Требования и рекомендации по оформлению кода в MATLAB

платформенно

Статус документа	Ожидает подтверждения	
Дата утверждения		
Дата последнего изменения		
Наблюдатели	Galim Mahmutov Anton Gromov Denis Odinokov Nikita Ivlev Alexander Starchenko Ilya Akimov Stanislav Koval Неизвестный пользователь (g.kuznetsov) Alexander Harlan Dmitry Pritykin Mikhail Skrob Maxim Koretskiy Неизвестный пользователь (m.moiseev) Sofiya Vlasova Tagir Sadretdinov Неизвестный пользователь (a.prokopeva) Oleg Mukukinov Неизвестный пользователь (r.vishnyakov) Veronika Galaeva Dmitry Glotin	
Владелец документа	Denis Odinokov	
Подтверждение актуальности	Denis Odinokov	СОГЛАСОВАНО 18.10.21
	Alexander Harlan	СОГЛАСОВАНО 18.03.22

В данном документе сформулированы требования по стилю кода в MATLAB и даны рекомендации относительно структуризации и оформления. Пункты отмечаются как требования и рекомендации:

- Требования являются обязательными к выполнению правилами. Требования отмечаются в этом документе как **(Т)** и могут иметь исключения.
- Рекомендации являются необязательными к выполнению советами по написанию кода. Рекомендации отмечаются в этом документе как **(Р)** и не должны трактоваться как требования.

Требования, описанные в документе, является актуальными только для написания нового кода и рефакторинга уже имеющегося. Не предполагается переписывание всего имеющегося кода в соответствии с требованиями. Предполагается, что в процессе рецензирования (review) код проверяется на соответствие с этим документом.

0. Зачем это нужно?

Мартин Фаулер писал: *"Любой может написать код, понятный для компьютера. Хороший программист пишет код, понятный для человека."*

Требования к оформлению кода необходимы по простой причине того, что код чаще читается, чем пишется. Аккуратный код не только помогает читателю быстрей вникнуть в суть алгоритма, но и способствует более удобному и быстрому поиску багов. Очень важно придавать значение оформлению кода с самого начала, поскольку аккуратно оформить код сразу проще, чем разбираться в плохо написанном. Скорость набора кода менее приоритетна, чем его понятность.

1. Имена

Для именований приняты следующие общие правила и рекомендации:

- **(Т)** Название любых переменных, полей классов и структур, функций необходимо писать с маленькой буквы, используется "lowerCamelCase". Пример: *pointArray, satCount.* Исключениями являются случаи, где удобство использования однобуквенных наименований переменных превышает удобство чтения кода, например в математических функциях и процедурах, где область видимости переменной ограничена одной функцией/процедурой.
- **(Т)** Название классов, пакетов и проектов (но не имени объектов) необходимо писать с большой буквы, используется "CamelCase". Пример: *AbstractMap.*
- **(Т)** Название папок внутри проекта необходимо писать с маленькой буквы, используется "lowerCamelCase".

- **(T)** Название веток в git-репозитории необходимо писать на английском языке (нужно для Jenkins) с маленькой буквы, используется "kebab-case". Также в начало названия ветки необходимо помещать идентификатор задачи из Jira, если релевантно. Пример: *KSMODEL-330-ico-from-hex.*
- **(T)** Название любых файлов (кроме файлов с определениями классов) пишется с использованием "lowerCamelCase".
- **(T)** Названия не должны содержать в себе транслитерацию.
- **(P)** Чем больше область видимости сущности (функции, переменной, объекта), тем более подробным и понятным должно быть у нее имя.

1.1. Имена переменных

1.1.1. (T) Для переменных, имеющих значение количества, требуется использовать постфикс "count". Примеры: *satCount, epochCount.*

1.1.2. (T) Акронимы, даже если они обычно пишутся в верхнем регистре, требуется писать в смешанном (первая буква заглавная, остальные строчные) или в нижнем регистре. Следует писать: *html, isUsaSpecific, checkTiffFormat().* Не следует: *hTML, isUSASpecific, checkTIFFFormat().* Аббревиатуры следует интерпретировать как отдельные слова.

