POLITECNICO
MILANO 1863

Orbital Mechanics
Module 1: MATLAB fundamentals & numerical integration of dynamical systems

Academic year 2020/21

Juan Luis GONZALO GOMEZ, Giacomo BORELLI, Camilla COLOMBO
Department of Aerospace Science and Technology

Contacts

- Camilla **Colombo**
  - Email: camilla.colombo@polimi.it
  - Meeting time: please arrange via email
  - Tel: 8352
- Juan Luis **Gonzalo Gomez**
  - Email: juanluis.gonzalo@polimi.it
  - Meeting time: please arrange via email
  - Tel: 8401
- Giacomo **Borelli**
  - Email: giacomo.borelli@polimi.it
  - Meeting time: please arrange via email
  - Tel: 8401

About me

2016: Beng, Università di Parma, Mechanical Engineering

2019: Meng, Politecnico di Milano, Space engineering

2019-present: Research Fellow, Politecnico di Milano, “SUNRISE project: design of ADR mission for mega constellations – *ESA OneWeb*”

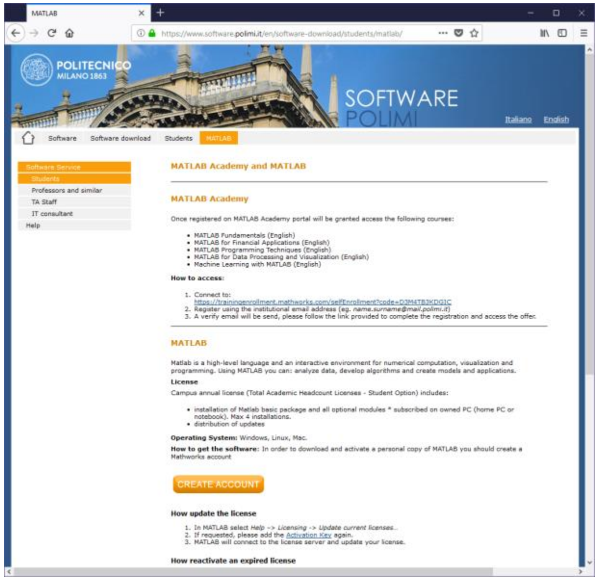
Research activities: Formation flying, active debris removal, proximity operations, GNC, low thrust trajectory design.

Before we begin…

- We will use **MATLAB** for the lab classes
  - You need to know the basics to follow the lab!

- **PoliMi** has a **Campus License**, giving access to the software and many resources:

  https://www.software.polimi.it/en/software-download/students/matlab



MATLAB Academy and MATLAB

MATLAB Academy

Once registered on MATLAB Academy portal will be granted access the following courses:

- MATLAB Fundamentals (English)
- MATLAB for Financial Applications (English)
- MATLAB Programming Techniques (English)
- MATLAB for Data Processing and Visualization (English)
- Machine Learning with MATLAB (English)

How to access:

1. Connect to: https://trainingenrollment.mathworks.com/selfEnrollment?code=D3M4TB3KDG1C
2. Register using the institutional email address (eg. *name.surname@mail.polimi.it*)
3. A verify email will be send, please follow the link provided to complete the registration and access the offer.

MATLAB

Matlab is a high-level language and an interactive environment for numerical computation, visualization and programming. Using MATLAB you can: analyze data, develop algorithms and create models and applications.

License

Campus annual license (Total Academic Headcount Licenses - Student Option) includes:

- installation of Matlab basic package and all optional modules * subscribed on owned PC (home PC or notebook). Max 4 installations.
- distribution of updates

Operating System: Windows, Linux, Mac.

How to get the software: In order to download and activate a personal copy of MATLAB you should create a Mathworks account

CREATE ACCOUNT

How update the license

1. In MATLAB select *Help -> Licensing -> Update current licenses…*
2. If requested, please add the Activation Key again.
3. MATLAB will connect to the license server and update your license.

How reactivate an expired license

Module Contents

MATLAB fundamentals & numerical integration of dynamical systems

- **MATLAB fundamentals**
  - MATLAB learning resources
  - Code structuring
  - Best coding practices
  - Figures
- **Numerical Integration of Dynamical Systems**
  - Ordinary Differential Equations
  - Numerical resolution of ODEs
  - **Exercise 1:** The underdamped harmonic oscillator
  - **Exercise 2:** Orbital dynamics
- **Root-finding**
  - **Exercise 3:** Kepler’s equation

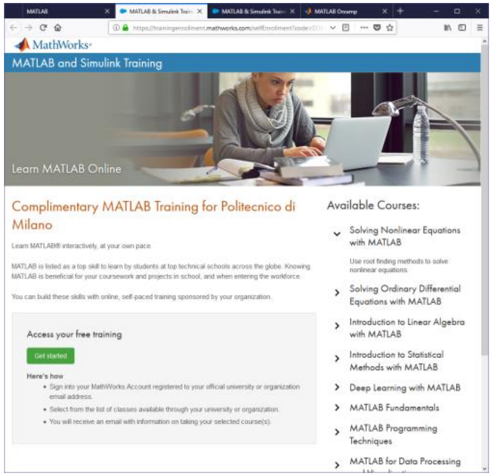
MATLAB FUNDAMENTALS

MATLAB learning resources

Mathworks online courses

Visit the courses in MATLAB Academy (included in the Campus license) to improve your MATLAB competencies

https://trainingenrollment.mathworks.com/selfEnrollment?code=D3M4TB3KDGIC



Complimentary MATLAB Training for Politecnico di Milano

Learn MATLAB® interactively, at your own pace.

MATLAB is listed as a top skill to learn by students at top technical schools across the globe. Knowing MATLAB is beneficial for your coursework and projects in school, and when entering the workforce.

You can build these skills with online, self-paced training sponsored by your organization.

Access your free training

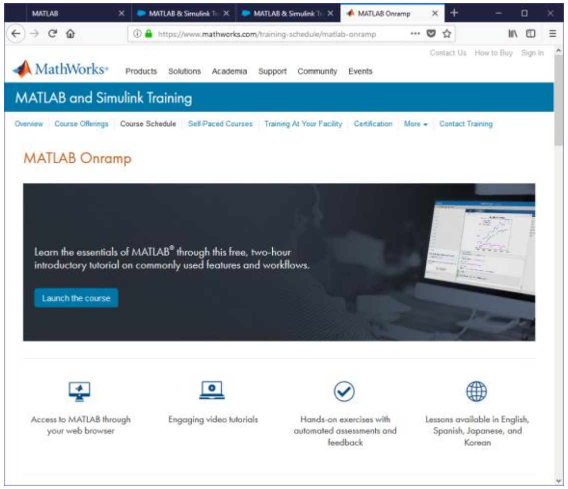
Here's how

- Sign into your MathWorks Account registered to your official university or organization email address.
- Select from the list of classes available through your university or organization.
- You will receive an email with information on taking your selected course(s).

Available Courses:

- Solving Nonlinear Equations with MATLAB

  Use root finding methods to solve nonlinear equations
- Solving Ordinary Differential Equations with MATLAB
- Introduction to Linear Algebra with MATLAB
- Introduction to Statistical Methods with MATLAB
- Deep Learning with MATLAB
- MATLAB Fundamentals
- MATLAB Programming Techniques
- MATLAB for Data Processing



MATLAB Onramp

Learn the essentials of MATLAB® through this free, two-hour introductory tutorial on commonly used features and workflows.

Access to MATLAB through your web browser

Engaging video tutorials

Hands-on exercises with automated assessments and feedback

Lessons available in English, Spanish, Japanese, and Korean

For those with little or no MATLAB experience, it is **strongly recommended** to follow the **MATLAB Onramp** online course (approx. 2 hours)

https://www.mathworks.com/training-schedule/matlab-onramp

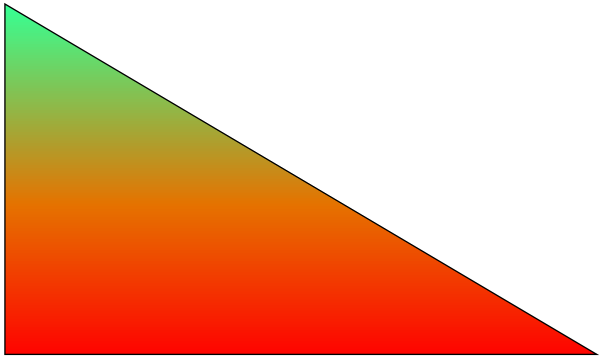
MATLAB learning resources

MATLAB functions: too many things to remember

MATLAB has:

- **Thousands** of included functions
  - Most of them with multiple inputs and outputs
    - To be provided in a given way (can affect behaviour)
    - Many are optional, with pre-set default values

Effort to remember it all:



Impossible to remember/know them all!

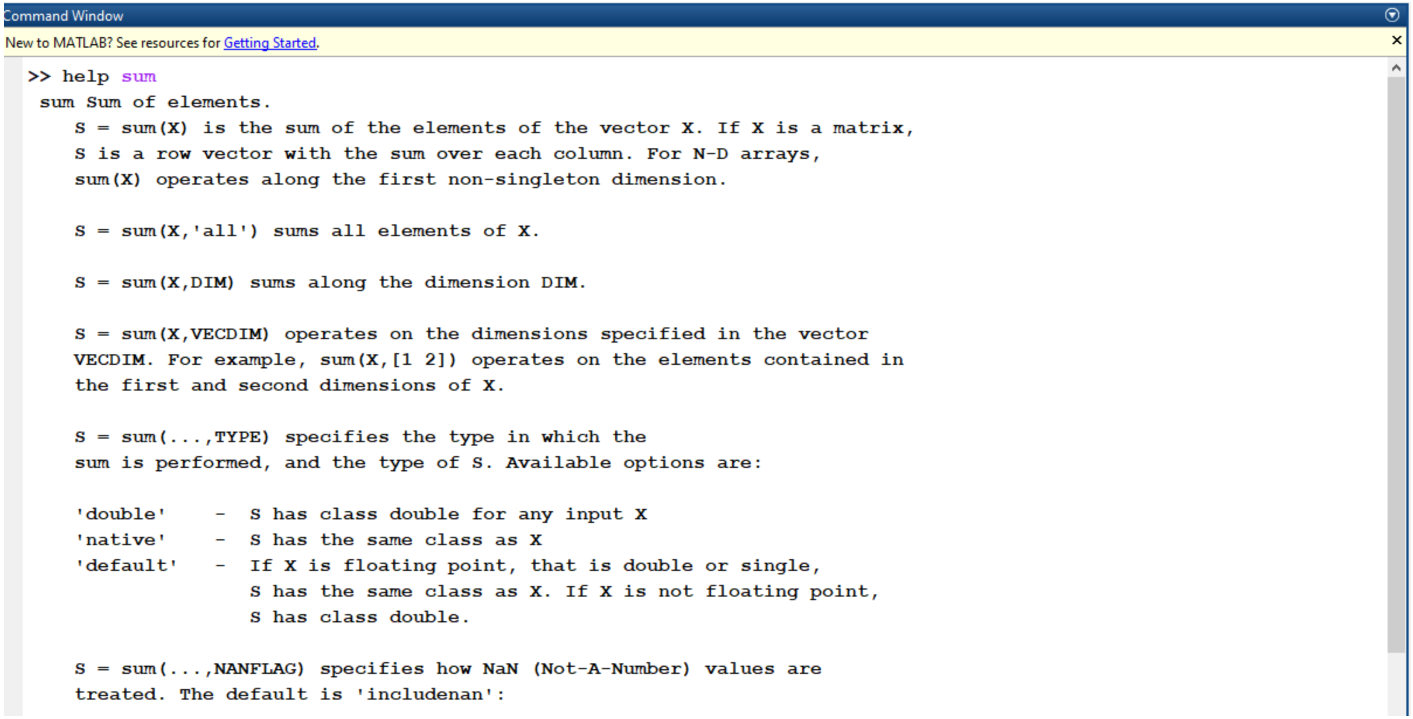
But it is easy remembering to check MATLAB’s documentation (even the pros use it)

