*

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}.$$

Then $x^T y = 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 1 = 13$. Here we make use of the symbol “ \cdot ” to denote the arithmetic product.

A traditional loop for implementing the updating of a scalar by adding an inner product to it, $\alpha := x^T y + \alpha$, is given by

```

k := 0
while k < m do
  α := χkψk + α
  k := k + 1
endwhile

```

Some will find this loop a bit cumbersome, since the algorithm can be more easily expressed with “for” loop. Why we use a while loop will become clear later.

Algorithm: $\alpha := \text{SAPDOT_VAR1}(x, y, \alpha)$
$x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array} \right), y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array} \right)$ <p>where x_T and y_T have 0 elements</p> <p>while $m(x_T) < m(x)$ do</p> $\left(\begin{array}{c} x_T \\ \hline x_B \end{array} \right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array} \right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array} \right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array} \right)$ <p>where χ_1 and ψ_1 are scalars</p>
$\alpha := \chi_1 \psi_1 + \alpha$
$\left(\begin{array}{c} x_T \\ \hline x_B \end{array} \right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array} \right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array} \right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array} \right)$ <p>endwhile</p>

Figure 1.5: Algorithm for computing $\alpha := x^T y + \alpha$.

1.2.2 A new notation for presenting algorithms

in Figure 1.5 we present a loop that updates with a dot product with an alternate notation, which we call the *FLAME notation*. The name of the algorithm in that figure reflects that it performs an *alpha plus dot product* (APDOT). To interpret the algorithm in Figure 1.5 note the following:

- We bid farewell to intricate indexing: In this example only indices from the sets $\{T, B\}$ (*Top* and *Bottom*) and $\{0, 1, 2\}$ are required.
- Each vector has been subdivided into two subvectors, separated by thick lines. This is how we will represent systematic movement through vectors (and later matrices).
- Subvectors x_T and y_T include the “top” elements of x and y that, in this algorithm, have already been used to compute a partial update to α . Similarly, subvectors x_B and y_B include the “bottom” elements of x and y that, in this algorithm, have not yet been used to update α . Referring back to the traditional loop, x_T and y_T consist of elements $0, \dots, k-1$ and x_B and y_B consist of elements

$k, \dots, m-1$:

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} = \begin{pmatrix} \chi_0 \\ \vdots \\ \chi_{k-1} \\ \chi_k \\ \vdots \\ \chi_{m-1} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{k-1} \\ \psi_k \\ \vdots \\ \psi_{m-1} \end{pmatrix}.$$

- The initialization before the loop starts

$$x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix}, y \rightarrow \begin{pmatrix} y_T \\ y_B \end{pmatrix}$$

takes the place of the assignment $k := 0$ in the traditional loop.

- The loop is executed as long as $m(x_T) < m(x)$ is *true*, which takes the place of $k < m$ in the traditional loop. Here $m(x)$ equals the length of vector x so that the loop terminates when x_T includes all elements of x .
- The statement

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \rightarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \rightarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$$

exposes the top elements of x_B and y_B , χ_1 and ψ_1 respectively, which were χ_k and ψ_k in the traditional loop.

- The exposed elements χ_1 and ψ_1 are used to update α in

$$\alpha := \chi_1 \psi_1 + \alpha,$$

which takes the place of the update $\alpha := \chi_k \psi_k + \alpha$ in the traditional loop.

It is important not to confuse the single elements exposed in our repartitionings, such as χ_1 or ψ_1 , with the second entries of corresponding vectors.

- The statement

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$$

Algorithm: $\alpha := \text{SAPDOT_VAR2}(x, y, \alpha)$

$$x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix}, y \rightarrow \begin{pmatrix} y_T \\ y_B \end{pmatrix}$$

where x_T and y_T have 0 elements

while $m(x_T) < m(x)$ do

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \rightarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \rightarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$$

where χ_1 and ψ_1 are scalars

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$$

endwhile

Figure 1.6: Partial algorithm for computing $\alpha := x^T y + \alpha$.

moves the top elements of x_B and y_B to x_T and y_T , respectively. This means that these elements have now been used to update α and should therefore be added to x_T and y_T .

Homework 1.2.2.8 Consider the following loop, which computes $\alpha := x^T y + \alpha$ by traversing the vectors backwards:

```

k := m - 1
while k ≥ 0 do
    α := χkψk + α
    k := k - 1
endwhile

```

Complete the algorithm in Figure 1.6 so that it expresses this alternative algorithmic variant.