1.1.3. (T) Требуется избегать ключевых или специальных слов. В MATLAB присутствует список зарезервированных слов, которые не стоит использовать для любых названий:

```
break, case, catch, classdef, continue, else, elseif, end, for, function, global, if, otherwise, parfor,
persistent, return, spmd, switch, try, while
```

Актуальный список зарезервированных слов можно узнать с помощью команды *iskeyword()*.

1.1.4. (T) Следует избегать переопределения названий встроенных в MATLAB и тулбоксы функций. Есть некоторые имена, которые уже используются MATLAB'ом и их использование для других целей может привести к ошибкам и багам. Например, эти имена уже используются в MATLAB и их следует избегать:

```
alpha, angle, axes, axis, balance, beta, contrast, gamma, image, info, input, length, line, mode, power, rank,
run, start, text, type
```

Использование стандартных, широко известных, слов в качестве имени переменных снижает читаемость. Если есть желание, например, использовать стандартное слово *length* в имени переменной, следует добавить префиксы или постфиксы: *pointArrayLe ngth*, *trajectoryLengthMeters*. См. пункт 1.2.11. относительно проверки наличия функций с определенным именем.

1.1.5. (P) Имена переменных должны однозначно декларировать их назначение. Не рекомендуется использовать имена переменных из одной буквы, существенные сокращения (кроме общепринятых). Длинное, но более понятное имя переменной всегда лучше, чем короткое.

Такой код заставляет читателя тратить время и разбираться в смысле каждой переменной:

```
sc = sp * np;
```

Такой код помогает читателю понять смысл каждой переменной без комментариев:

```
satCount = satPerPlane * planeCount;
```

Допускается использование коротких и однобуквенных имен переменных в математических выражениях. Рекомендуется описывать в комментариях назначение переменных в случае использования коротких имен.

1.1.6. (P) Рекомендуется использовать одинаковый подход к множественному числу. Рекомендуется избегать ситуаций, при которой имена двух переменных отличаются только окончанием *s.* Так именовать переменные не следует:

```
point = points(1,:);
```

Допустимым указанием на множественное число являются суффиксы *Array* или *List.* Для уточнения, что переменная несет в себе только одно значение, можно использовать, например, префикс *this:*

```
thisPoint = pointArray(1,:);
```

1.1.7. (P) Для циклов for не рекомендуется использовать в качестве счетчиков цикла буквы *i, j, k*. Название счетчика цикла должно быть информативным, это помогает не запутаться, особенно при наличии вложенных циклов. Удобным вариантом является постфикс *idx* (сокращение от *index)* после осмысленного имени переменной:

```
for epochIdx = 1:epochCount
        for satIdx = 1:satCount
                bar = foo(epochIdx, satIdx);
        end
end
```

1.1.8. (P) Для булевых переменных рекомендуется использовать префикс *is.* Рекомендуется избегать имен с отрицанием для булевых переменных. Выражение *~isNotFound* менее понятно, чем *isFound* или *~isFound*.

1.1.9. (P) Рекомендуется использовать общепринятые для данной технической сферы аббревиатуры. Правильно: *roi* или *regionOf Interest*. Неправильно: *mapRegionForAnalysis*. Правильно: *ecefPosition*, неправильно: *earthFixedPosition*. Список принятых сокращений можно найти в глоссарии.

1.1.10. (P) Имя структуры не должно включаться в имена полей. Повторение имени излишне и снижает читаемость. Пример: s*eg ment.segmentArea* не так удобно как *segment.area.*

1.2. Имена функций

1.2.1. (P) Функции пишутся с маленькой буквы в нижнем регистре или "lowerCamelCase". Все стандартные функции MATLAB пишутся в нижнем регистре, однако при реализации своих функций следует придерживаться написания в "lowerCamelCase".

1.2.2. (P) Рекомендуется давать функциям осмысленные имена. Короткие имена некоторых стандартных функций MATLAB могли быть обусловлены ограничениями DOS в 8 символов. Сейчас таких ограничений нет и главный приоритет отдается читаемости кода. Пример: *computeTotalWidth* более понятно, нежели *compwid*. В пунктах 1.2.3-1.2.11 даются рекомендации относительно именований функций в зависимости от назначения функции.