MATLAB learning resources

MATLAB’s help command

MATLAB’s `help` command is run from the command window, and provides usage information about a given function:

```
>> help name_of_function
```

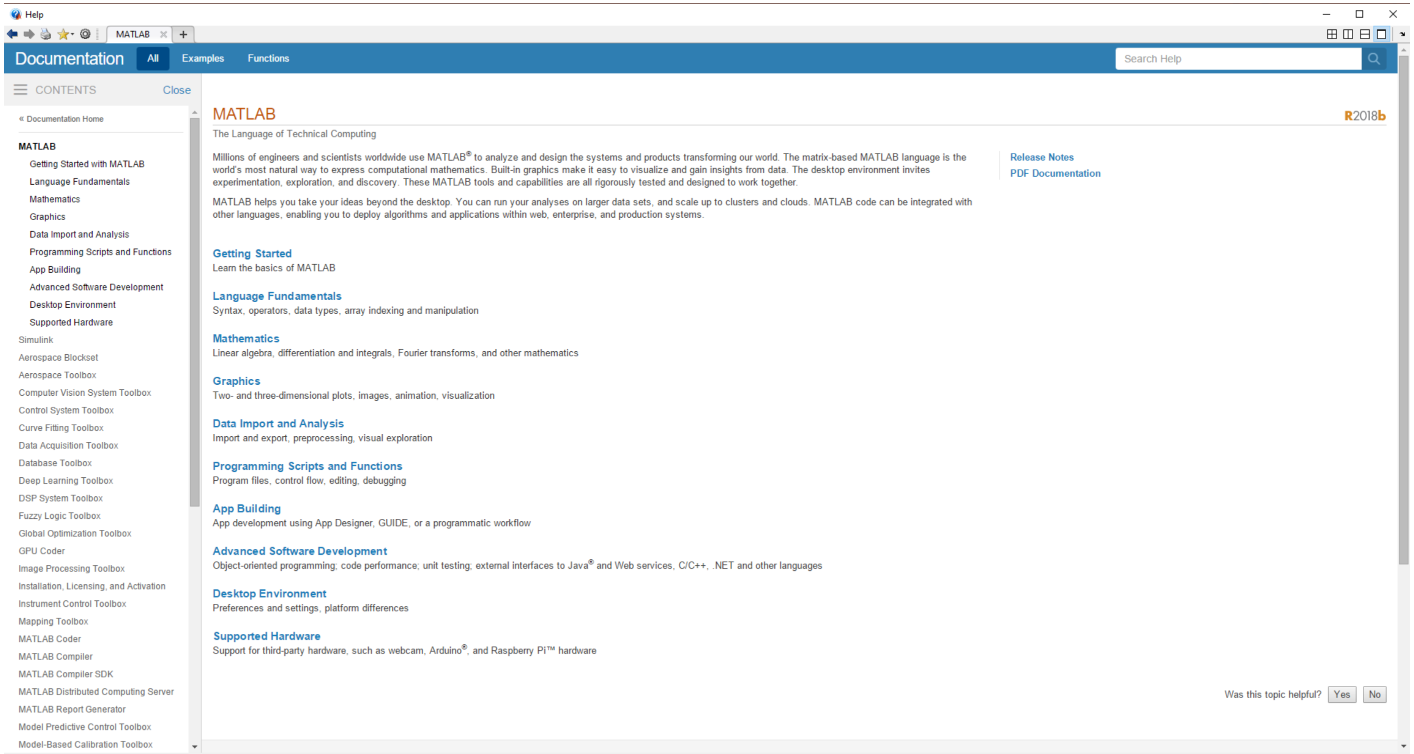


```
>> help sum
 sum Sum of elements.
    S = sum(X) is the sum of the elements of the vector X. If X is a matrix,
    S is a row vector with the sum over each column. For N-D arrays,
    sum(X) operates along the first non-singleton dimension.

    S = sum(X,'all') sums all elements of X.

    S = sum(X,DIM) sums along the dimension DIM.

    S = sum(X,VECDIM) operates on the dimensions specified in the vector
    VECDIM. For example, sum(X,[1 2]) operates on the elements contained in
    the first and second dimensions of X.

    S = sum(...,TYPE) specifies the type in which the
    sum is performed, and the type of S. Available options are:

    'double'    -  S has class double for any input X
    'native'    -  S has the same class as X
    'default'   -  If X is floating point, that is double or single,
                   S has the same class as X. If X is not floating point,
                   S has class double.

    S = sum(...,NANFLAG) specifies how NaN (Not-A-Number) values are
    treated. The default is 'includenan':
```

MATLAB learning resources

MATLAB documentation center

- The **documentation center** provides:
  - **Detailed information on functions** (usage, inputs/outputs, examples, etc.)
  - **General and specific information about how to use MATLAB**
  - **Practical examples and tutorials**



MATLAB

The Language of Technical Computing

Millions of engineers and scientists worldwide use MATLAB® to analyze and design the systems and products transforming our world. The matrix-based MATLAB language is the world's most natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data. The desktop environment invites experimentation, exploration, and discovery. These MATLAB tools and capabilities are all rigorously tested and designed to work together.

MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets, and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems.

Getting Started
Learn the basics of MATLAB

Language Fundamentals
Syntax, operators, data types, array indexing and manipulation

Mathematics
Linear algebra, differentiation and integrals, Fourier transforms, and other mathematics

Graphics
Two- and three-dimensional plots, images, animation, visualization

Data Import and Analysis
Import and export, preprocessing, visual exploration

Programming Scripts and Functions
Program files, control flow, editing, debugging

App Building
App development using App Designer, GUIDE, or a programmatic workflow

Advanced Software Development
Object-oriented programming; code performance; unit testing; external interfaces to Java® and Web services, C/C++, .NET and other languages

Desktop Environment
Preferences and settings, platform differences

Supported Hardware
Support for third-party hardware, such as webcam, Arduino®, and Raspberry Pi™ hardware

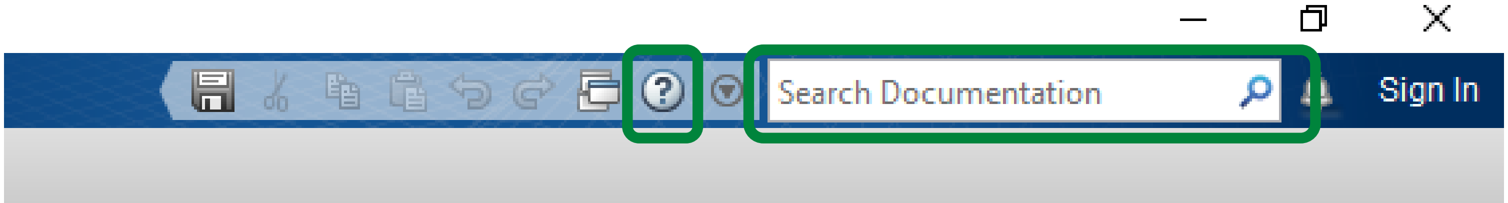
MATLAB learning resources

MATLAB documentation center

- You can access the **documentation center**:
  - Running command `doc` from the command window
    - If you use command `doc name_of_function`, you will go directly to the documentation page for that function

```
>> doc
>> doc name_of_function
```

  - Using the **help button** or the **Search Documentation** box in the toolbar:

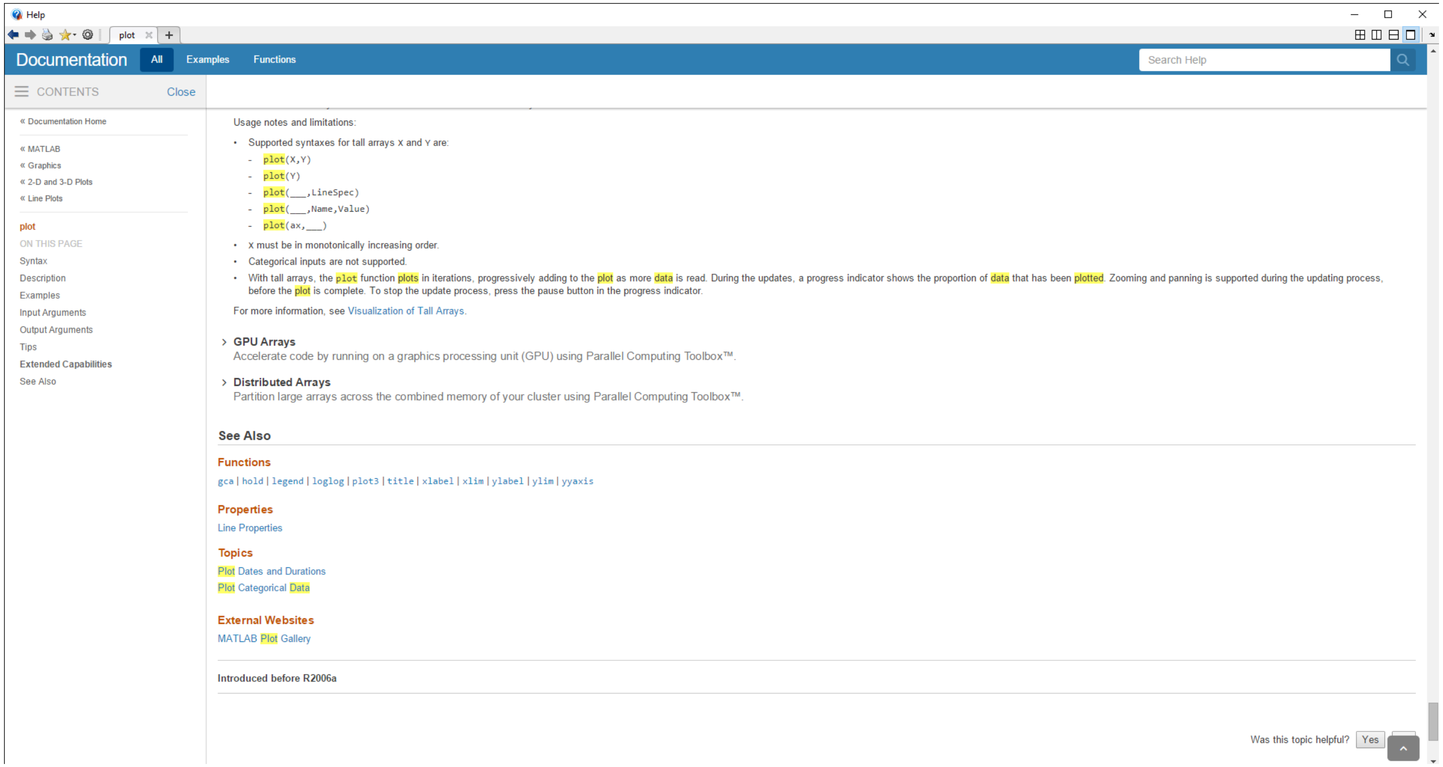


  - Pressing **F1** on your keyboard

MATLAB learning resources

MATLAB documentation center

- Many times MATLAB already has the functionality you need:
    - Search the **documentation center** before implementing new functions
    - At the end of each documentation page, you will find a list of **related functions and topics** that can be very handy