👉 SEE ANSWER

1.2.3 Typesetting algorithms with FLAME notation and \LaTeX

One drawback of how we typeset our algorithms with the FLAME notation is that it takes considerably more room and time to write them down. To overcome this, we suggest using \LaTeX . In Week 0 you presumably installed a tool, TeXstudio, for writing \LaTeX documents.

Homework 1.2.3.9 Follow the instructions in the following video to duplicate Figure 1.5.

Useful links:

- A partial \LaTeX document can be found in

[Assignments/Week1/LaTeX/Sapdot_unb_var1.tex](#).

- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).

 [SEE ANSWER](#)

Homework 1.2.3.10 Typeset the algorithm from Homework 1.2.2.8.

Useful links:

- A partial \LaTeX document can be found in

[Assignments/Week1/LaTeX/Sapdot_unb_var2.tex](#).

- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).

Make sure you "march" through the vectors in the correct direction.

 [SEE ANSWER](#)

1.2.4 Representing (FLAME) algorithms in code

We now have a new notation for expressing algorithms. What we will find out later in the course is how to derive correct algorithms using this notation. The problem is that a correct algorithm still has to be translated into code. Programming bugs can be easily introduced as part of that translation. To overcome this, we will use an Application Programming Interface (API) that allows the code to closely resemble the algorithm.

Homework 1.2.4.11 Follow the instructions in the following video and the Live Script in

[Assignments/Week1/C/PfC_sapdot_unb_var1_LS.mlx](#)

to translate the algorithm in Figure 1.5 into code using the FLAME@lab API.

Useful links:

- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).

 [SEE ANSWER](#)

Homework 1.2.4.12 Translate the algorithm from Homework 1.2.2.8 into code.

Useful links:

- [Spark/index.html](#)
- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).
- Live Script to test your implementation:

[Assignments/Week1/C/PfC_Sapdot_unb_var2_LS.mlx](#)

Make sure you "march" through the vectors in the correct order.

 [SEE ANSWER](#)

1.3 Correctness of a Loop

1.3.1 Predicates as assertions about the state

To reason about the correctness of algorithms, predicates will be used to express assertions about the state (contents) of the variables. Recall that a *predicate*, P , is simply a *(Boolean) expression* that evaluates to *true* or *false* depending on the state of the variables that appear in the predicate. The placement of a predicate at a specific point in an algorithm means that it must evaluate to *true* so that it asserts the state of the variables that appear in the predicate. An *assertion* is a predicate that is used in this manner.

For example, after the command

$$\alpha := 1$$

which assigns the value 1 to the scalar variable α , we can assert that the predicate " $P : \alpha = 1$ " holds (is true). An assertion can then be used to indicate the state of variable α after the assignment as in

$$\begin{array}{l} \alpha := 1 \\ \{P : \alpha = 1\} \end{array}$$

Assertions will be enclosed by curly brackets in the algorithms.

If P_{pre} and P_{post} are predicates and S is a sequence of commands, then $\{P_{pre}\}S\{P_{post}\}$ is a predicate that evaluates to true if and only if the execution of S , when begun in a state satisfying P_{pre} , terminates in a finite amount of time in a state satisfying P_{post} . Here $\{P_{pre}\}S\{P_{post}\}$ is called the *Hoare triple*, and P_{pre} and P_{post} are referred to as the *precondition* and *postcondition* for the triple, respectively.

Example 1.2 $\{\alpha = \beta\} \alpha := \alpha + 1 \{\alpha = \beta + 1\}$ evaluates to true. Here " $\alpha = \beta$ " is the precondition while " $\alpha = (\beta + 1)$ " is the postcondition.

1.3.2 Verifying loops

Consider the loop in Figure 1.5, which has the form

```

while  $G$  do
   $S$ 
endwhile

```

Here, G is a Boolean expression known as the *loop-guard* and S is the sequence of commands that form the *loop-body*. The loop is executed as follows: If G is false, then execution of the loop terminates; otherwise S is executed and the process is repeated. Each execution of S is called an *iteration*. Thus, if G is initially false, no iterations occur.

We now formulate a theorem that can be applied to prove correctness of algorithms consisting of loops. For a proof of this theorem (using slightly different notation), see [?].