1.2.3. (P) При единственном выходном аргументе имя функции должно указывать на его единственность. Примеры: *calcMeanAngle* , *calcStandartError*.

1.2.4. (P) Функции (процедуры), не возвращающие аргументы, не должны вводить в заблуждение своим названием. Название должно объяснять, что функция должна (и, возможно, что не должна) делать. Примеры: plotPieChart, showBestPoint.

1.2.5. (P) Префиксы *get/set* рекомендуется использовать только для доступа к объекту или его свойства. Это общая практика в MATLAB и в других языках программирования. Методы, включающие в себя префикс *get* не должны менять состояние объекта. Примеры: *getObj*, *setParams*.

1.2.6. (P) Префикс *find* рекомендуется использовать для функций, где что-то отыскивается (без серьезных вычислений). Подобн ый подход помогает читателю понять, что в функции используется простой метод поиска с минимумом вычислений. В большинстве случаев рекомендуется использовать *find* вместо чрезмерно используемого *get*. Примеры: *findLowestPoint*, *findImage Idx*.

1.2.7. (P) Префикс *calc* должен использоваться только для функций или методов, где происходят вычисления (но не обязательно должен использоваться именно *calc*). Одинаковый подход к использованию префиксов улучшает читаемость кода. Следует избегать путаницы с префиксами *find* и *make*. Примеры: *calcAverage*, *calcSpread*.

1.2.8. (P) Префикс *init* рекомендуется использовать для функций, где инициализируются объекты или переменные. Пример: *initPr oblemState*.

1.2.9. (P) Префикс *is* рекомендуется использовать для булевых функций, так же как и для булевых переменных. Примеры: *isCom plete*, *isOverpriced*. В некоторых случаях префиксы *has, can, should* будут более подходящими: *hasLicense, canEvaluate, shouldSort* . (Подумать про разницу между булевыми функциями и переменными).

1.2.10. (P) Рекомендуется использовать симметричные имена для симметричных функций. Симметрия помогает читаемости. Примеры: *get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up /down, min/max, next/previous, open/close, show/hide, suspend/resume* и другие.

1.2.11. (P) Следует давать функциям уникальные (по возможности) имена. Две и более функций с одинаковыми именами увеличивают вероятность непредвиденного поведения программы или ошибки. В случае сомнений рекомендуется проверять название функций с помощью команд *which -all* или *exist*. Например, проверить наличие функций с именем *mean* можно командами *which mean -all* или *exist mean.*

1.2.12. (P) Рекомендуется в имени функции использовать глагол (префикс с глаголом) для отличия функции от переменной.

1.3. Разное

1.3.1. (T) Использовать платформенно независимые решения, то есть независящие от операционной системы. Использовать "filesep" или "/" вместо "\". Не использовать .exe .dll и так далее.
Для замены "filesep" ещё лучше пользоваться: либо projPath() из matlab-models; либо встроенной Matlab функцией path().

1.3.2. (P) Рекомендуется добавлять суффикс единицы измерения для переменных с размерностью, особенно, если используется не СИ. Добавление суффикса единицы измерения помогает избежать путаницы с единицами измерения. Пример: *inclinationAngle Degrees*, *positionKilometers*.

1.3.3. (P) Рекомендуется избегать использования неспецифичных аббревиатур в наименованиях. Использование полноценных слов помогает избежать неоднозначности и помогает коду быть самодостаточным для объяснения. Пример: *computeArrivalTime* гораздо понятнее, чем *comparr*. Однако специфичные акронимы и аббревиатуры, общепринятые в технической сфере, не стоит расшифровывать без необходимости (см. пункт 1.1.8).

1.3.4. (P) Рекомендуется давать легко произносимые имена. Название, которое легко произносится, легче читается и запоминается.

2. Структуризация кода

Структуризация как и кода внутри файлов, так и файлов в целом важна для удобства восприятия.

2.1. Модульность, классы и функции

2.1.1. (T) Функция, которая может использоваться только одним скриптом или функцией, должна быть оформлена в этом же файле.

2.1.2. (T) Функция всегда должна закрываться ключевым словом *end*.