Usage notes and limitations:

- Supported syntaxes for tall arrays X and Y are:
    - `plot(X,Y)`
    - `plot(Y)`
    - `plot(___,LineSpec)`
    - `plot(___,Name,Value)`
    - `plot(ax,___)`
- X must be in monotonically increasing order.
- Categorical inputs are not supported.
- With tall arrays, the `plot` function plots in iterations, progressively adding to the plot as more data is read. During the updates, a progress indicator shows the proportion of data that has been plotted. Zooming and panning is supported during the updating process, before the plot is complete. To stop the update process, press the pause button in the progress indicator.

For more information, see Visualization of Tall Arrays.

› **GPU Arrays**
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

› **Distributed Arrays**
Partition large arrays across the combined memory of your cluster using Parallel Computing Toolbox™.

See Also

Functions

`gca` | `hold` | `legend` | `loglog` | `plot3` | `title` | `xlabel` | `xlim` | `ylabel` | `ylim` | `yyaxis`

Properties

Line Properties

Topics

Plot Dates and Durations
Plot Categorical Data

External Websites

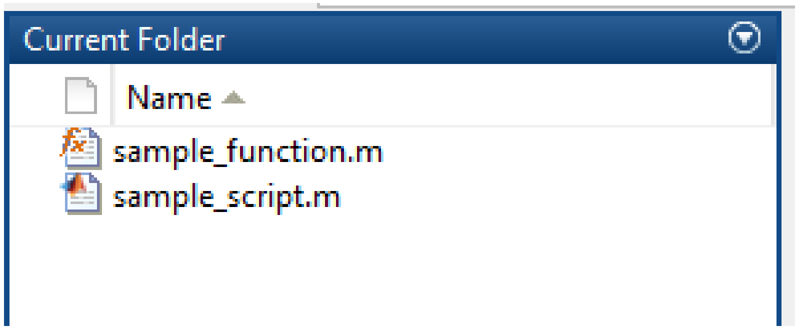
MATLAB Plot Gallery

Introduced before R2006a

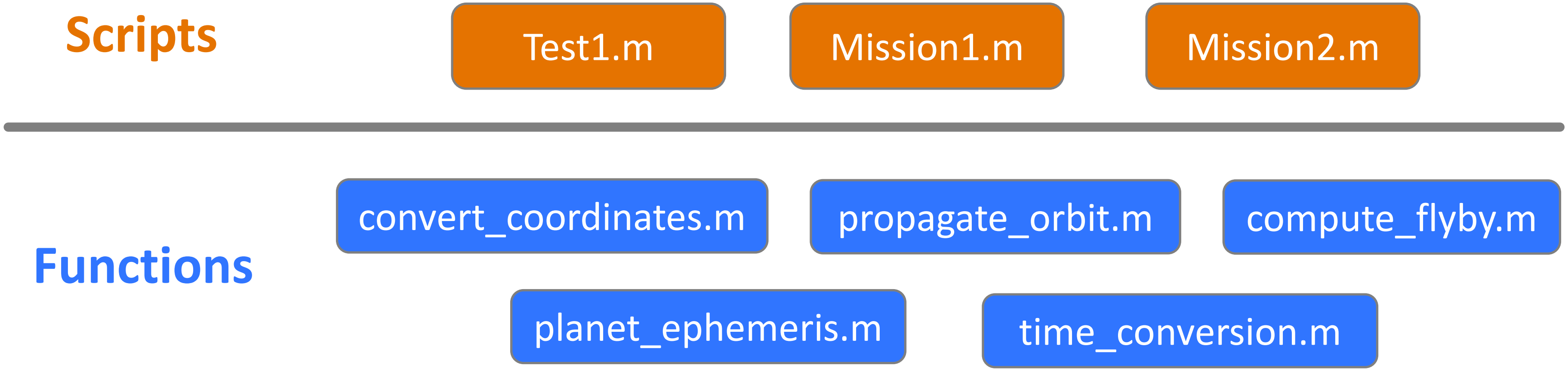
Code structuring

Scripts and functions

- MATLAB code is organized in **scripts** and **functions**
    - Both have file extension `.m`
    - They get different icons in MATLAB file explorer
    - **Scripts** should be the top-level part of your code, where you organize the whole program. **Functions** should correspond to specific tasks.



sample_function.m
sample_script.m



Scripts

Test1.m
Mission1.m
Mission2.m

Functions

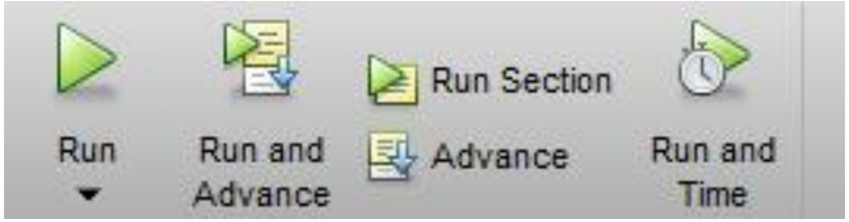
convert_coordinates.m
propagate_orbit.m
compute_flyby.m
planet_ephemeris.m
time_conversion.m

Code structuring

Scripts

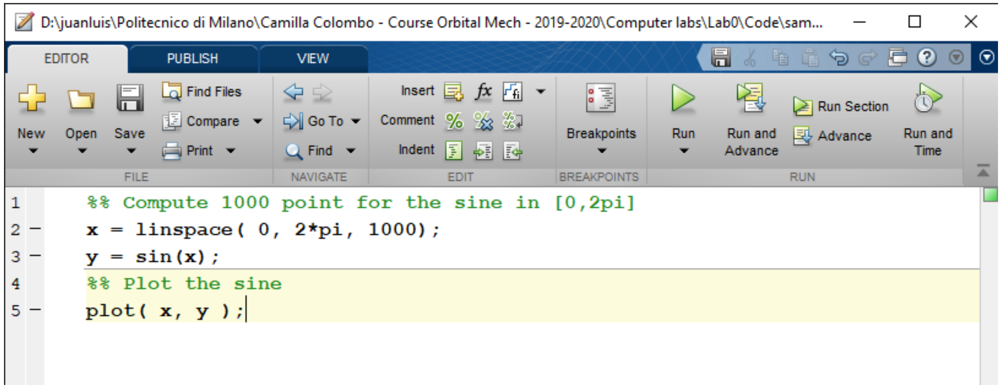
Scripts are the main files of your code. They can be executed directly:

- From the **editor**, using the toolbar or the keyboard
- From the command window, typing the name of the script (without file extension)
- You can also define **sections**, to execute your scripts part by part
  - New sections are created typing **%%**
  - The current section is the one containing the cursor, and is highlighted in yellow



Keep your cursor over a button in the toolbar to see the keyboard shortcut

```
>> sample_script
```

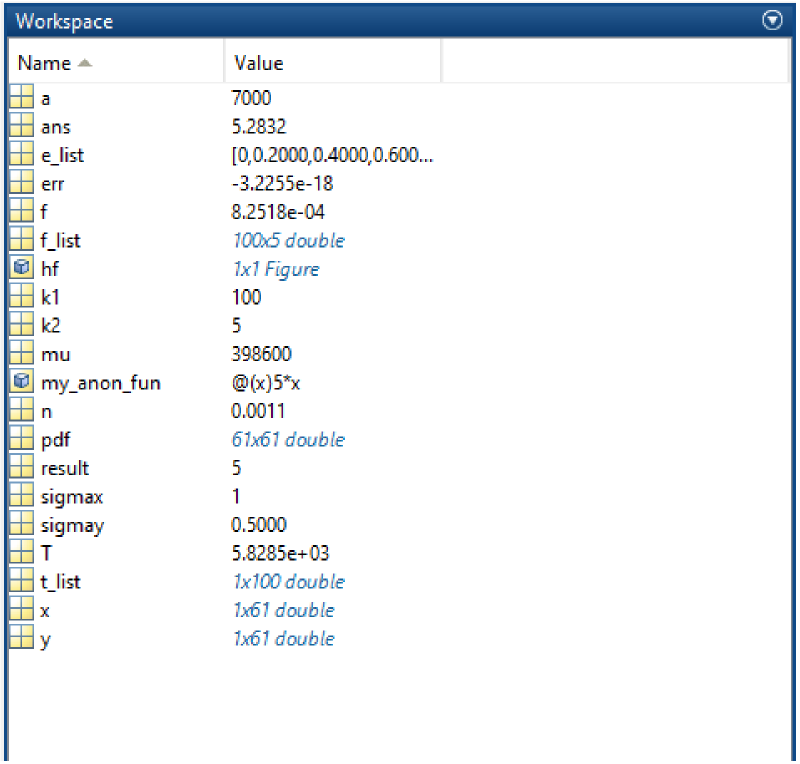


```
%% Compute 1000 point for the sine in [0,2pi]
x = linspace( 0, 2*pi, 1000);
y = sin(x);
%% Plot the sine
plot( x, y );
```

Code structuring

Scripts

- Scripts use the **global memory workspace**:
  - They can access any variable previously defined in the workspace
  - They can add new variables to the workspace
- This is very dangerous when calling scripts from scripts. Because they share the workspace, if they accidentally use the same name for different variables, they will overwrite each other:
  - Very difficult to debug
  - You have better control calling **functions** instead



Name	Value
a	7000
ans	5.2832
e_list	[0,0.2000,0.4000,0.600...
err	-3.2255e-18
f	8.2518e-04
f_list	*100x5 double*
hf	*1x1 Figure*
k1	100
k2	5
mu	398600
my_anon_fun	@(x)5*x
n	0.0011
pdf	*61x61 double*
result	5
sigmax	1
sigmay	0.5000
T	5.8285e+03
t_list	*1x100 double*
x	*1x61 double*
y	*1x61 double*

Code structuring

Functions

- **Functions** can be called from the **command window**, **scripts**, or **other functions**:

```
>> [ouput1,output2] = sample_function(input1,input2)
```