Theorem 1.3 (Fundamental Invariance Theorem) *Given the loop*

```

while  $G$  do  $S$  endwhile

```

and a predicate P_{inv} assume that

1. $\{P_{inv} \wedge G\}S\{P_{inv}\}$ holds – execution of S begun in a state in which P_{inv} and G are true terminates with P_{inv} true – , and
2. execution of the loop begun in a state in which P_{inv} is true terminates.

Then, if the loop is entered in a state where P_{inv} is true, it will complete in a state where P_{inv} is true and the loop-guard G is false.

Here the symbol “ \wedge ” denotes the *logical and* operator.

This theorem can be interpreted as follows. Assume that the predicate P_{inv} holds before and after the loop-body. Then, if P_{inv} holds before the loop, obviously it will also hold before the loop-body. The commands in the loop-body are such that it holds again after the loop-body, which means that it will again be true before the loop-body in the next iteration. We conclude that it will be true before and after the loop-body every time through the loop. When G becomes false, P_{inv} will still be true, and therefore it will be true after the loop completes (if the loop can be shown to terminate), we can assert that $P_{inv} \wedge \neg G$ holds after the completion of the loop, where the symbol “ \neg ” denotes the *logical negation*. This can be summarized by

$$\begin{array}{ccc}
 \{P_{inv} \wedge G\} & & \{P_{inv}\} \\
 S & \Rightarrow & \textbf{while } G \textbf{ do} \\
 \{P_{inv}\} & & \{P_{inv} \wedge G\} \\
 & & S \\
 & & \{P_{inv}\} \\
 & & \textbf{endwhile} \\
 & & \{P_{inv} \wedge \neg G\}
 \end{array}$$

if the loop can be shown to terminate. The assertion P_{inv} is called the *loop-invariant* for this loop.

Step	Algorithm: $\alpha := \text{SAPDOT}(x, y, \alpha)$
1a	$\alpha = \hat{\alpha} \wedge 0 \leq m(x) = m(y)$
4	$x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix}, y \rightarrow \begin{pmatrix} y_T \\ y_B \end{pmatrix}$ where x_T and y_T have 0 elements
2	$(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))$
3	while $m(x_T) < m(x)$ do
2,3	$(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)) \wedge m(x_T) < m(x)$
5a	$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \rightarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \rightarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$ where χ_1 and ψ_1 are scalars
6	$(\alpha = x_0^T y_0 + \hat{\alpha}) \wedge (0 \leq m(x_0) = m(y_0) < m(x))$
8	$\alpha := \chi_1 \psi_1 + \alpha$
5b	$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \begin{pmatrix} y_T \\ y_B \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix}$
7	$(\alpha = x_0^T y_0 + \chi_1 \psi_1 + \hat{\alpha}) \wedge (0 < m(x_0) + 1 = m(y_0) + 1 \leq m(x))$
2	$(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))$
	endwhile
2,3	$(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)) \wedge \neg(m(x_T) < m(x))$
1b	$\alpha = x^T y + \hat{\alpha}$

Figure 1.7: Annotated algorithm for computing $\alpha := x^T y + \alpha$.

1.3.3 Example

Let us again consider the computation $\alpha := x^T y + \alpha$. Let us use $\hat{\alpha}$ to denote the *original contents* (or state) of α . Then we define the precondition for the algorithm as

$$P_{pre} : \alpha = \hat{\alpha} \wedge 0 \leq m(x) = m(y),$$

and the postcondition as

$$P_{post} : \alpha = x^T y + \hat{\alpha};$$

that is, the result the operation to be computed.

In order to prove the correctness, we next annotate the algorithm in Figure 1.5 with assertions that describe the state of the variables, as shown in Figure 1.7. Each command in the algorithm has the property that, when entered in a state described by the assertion that precedes it, it will complete in a state where the assertion immediately after it holds. In that *annotated algorithm*, the predicate

$$P_{inv} : (\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))$$

is a loop-invariant as it holds

1. immediately before the loop (by the initialization and the definition of $x_T^T y_T = 0$ when $m(x_T) = m(y_T) = 0$),
2. before the loop-body, and
3. after the loop-body.