2.1.3. (T) Все вызовы функций, процедур, методов без аргументов должны вызываться с использованием скобок (). Это дает понять, что вызывается именно функция, а не переменная. Исключениями являются некоторые встроенные в Matlab процедуры и функции, например *clc, clear, close all* и т.п.

2.1.4. (T) В определении всех методов класса, кроме статичных и конструктора, первым аргументом должен быть *this*.

2.1.5. (T) При вызове метода класса необходимо использовать синтаксис *obj.method(args)*, а не *method(obj, args)*, где *obj* – это название экземпляра класса, *method* – название его метода, а *args* – аргументы метода. Внутри класса следует писать this. method(args).

2.1.6. (P) Необходимо поддерживать модульность. Лучший способ написания большой программы – это её составление из небольших частей (функций и/или классов). Такой подход улучшает читаемость, понимание и тестирование, т.к. уменьшает количество текста, которое нужно прочесть для понимания логики кода. Файл с кодом, выходящий за пределы двух экранов редактора, является кандидатом для рассечения на части или превращения в класс.

2.1.7. (P) Необходимо поддерживать лаконичность интерфейсов функций и методов классов. Длинный список обязательных аргументов лучше заменить, например, структурой или классом. Для длинного списка необязательных аргументов рекомендуется использование *varargin*.

2.1.9. (P) Любой повторяющийся участок кода должен быть рассмотрен на предмет возможности оформления в отдельной функции или методе класса.

2.2. Переменные и константы

2.2.1. (T) Запрещается использование глобальных переменных, которые задаются ключевым словом *global*.

2.2.2. (P) Для лучшей читаемости рекомендуется использовать отдельные переменные для промежуточных результатов. Исключения возможны, когда важны производительность и ограничения памяти. В некоторых случаях рекомендуется очищать память от переменной сразу после ее использования. Пример:

```
intermediateResult = foo(arg);
variableArray = bar(intermediateResult);
clear intermediateResult;
```

2.2.3. (P) На использование *persistent*-переменных не налагается ограничений. Однако следует рассмотреть возможность реализации без *persistent*-переменных, но с использованием ООП.

2.3. Циклы

2.3.1. (T) Переменные, изменяемые в теле цикла, должны, когда это возможно, быть инициализированы перед циклом. Это улучшает производительность кода и способствует отладке. Пример:

```
state = zeros(epochCount, 3);
for epochIdx = 1:epochCount
        state(epochIdx, :) = foo(epochIdx);
end
```

2.3.2. (P) Для массивов структур, классов и т.п. рекомендуется использование инициализации через последний индекс с убывающим счетчиком в цикле. Пример:

```
for layerIdx = length(jsonStruct):-1:1
    demandMapLayers{layerIdx} = createLayer(jsonStruct{layerIdx});
end
```

2.3.3. (P) В многократно вложенных циклах рекомендуется использовать комментарии в строках с *end*. Добавление комментариев помогает определить, что происходит в каждом из циклов. Пример:

```
for simulationIdx = 1:simulationCount
    ...
    ...
    ...
    for epochIdx = 1:epochCount
        ...
        ...
        for satIdx = 1:satCount
            ...
        end %
        ...
    end %
    ...
    ...
end %
```

2.4. Условные операторы

2.4.1. (T) Выражение *switch* всегда должно включать в себя условие *otherwise*. Отсутствие *otherwise* может привести к непредсказуемым результатам, а его наличие способствует отладке.

```
switch type
        case TerminalTypes.household
                ...
        case TerminalTypes.establishment
                ...
        case TerminalTypes.backhaul
                ...
        otherwise
                error('  !');
end
```

2.4.2. (P) Следует избегать громоздких условных конструкций. Вместо этого следует объявлять временные переменные. Через определение переменных код документирует сам себя. Такой код следует переписать:

```
if (value >= lowerLimit) && (value <= upperLimit) && ~ismember(value, valueArray)
        foo(bar)
end
```

Такой код более понятен:

```
isValid = (value >= lowerLimit) && (value <= upperLimit);
isNew   = ~ismember(value, valueArray);

if (isValid && isNew)
        foo(bar)
end
```

2.4.3. (P) Предполагаемый путь выполнения условного выражения должен быть описание в секции *if*, а исключение (например, обработка ошибок и т.п.) должно быть в секции *else*. Такая практика улучшает читаемость, т.к. ставит на первых план нормальный ход исполнения программы. Важным исключением являются случаи, описанные в пункте 3.1.5.