- **Functions** should focus on **performing specific tasks**:
  - Breaking code in small pieces makes it easier to **test**, **develop** and **maintain**
  - Try making them as general as possible, so that they can be **reused**
- **Functions** have their own **memory workspace**:
  - They don’t “see” variables created outside of their workspace, except for those defined as **global** (strongly discouraged)
  - Data exchange is controlled through **input/output variables**
  - Their internal variable names will not conflict with other scripts/functions
  - All internal variables are erased when the function ends, unless they are defined as **persistent** (be careful when using them)

Code structuring

Functions

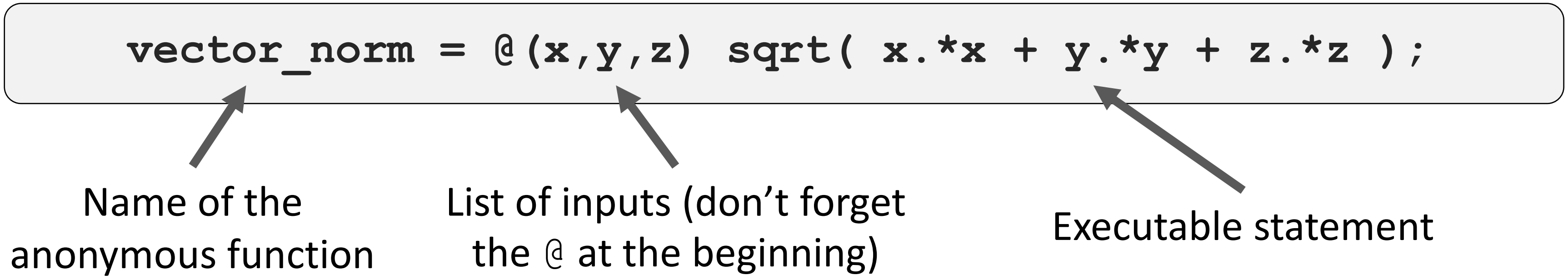
```
function n = mean_motion( a, mu )
% mean_motion Mean motion of a Keplerian orbit
%
% Function to compute the mean motion of an orbit. This is a very simple
% example to put in the slides of the lab
%
% PROTOTYPE
%   n = mean_motion( a, mu )
%
% INPUT:
%   a [1]       Semimajor axis [L]
%   mu [1]      Gravitational parameter of the primary [L^3/T^2]
%
% OUTPUT:
%   n [1]       Orbit mean motion [1/T]
%
% CONTRIBUTORS:
%   Juan Luis Gonzalo Gomez
%
% VERSIONS
%   2018-09-26: First version
%

n = sqrt( mu/a^3 );

end
```

Code structuring

Anonymous functions

- An **anonymous function** is a function that is not written in a file, but defined directly as a variable
  - They can have several inputs and outputs, as a normal **function**
  - But they can only contain a single executable statement
  - You can define them inside **scripts**, **functions**, or in the command line

- **Anonymous functions** are created as follows:



```
vector_norm = @(x,y,z) sqrt( x.*x + y.*y + z.*z );
```

Name of the anonymous function

List of inputs (don’t forget the @ at the beginning)

Executable statement

- **Anonymous functions** are called as any normal **function**

Code structuring

Anonymous functions

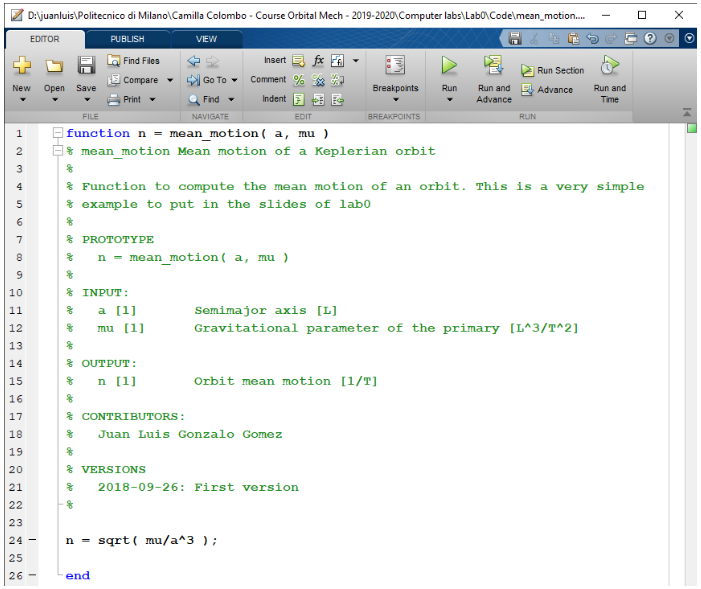
- **Anonymous functions** are normally used to:
  - Define small auxiliary functions within a **function** or **script**, without the need of creating a new file
  - Adapt the **interface** (inputs/outputs) of another function. Example:

```
% MATLAB’s ‘integral’ function can be used to perform the
% numeric integral of a 1D function. ‘integral’ expects
% a function with a single input variable.
% I want to perform the integral of function my_fun
% between 0 and 1, but it has 2 inputs (the second one
% is a parameter, of value 5 for this example).
% I can do it using an anonymous function:

my_anon_fun = @(x) my_fun( x, 5 );
result = integral( my_anon_fun, 0, 1 );
```

Best coding practices

Documentation

- **Document your code!**
  - Software is normally developed by **teams**; others must be able to understand/use your code
  - Not just for others, also **for the future you**
  - Use easily identifiable names for functions and variables
  - Include a header with:
    - Usage of the function
    - List of inputs and outputs (with physical units)
    - Authors and dates
    - Version log
    - External dependencies



```matlab
function n = mean_motion( a, mu )
% mean_motion Mean motion of a Keplerian orbit
%
% Function to compute the mean motion of an orbit. This is a very simple
% example to put in the slides of lab0
%
% PROTOTYPE
%   n = mean_motion( a, mu )
%
% INPUT:
%   a [1]       Semimajor axis [L]
%   mu [1]      Gravitational parameter of the primary [L^3/T^2]
%
% OUTPUT:
%   n [1]       Orbit mean motion [1/T]
%
% CONTRIBUTORS:
%   Juan Luis Gonzalo Gomez
%
% VERSIONS
%   2018-09-26: First version
%

n = sqrt( mu/a^3 );

end
```

Comments in MATLAB begin with symbol %

They can begin at any point of the line (e.g. after a command)

Best coding practices

Testing

- **Test your code as you go!**
  - No large code is bug-free (empirically demonstrated)
  - **Unit testing**: test each function independently as you program them
    - If you only debug the whole program at the end, it is very difficult to identify the source of each problem
  - Breaking up your code into small **functions** eases validation
  - One great way of validation is to check if the code reproduces **known results** (analytic solutions for particular cases, conservation principles, etc.)

MATLAB includes very useful **debugging tools**:

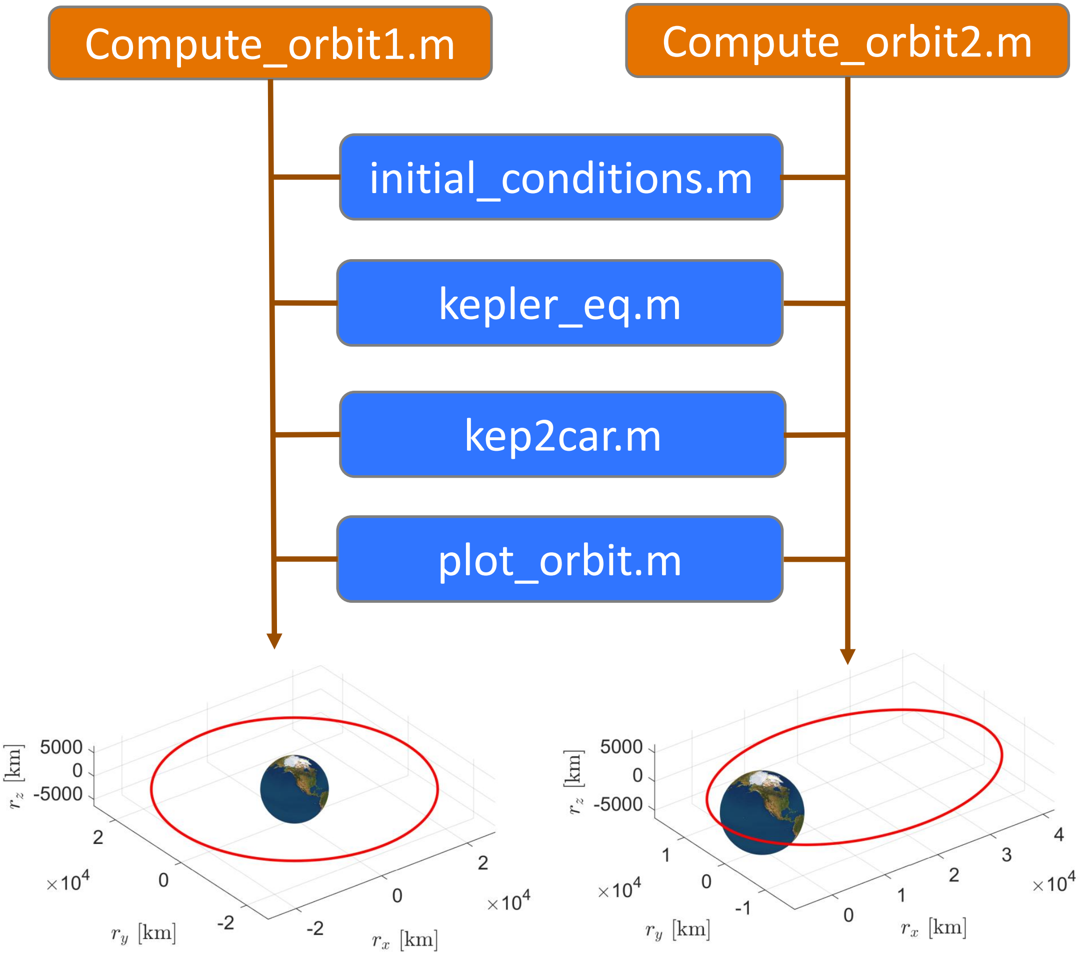
- Pause the execution and advance line by line,
- Check the memory workspace of each function,
- And much more.

Read the documentation page **“Debug a MATLAB Program”** to learn more!

Best coding practices

Reusability

- **Reuse as much as possible!**
  - Break your code into small, single-task **functions** with general inputs, so that they can be reused
  - This is not the same as copy-pasting blocks of code
  - This eases development and testing



Compute_orbit1.m

Compute_orbit2.m

initial_conditions.m

kepler_eq.m

kep2car.m

plot_orbit.m

By writing small, task-specific, reusable **functions** you will have less code to write, test and maintain

Figures

Are they really important?

