Introduction to Research Software Development With MATLAB

Clean Code and Fundamentals of Research Software Development with MATLAB

Hands-On Workshop

Dr. Thomas Künzel, MathWorks, 2024

Table of Contents

Learning Outcomes

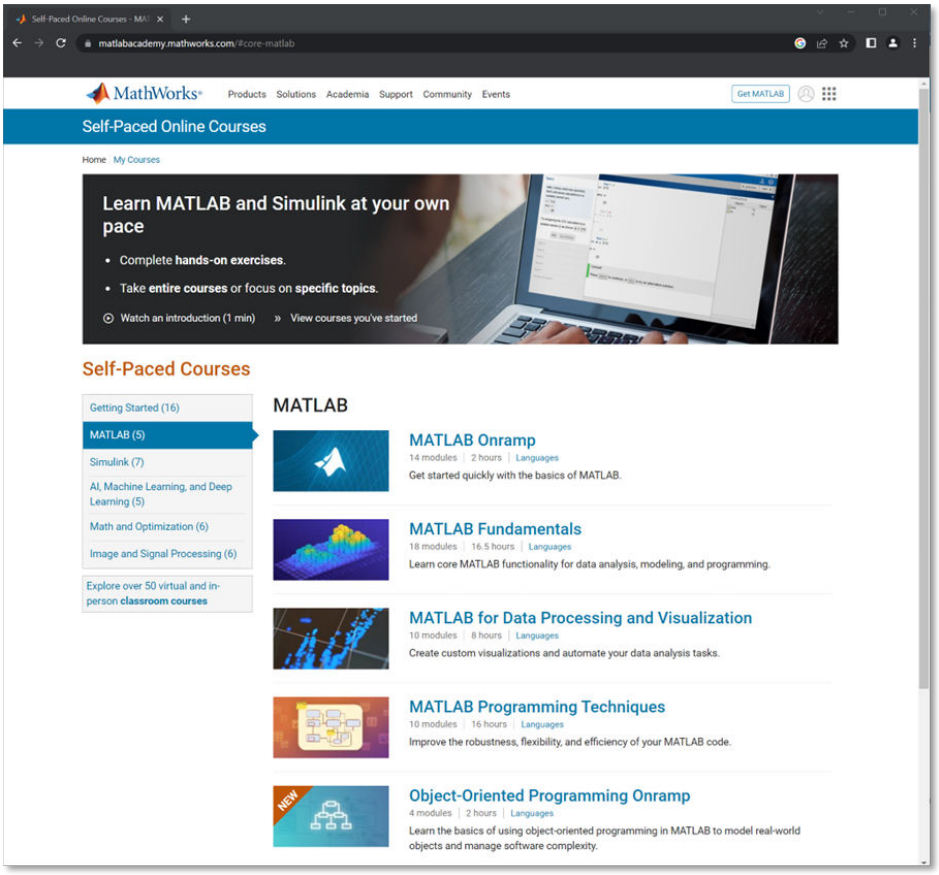
At the end of this workshop, you…

- Better understand how to write Clean Code and know modern MATLAB language and IDE features to achieve it
- Learned about the concept of writing Tests for your code and implement them with MATLAB
- Understand how to use git for local Source Control of your MATLAB code and why it is important for maintaining code quality
- Know why (and how!) to use git branches during development and how to merge
- Understand how to share and collaborate in a reproducible manner with MATLAB projects

You get a running start at pragmatic and collaborative development of maintainable research software with MATLAB.

This workshop is not about knowing the correct code syntax or writing efficient MATLAB code!

If you want to become better at programming in MATLAB, consider these self-paced online courses:



Self-Paced Online Courses

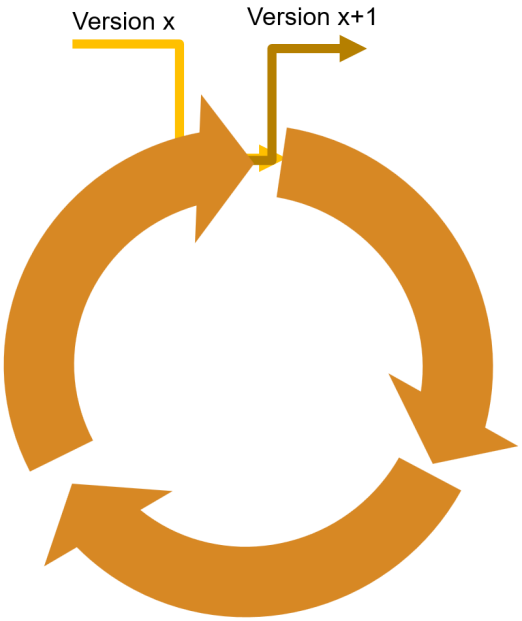
Background / Philosophy

I am proposing that most software development in research is prototyping, tinkering and incremental improvements.

In order to do this in a safe, reproducible and scalable manner I propose following the...

Incremental Research Software Development Cycle

Source Control: Committing code changes to a source control system concludes this incremental step. It ensures that your work can be undone and the development is retraceable. This is your **safety net**!



Refactoring: Small code that improve or efficiency but do **behavior** of the co

Also: Incrementally **without breaking functionality**

Tests: “Extra” project code that does not contribute to the function but asserts that the behavior of all components of the code (and maybe the whole program) is as expected. Passing the tests makes unequivocally clear that your refactoring was successful

Key messages / activities

- "Refactoring" - make your code CLEAN
- "Testing" - automatic confirmation that refactoring did not break functionality or that the new feature works as intended
- "Source Control" - Make your work retracable, divide work into logical chunks (i.e. branches), collaborate seamlessly with others
- "Reproducibility" - take care that the expected environment of the code (dependencies, variables, data, folder structure etc.) can be rebuild easily

Conclusions

- Writing functionally working code is only the first step
- Research code is often “prototype” code that gets used maybe once… however you never know when it may become critical for future you
- Code must be maintained through incremental changes

- A research software development project is **not** like building a house: Architect makes plan, builders construct, customers live in it for years and only occasionally paint the walls
- A research software development project is more like **gardening**, it involves constant planting, observing outcomes, formulating new plans, weeding, fighting bugs, replanting in new locations, etc.
- Writing code is **human communication** (with future-you or other developers), so aim to express your intent clearly. Code that cannot be read or understood by you or other people is "write-only" code. This code can in the future become useless at best, or even a severe liability that can bog down a whole research project for weeks!

Almost everything in this workshop is meant as a suggestion and is up to debate! Pick what you need for your project!

Modify and develop your own perspective!

Workshop Sections

The main parts of the workshop are stand-alone livescripts. First, let us make sure we are in the correct folder and that all subfolders are added to the MATLAB search path:

```
if exist("MATLABGoodCodingPractice_main.mlx", "file") && exist("sections", "dir")
    addpath(genpath(fullfile(".","sections")));
else
    error("Please go to the correct workshop folder!");
end
```

Run the sections to set up the next workshop section.

Section 0: MATLAB IDE Tools

```
edit MATLABGoodCodingPractice_tools.mlx
```

Section 1: Refactoring / Clean code

```
edit MATLABGoodCodingPractice_cleancode.mlx
```

Section 2: Testing

```
edit MATLABGoodCodingPractice_testing.mlx
```

Section 3: Source Control

```
edit MATLABGoodCodingPractice_sourcecontrol.mlx
```

Section 4: Reproducible Environment

```
edit MATLABGoodCodingPractice_projects.mlx
```

Section 0: Development Tools

Before we start the workshop for real let us quickly talk about your toolkit for software development in MATLAB.

You can of course also develop MATLAB code in Jupyter Notebooks or in VS Code with the MATLAB Extension, but MATLAB already comes with a great IDE that includes low-code Apps for productivity!

Table of Contents

MATLAB IDE Features

Here are just a few examples of powerful Editor/IDE features of MATLAB:

See the **Code Analyzer** ("Linter") at work?

```
mean(A,) % note the squiggly line!
```

Auto renaming: Change the first instance of "A." to "B." and accept suggestion with Shift+Enter!

Multi-Edit: hold ALT Key and drag mouse to generate multiple cursors! Then move all cursors to the right and add a semicolon to all lines at once!

```
% Create UserStruct
A.firstName = 'Hans'
A.lastName = 'Imglück'
A.age = 42
A.height = 179
A.ID = 124738
```

Multi-line commands (...) make code more readable and better for source-control. Try it, and make line 9 (the randn statement) into a multi-line command by adding ... after the comma and pressing RETURN

Automatic "restyling" of the code with smart-indent (icon or CTRL+I). Select lines 8-18 and press CTRL+I.

```
for idx = 1:10
k{idx} = randn(1,10);
disp(idx);
end
time = linspace(0, 1.0,200;
y = sin(2 .* pi .* 10 .* time)
```

```
plot(time, y, ...
 'LineStyle','-',...
'Color','r',...
    'Marker','square',...
 'MarkerFaceColor','w');
```

Automatic refactoring into functions, for example the body of for loops. Select line 21 and choose Refactor -> Convert to Function from the Live-Editor Tab.

```
x = linspace(-10,10,21);
for idx = 1:length(x)
    y(idx) = 10/x(idx);
end
```

MATLAB Debugger

The first call to the function "debugme.m" works...

```
y = debugme(150);
```

But the other call produces an error...

```
y = debugme(85);
```

Let us pretend this is a complicated case and look into the function:

```
edit debugme.m
```

Use left-click on line-numbers to set breakpoints...

```
1  function y = debugme(N)
2  %debugme - a function that produces errors
3
4  x = randn(1,N);
5  x(x<0) = 0;
6  for idx = 1:N
7      randomPick = randi(100,1);
8      y(idx) = x(randomPick);
9  end
```

...that halt the execution of the function at a specific line. This allows you to inspect the "internal state" of the function.

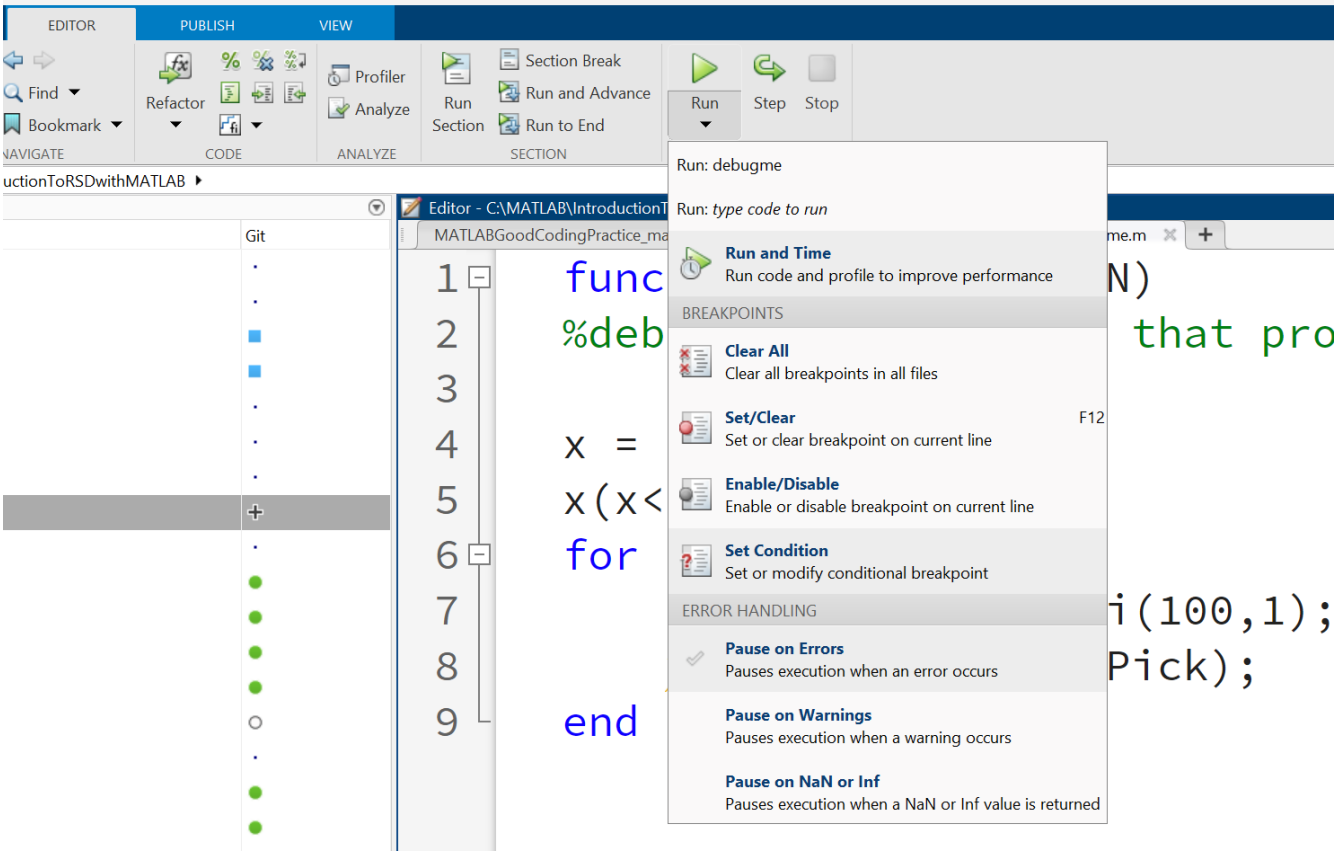
Breakpoints can be "conditional" as well, just try a right-click on a line number...

```
function y = debugme(N)
%debugme - a function that produces errors

x = randn(1,N);
x(x<0) = 0;
for idx = 1:N
    randomPick = randi(100,1);
    y(idx) = x(randomPick);
end
```

Set Breakpoint
Set Conditional Breakpoint...
Clear All Breakpoints
Show Code Folding Margin

... and enter a MATLAB expression that has to be "true" to halt the execution at that line.