Now, it is also easy to argue that the loop terminates so that, by the Fundamental Invariance Theorem, $\{P_{inv} \wedge \neg G\}$ holds after termination. Therefore,

$$\begin{aligned}
 & P_{inv} \wedge \neg G \\
 \equiv & \underbrace{(\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))}_{P_{inv}} \wedge \underbrace{\neg(m(x_T) < m(x))}_{\neg G} \\
 \Rightarrow & (\alpha = x_T^T y_T + \hat{\alpha}) \wedge \underbrace{(0 \leq m(x_T) = m(y_T) \leq m(x)) \wedge (m(x_T) \geq m(x))}_{\Rightarrow} \\
 & \qquad \qquad \qquad m(x_T) = m(y_T) = m(x) \\
 \Rightarrow & (\alpha = x^T y + \hat{\alpha}),
 \end{aligned}$$

since x_T and y_T are subvectors of x and y and therefore $m(x_T) = m(y_T) = m(x)$ implies that $x_T = x$ and $y_T = y$. Thus we can claim that the algorithm correctly computes $\alpha := x^T y + \alpha$.

1.4 Deriving a Correct Loop

1.4.1 Goal-Oriented Development of Algorithms

While in the previous sections we discussed how to add annotations to an existing algorithm in an effort to prove it correct, we now demonstrate how one can methodically and constructively *derive* correct algorithms. *Goal-oriented* derivation of algorithms starts with the specification of the operation for which an

algorithm is to be developed. From the specification, assertions are systematically determined and inserted into the algorithm before commands are added. By then inserting commands that make the assertions true at the indicated points, the algorithm is developed, hand-in-hand with its proof of correctness.

We draw the attention of the reader again to Figure 1.7. The numbers in the left column, labeled **Step**, indicate in what order to fill out the annotated algorithm.

Step 1: Specifying the precondition and postcondition. The statement of the operation to be performed, $\alpha := x^T y + \alpha$, dictates the precondition and postcondition indicated in Steps 1a and 1b. The precondition is given by

$$P_{pre} : \alpha = \hat{\alpha} \wedge 0 \leq m(x) = m(y),$$

and the postcondition is

$$P_{post} : \alpha = x^T y + \hat{\alpha}.$$

Step 2: Determining loop-invariants. As part of the computation of $\alpha := x^T y + \alpha$ we will sweep through vectors x and y in a way that creates two different subvectors of each of those vectors: the parts that have already been used to update α and the parts that remain yet to be used in this update. This suggests a partitioning of x and y as

$$x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix} \quad \text{and} \quad y \rightarrow \begin{pmatrix} y_T \\ y_B \end{pmatrix},$$

where $m(x_T) = m(y_T)$ since otherwise $x_T^T y_T$ is not well-defined.

We take these partitioned vectors and substitute them into the postcondition to find that

$$\alpha = \begin{pmatrix} x_T \\ x_B \end{pmatrix}^T \begin{pmatrix} y_T \\ y_B \end{pmatrix} + \hat{\alpha} = \left(x_T^T \mid x_B^T \right) \begin{pmatrix} y_T \\ y_B \end{pmatrix} + \hat{\alpha},$$

or,

$$\alpha = x_T^T y_T + x_B^T y_B + \hat{\alpha}.$$

This *partitioned matrix expression (PME)* expresses the *final* value of α in terms of its original value and the partitioned vectors.

The partitioned matrix expression (PME) is obtained by substitution of the partitioned operands into the postcondition.

Now, at an intermediate iteration of the loop, α does not contain its final value. Rather, it contains some *partial* result towards that final result. This partial result should be reflected in the loop-invariant. One such intermediate state is given by

$$P_{inv} : (\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)),$$

which we note is exactly the loop-invariant that we used to prove the algorithm correct in Figure 1.7.

Once it is decided how to partition vectors and matrices into regions that have been updated and/or used in a consistent fashion, loop-invariants can be systematically determined *a priori*.

Step 3: Choosing a loop-guard. The condition

$$P_{inv} \wedge \neg G \equiv ((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x))) \wedge \neg G$$

must imply that “ $P_{post} : \alpha = x^T y + \hat{\alpha}$ ” holds. If x_T and y_T equal all of x and y , respectively, then the loop-invariant implies the postcondition: The choice ‘ $G : m(x_T) < m(x)$ ’ satisfies the desired condition that $P_{inv} \wedge \neg G$ implies that $m(x_T) = m(x)$, as x_T must be a subvector of x , and

$$\underbrace{((\alpha = x_T^T y_T + \hat{\alpha}) \wedge (0 \leq m(x_T) = m(y_T) \leq m(x)))}_{P_{inv}} \wedge \underbrace{(m(x_T) \geq m(x))}_{\neg G} \\ \Rightarrow \alpha = x^T y + \hat{\alpha},$$

as was already argued in the previous section. This loop-guard is entered in Step 3 in Figure 1.7.