- **Figures** (plots, graphs, charts, etc.) are a key component of scientific and technical communication
  - Good figures convey a lot of information in a clear and concise way
- A good figure should:
  - Have its axes clearly labelled, including physical units
  - Include a caption, placed below the figure, presenting its contents
  - If more than one data set is represented, include a legend and use different colours and markers/line styles
  - Use a font size clearly readable
  - Be self-contained (i.e. should be understandable even without reading the whole document)

⚠ Badly presented figures in the reports will be harshly punished

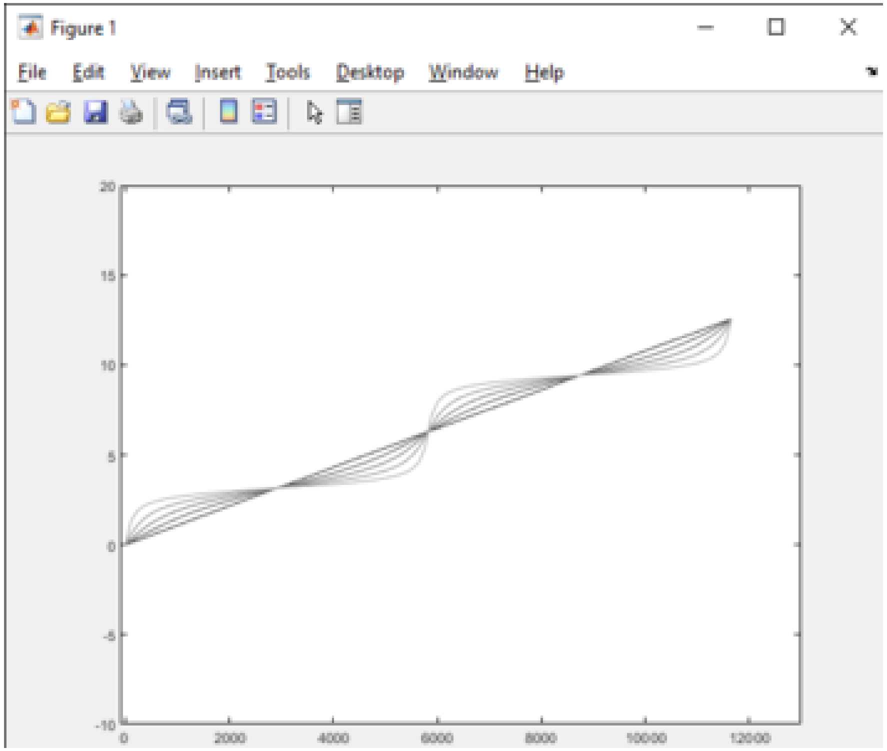
Figures

This is a BAD figure

It is just a low-resolution screenshot

Font is too small to read

I don't know what the axes represents, or their physical units



Very difficult to distinguish the lines

I don't know the meaning of each line

Plenty of wasted blank space

Without a title or a caption, I cannot know what is being represented in the plot

Figures

This is a GOOD figure

Good-quality image

Font size is similar to the text of the slide

The variable represented in each axis is indicated, along with its units

The unit of time (orbital period) is chosen to get numerical values easy to understand

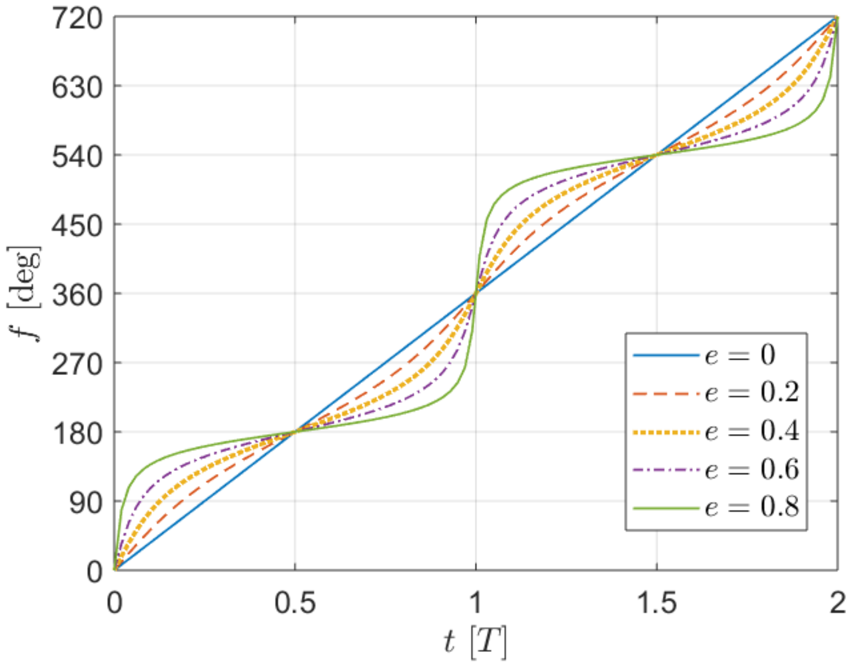


Figure 1. Evolution of true anomaly with time for several orbits with $a = 7000$ km, $\mu = 398600\ \mathrm{km^3/s^2}$, and different e

Lines are set apart using different colors and line styles

A legend is included

A grid is included to improve readability

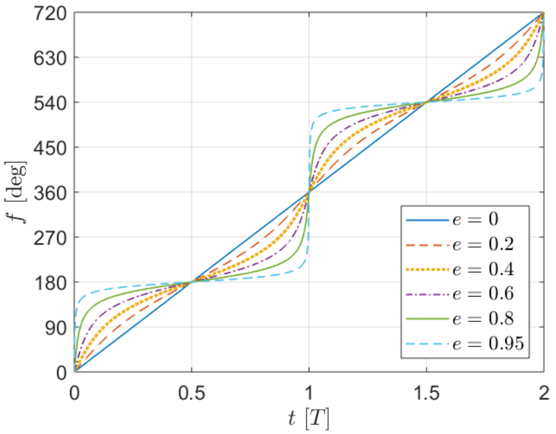
Ranges are set to make use of all available space

Figure contents are clearly introduced in a caption, together with a number to reference the figure

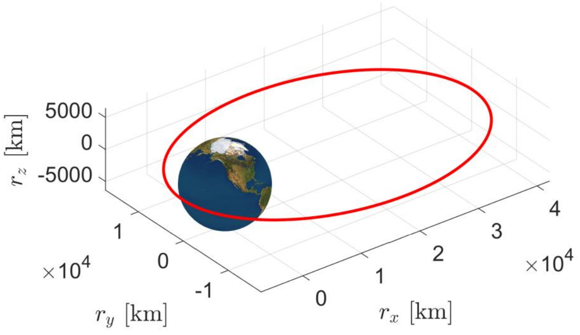
Figures

Plotting in MATLAB

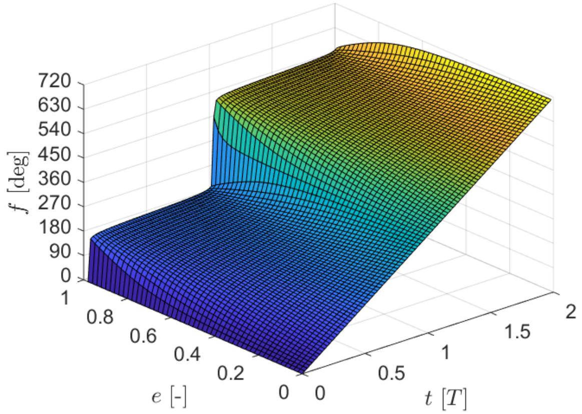
- MATLAB can represent many types of figures (check the **documentation center** to learn how to use them)



`plot`



`plot3`



`surface`

- Other types of MATLAB plots:
  - `quiver` and `quiver3`: 2D and 3D vector field plots
  - `comet` and `comet3`: animations in 2D and 3D plots
  - `sphere`: Generates a sphere
  - `mesh`: 3D surface plots

Figures

Customizing plots in MATLAB

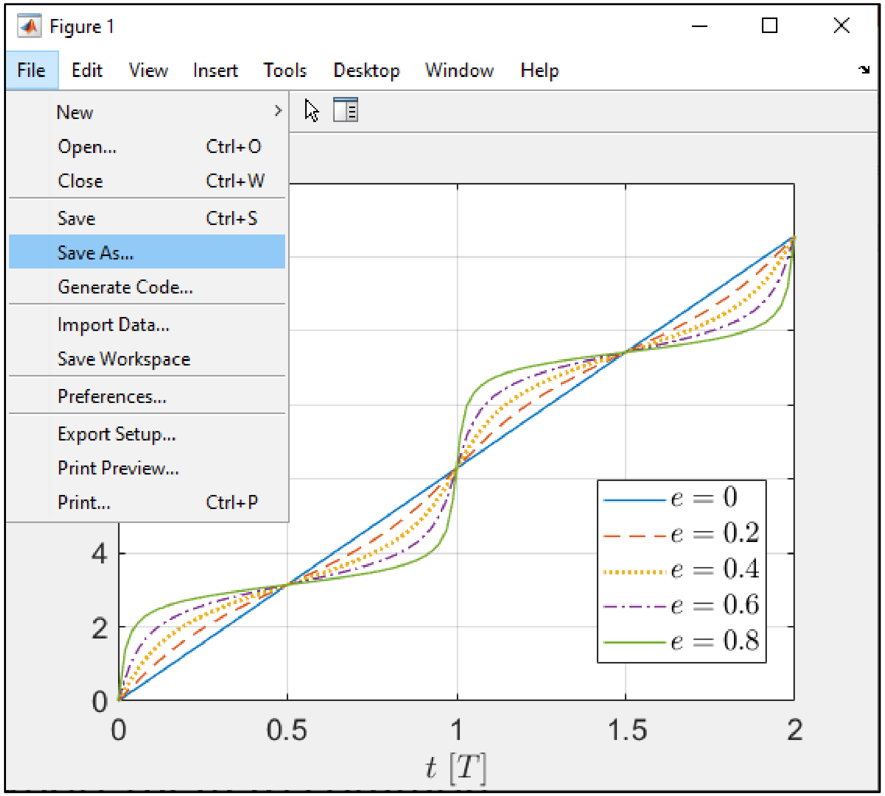
- Properties of plots can be adjusted from the graphical interface or using commands:
  - `xlabel`, `ylabel`, and `zlabel`: Set labels for x, y, and z axes
  - `xlim`, `ylim`, and `zlim`: Set the plotting range for each axis
  - `title`: Set the title of the plot
  - `legend`: Add a legend to the plot
  - `grid on`: Add a grid to the plot
  - `hold on`: Hold the plot, so new lines can be added
  - `axis equal`: Use equal unit length along all axes
  - and many more

ⓘ Check the documentation page for each kind of plot to discover all the available options and related commands

Figures

Saving figures in MATLAB

- MATLAB uses its own format (`.fig`) to save figures
  - You can open fig files in MATLAB to edit them
- You can export figures to common formats (png, jpg, eps, etc) through menu “File > Save as…”
  - MATLAB cannot edit these formats after exporting
- You can also save figures from code using commands such as `saveas` and `print`