Other useful conditional breakpoints can be set through the "Run" menu in the Editor-Tab:



As bug-fixing can often be detective work, these tools give you a lot of options to solve your case!

Workshop Section 1: Refactoring and Clean-Code

Table of Contents

What is Clean Code?

There are many definitions of clean code, I prefer these:

Clean code is easy to read, easy to understand and easy to change!

In other words:

Clean code is maintainable, modifiable, extendable!

Clean code **is not** (exclusively) about coding style. Following the agreed style of the language you use (like Python PEP8 style-guide) or the “scene”/group you work in is clearly recommended. However: this does not ensure any of the points above! MathWorks does not offer an official style-guide for MATLAB.

Again, please note: you can find many books, articles and opinions about Clean Code and good coding practices. This workshop is just a suggestion. Take from it what you can use, drop the rest!

How do I write Clean Code?

Some people say you should **"Write code as if the next person to read it is a violent psychopath who knows where you live."**

Good advice, good advice... But let us get a bit more specific.

Name Things Sensibly!

- **Don’t overthink it!**
- Use words that convey your intention

```
% unclear intention
idx
iii
k

% clear intention
index
counter
```

- Avoid very generic names, avoid lab-jargon, avoid cute jokes

```
myfunc
foo
dV
```

- Functions often use verbs or actions as names that tell you what it does

```
parseUserInput
```

```
classifyImage
saveData
```

- Variables get adjectives or nouns that tell you about the contained data

```
isValidUser
hasConnection

userName
voltageTrace
```

- Too long and too short is not good

```
% Which variable name has the correct lenght?
uN
usrN
userName
userNameWebInputNotSanitized
```

- Suggestion: functions and variables start lower-case and use camel case/mixed-case to separate words, classes start uppercase

```
variableName
functionName

ClassName
```

DRY - Don't Repeat Yourself!

Look at this (very typical) MATLAB code:

```
[x,y] = receiveData();% x ranges 0 - 2*pi in 32 steps, contains 3x data
highresx = linspace(0,2*pi,256);

p1 = polyfit(x{1},y{1},9);
fity1 = polyval(p1, highresx);

p2 = polyfit(x{2},y{2},9);
fity2 = polyval(p2, highresx);

p3 = polyfit(x{3},y{3},9);
fity3 = polyval(p3, highresx);
```

What if you want to change the polynomial order of the fit? You have to make three changes (what if you forget one?). Also, what if your new data contains 4 or only 2 measurements?

Now look at the refactored version:

```
for n = 1:length(x)
    [p{n}, fity{n}] = performFit(x{n}, y{n}, highresx);
end

function [p, fity] = performFit(x, y, highresx)
p = polyfit(x, y, 9);
fity = polyval(p, highresx);
end
```

Same lines of code, but more versatile ("length(x)") and more modifiable (only one place to change the polynomial order).

Avoid Mixed Levels of Abstraction!

The following code (from a real project of mine) is really hard to read! I propose that this is mainly because low-level and high-level operations are intermixed.

```
prompt = {"Name:","Experiment-Nr:"};
dlgtitle = "Request data";
dims = [1 35];
answer = inputdlg(prompt,dlgtitle,dims);
userList = load("userList.mat");
if ismember(lower(answer{1}),userList)
    experiments = requestExperiments(answer{1});
    if ismember(str2double(answer{2}), experiments)
        data = requestData(lower(answer{1}), str2double(answer{2})); %#ok<NASGU>
    else
        error("DataProvider:BadExpID","Invalid Experiment-ID");
    end
else
    error("DataProvider:BadUser","Unknown Experimenter");
end
```

The refactored version is a lot easier to understand. This is because the main program now only has "high-level" operations with good names which make very clear what the program does

```
[rawName, rawID] = getUserQuery();
userName = validateName(rawName);
expID = validateID(userName, rawID);
data = requestData(userName, expID);
```

Of course I had to create three function for that:

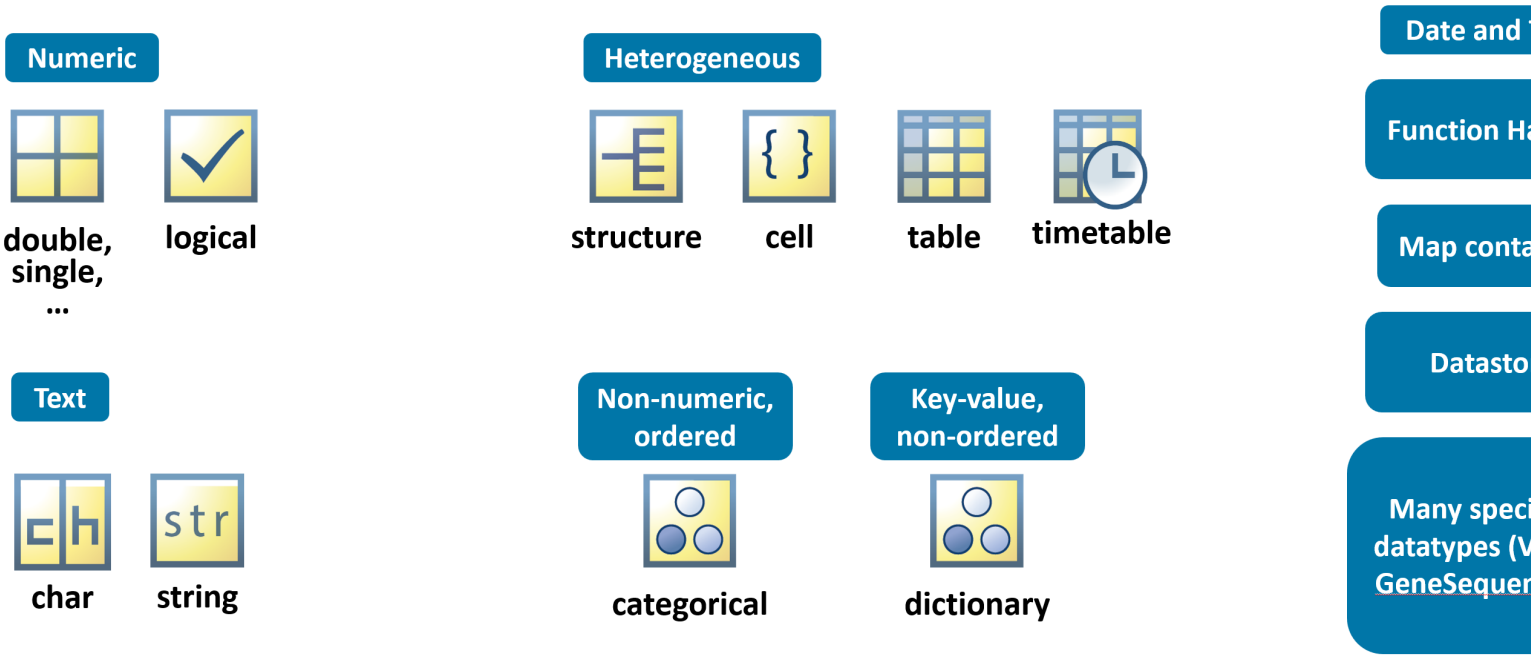
```
function [name, expID] = getUserQuery()
prompt = {"Name: ", "Experiment-Nr.: "};
dlgtitle = "Request data";
dims = [1 35];
answer = inputdlg(prompt, dlgtitle, dims);
name = lower(answer{1});
expID = str2double(answer{2});
end

function userName = validateName(userName)
userList = load("userList.mat");
errorMsg = "Unknown Username";
assert(ismember(userName, userList), errorMsg); %guard statement instead of nested
if-else...
end

function expID = validateID(userName, expID)
experiments = requestExperiments(userName);
errorMsg = "Unknown Experiment-ID";
assert(ismember(expID, experiments), errorMsg); %guard statement instead of nested
if-else...
end
```

Use Abstract Representations When Available!

As we just learned hiding the "nitty gritty" details behind function calls can make a program very readable. Similarly, high-level constructs like specialized data-types and containers can make very readable code.

High-Level Data-Types and Containers

Of course you can represent all your data in low-level types like arrays of double and char. But this make you write (and read!) a lot of extra code on different levels of abstractions. MATLAB offers a lot of specialized data types with dedicated methods.



Another good example are datastores or tall-arrays - they are needed to work with data that exceeds local memory, but they also allow you to do technically complex things in a few lines of very clean code!

Object-Oriented Programming

What if the MATLAB datatypes are not abstract enough to express your intent?

Let us say, your unit of processing is the “Experimental-Session” (which is a combination of images, time-series and metadata). All operations in your code deal with processing those “conceptually related” data.

Consider defining a Class (“ExperimentalSession”) that encapsulates data (“Properties”) and custom operations (“Methods”)! This makes VERY readable code because of great separation of the abstraction levels but is considered relatively advanced programming.

```
classdef Rectangle
    properties
        X (1,1) double {mustBeReal} = 0
        Y (1,1) double {mustBeReal} = 0
        Width (1,1) double {mustBeReal} = 0
        Height (1,1) double {mustBeReal} = 0
    end

    methods
        function R = enlarge(R,x,y)
            arguments (Input)
                R (1,1) Rectangle
                x (1,1) {mustBeNonnegative}
                y (1,1) {mustBeNonnegative}
            end
            arguments (Output)
                R (1,1) Rectangle
            end
            R.Width = R.Width + x;
            R.Height = R.Height + y;
        end
    end
end
```

Note: the code is not runnable because defining a class in a script is not allowed.

The actual code to create and enlarge a rectangle is very clean now:

```
rect1 = Rectangle;
rect1.enlarge(5,1)
```

Write good functions!

A large part of what we talked about so far leads to the following conclusion: structure your code in well written functions!

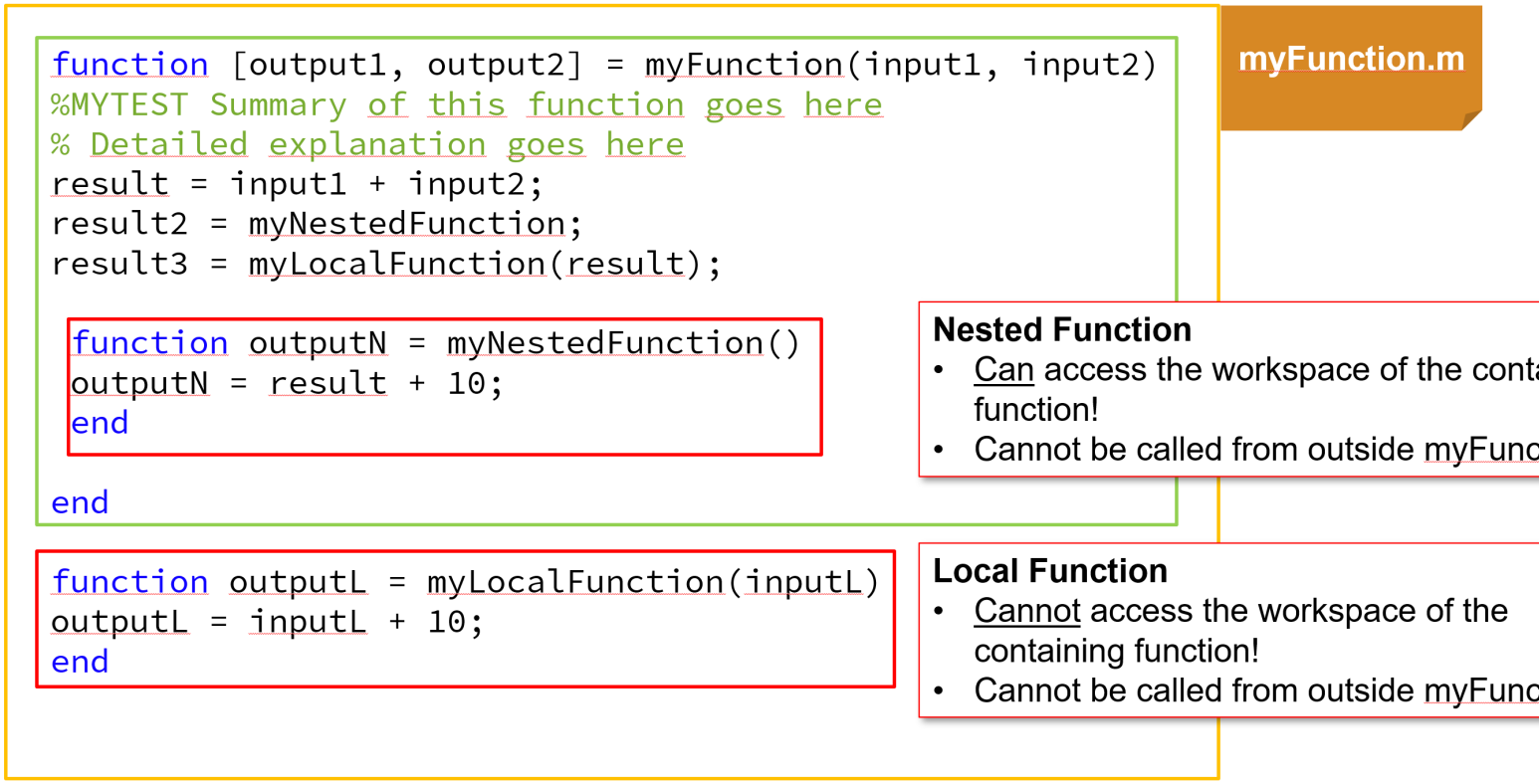
Functions in MATLAB are superior to scripts in many ways. **A good function "does one thing".** Thus, good functions are automatically short and concise. If you find yourself scrolling up and down in a very complicated and long function - maybe it can be refactored into smaller functions that do more specific tasks?