2.4.4. (P) Между строками и перечислениями в качестве условной переменной для выражения switch рекомендуется выбирать пе речисления (см. enumeration). Пример объявления перечисления:

```
classdef TerminalTypes < int32
    enumeration
        household     (1)
        establishment (2)
        backhaul      (3)
    end
end
```

2.5. Общее

2.5.1. (T) Десятичные числа следует записывать с цифрой перед запятой. 0.5 более читаемо, чем .5, более того это исключает восприятие числа как целого.

2.5.2. (P) В случае любых сомнений относительно порядка выполнения операций, особенно в длинных выражениях, следует использовать скобки.

2.5.3. (P) Использование чисел в выражениях должно быть минимизировано. Параметры, которые могут быть изменены, должны быть переменными. Следует избегать:

```
satCount = 20 * 40;
```

Вместо этого следует писать:

```
satPerPlane = 20;
planeCount = 40;
satCount = satPerPlane * planeCount;
```

2.5.4. (P) Рекомендуется избегать прямого сравнения чисел с плавающей запятой. Прямое сравнение двух чисел с плавающей запятой может привести к некорректным результатам. Пример:

```
criteria = 1e-5;
if (abs(floatingPointNumber1 - floatingPointNumber2) <= criteria)
        foo(bar)
end
```

Такой код может привести к непредсказуемым результатам:

```
if floatingPointNumber1 == floatingPointNumber2
        foo(bar)
end
```

2.5.5. (P) Рекомендуется использовать встроенную функцию *error* в секциях условных операторов и switch, если необходимо обрабатывать ошибки. См. пример в пункте 2.4.3. и 3.1.4.

3. Форматирование и комментарии

Цель форматирования – помочь читателю понять код. Форматирование (особенно использование отступов) способствует структуризации кода.

3.1 Длина строк и отступы

3.1.1. (T) Размер отступа всегда равен 4 пробелам. Использовать символы табуляции запрещается. Нажатие клавиши Tab делает отступ, равный по умолчанию 4 пробелам. (см. Editor/Debugger Tab Preferences)

3.1.2. (T) Отступы необходимо всегда использовать в теле циклов, внутри операторов *if, else, switch,* в теле функций и методов. Примеры:

```
if isValid
    foo(bar)
else
    error("  !");
end
```

```
switch calcMethod
        case "foo"
                result = foo();
    case "bar"
        result = bar();
    otherwise
        error(" !")
end
```

```
for satIdx = 1:satCount
    satState(satIdx, :) = calcSatState(satIdx);
end
```

```
function resultName = awesomeFunctionName(greatArgumentName)
    firstResult  = foo(greatArgumentName);
    secondResult = bar(firstResult);
    resultName   = firstResult / secondResult;
end
```

Для автоматического отступа во всех функциях необходимо в настройках Matlab в пункте *Language* раздела *Editor/Debugger* напротив *Function indenting format:* в выпадающем списке выбрать *Indent all functions*.

3.1.3. (T) Строка кода должна содержать только одно исполняемое выражение.

3.1.4. (P) Крайне не рекомендуется превышать ограничение в 120 символов для одной строки кода. Исключения из правила возможны, однако слишком длинные выражения существенно ухудшают читаемость.

3.1.5. (P) Строки кода рекомендуется разделять в особых местах: после запятой или после оператора. Начало новой строки следует разместить вровень с началом выражения на предыдущей строке. Допустимые варианты:

```
layerParamsList{1} = LayerParams("General", ...
                                 "generalDefault", ...
                                 "none", ...
                                 generalDemandParams, ...
                                 terminalTypes, ...
                                 [1337, 1338, 1339]);
```

```
res = foo(...
        arg1, ...
        arg2, ...
        arg3, ...
);
```

```
totalLongExpressionSum =
        firstLongExpression(foo) + secondLongExpression(foo);
```