Screenshots are not a good way to include figures in the report

NUMERICAL INTEGRATION OF DYNAMICAL SYSTEMS

Ordinary Differential Equations

An **ordinary differential equation** (ODE) is an equation F involving one or more functions $\mathbf{x}$ of an independent variable t (usually **time**) and their derivatives:

$$F\left(t; \mathbf{x}(t), \dot{\mathbf{x}}(t), \ddot{\mathbf{x}}(t), \dots, \mathbf{x}^{(n)}(t)\right) = 0$$

Many dynamical systems in nature can be modelled through a system of ODEs

Example: Damped harmonic oscillator

Applying 2nd Newton's law:

$$m\,\ddot{x} = -k\,x\; - c\,\dot{x}$$

Rearranging terms:

$$\ddot{x} + 2\gamma\dot{x} + \omega_0^2 x = 0$$

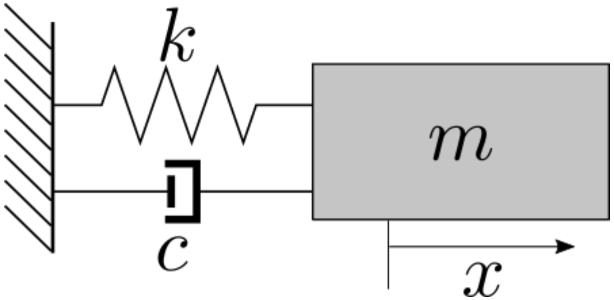
where:

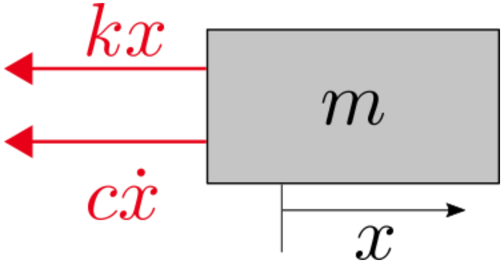
$$\gamma = \frac{c}{2m}, \;\; \omega_0 = \sqrt{k/m}$$

k: spring constant

c: viscous damping coefficient

m: mass of the block





Reduction to a first-order system

The **order** of an ODE is that of the highest-order derivative involved.

Any ODE can be reduced to a **first-order system** by treating the derivatives up to order *n-1* as **independent variables**. In explicit form:

$$\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{dt}} = \mathbf{f}(t; \mathbf{y}(t))$$

where $\mathbf{y} = \left[\mathbf{x}, \dot{\mathbf{x}}, \ \dots, \mathbf{x}^{(n-1)}\right]$.

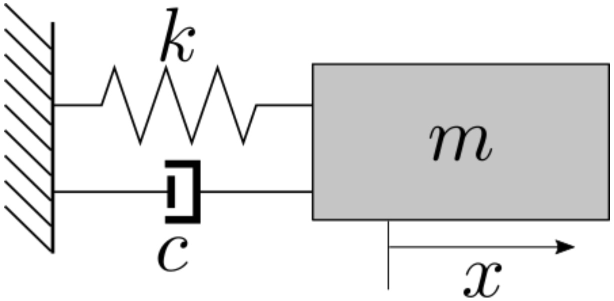
Example: Damped harmonic oscillator, reduction to a first-order system

New state vector (including velocity): $\mathbf{y} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$

Equation of motion: $\dot{y}_2 = -2\gamma y_2 \ - \omega_0^2 y_1$

Velocity definition: $y_2 = \dot{x} = \dot{y}_1$

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ -2\gamma y_2 \ - \omega_0^2 y_1 \end{bmatrix}$$



Resolution of an ODE system

If $\mathbf{f}$ is *regular enough*, the solution of the first order ODE system for given initial conditions $\mathbf{y}(t_0) = \mathbf{y}_0$ exists and is unique (Cauchy-Lipschitz or Picard-Lindelöf theorem)

- In some cases, a closed form (analytic) solution can be found
- In general, we will resort to **numerical integration schemes**

Example: Damped harmonic oscillator, analytic solution

Initial conditions: $x(0) = x_0, \dot{x}(0) = \dot{x}_0$

Overdamped
$\gamma^2 > \omega_0^2$

$$x(t) = \mathrm{e}^{-\gamma t}\left[x_0 \cosh \omega^* t + \frac{\dot{x}_0 + \gamma x_0}{\omega^*} \sinh \omega^* t\right] \qquad \omega^* = \sqrt{\gamma^2 - \omega_0^2}$$

Critical damping
$\gamma^2 = \omega_0^2$

$$x(t) = \mathrm{e}^{-\gamma t}[x_0 + (\dot{x}_0 + \gamma x_0)t]$$

Underdamped
$\gamma^2 < \omega_0^2$

$$x(t) = \mathrm{e}^{-\gamma t}\left[x_0 \cos \omega t + \frac{\dot{x}_0 + \gamma x_0}{\omega} \sin \omega t\right] \qquad \omega = \sqrt{\omega_0^2 - \gamma^2}$$

Numerical resolution of ODEs

General aspects

Let us approximate function $\mathbf{y}(t)$, solution to the first-order ODE system $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y})$, through its values at a set of times $\left[t_0 \ t_1 \dots t_k \ t_{k+1} \ \dots \ t_f\right]$:

$$y_k = y(t_k) \qquad f_k = f(t_k, y_k)$$

It is possible to construct numerical schemes that compute the value of $\mathbf{y}$ at the next time step from the values of $\mathbf{y}$ and $\mathbf{f}$ at previous time steps:

$$y_{k+1} = \Phi(t_{k+1}, t_k, t_{k-1}, \dots, y_{k+1}, y_k, y_{k-1}, \dots, f_{k+1}, f_k, f_{k-1}, \dots)$$

There exist many schemes/algorithms, with different characteristics:

- The number of previous steps involved depends on the scheme
- Schemes can be implicit (Φ depends on the values at t_{k+1}) or explicit
- Time steps can be fixed, or adjusted automatically by the solver to ensure a preset accuracy
- In all cases, you have to provide the solver with
  - The initial condition $(t_0, \mathbf{y}_0)$
  - A way to evaluate $\mathbf{f}_k$ (i.e. a function). The values of t_k and $\mathbf{y}_k$ are not known a priori, so $\mathbf{f}_k = \mathbf{f}(t_k, \mathbf{y}_k)$ has to be evaluated as the numerical solver advances.

Numerical resolution of ODEs

Euler scheme

The simplest numerical scheme for solving ODEs is the **Euler scheme**.

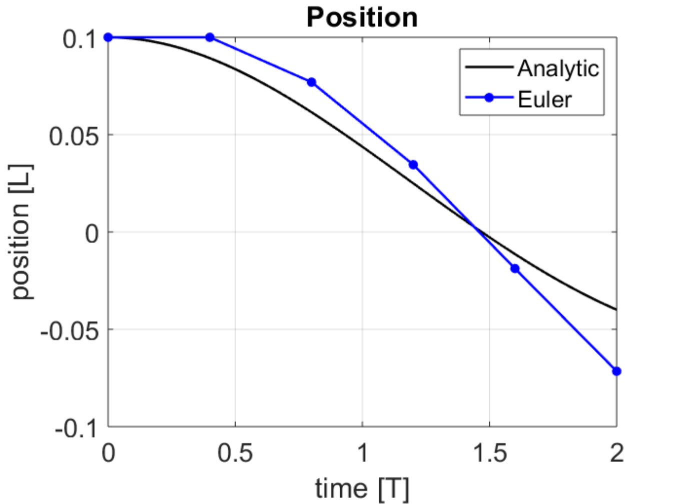
Consider the Taylor expansion of $\mathbf{y}(t)$ around t_k:

$$\mathbf{y}(t) = \mathbf{y}(t_k) + \dot{\mathbf{y}}(t_k)(t - t_k) + O(t - t_k)^2$$

Retaining only the first order term, **and recalling that $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y})$**:

$$\mathbf{y}(t_{k+1}) = \mathbf{y}_{k+1} \approx \mathbf{y}_k + \mathbf{f}(t_k, \mathbf{y}_k)\Delta t$$

That is, we approximate the value of $\mathbf{y}$ at time step $t_{k+1} = t_k + \Delta t$ from its value at t_k. Geometrically, this is equivalent to treating $\mathbf{y}$ between t_k and t_{k+1} as a straight line with the slope given by $\mathbf{f}_k$.



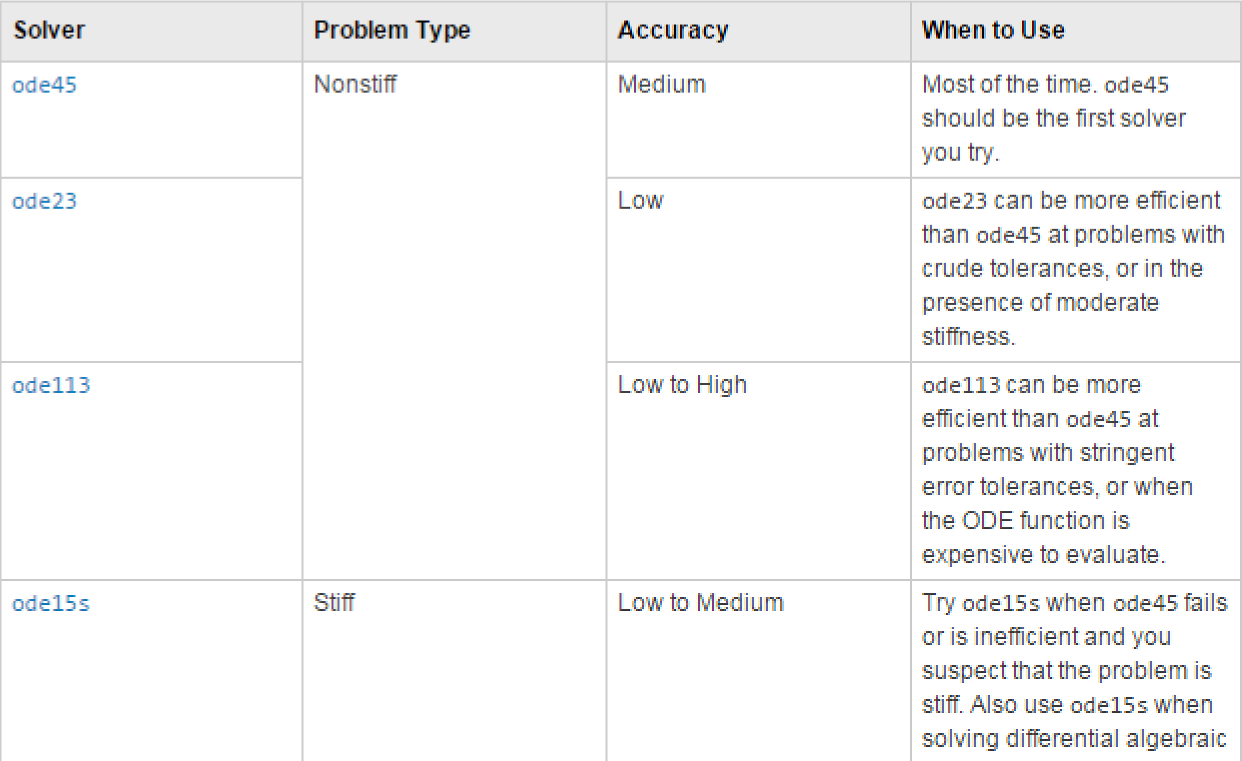
Euler scheme requires very small time steps to achieve high precision (local error is of order $\ddot{\mathbf{y}}_k \Delta t^2$). This leads to large number of steps and computational time.