The chosen loop-invariant and the postcondition together *prescribe* a (non-unique) loop-guard G .

Step 4: Initialization. If we partition

$$x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix} \quad \text{and} \quad y \rightarrow \begin{pmatrix} y_T \\ y_B \end{pmatrix},$$

where x_T and y_T have *no* elements, then we place variables α , x_T , x_B , y_T , and y_B in a state where the loop-invariant is satisfied. This initialization appears in Step 4 in Figure 1.7.

The chosen loop-invariant and the precondition together *prescribe* the initialization.

Step 5: Progressing through the vectors. We now note that, as part of the computation, x_T and y_T start by containing no elements and must ultimately equal all of x and y , respectively. Thus, as part of the loop, elements must be taken from x_B and y_B and must be added to x_T and y_T , respectively. This is denoted in Figure 1.7 by the statements

Repartition

$$\begin{pmatrix} x_T \\ x_B \end{pmatrix} \rightarrow \begin{pmatrix} x_0 \\ \chi_1 \\ x_2 \end{pmatrix}, \quad \begin{pmatrix} y_T \\ y_B \end{pmatrix} \rightarrow \begin{pmatrix} y_0 \\ \psi_1 \\ y_2 \end{pmatrix} \\ \text{where } \chi_1 \text{ and } \psi_1 \text{ are scalars,}$$

and

Continue with

$$\left(\frac{x_T}{x_B} \right) \leftarrow \left(\frac{\frac{x_0}{\chi_1}}{x_2} \right), \quad \left(\frac{y_T}{y_B} \right) \leftarrow \left(\frac{\frac{y_0}{\psi_1}}{y_2} \right).$$

This notation simply captures the movement of χ_1 , the top element of x_B , from x_B to x_T . Similarly ψ_1 moves from y_B to y_T . The movement through the vectors guarantees that the loop eventually terminates, which is one condition required for the Fundamental Invariance Theorem to apply.

The initialization and the loop-guard together prescribe the movement through the vectors.

Step 6: Determining the state after repartitioning. The repartitionings in Step 5a do not change the contents of α : it is an “indexing” operation. We can thus ask ourselves the question of what the contents of α are in terms of the exposed parts of x and y . We can derive this state, P_{before} , via *textual substitution*: The repartitionings in Step 5a imply that

$$\frac{x_T = x_0}{x_B = \left(\frac{\chi_1}{x_2} \right)} \quad \text{and} \quad \frac{y_T = y_0}{y_B = \left(\frac{\psi_1}{y_2} \right)}.$$

If we substitute the expressions on the right of the equalities into the loop-invariant we find that

$$\alpha = x_T^T y_T + \hat{\alpha}$$

implies that

$$\alpha = \underbrace{x_0}_{x_T}^T \underbrace{y_0}_{y_T} + \hat{\alpha},$$

which is entered in Step 6 in Figure 1.7.

The state in Step 6 is determined via textual substitution.

Step 7: Determining the state after moving the thick lines. The movement of the thick lines in Step 5b means that now

$$\frac{x_T = \left(\frac{x_0}{\chi_1} \right)}{x_B = x_2} \quad \text{and} \quad \frac{y_T = \left(\frac{y_0}{\psi_1} \right)}{y_B = y_2},$$

so that

$$\alpha = x_T^T y_T + \hat{\alpha}$$

implies that

$$\alpha = \underbrace{\begin{pmatrix} x_0 \\ \chi_1 \end{pmatrix}}_{x_T}^T \underbrace{\begin{pmatrix} y_0 \\ \psi_1 \end{pmatrix}}_{y_T} + \hat{\alpha} = x_0^T y_0 + \chi_1 \psi_1 + \hat{\alpha},$$

which is then entered as state P_{after} in Step 7 in Figure 1.7.

The state in Step 7 is determined via textual substitution and the application of the rules of linear algebra.