Another useful rule of thumb is: **A good function should have "as few inputs as possible"**. If your function has dozens of input parameters, maybe it is time to refactor it?

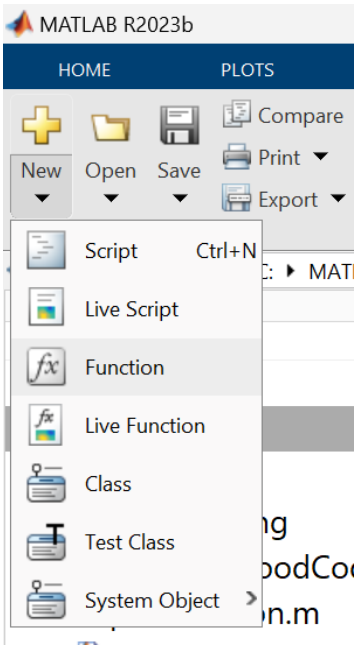
Functions have their own **protected variable workspace (scope)**! This is good, as you can (re-)use clear variable names without the fear of overwriting other data. However, this makes you work a bit more during coding, as you have to consider which variables are available inside the function and which are not.

My opinion (we can argue!): use live-scripts [*.mlx] for workflow examples and functions [in separate *.m files] for all operational code

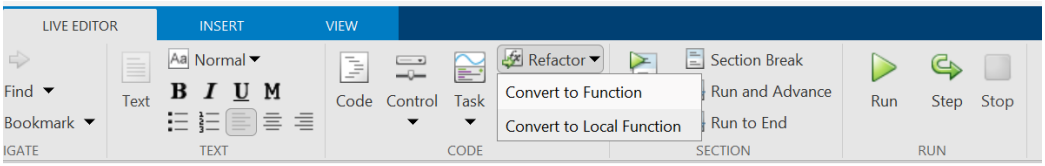
Anatomy of MATLAB Functions



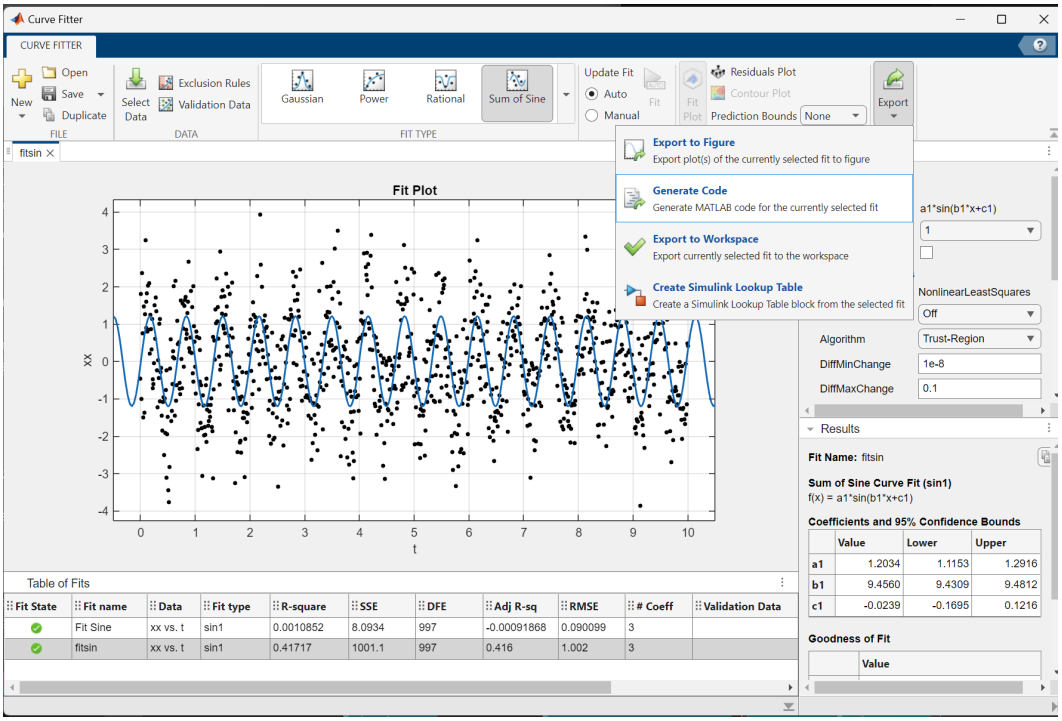
If you forget how to write a function, MATLAB will help you with this: you can create a correctly styled, empty "New" Function with the GUI



...automatically refactor code into functions



or export functionality from Apps and LiveTasks into auto-generated functions!




```
% (...)
[xData, yData] = prepareCurveData( t, xx );
```

```
% Set up fittype and options.
ft = fittype( 'sin1' );
opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
opts.Display = 'Off';
opts.Lower = [-Inf 0 -Inf];
opts.StartPoint = [1.19675885778667 9.42477796076938 0.12581876820584];

% Fit model to data.
[fitresult, gof] = fit( xData, yData, ft, opts );
```

Function Call Precedence

How does MATLAB know that you want to call a custom function you wrote? When MATLAB encounters a word in program code it will stick to the following order when trying to figure out what you are referring to:

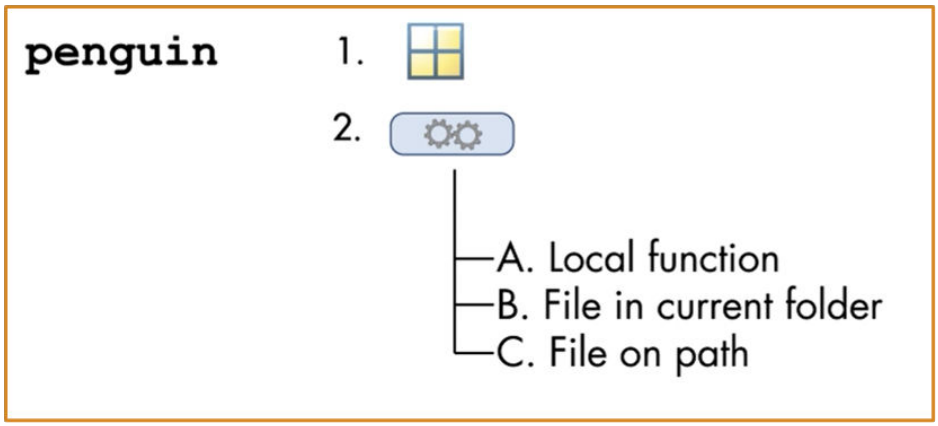
1: Look for a variable called penguin in the current workspace

2: Look for a script or function called penguin

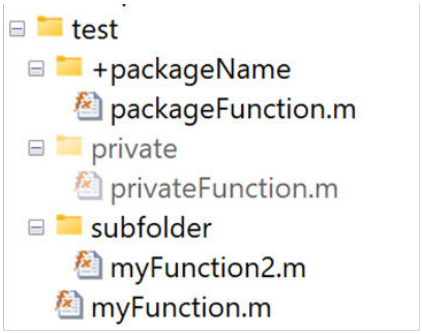
2.1: Is it a nested or local function in the current file?

2.2: Is it a file of that name in the current folder or subfolders of the current folder?

2.3: Is a file of that name anywhere on the MATLAB search path?



Structuring your code as good functions with good names will make your code project more accessible. Putting functions in subfolders, possibly with special names, will support this:



If the folder test (and subfolders) are on path…

- myFunction and myFunction2 can be called from anywhere
- privateFunction can only be called if test is the current folder of the calling "agent"
- packageFunction must be called by packageName.packageFunction

Input Argument Validation

The use of function argument validation is optional in function definitions. But argument validation is very useful in functions that can be called by any code and where validity of the arguments must be determined before executing the function code.

Argument validation also allows definition of default values

Functions that are designed for use by others can benefit from the appropriate level of restriction on arguments and the opportunity to return specific error messages based on the argument validation checks.

```
function myFunction(inputArg)
    arguments
        inputArg (dim1,dim2,...) ClassName {fcn1,fcn2,...} = defaultValue
    end
     % Function code
end
```

Size Class Functions

Check out the Documentation on the arguments block!

Handle Errors and Write Robust Code!

- **Fail early:** validate inputs or intermediate results as soon as possible
- Use assert statements to guard your program (instead of the horror of nested if-else statements)
- **Fail gracefully:** throw human-readable error messages and make sure to leave the machine in a reasonable state
- Consider try/catch blocks to handle "non-breaking" errors or clean-up upon error

```
try
 filehandles = unrealiableInput();%generates many files
 analysis = analyzeData(filehandles);%this might fail
catch
 %delete temporary files
 %generate meaningful message to user
end
```

Robust Code makes as little assumptions as possible

- Avoid hardcoded path names (mind the file-separators and different home folder etc)! Functions like filesep, userpath or fullfille can be helpful to write OS-agnostic and thus robust code

```
imread("/home/user/thomas/Documents/MATLAB/myimages/urlaubsfoto.jpg") % only works
on linux!
imread("C:\Users\tkuenzel\Documents\MATLAB\myimages\urlaubsfoto.jpg") % only works
on windows!
imread(fullfile(userpath, "myimages","urlaubsfoto.jpg")) % works on all OS
```

- Check we run in the correct folder, check if output folders exist

```
if ~exist("output", "dir")
    mkdir("output")
end
```

- Check for existence of files (even the ones your program just created)

```
assert(exist("myOutput.mat","file"), "Output-file does not exist!")
```

- Use dynamic output file names (otherwise results will be overwritten)

```
tempdir % returns the folder used for "temporary" files (OS-specific!)
fileName = fullfile(tempname) % returns a unique, random filename
```

- If you change directories, move the user back where we started in the end

```
oldDir = cd(newDir);
% ...
% Some work is done
%...
cd(oldDir);
```

Avoid Comments! (yes, I am serious!) [at least a little bit!]

- Comments must be maintained alongside the code: twice the work and can be difficult
- Comments can do more harm than good if not maintained

- Comments are not a source control system

```
function output = complicatedAnalysis(input)
%COMPLICATEDANALYSIS Perform the complicated analysis on data
% The complicated analysis follows Mustermann et al., 2023,
% Journal of complicated algorithms 34:1-322

inputp1 = input + 1; % add one to input
% inputp1 = doStuff(inputp1);%DOES NOT WORK

% now we fit a gaussian to the data
polynomial = polyfit(inputp1);% fit a polynomial

compDat = veryAbstractFun();% watch out, this pulls data from database
```

My advice: avoid comments as much as possible, MATLAB code is already the most concise way to express your intent!

Exercise 1: Refactor a typical MATLAB Script

Take a look at this typical MATLAB script:

```
edit plotscript.m
```

Can you refactor it into a function, making it more versatile and expandable by applying the Clean Code principles we discussed above?

Consider to make the timetable T an input to the function! Can you make sure the user provides the correct input?

(Hint: if you need inspiration you can look into the "plotfunction.m" file... this is what I thought could be a refactored version of the script)

Workshop Section 2: Testing

What is Testing?

- Testing or “Unit Testing” uses code that calls your operational program units (most often functions) in various ways and checks whether the behavior is as expected
- Testing frameworks are used to automate running this code
- “Passing the unit tests” as a requirement for code to be committed to source control helps write maybe more bug-free code
- Unit tests are the first users of the code and show the expected behavior of your program. If unit tests exist in a project, try studying them!

Do I really need so much extra code for my small research software project?

- For analysis tools I felt at least an “completion test” (running e.g. an example dataset that creates a known result x) to be necessary
- **Be pragmatic**, the amount of overhead work should match the size and scope of the project

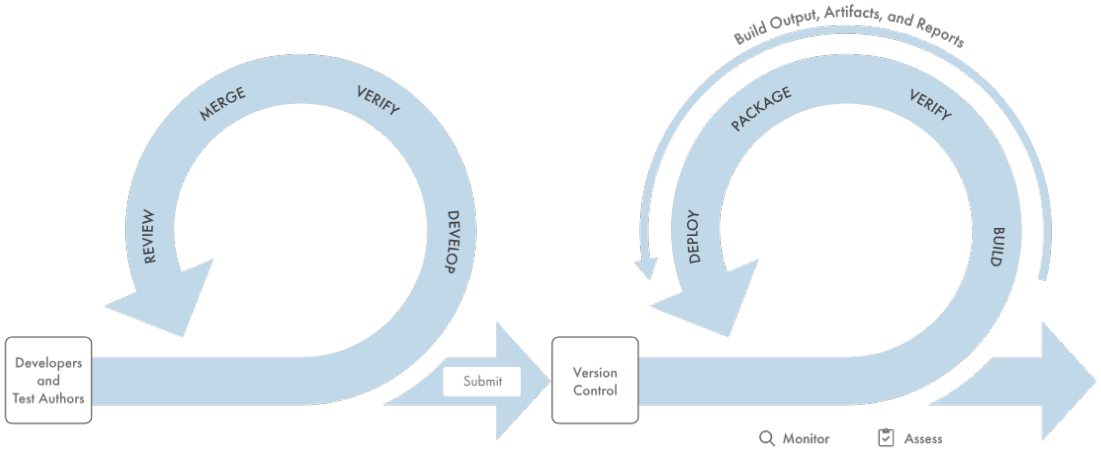
Next time: aim to write ONE test for your next function

MATLAB Test

MATLAB now has a Test Toolbox called "MATLAB Test". MATLAB Test provides tools for developing, executing, measuring, and managing dynamic tests of MATLAB code, including deployed applications and user-authored toolboxes. It helps you go to scale with your projects and test suites. This is however **out of scope for this workshop**.