In practice, we will use more advanced integration schemes already implemented in MATLAB

Integrating ODEs with MATLAB

MATLAB provides several solvers for systems of ODEs

- Based on different integration schemes, with different properties
- **We have to select the most appropriate one for our dynamics**



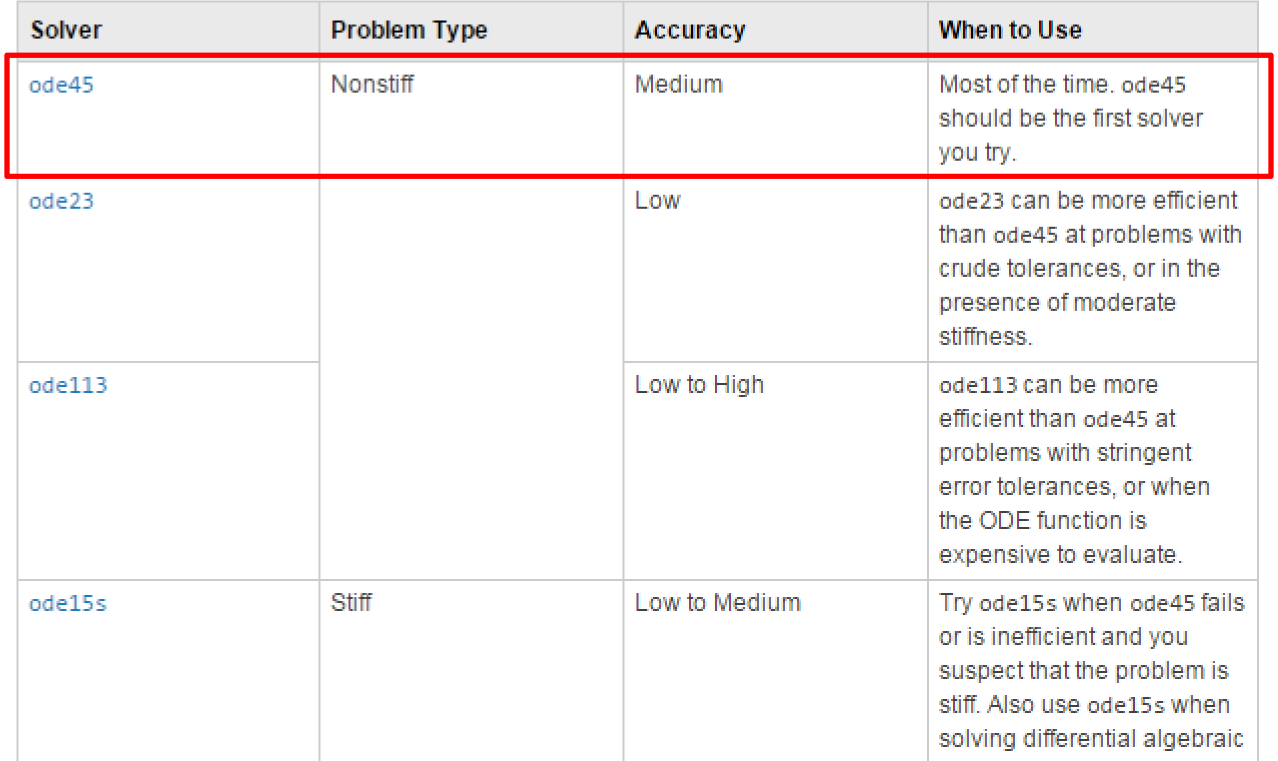
Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. ode45 should be the first solver you try.
ode23		Low	ode23 can be more efficient than ode45 at problems with crude tolerances, or in the presence of moderate stiffness.
ode113		Low to High	ode113 can be more efficient than ode45 at problems with stringent error tolerances, or when the ODE function is expensive to evaluate.
ode15s	Stiff	Low to Medium	Try ode15s when ode45 fails or is inefficient and you suspect that the problem is stiff. Also use ode15s when solving differential algebraic

Integrating ODEs with MATLAB

ode45

MATLAB provides several solvers for systems of ODEs

- Based on different integration schemes, with different properties
- **We have to select the most appropriate one for our dynamics**



Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. ode45 should be the first solver you try.
ode23		Low	ode23 can be more efficient than ode45 at problems with crude tolerances, or in the presence of moderate stiffness.
ode113		Low to High	ode113 can be more efficient than ode45 at problems with stringent error tolerances, or when the ODE function is expensive to evaluate.
ode15s	Stiff	Low to Medium	Try ode15s when ode45 fails or is inefficient and you suspect that the problem is stiff. Also use ode15s when solving differential algebraic

ode45

Explicit Runge-Kutta (4,5) formula (the Dormand-Prince pair)

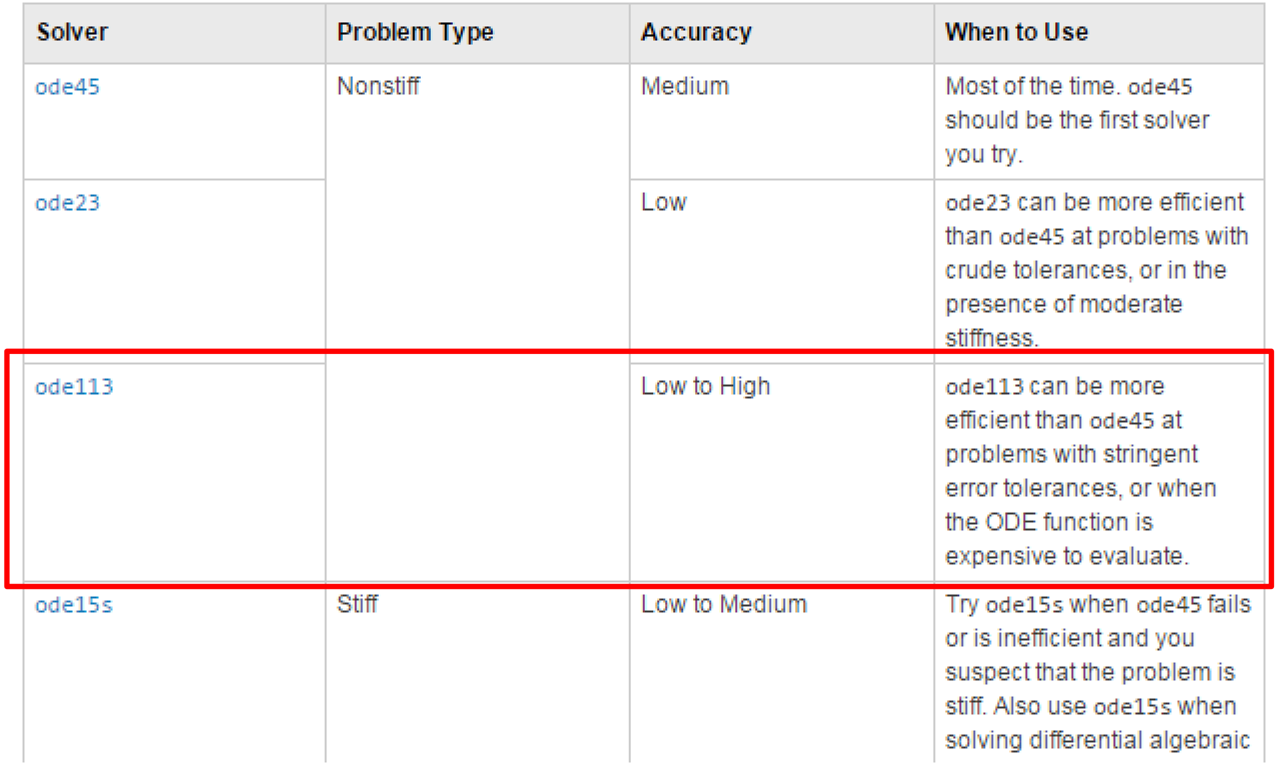
- Very versatile
- Low performance for stiff problems
- Low performance for high accuracy requirements

Integrating ODEs with MATLAB

ode113

MATLAB provides several solvers for systems of ODEs

- Based on different integration schemes, with different properties
- **We have to select the most appropriate one for our dynamics**



Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. ode45 should be the first solver you try.
ode23		Low	ode23 can be more efficient than ode45 at problems with crude tolerances, or in the presence of moderate stiffness.
ode113		Low to High	ode113 can be more efficient than ode45 at problems with stringent error tolerances, or when the ODE function is expensive to evaluate.
ode15s	Stiff	Low to Medium	Try ode15s when ode45 fails or is inefficient and you suspect that the problem is stiff. Also use ode15s when solving differential algebraic

ode113

Variable-step, variable-order Adams-Bashforth-Moulton Predictor-Corrector solver of orders 1 to 13

- Less function calls than ode45 (less evaluations of the ODE function)

Syntax for ODE solvers

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

Syntax for ODE solvers

Inputs: ODE function

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

`odefun` is a MATLAB function representing the **right-hand side of the first order ODE system**, that is, function $\mathbf{f}$ in:

$$\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = \mathbf{f}(t; \mathbf{y}(t))$$

This function has to be of the form:

```
function dy = odefun( t, y )
    .....
    .....
end
```

⚠

`t` is a scalar
`y` and `dy` are column vectors of dimension [n x 1]

How do we introduce additional parameters (such as γ and ω_0 in the harmonic oscillator)?

Syntax for ODE solvers

Inputs: ODE function

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

Two ways of introducing additional parameters:

1. Additional inputs added at the end of the argument list of `odeXX` (i.e. after `options`) are passed directly to `odefun`:

```
[t, y] = odeXX( odefun, tspan, y0, options, par1, par2, par3, … )

          dy = odefun( t, y, par1, par2, par3, … )
```

2. Use an **anonymous function**:

```
[t, y] = odeXX(@(t,y) odefun(t,y,par1,par2), tspan, y0, options )
```

This creates an unnamed function with inputs `t` and `y`, to be used by `odeXX` as a wrapper for `odefun`.

Syntax for ODE solvers

Inputs: Time span

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

Time span for the integration. There are two possibilities:

1. **`tspan = [ tstart  tend  ]`**
   If `tspan` is a 2-elements array, they represent the initial and final times for the integration, respectively. Output is given at the **internal time steps used by the solver.**

2. **`tspan = [ tstart  t1  t2  …  tend  ]`**
   If `tspan` is a monotonic array of more than 2 elements, the solver returns the value of `y` only at the times in `tspan`. The first and last elements of `tspan` still represent the initial and final time of the integration.
   This does not affect the internal time steps automatically decided by the solver (it uses its own time steps to integrate, and afterwards interpolates to get the values for the requested times).