Step 8: Determining the update. Comparing the contents in Step 6 and Step 7 now tells us that the state of α must change from

$$P_{before} : \alpha = x_0^T y_0 + \hat{\alpha}$$

to

$$P_{after} : \alpha = \underbrace{x_0^T y_0 + \hat{\alpha}}_{\text{already in } \alpha} + \chi_1 \psi_1,$$

which can be accomplished by updating α as

$$\alpha := \chi_1 \psi_1 + \alpha.$$

This is then entered in Step 8 in Figure 1.7.

It is *not* the case that $\hat{\alpha}$ (the original contents of α) must be saved, and that the update $\alpha = x_0^T y_0 + \chi_1 \psi_1 + \hat{\alpha}$ must be performed. Since α already contains $x_0^T y_0 + \hat{\alpha}$, only $\chi_1 \psi_1$ needs to be added. Thus, $\hat{\alpha}$ is only needed to be able to reason about the correctness of the algorithm.

Final algorithm. Finally, we note that all the annotations (in the grey boxes) in Figure 1.7 were only introduced to derive the statements of the algorithm. Deleting these produces the algorithm already stated in Figure 1.5.

1.4.2 Typesetting a derivation

Homework 1.4.2.13 Follow the instructions in the following video to duplicate Figure 1.7.

Useful links:

- A partial \LaTeX document can be found in [Exercises/Week01/TypesetSapdotVar1WS.tex](#).
- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).

 [SEE ANSWER](#)

FL ^A T _E X command	Result
<code>\FlaTwoByOne{x_T}</code> <code>{x_B}</code>	$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$
<code>\FlaThreeByOneB{x_0}</code> <code>{x_1}</code> <code>{x_2}</code>	$\left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$
<code>\FlaThreeByOneT{x_0}</code> <code>{x_1}</code> <code>{x_2}</code>	$\left(\begin{array}{c} x_0 \\ \hline x_1 \\ \hline x_2 \end{array}\right)$
<code>\FlaOneByTwo{x_L}{x_R}</code>	$\left(\begin{array}{c c} x_L & x_R \end{array}\right)$
<code>\FlaOneByThreeR{x_0}{x_1}{x_2}</code>	$\left(\begin{array}{c c c} x_0 & x_1 & x_2 \end{array}\right)$
<code>\FlaOneByThreeL{x_0}{x_1}{x_2}</code>	$\left(\begin{array}{c c c} x_0 & x_1 & x_2 \end{array}\right)$
<code>\FlaTwoByTwo{A_{TL}}{A_{TR}}</code> <code>{A_{BL}}{A_{BR}}</code>	$\left(\begin{array}{c c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
<code>\FlaThreeByThreeBR{A_{00}}{A_{01}}{A_{02}}</code> <code>{A_{10}}{A_{11}}{A_{12}}</code> <code>{A_{20}}{A_{21}}{A_{22}}</code>	$\left(\begin{array}{c c c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
<code>\FlaThreeByThreeBL{A_{00}}{A_{01}}{A_{02}}</code> <code>{A_{10}}{A_{11}}{A_{12}}</code> <code>{A_{20}}{A_{21}}{A_{22}}</code>	$\left(\begin{array}{c c c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
<code>\FlaThreeByThreeTR{A_{00}}{A_{01}}{A_{02}}</code> <code>{A_{10}}{A_{11}}{A_{12}}</code> <code>{A_{20}}{A_{21}}{A_{22}}</code>	$\left(\begin{array}{c c c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
<code>\FlaThreeByThreeTL{A_{00}}{A_{01}}{A_{02}}</code> <code>{A_{10}}{A_{11}}{A_{12}}</code> <code>{A_{20}}{A_{21}}{A_{22}}</code>	$\left(\begin{array}{c c c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

Table 1.1: Various FL^AT_EX commands for typesetting partitioned matrices.

1.4.3 Representing algorithms in code

We now have a notation and methodology for presenting and deriving correct algorithms for computing the APDOT operation. The problem is that those algorithms still need to be translated into code. The leap from the algorithm in Figure 1.5 to MATLAB implementation introduces opportunities for the introduction of programming errors. For this reason, we have created an *Application Programming Interface* (API) on top of MATLAB that allows the code to closely resemble the algorithm so that the correctness of the algorithm translates into correctness of the implementation. We call this API the *FLAME@lab* API.

Homework 1.4.3.14 Follow the instructions in the following video to translate the algorithm in Figure 1.5 into MATLAB code with the FLAME@lab API.