Testing & CI/CD (beyond our scope today!)

In larger project or production settings testing is often deeply integrated in the workflow. This allows agile development AND production and deployment of high-quality code artifacts. These workflows are often described under the term "continuous integration" and "continuous deployment". Here tests ("verify") are automatically performed when it is submitted to a version control system.



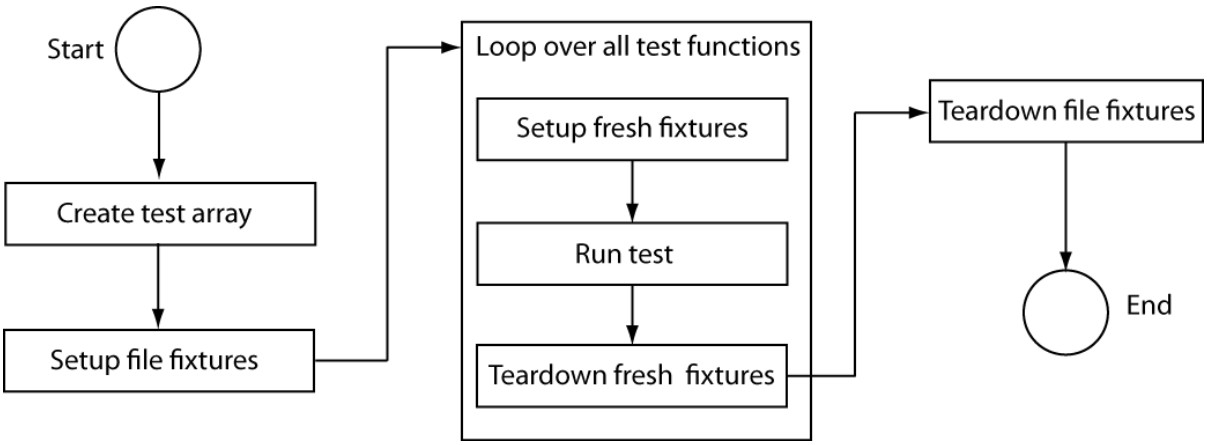
MATLAB and Simulink offer a wide range of capabilities and integrations for leading CI/CD systems. If this sounds like something your team must use, please refer to the documentation on CI or contact us! As fascinating as this is, this topic is **out of scope for this workshop** as well.

MATLAB Unit Testing Framework

How do you write test code? In principle you could create your own functions and scripts as tests and run them as needed. But you will soon notice it is more convenient to use the MATLAB Testing Framework.

Unit tests can be script-based, function-based and class-based - versatility but also complexity increases from left to right.

Here we will aim for the middle-ground and discuss **function-based unit-testing**



To understand this better we follow a small documentation example. Let "*quadraticSolver.m*" be the function we develop...

```
edit quadraticSolver.m
```

Lets quickly visualize what it does:

```
x = linspace(-3,3,600);
a = -1;
```

```
b = -2;
c = 2;
y = a.*x.^2 + b.*x + c;
plot(x,y);
zc = quadraticSolver(a, b, c);
hold on
plot(zc,[0 0],"rx");
```

Makes sense, right?

Now let us take a look at the corresponding test function. Test functions must have the same name as the function under test plus the keyword "test" at the beginning or end of the name. Thus our test function for quadraticSolver.m is called *quadraticSolverTest.m*:

```
edit quadraticSolverTest.m
```

You run the suite of tests for your function from the commmand line (or code) with

```
results = runtests("quadraticSolverTest.m")
```

We see all the test succeeded. The result struct contains valuable diagnostic information, which is probably useful when a test fails unexpectedly!

Please also note how the "fixture" functions were executed: the **setupOnce** and **teardownOnce** were run **at the beginning and end of the suite**, the **setup** and **teardown** were run **repeatedly at the beginning and end of each test** in the suite. This allows you to create the appropriate environment for your functions under test so they can actually run or tidy-up generated files, figures etc. after a function ran.

The function-based tests in MATLAB are an ideal compromise of versatility and ease of use! They are sufficient for almost all use-cases.

Test-Driven Development

Some developers practice test driven development: they write the test first and then implement the feature to pass the test!

Let us explore this philosophy and add a feature to our function with test-driven development: The function *quadraticSolver.m* could also tell us whether the solution it found was real or imaginary. Thus we could demand a second output argument which is a boolean which is true if the quadratic function has a real solution. Easy right?

We'll add a new test to the suite first, i.e. uncomment lines 19-22 in *quadraticSolverTest.m*!

```
%function testBooleanOutput(testCase)
%[actSolution, hasRealSolution] = quadraticSolver(1,2,10);
%assert(hasRealSolution);
%end
```

We run the test suite and see - quite expectedly - one test fails with errors!

```
results = runtests("quadraticSolverTest.m")
```

Of course it fails, since we haven't even implemented the new feature yet. However, this helps us structure our work. We can go home and come back next week. When we run the test suite we immediately remember what the next steps were. Or someone else can run the test suite and see the missing feature and implement it?

Let us add the feature now change line 1 to this code (without the % of course):

```
%function [roots, hasRealSolution] = quadraticSolver(a,b,c)
```

and add this code to line 13 (uncommented):

```
%hasRealSolution = isreal(roots);
```

When we now run the test suite...

```
results = runtests("quadraticSolverTest.m")
```

... it passes. We successfully implemented a new feature with Test-Driven Development! This code is ready for committing to source control... but that is the next section of the workshop!

Exercise 2: Write a Unit Test

Now it is your turn! Complete the *plotfunctionTest.m* by implementing a unit test for the *plotfunction.m* (remember, this was my suggested solution of the refactoring exercise). Think about adding some cleanup code to the teardown fixture, otherwise you'll clutter your desktop with figure windows each time you run the test suite.

All hints should be in the file. If you really get stuck take a look at "*solution_plotfunctionTest.m*".

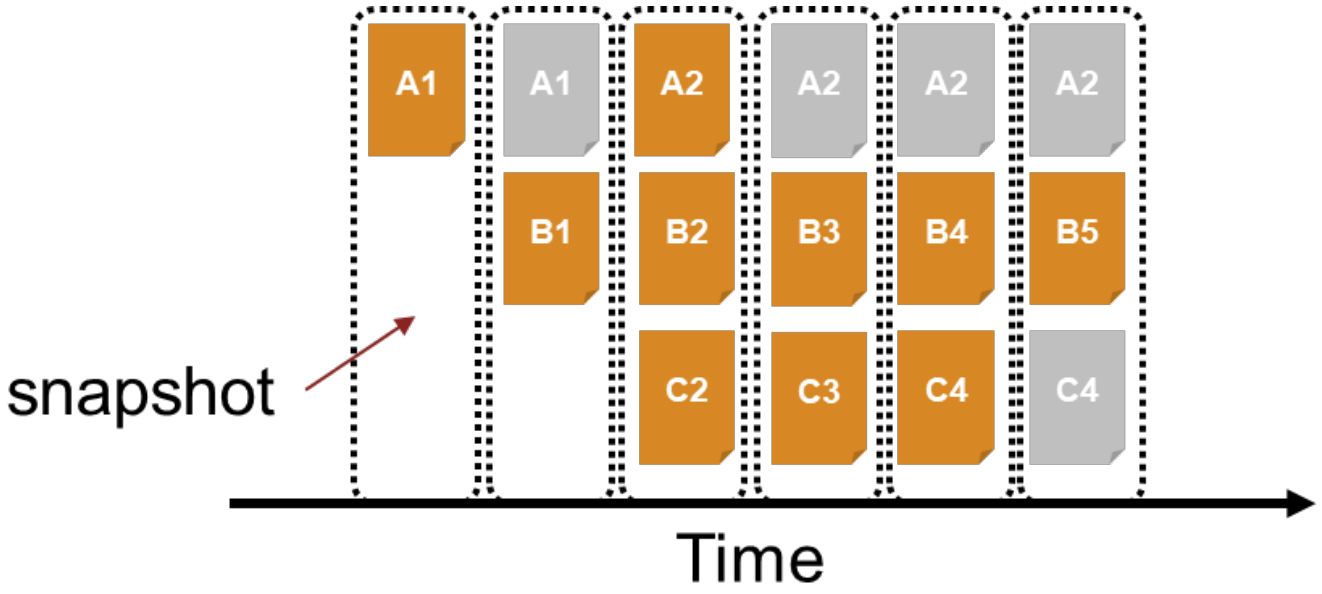
```
edit plotfunctionTest.m
```

Workshop Section 3: Source-Control & git Integration

Table of Contents

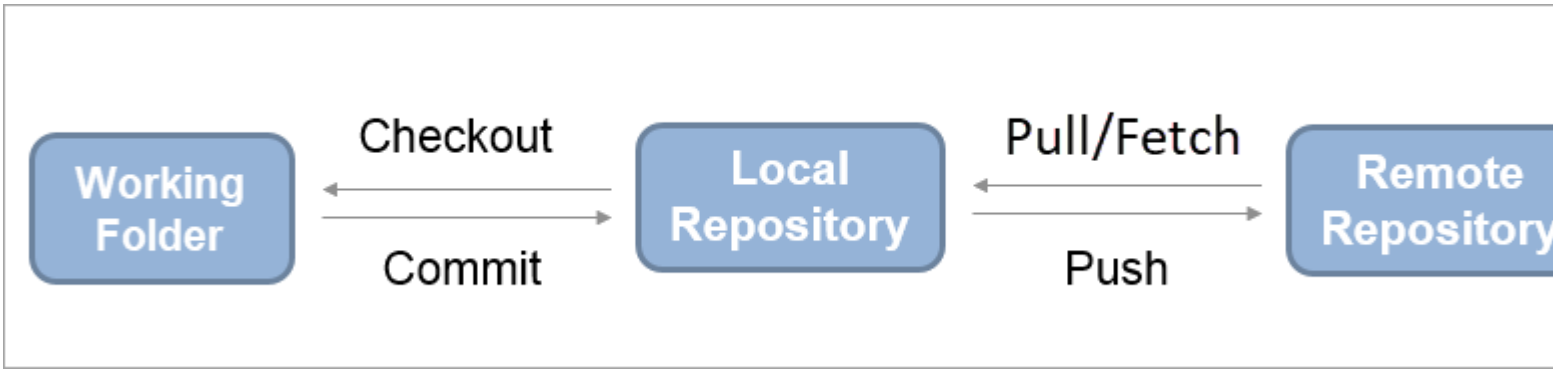
What is git and how does it work?

- Git is a distributed version control system, a software that records changes in files over time and allows recall of specific versions
- “Distributed” means every client has the complete history (not just the latest state), so many identical copies of the project might exist. There can be a remote repository (with lots of tools to synchronize local states to the remote states) but it is not required. Often git it is used locally only.
- Git is focused on saving the complete state of the project (“snapshot”) at a given time, not only the differences (in a smart way, however, to save space and processing time)
- Git calculates a cryptographic checksum from every snapshot and uses that to verify and reference the state



Basic git workflow

Workflows for git are usually with remote repositories and collaboration in mind. It is however perfectly fine (and actually often desirable) to only work with a local repository.



Branches

- Branches are encouraged in git! Use branches of your project for logically separate aspects of your work, testing, alternate ideas, etc. especially when working in a collaborative project!
- A repository has at least one branch: the current commit and its series of parent commits (most often called “master” or “main” branch)
- Commits can have multiple “children” (a branching point) and multiple “parents” (a merge point)
- git maintains a pointer called HEAD which identifies the currently active commit.

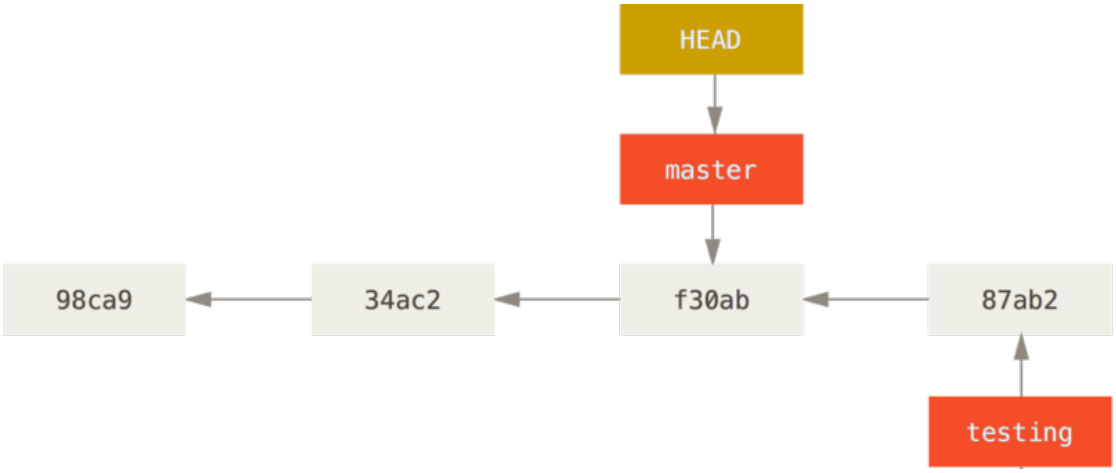


image from Git - Branches auf einen Blick (git-scm.com)

Using git in the MATLAB IDE

MATLAB ships a built-in version of git, so in principle you do not have to install anything.