3.1.6. (P) Следует избегать многократно вложенного кода. Если код содержит более четырех отступов относительно начала строки, рекомендуется его переписать, исключения возможны для методов внутри классов. Также, если функция выполняется только при определенном условии, следует вынести тело функции из условного оператора, а в начале функции сделать проверку с возможным выходом или обработкой ошибки. Аналогичная рекомендация применима к циклам. Примеры:

```
function res = foo(structArg)
        if ~structArg.isValid
         error('   foo  !');
        end

    %   ,
    ...
    ...
    ...
    ...
    ...
    ...
end
```

```
parfor computePartIdx = firstIdx : lastIdx
    if valuesArray(computePartIdx) ~= 0
        continue
    end

    %   ,
    xVecefFace = xVecef(:, computePartIdx - shift);
    yVecefFace = yVecef(:, computePartIdx - shift);
    zVecefFace = zVecef(:, computePartIdx - shift);

    fCenter = mean([xVecefFace, yVecefFace, zVecefFace]);

    V1Ecef = [xVecefFace(1), yVecefFace(1), zVecefFace(1)];
    V2Ecef = [xVecefFace(2), yVecefFace(2), zVecefFace(2)];
    V3Ecef = [xVecefFace(3), yVecefFace(3), zVecefFace(3)];

    pthere = truncEcef(fCenter, epsMajor, ptableEcef);
    [idxsIn, ~] = getIdxsIn(pthere, V1Ecef, V2Ecef, V3Ecef);

    valuesArray(computePartIdx) = sum(mapColumnReduced(idxsIn));
    temp = zeros(size(ptableEcef, 1), 1, 'logical');
    temp(idxsIn) = 1;

    if ~isempty(idxsIn)
        found = found | temp;
    end
end
```

3.2. Пробелы и пустые строки

Пробелы улучшают читаемость кода выделяя отдельные компоненты выражений.

3.2.1. (T) Все операторы, не являющиеся арифметическими, следует выделять пробелами.

3.2.2. (T) Пробелы следует ставить после запятых. Пример:

```
result = function(firstArgument, secondArgument, thirdArgument)
```

3.2.3. (T) Если функция определяется не в собственном файле, то она должна быть обособлена от остального кода пустыми строками.

3.2.4. (P) Операторы рекомендуется выделять пробелами. Пример:

```
valueExample = ((firstExpression * secondExpression) + thirdExpression) / fourthExpression;
```

3.2.5. (P) Группы выражений, объединенных общей логикой, должны отделяться одной пустой строкой.

3.3. Выравнивание

Во многих случаях выравнивание кода улучшает читаемость.

3.3.1. (P) Блок, в котором присваиваются значения переменным, может быть выровнен относительно оператора присваивания. Пример:

```
this.configName               = configName;
this.demandParams             = demandParams;
this.terminalTypes            = terminalTypes;
this.terminalCountRngSeedArray = terminalCountRngSeedArray;
```

3.3.2. (P) Сложное выражение с повторяющимися элементами может быть выровнено относительно повторяющихся блоков с переносом строк. Пример:

```
weightedSum = (firstWeight  * firstElement) + ...
                         (secondWeight * secondElement) + ...
                         (thirdWeight  * thirdElement);
```

3.4. Комментарии

Цель комментариев – добавить информативности коду. Комментарии объясняют использование кода, добавляют ссылки на источники, оправдывают решения, описывают ограничения, отмечают необходимые улучшения. Лучше всего писать комментарии одновременно с кодом.

3.4.1. (T) Комментарии должны быть написаны на русском языке. Исключения могут быть для заимствованного кода и для цитат.

3.4.2. (T) Требуется удалять или переписывать устаревшие и неактуальные комментарии.

3.4.3. (T) Старый закомментированный код запрещено коммитить в репозиторий. История кода сохраняется в репозитории, а закомментированный код ухудшает читаемость.

3.4.4. (T) Комментарии должны легко читаться. Между % и комментарием должно быть не менее одного пробела, комментарий рекомендуется начинать с заглавной буквы.

3.4.5. (T) Комментарии должны иметь такой же отступ как и выражение, к которому они адресуются. Если комментарий пишется на той же строке, что и выражение, то перед знаком % необходимо ставить минимум один пробел.