Useful links:

- The local install of the Spark webpage: [Spark/index.html](#).
(If version does not update correctly, click [here](#)).

 [SEE ANSWER](#)

A legitimate criticism of this API is that it creates a lot of overhead, since MATLAB will create intermediate arrays for subvectors. One answer to this is that we are only using MATLAB to illustrate how algorithms can be translated into code. A similar API for the C programming language, *FLAMEC*, is discussed in an enrichment from this week. That API does not incur as much overhead since no explicit copies of arrays are made.

[add more](#)

1.5 Enrichment

1.6 Wrap Up

1.6.1 Additional Exercises

A number of other commonly encountered operations involving vectors is presented in Table 1.2. These operations will resurface in later chapters and are therefore natural choices for further exercises.

Homework 1.6.1.14 Use the worksheet in Figure ?? to derive an algorithm for computing $\alpha := \alpha + x^T y$ by choosing the loop-invariant as $P_{inv} : \alpha = \hat{\alpha} + x_B^T y_B$.

Homework 1.6.1.14 Use the worksheet in Figure ?? to derive algorithms for the following two operations (see Table 1.2):

(a) The scaling of a vector.

Hint: Since x is modified in this operation, we recommend using \hat{x} to denote its original contents. In reasoning about correctness, \hat{x}_T , \hat{x}_B , \hat{x}_0 , \hat{x}_1 , and \hat{x}_2 will be encountered.

(b) The AXPY.

The name of this operation comes from scalar alpha times x plus y.

Homework 1.6.1.14 Prove the costs for the scaling of a vector and the AXPY shown in Table 1.2.

Step	Algorithm:
1a	
4	where
2	
3	while do
2,3	\wedge
5a	where
6	
8	
5b	
7	
2	
	endwhile
2,3	$\wedge \neg (\quad)$
1b	

Figure 1.8: Blank worksheet. You may want to print [Resources/BlankWorksheet.pdf](#).

Operation	Math. Notation	Defined as ($0 \leq i < m$)	Cost (flops)
Scaling (SCAL)	$x := \alpha x$	$\chi_i := \alpha \chi_i$	m
Addition (ADD)	$y := x + y$	$\psi_i := \chi_i + \psi_i$	m
Inner product (DOT)	$\alpha := x^T y$	$\alpha := \sum_{k=0}^{m-1} \chi_k \psi_k$	$2m - 1$
APDOT	$\alpha := \alpha + x^T y$	$\alpha := \alpha + \sum_{k=0}^{m-1} \chi_k \psi_k$	$2m$
AXPY	$y := \alpha x + y$	$\psi_i := \alpha \chi_i + \psi_i$	$2m$

Table 1.2: Basic operations involving vectors. In the table, α is a scalar and x, y are vectors of length m with their corresponding i th elements denoted as χ_i, ψ_i , respectively.

1.6.2 Summary

Bibliography

Index

- $m()$, 18
- $=$, 16
- P_{inv} , 21
- P_{post} , 20
- P_{pre} , 20
- $:=$, 16
- \cdot , 16
- \wedge , 21
- \neg , 21
- \wedge , 21
- APDOT, 17
- algorithm, annotated, 23
- algorithm, correctness of, 21
- annotated algorithm, 23
- API, 27
- Application Programming Interface, 27
- arithmetic product, 16
- assertion, 20
- assignment, 16
- becomes, 16
- Boolean expression, 20
- command, 20
- correctness, 20, 21
- dot product, 16, 29
- equality, 16
- FLAME
 - notation, 17
- lab, 29
- FLAMEC, 29
- Fundamental Invariance Theorem, 21
- goal-oriented derivation, 23
- Hoare triple, 20
- inner product, 16, 29
- iteration, 21
- logical and, 21
- logical negation, 21
- loop, 16, 20, 21
 - verifying, 20
- loop-body, 21
- loop-guard, 21
- loop-invariant, 21
- loop-invariant, determining, 24
- original contents, 21
- partitioned matrix expression, 24
- PME, 24
- postcondition, 20, 21, 24
- precondition, 20, 21, 24
- predicate, 20
- preface, iii
- sapdot, 17, 29
- saxpy, 29, 31
- scaling, 29
- state, 20
- vector, 16
 - slicing and dicing, 29
- vector, addition, 29
- vector, length, 18
- vector, scaling, 29
- vector,element, 16