There is several ways to interact with git from within MATLAB. You should experiment and use the approach that makes you most productive!

System Calls

This works well when you have git installed and configured system-wide. This is advisable anyway, so maybe this is for you:

```
error("do not run this section as whole please"); % this code-section should not be
run

% create a new repository in current folder
!git init

% clone a remote repository in current folder
!git clone <remoterepository.git> <local folder name>

% merge remote changes into your local repository
!git pull

% stage changed <file>
!git add <file>
!git add .

% commit (with message) / -a flag skips the staging (= add .)
!git commit -m "Message”
!git commit –a -m "Message”

% shows state of current directory
!git status
```

```
% shows timeline of commits
!git log

% push local changes to remote repository
!git push

% creates a new branch at the current commit state
!git branch <name>

% points HEAD to the branch <name> (the working directory will be set-up in the
appropriate state).
% All following commits will extend the series of this branch
!git checkout <name>

%integrates the changes in <branch> into the branch that the current HEAD points to
[conflicts could arise!]
!git merge <branch>
```

New programmatic git commands

In R2023b new programmatic tools to interact with git were introduced. These can be in code (think about CI/CD or other complex workflows) or entered into the Command Window

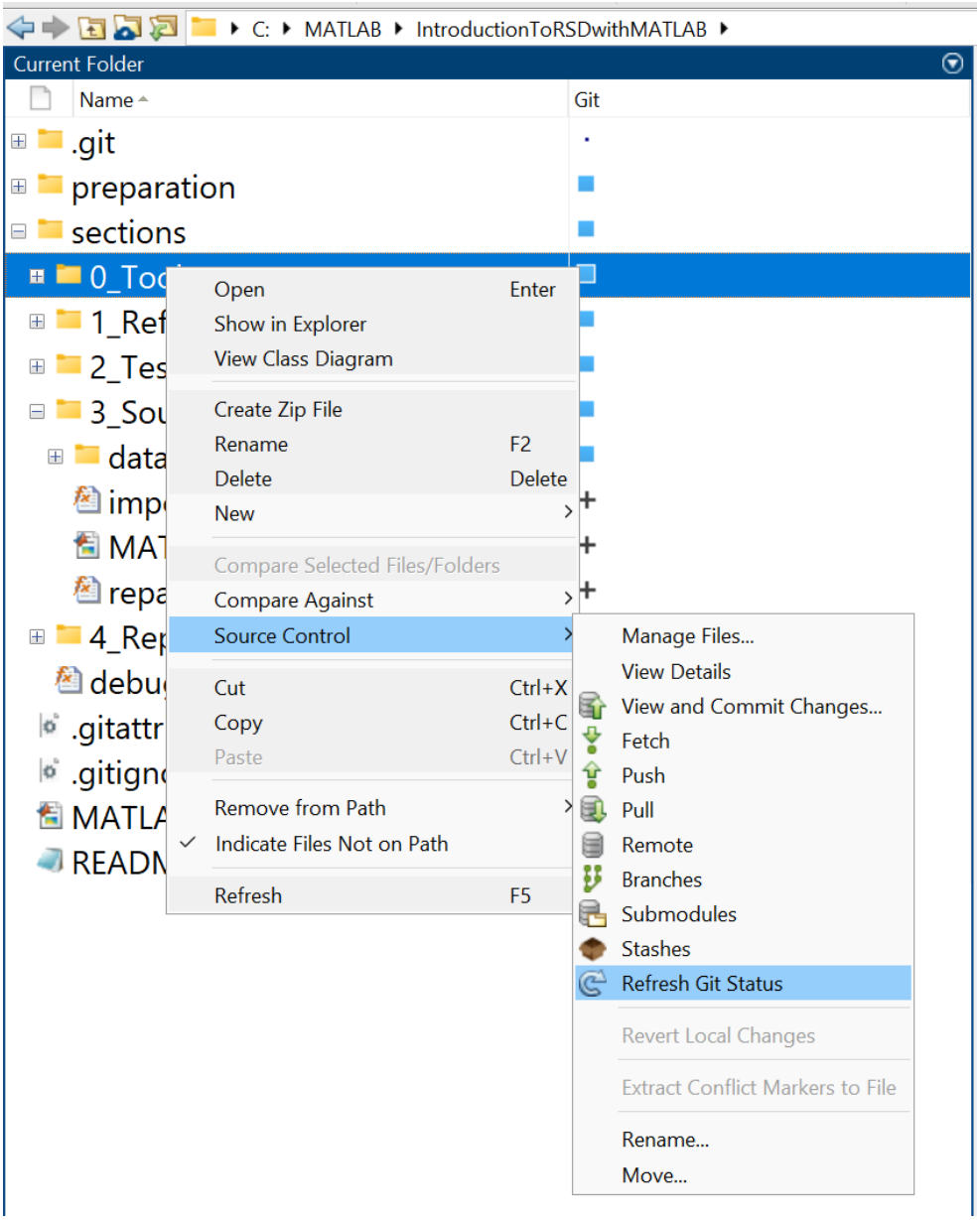
```
error("do not run this section as whole please"); % this code-section should not be
run

% create a new repository in current folder
repo = gitinit

% clone a remote repository in current folder
repo = gitclone(URL)

%get repo-object for existing repository
repo = gitrepo

% merge remote changes into your local repository
pull(repo)

% stage changed <file>
add(repo,filename)

% commit (with message) / -a flag skips the staging (= add .)
commitDetails = commit(repo,Message=commitMessage)

% shows state of current directory
statusDetails = status(repo)

% shows timeline of commits
commitHistory = log(repo)
```

```
% push local changes to remote repository
push(repo)

% creates a new branch at the current commit state
branchDetails = createBranch(repo,name)

% points HEAD to the branch <name> (the working directory will be set-up in the
appropriate state).
% All following commits will extend the series of this branch
branchDetails = switchBranch(repo,name)

%integrates the changes in <branch> into the branch that the current HEAD points to
[conflicts could arise!]
merge(repo,commitIdentifier) % commmitIdentifier can be  branch, commit, or tag
```

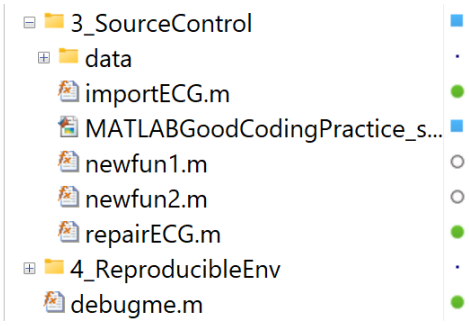
Git IDE Integration (classical Desktop)

You can also interact with git in a graphical manner in the IDE. Here I am showing Screenshots from MATLAB classical Desktop.

Your main interaction principle is to right-click in the Current Folder tab. You can then interact with git in many ways with the commands under the "Source Control" menupoint:

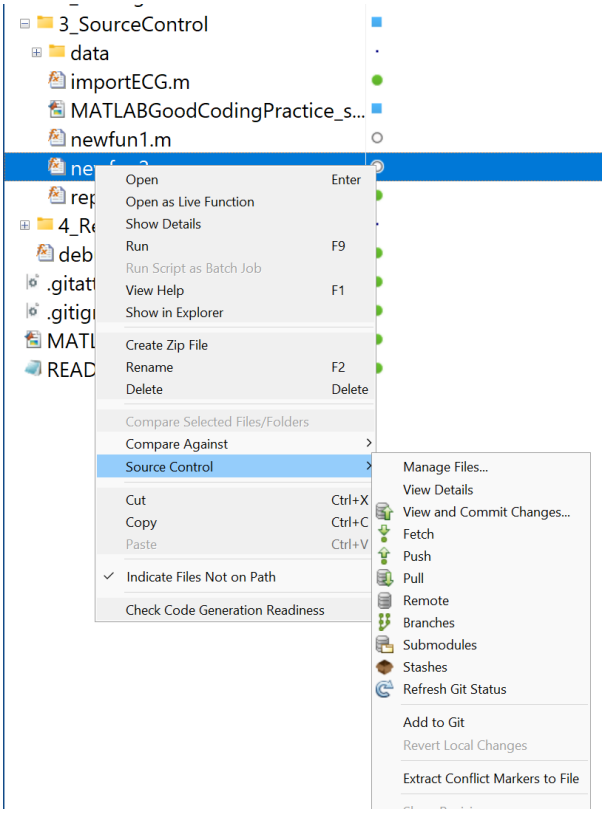


Note the markers next to the files in the Current Folder tab:

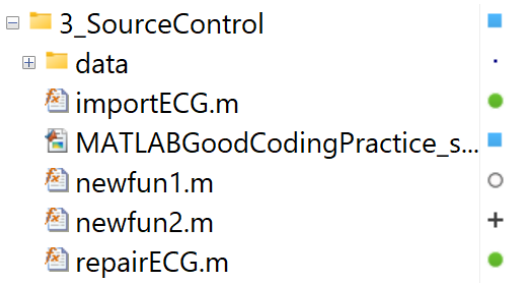


Here we have unstaged files (white), files that are in the repository with changes (blue) and without changes.

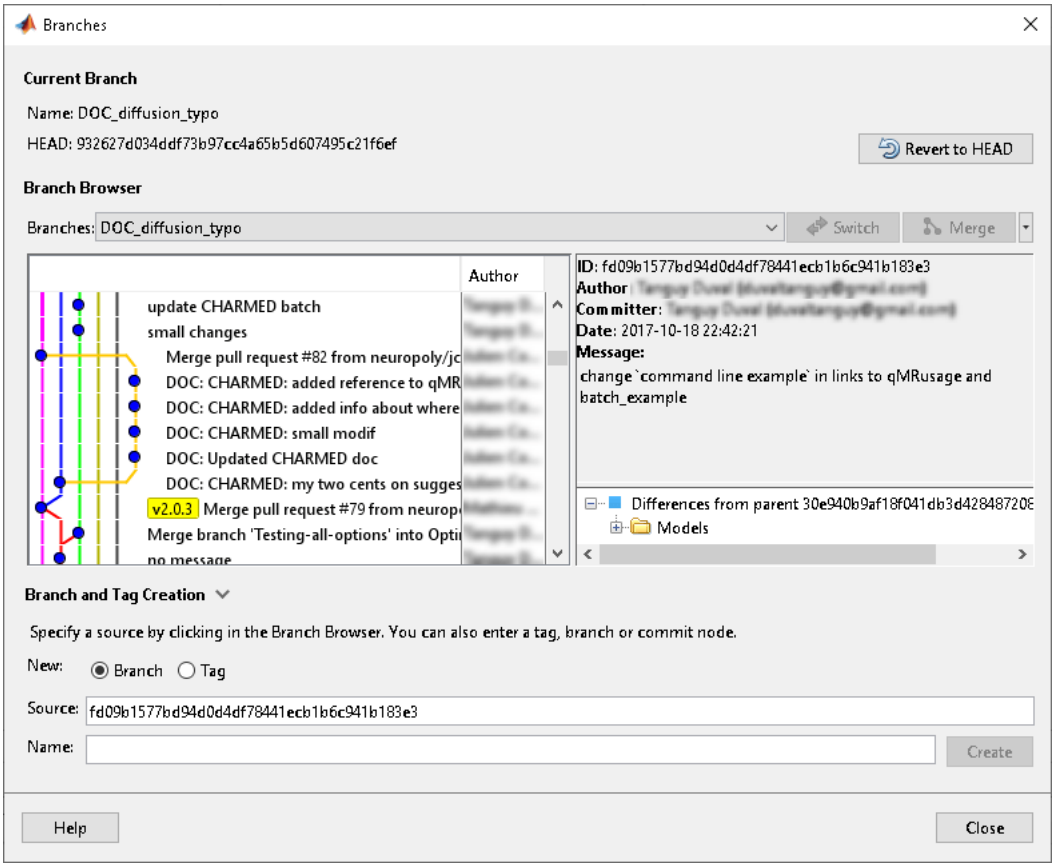
If you want to add one of the unstaged files to future commits: right-click - Source Control - Add to git



Once it is staged (but not committed) the file gets a new marker:



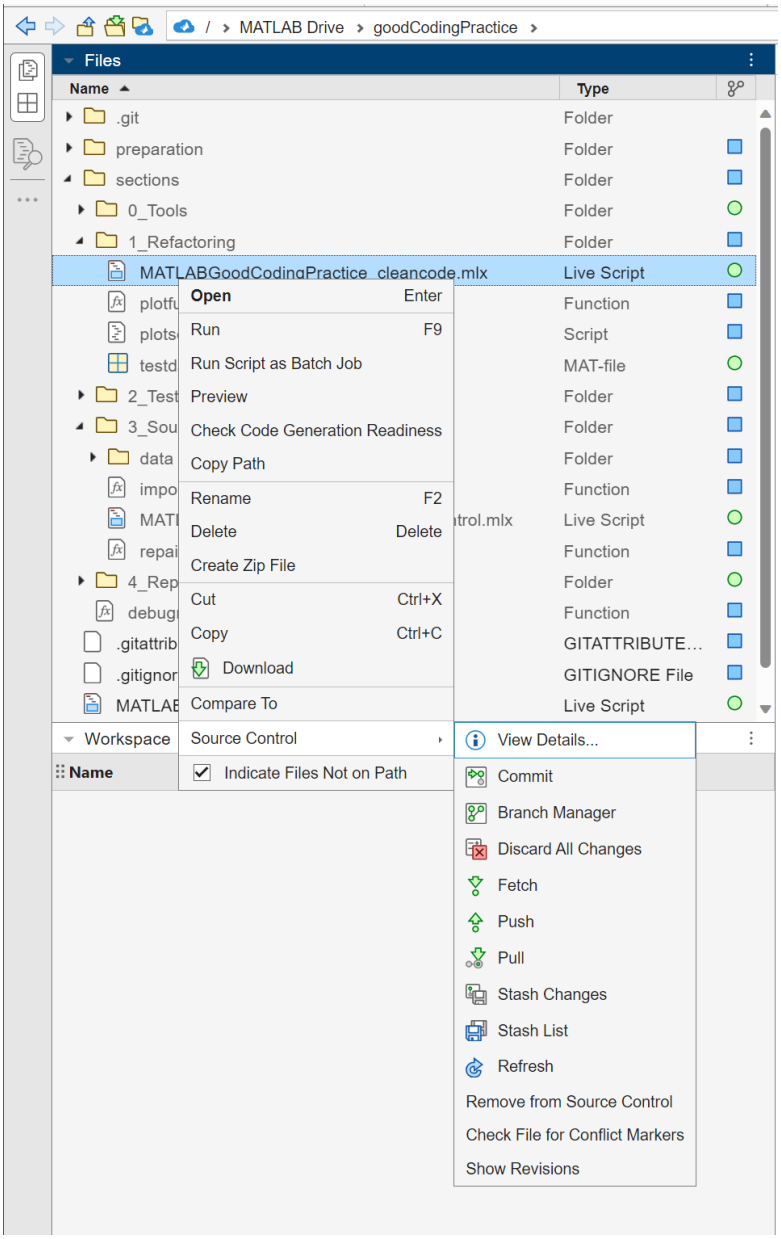
Under the "Source Control - Branches" Submenu you can find the graphical branch-manager App. Here you can create, checkout and merge branches.



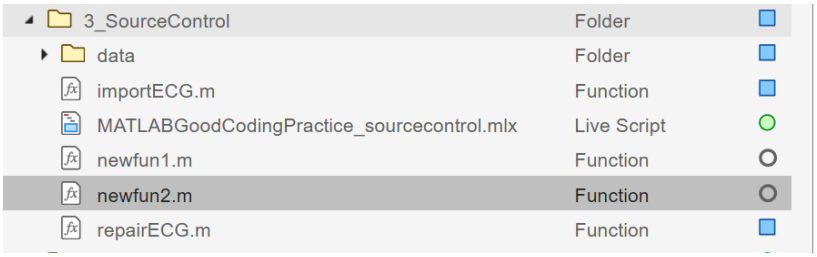
Git IDE Integration (new Desktop and MATLAB Online)

You can also interact with git in a graphical manner in the IDE. Here I am showing Screenshots from MATLAB Online. This also applies to the new Desktop.

Your main interaction principle is to right-click in the Current Folder tab. You can then interact with git in many ways with the commands under the "Source Control" menupoint:

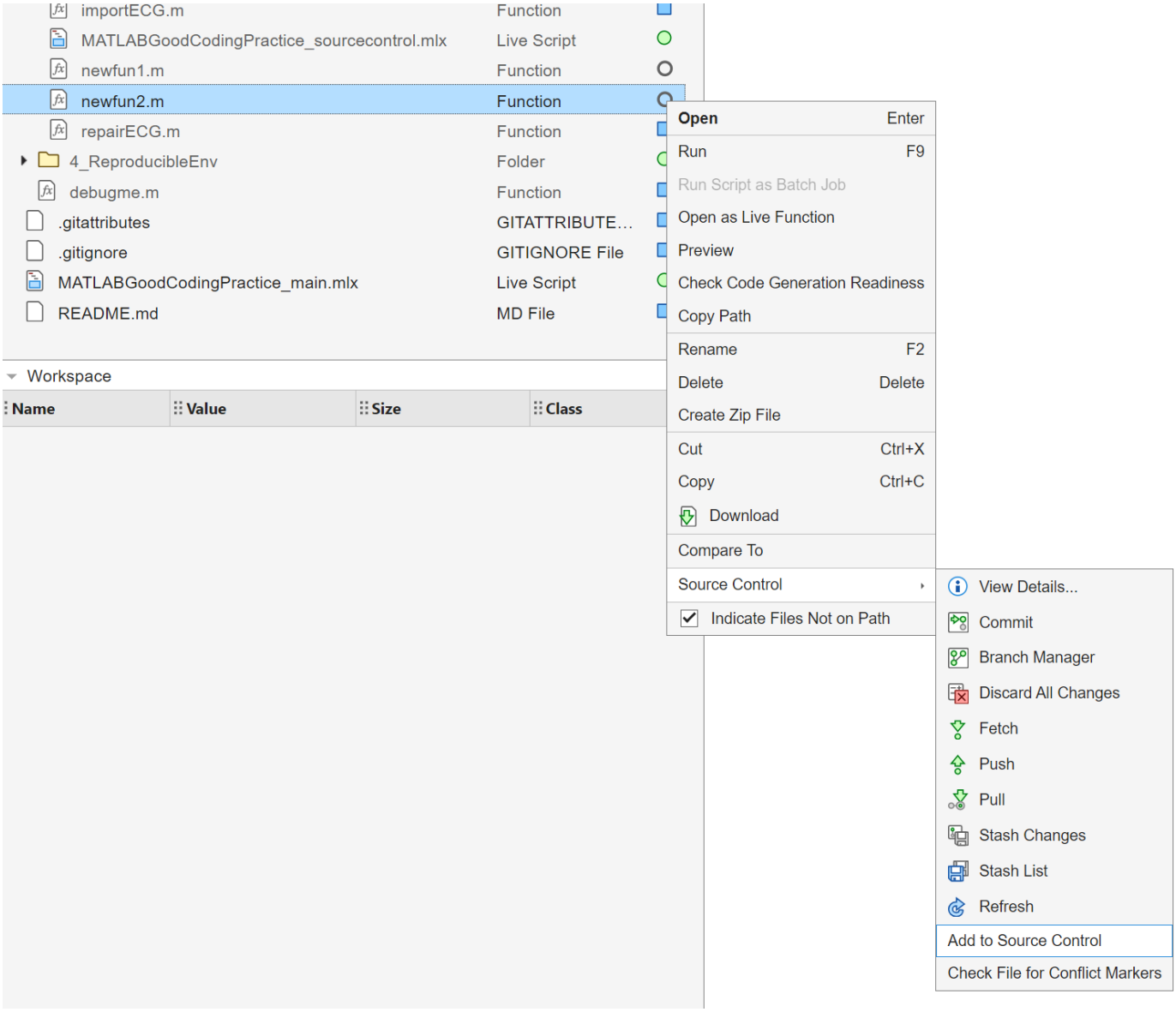


Note the markers next to the files in the Current Folder tab:

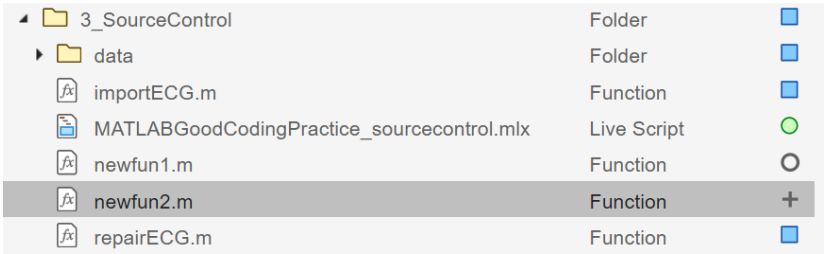


Here we have unstaged files (white), files that are in the repository with changes (blue) and without changes.

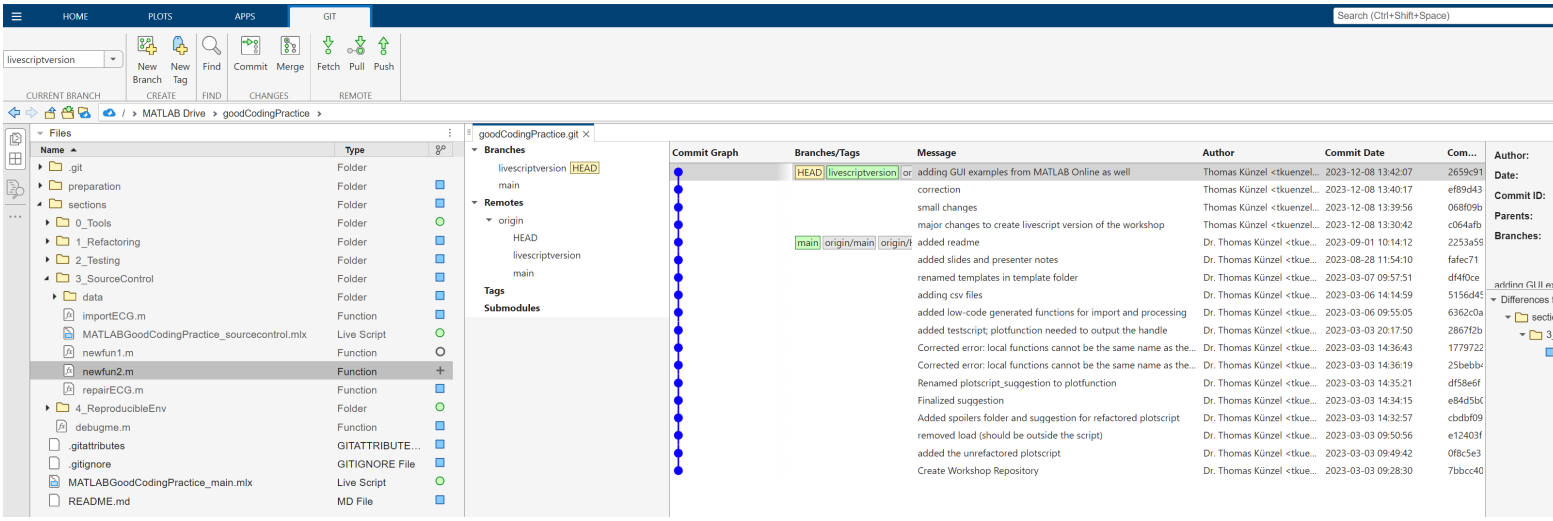
If you want to add one of the unstaged files to future commits: right-click - Source Control - Add in Source Control



Once it is staged (but not committed) the file gets a new marker:



The branch manager opens as a new graphical tab in MATLAB Online. The controls to create, checkout and merge branches are moved into the GIT ribbon on top.



An extended git exercise/code-along with branching

We will now, building upon our plotfunction example from the refactoring and testing sections, develop an exemplary git repository. Please follow along while I go through the exercise. And please interrupt me at any moment I am unclear, fast or slow!!

Step 1.: Setup and initial commit of our repository

Step 1.1: create a new folder

Step 1.2: initialize an empty git repository in this folder

```
% make a new folder
% cd into the new folder
gitinit % or use the GUI
```

Step 1.3: copy plotfunction.m and plotfunctionTest.m and the data folder into the new folder

Step 1.4: add the files to Source Control

Step 1.5: make initial commit

https://xkcd.com/1296/

Next we want to create an import-function and a preprocess function to be able to load the csv-files in the data folder and correct some artifacts in the data. We will either develop these from low-code tools or copy from the workshop folder "3_SourceControl" in case we are short on time. In any event, lets make several commits. so we have something to work with.

Step 2.1: create or copy import function

Step 2.2: add import function to source control

Step 2.3: commit import function

Step 3.1: create or copy repair function

Step 3.2: add repair function to source control

Step 3.3: commit repair function

Step 4.1: call repair function in import function

Step 4.2: commit import function

We should now run the test(s) again and make sure everything works. Let us check the log:

```
repo = gitrepo;
log(repo)
```

Let us now create a branch called "feature" based on the last commit and demonstrate how development can move on in different branches.

Step 5.1: Use the BranchManager (Right-Click --> Source Control --> Branches) to create a branch at the current commit, call it "feature".

Step 5.2: in the "main" branch - make cosmetic change to plotfunction.m to demonstrate divergent development

Step 5.2b (optional): If you want to "experience" a merge conflict, change the title of the first subplot in line from "All data" to something else, maybe "Complete Recording".

Step 5.3: commit changed plotfunction to main branch

Step 6.1: switch to "feature" branch

Step 6.2: note that the cosmetic changes to plotfunction are gone!