3.4.6. (T) Если в функции используется varargin, в комментарии обязательно должны быть указаны и описаны все возможные аргументы функции.

3.4.7. (P) Комментарии не могут оправдать плохо написанный код. Комментарии не должны исправлять плохое именование и неявную логическую структуру. Такой код должен быть переписан, а не дополнен комментариями.

3.4.8. (P) Комментарии должны быть согласованы с кодом, но больше объяснять, чем пересказывать код. В комментарии важно пояснить, почему или как, нежели что именно происходит. Пример:

```
%
centersHandle = scatter3(this.spheroid.centers(:,1), ...
                         this.spheroid.centers(:,2), ...
                         this.spheroid.centers(:,3), ...
                         600, ...
                         'MarkerEdgeAlpha', ...
                         0);
```

3.4.9. (P) Комментарии к функции должны отражать любые специальные требования к аргументам. Пользователь должен понимать, какие типы, единицы измерения и форматы входных аргументов используются функцией. Рекомендуется в целом подробно описывать часто используемые функции, с описанием в том числе и выходных аргументов. Пример:

```
% vecAng -
% . , 30.08.2021
%
%   ang = vecAng(u, v)
%   ang = vecAng(u, v, dim)
%
% :
%   ang = vecAng(u, v, dim)
%
%       .
%         dim (  -
%     u,  3).

%   u  v -  ,     3
%       ,    dim    .
%
%   u  v    ,
%          .
%
%   u  v    ,     ,
%         u,  v,  "dim",  1.
%              ,
%         -  .
%
%  :
%   u, v -  ,      3,
%          .
%   dim (,   1) -   ,
%         .   ,  u  v
%           3.
%
%  :
%   ang -     . ,
%        dim  1,     -
%         u  v (   ,
%        ,
%         )
```

3.4.10. (P) Комментарии к функции должны отражать любые действия помимо присваивания выходного значения (если они имеются). Если функция, например, помимо возвращения значений, отрисовывает график, это должно быть указано в комментар ии или в названии функции.

4. Тестирование

Требования к написанию авто-тестов

4.1. (T) Одна функция/класс = один Test Case класс. Один Test Case класс = несколько Unit тестов как методы в этом классе.

4.2. (T) Класс Test Case обязательно наследуется от соответствующего класса фреймворка используемого языка программирования (Например, Testing Frameworks в Matlab).

4.3. (T) Название Test Case (класса теста) класс повторяет название функции/класса с большой буквы с добавкой "Test" на конце. Например, *MyFunTest, MyClassTest.*

4.4. (T) Один Unit тест должен проверять одну единицу функциональности. Не стоит писать длинные Unit тесты, лучше много маленьких.

4.5. (T) Название Unit теста должно отражать суть теста. Чтобы сразу было видно, где ошибка, если она будет выявлена. Избегайте сокращений.

4.6. (T) Необходимо в комментарии указать ссылку на источник эталонных значений.

4.7. (T) Unit тесты должны быть атомарные и изолированные. Порядок запуска не должен иметь значения.

4.8. (T) Файл с новым Test Case класс необходимо добавить в путь проекта. Иначе тест не будет запускаться на сервере автоматизированного тестирования.

4.9. (P) Проверять всегда лучше как можно больше значений, даже самые очевидные. Если вы пишете тест для класса, проверяйте все его поля с данными.

4.10. (P) Рекомендуется сформировать датахаб данных для тестов. То есть данные для тестов лучше хранить на сервере компании, а не скачивать от первоисточника при каждом запуске тестов.

4.11. (P) Тесты не должны обращаться к одному и тому же изменяемому ресурсу (файлу, массиву, т.д.).

4.12. (P) Тесты не должны создавать/изменять один и тот же ресурс (файл, массив, т.д.).

4.13. (P) Тест должен быть написан таким образом, чтобы тестировать ожидаемое поведение, логику, а не внутреннюю реализацию. Тревожный звонок, когда необходимо переписывать тесты при рефакторинге класса/функции или добавлении новой фичи. Тесты должен быть независим от внутренней имплементации и структуры (см. 10 эффективных практик реализации тестирования).

4.14. (P) Следовать методологии, изложенной в статье 10 эффективных практик реализации тестирования.