Step 6.3: develop or copy heartRateEstimator function

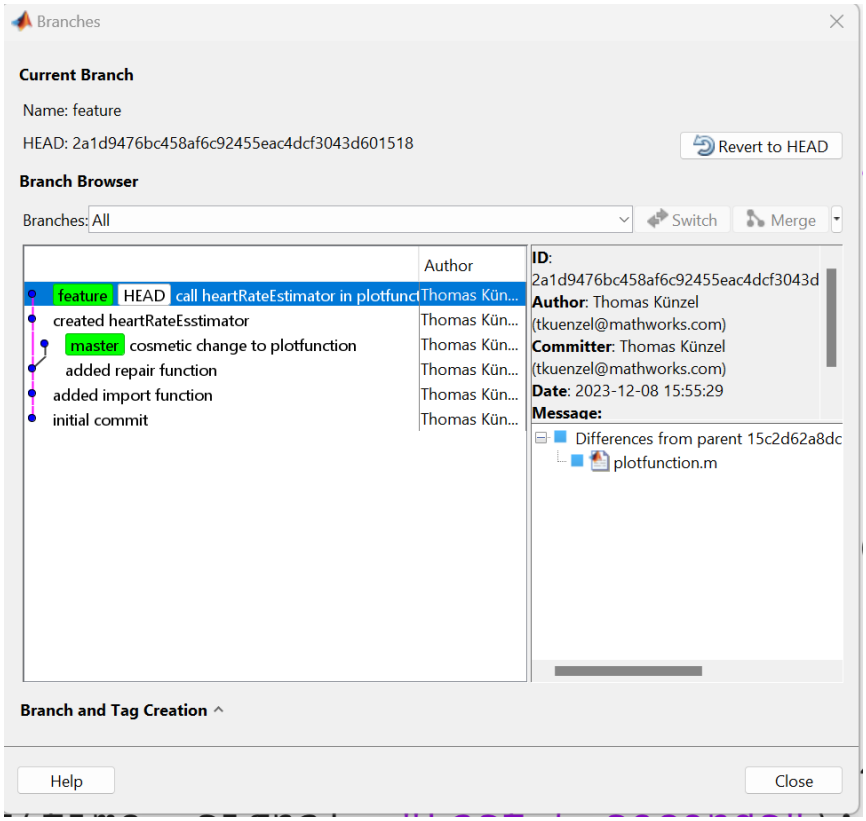
Step 6.4: add heartRateEstimator function to source control

Step 6.5: commit heartRateEstimator to "feature" branch

Step 7.1: call heartRateEstimator in the plotfunction and add result to figure by using the estimated heartrate in the title of the first subplot

Step 7.2. commit modified plotfunction to "feature" branch

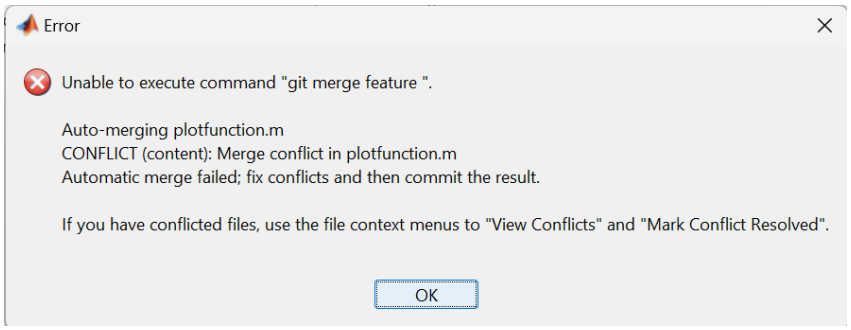
The project looks like this now:



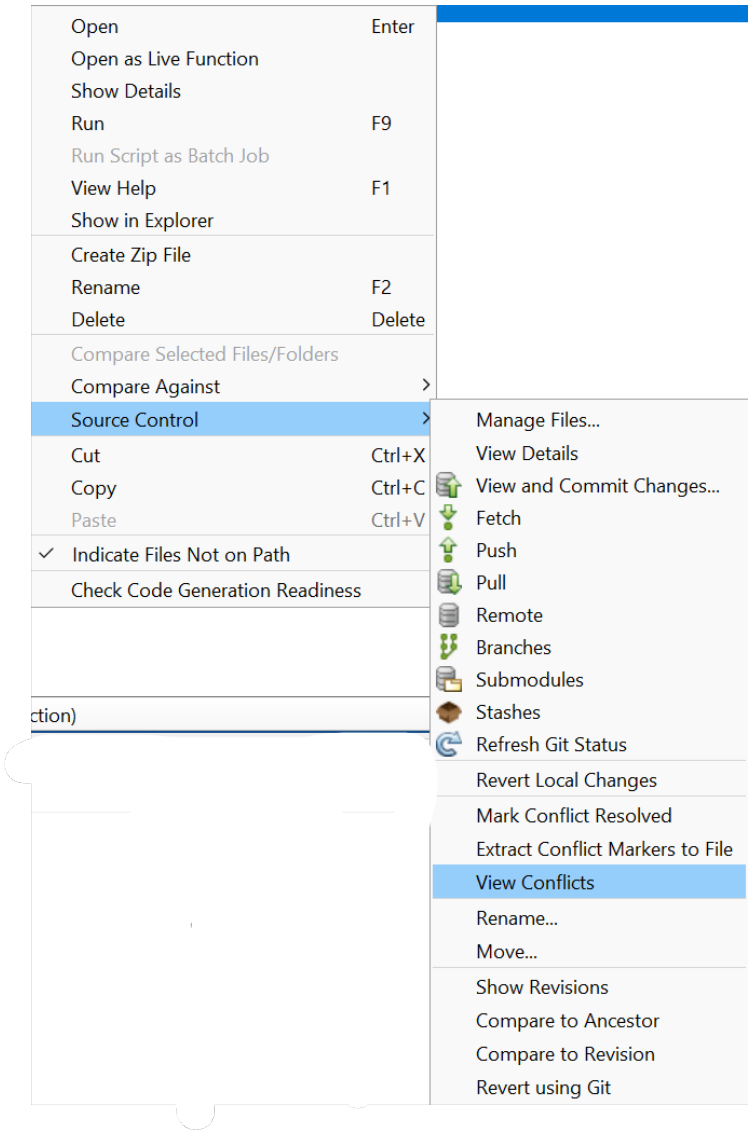
When you now switch back to the "main" branch, all your work is gone! Panic? No, remember git recreates the working directory as it was when the commit was performed.

Step 8.1: merge "feature" branch into "main" branch. If there are no "conflicts", this will be automatic.

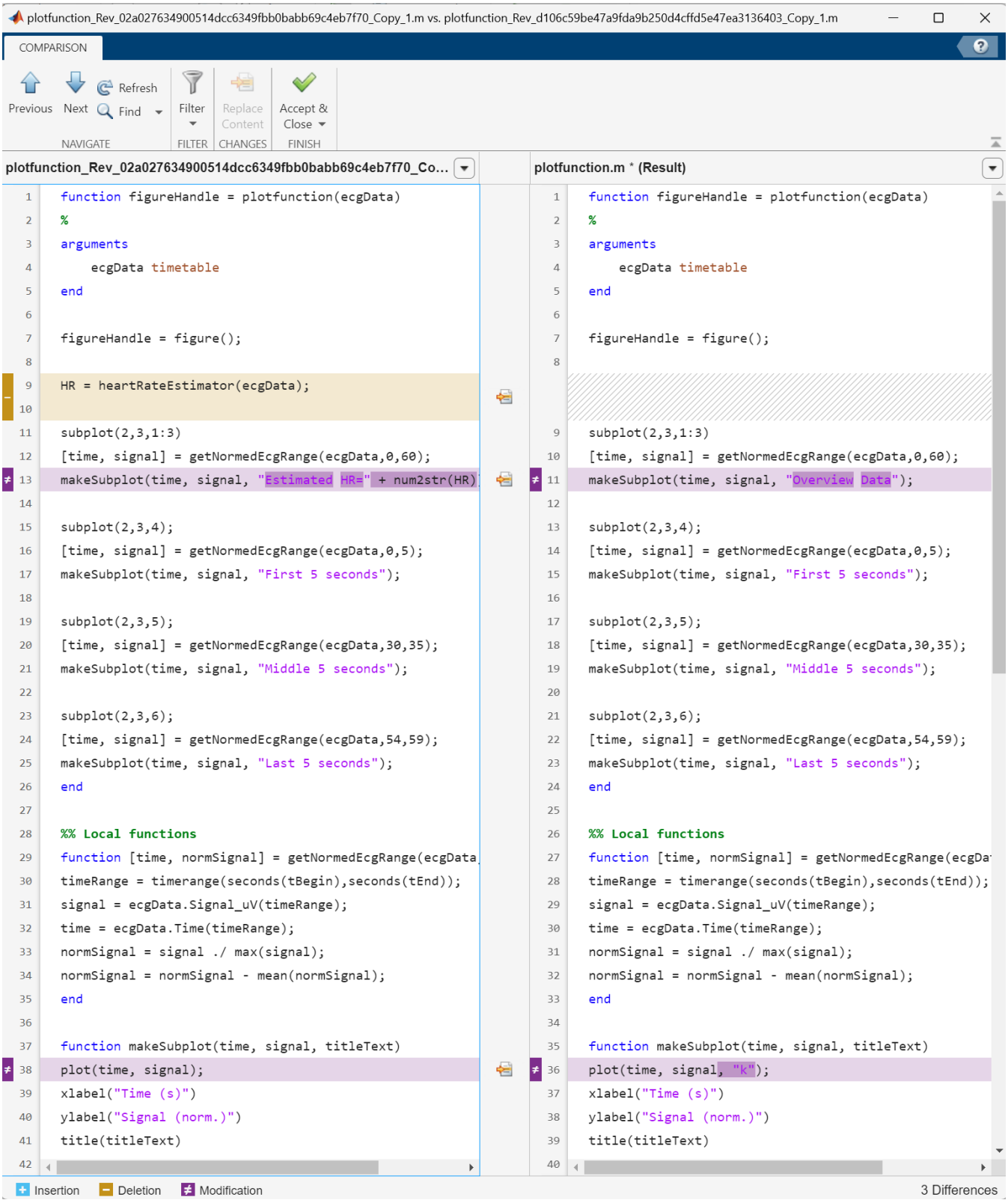
(optional Step 8.1b: resolve conflicts with the diff tool)



The merge has not been completed. To keep on working you first have to resolve the conflicts and complete the merge, so a "merge commit" can be made.

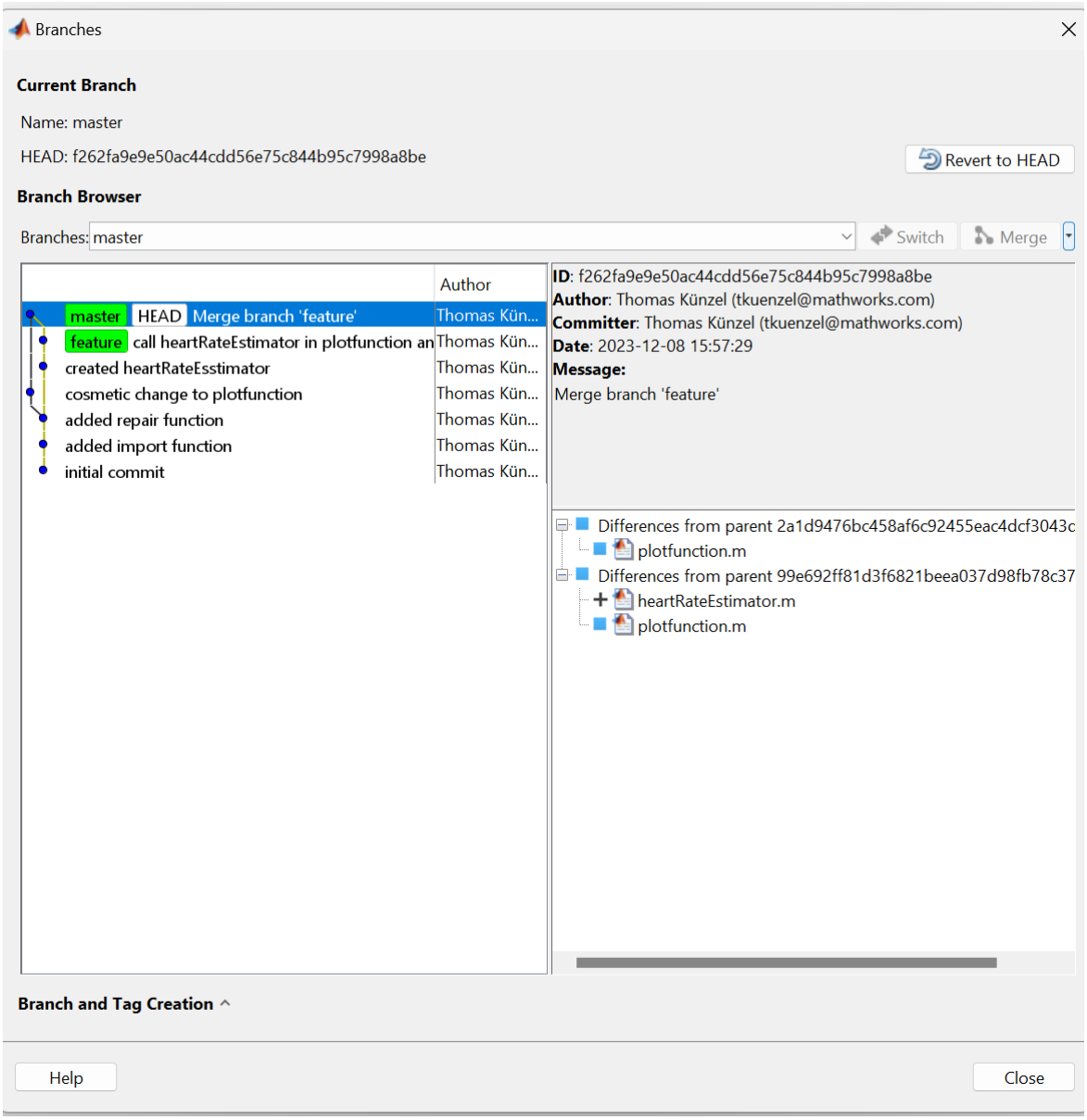


The diff-tool provides a convenient environment to resolve merge conflicts. Just pick the changes from left (which is the branch to be merged) or not (keep the version in the target branch). In our case we will take the first two from the left (click on the orange arrow icon in the middle) but not the third.



Then click on "Accept & Close". To complete the merge, now make a commit - the message is prefilled for you this time.

Done! The project should now look similar to this...



...and incorporate both the cosmetic changes and the heartrate estimation now!

By the way: you can always use the "Source Control - Compare to Revision" functionality to compare different version of a file in your repo. This is the MATLAB diff-tool, which is also used to resolve merge conflicts. This can also render changes in live-scripts.

Workshop Section 4: Achieving Reproducible Code Environments With MATLAB Projects

Why Do We Care about Reproducible Environments?

Sometimes having the source code of a project is not at all sufficient to run it. Code can depend on (specific versions of...) third-party toolboxes, expect certain environment variables or paths to be set and have prerequisites of when and how to run the code that are not immediately transparent.

This is a hard problem that many different tools try to address and is, among other things, a reason why software-containers like Docker, FlatPak or Snap are popular.

What Are Projects?

A "project" (in the context of our software platform) is a scalable environment where you can manage MATLAB files, data files, requirements, reports, spreadsheets, tests, and generated files together in one place.

Projects can help you organize your work and collaborate. Projects promote productivity and teamwork by helping you with common tasks.

- Find all the files that belong with your project.
- Create standard ways to set up and shut down the MATLAB environment across a team.
- Create, store, and easily access common operations.
- View and label modified files for peer review workflows.
- Share projects using built-in integration with Git™, Subversion® (SVN), or using external source control tools.

A big example project (Documentation)

The MATLAB Documentation contains an involved example project: **Please explore this to learn about advanced features of Projects**.

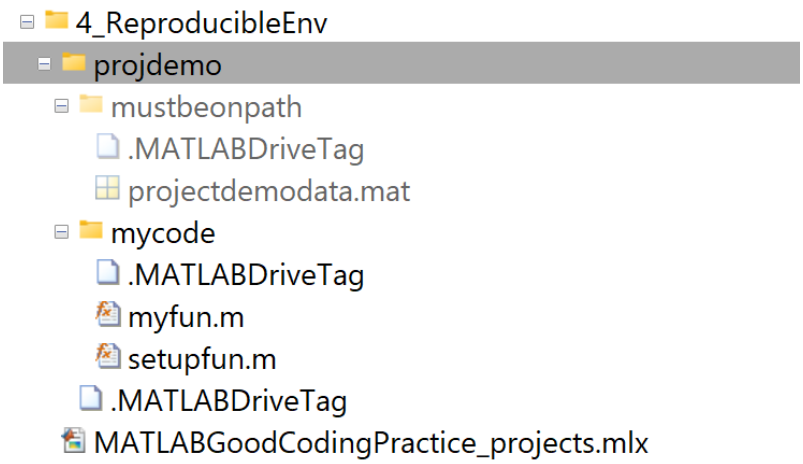
There is a nice video linked in the documentation that introduces the collaboration features of Projects: **watch it here on YouTube**.

Here we will just try to highlight a few points in a...

Small example project

In this rather artificial example the function myfun.m in ./projdemo/mycode has a few prerequisites.

For demonstration purposes make sure that "mustbeonpath" is *not* on the path. It should look like this:

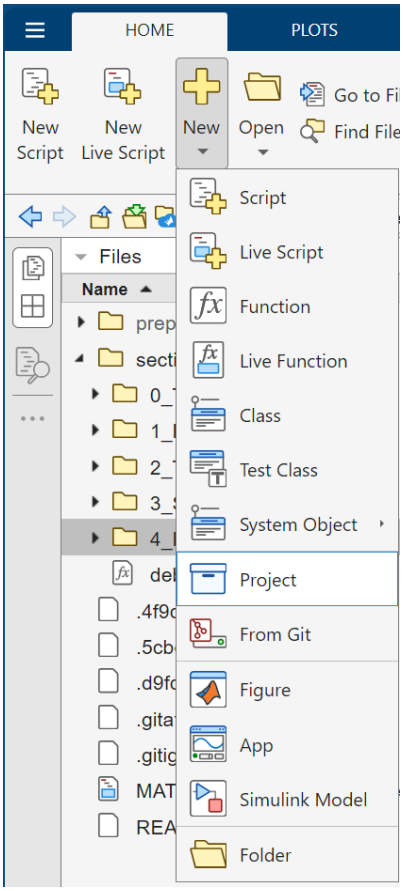



```
myfun
```

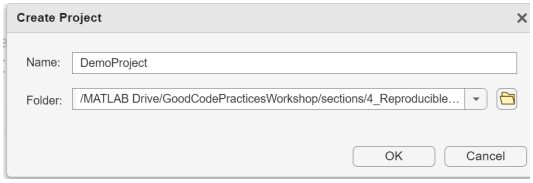
The function fails with errors, because the code environment it expects is not present.

Create Project

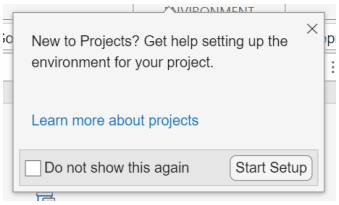
In the Home Tab choose New - Project



Because we already have files we pick the "projdemo" subfolder als Project Root Folder

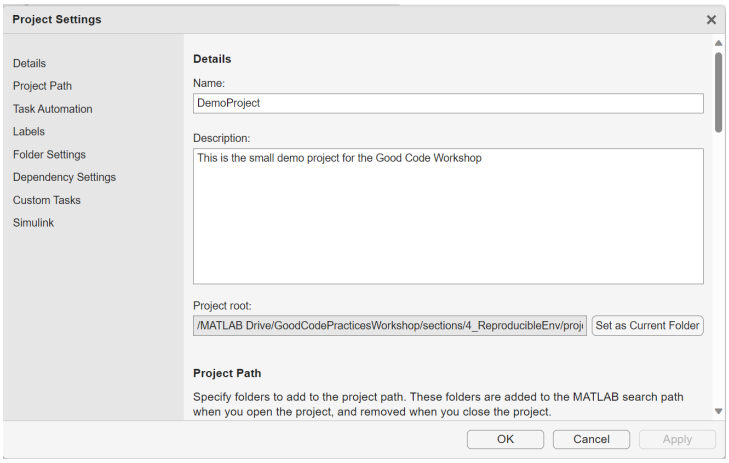


We could now follow the Setup-Wizard

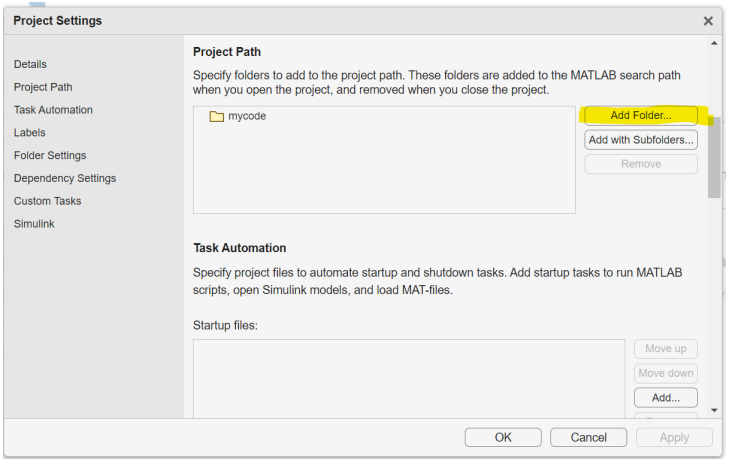


You can also **skip the Wizard** and **open the Project Settings**.

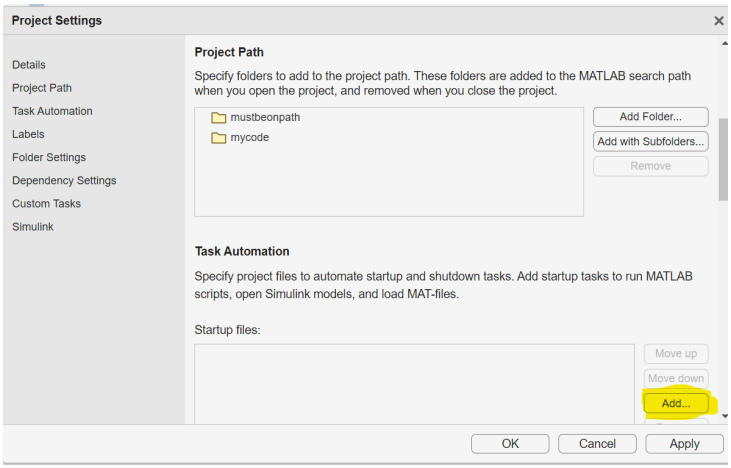
In the **Details Tab** you can write a good description or change the Project Root Folder.



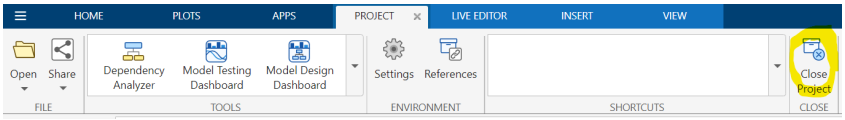
Under the **Project Path** Tab we make sure, that the "mustbeonpath" folder is added to the list of Project Path Folders.



And we add the "setupfun.m" function from the mycode subfolder as a startup-function under **Task Automation**.

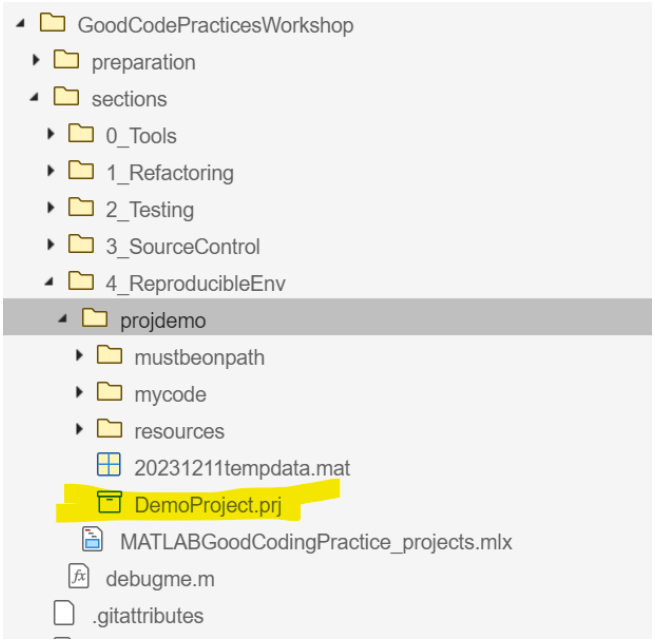


That is all for now. You can apply the changes and close the project.



Please note, that the Project-capabilities on MATLAB Online are limited. For the full version use MATLAB Desktop.

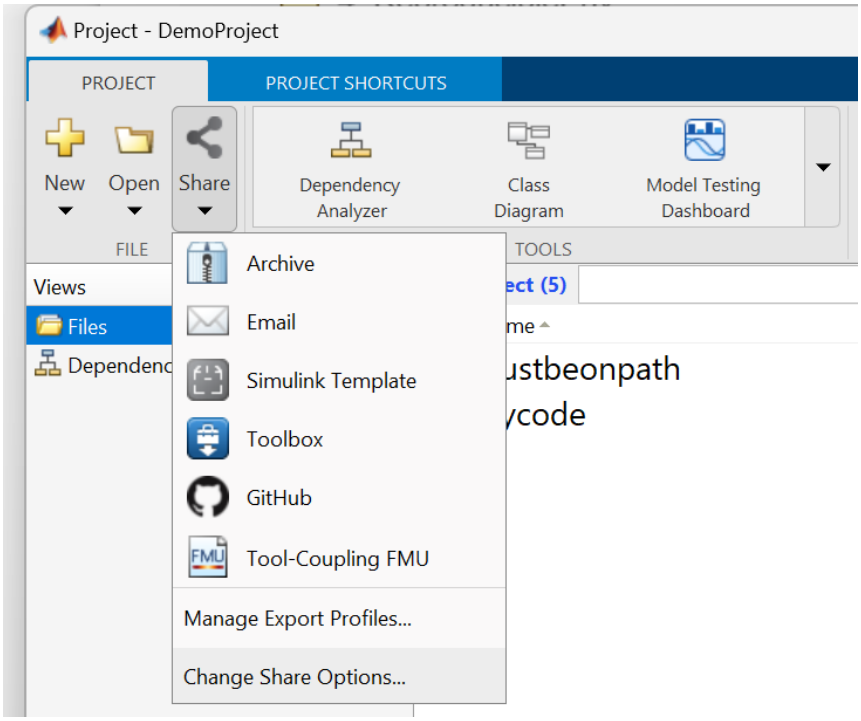
Now **reopen the project** by double-clicking the .prj file



MATLAB automatically takes you to the Project Root Folder, sets up the path and runs the custom startup function. We should be ready to go now!

```
myfun
```

Everyone you **share your project with** can setup the environment with one click now!


Project - DemoProject
PROJECT
PROJECT SHORTCUTS
New
Open
Share
Dependency Analyzer
Class Diagram
Model Testing Dashboard
FILE
TOOLS
Views
Files
Archive
Email
Simulink Template
Toolbox
GitHub
Tool-Coupling FMU
Manage Export Profiles...
Change Share Options...