⚠ The time values in `tspan` must be strictly monotonic!

Syntax for ODE solvers

Inputs: Initial conditions

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

Column array with the initial conditions for the state (i.e., the value of `y` at the first time given in `tspan`, $\mathbf{y}(t_{\text{start}}) = \mathbf{y}_0$). It must have the same dimensions as `y` and `dy`.

Syntax for ODE solvers

Inputs: Options

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

Object containing optional parameters for the ODE solver. It is created using the `odeset` function (check the documentation center). For example:

```
options = odeset( 'RelTol', 1e-13, 'AbsTol', 1e-14 );
```

For now, we consider only the relative and absolute tolerances (`RelTol` and `AbsTol`, respectively). **The internal time steps used by the solver are automatically chosen to fulfill the tolerances (they do not depend on `tspan`)**

This is an optional argument. You can omit it if no later arguments are given, or use the empty variable `[]`

There are many (powerful) options that can be configured. Check the **documentation center page** about `odeset` for more information

Default tolerance values, `RelTol=1e-3` and `AbsTol=1e-6`, are too loose for orbit propagation. Don't forget to set more stringent ones!

Syntax for ODE solvers

Outputs: Two possibilities

```
[t, y] = odeXX( odefun, tspan, y0, options )
```

`t`: m x 1 array with the times at each of the m time steps.

`y`: m x n array with the n states at each of the m time steps. That is, row m corresponds to the state at the m-th time step.

```
sol = odeXX( odefun, tspan, y0, options )
```

`sol`: MATLAB structure containing detailed information about the solution:

- Time and state at the time steps decided by the integrator are stored in `sol.x` and `sol.y`, respectively. **Beware**, they are transposed with respect to `t` and `y`: `sol.x` is a 1 x m array, and `sol.y` is a n x m array with each column corresponding to a different time step.
- The state for other times can be obtained using the function `deval`.

Back to the harmonic oscillator

ODE function

```
function dy = ode_harmonic_oscill( ~, y, omega0, gamma )
%ode_harmonic_oscill ODE system for the damped harmonic oscillator
%
% PROTOTYPE
%   dy = ode_harmonic_oscill( t, y, omega0, gamma )
%
% INPUT:
%   t[1]        Time (can be omitted, as the system is autonomous) [T]
%   y[2x1]      State of the oscillator (position and velocity)  [ L, L/T ]
%   omega0[1]   Natural frequency of the undamped oscillator [1/T]
%   gamma[1]    Damping coefficient [1/T]
%
% OUTPUT:
%   dy[2x1]     Derivative of the state  [ L/T^2, L/T^3 ]
%
% CONTRIBUTORS:
%   Juan Luis Gonzalo Gomez
%
% VERSIONS
%   2018-09-26: First version
%

% Set the derivatives of the state
dy = [  y(2)
        -2*gamma*y(2)-omega0^2*y(1) ];

end
```

Back to the harmonic oscillator

ODE function

```
function dy = ode_harmonic_oscill( ~, y, omega0, gamma )
%ode_harmonic_oscill ODE system for the damped harmonic oscillator
%
% PROTOTYPE
%   dy = ode_harmonic_oscill( t, y, omega0, gamma )
%
% INPUT:
%   t[1]         Time (can be omitted, as the system is autonomous) [T]
%   y[2x1]       State of the oscillator (position and velocity)  [ L, L/T ]
%   omega0[1]    Natural frequency of the undamped oscillator [1/T]
%   gamma[1]     Damping coefficient [1/T]
%
% OUTPUT:
%   dy[2x1]      Derivative of the state  [ L/T^2, L/T^3 ]
%
% CONTRIBUTORS:
%   Juan Luis Gonzalo Gomez
%
% VERSIONS
%   2018-09-26: First version
%

% Set the derivatives of the state
dy = [  y(2)
        -2*gamma*y(2)-omega0^2*y(1)  ];

end
```

Use ~ to omit unneeded inputs (like time in our oscillator)

Always document your code!
You can take this example as template.

Back to the harmonic oscillator

Main script

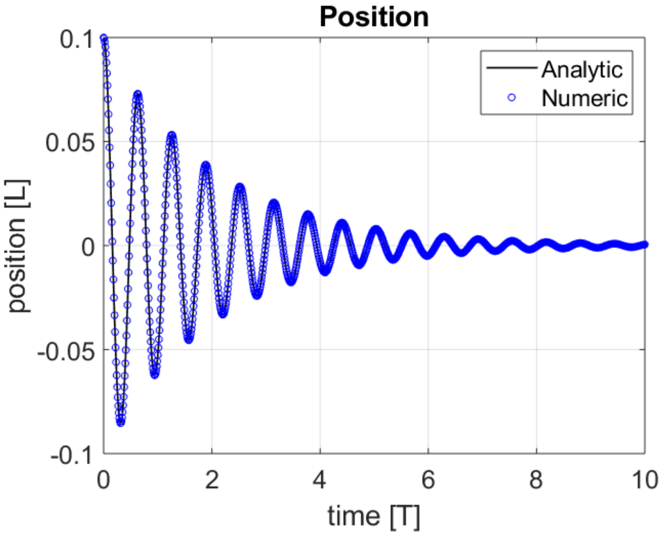
```
% Oscillator parameters
omega0 = 10;
gamma  = 0.1;

% Initial condition
y0 = [ 0.1; 0 ];

% Set options
options = odeset( 'RelTol', 1e-13, 'AbsTol', 1e-14 );

% Set time span
tspan = linspace( 0, 10, 1000 );

% Perform the integration
[ T, Y ] = ode113( @(t,y) ode_harmonic_oscill(t,y,omega0,gamma), tspan, y0, options);

% Plot the results
figure()
plot( T, Y(:,1), '-' )
xlabel('time [T]');
ylabel('position [L]');
title('Position');

figure()
plot( T, Y(:,2), '-' )
xlabel('time [T]');
ylabel('velocity [L/T]');
title('Velocity');
```

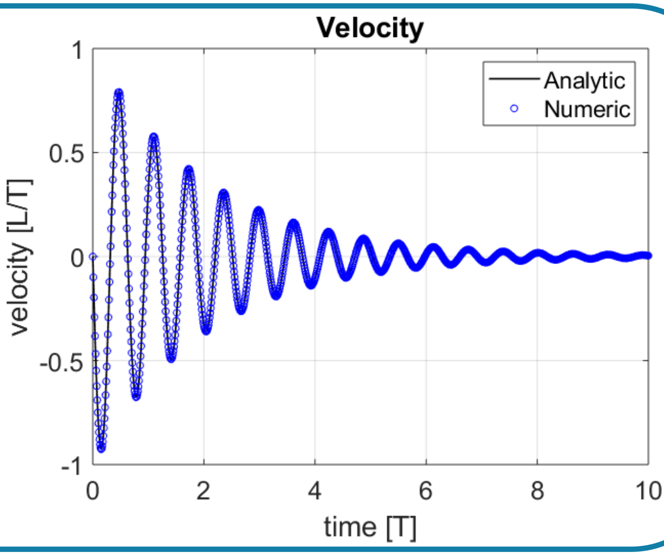
Back to the harmonic oscillator

Some results

Underdamped

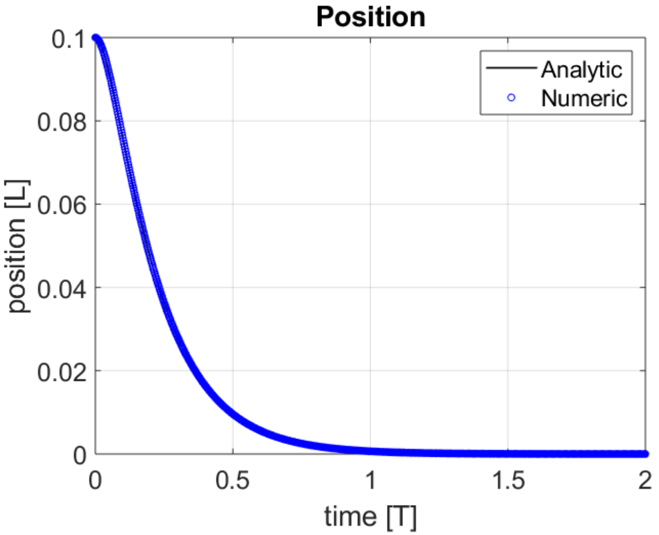
$\omega_0 = 10\ [\mathrm{T}^{-1}]$
$\gamma = 0.5\ [\mathrm{T}^{-1}]$
$x_0 = 0.1\ [\mathrm{L}]$
$\dot{x}_0 = 0\ [\mathrm{L/T}]$

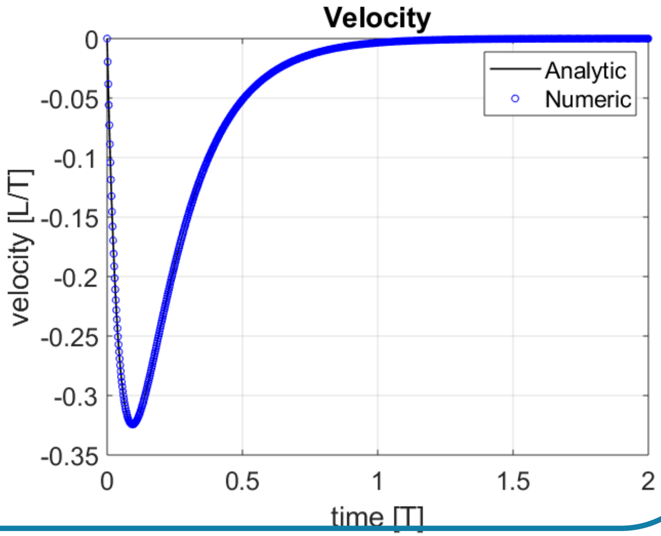




Overdamped

$\omega_0 = 10\ [\mathrm{T}^{-1}]$
$\gamma = 12\ [\mathrm{T}^{-1}]$
$x_0 = 0.1\ [\mathrm{L}]$
$\dot{x}_0 = 0\ [\mathrm{L/T}]$

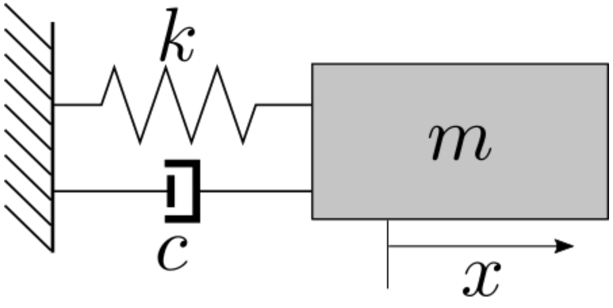




Exercise 1: Damped harmonic oscillator

Exercise 1: Compare the analytical and numerical solutions for the underdamped harmonic oscillator

1. Write a **function** for the analytic solution. *Which are the inputs?*
2. Write the **ODE function** for the system, and **solve it numerically using `ode113`**
3. Write a **main script** to compute the solutions and plot them
4. Try different initial conditions $(x_0, \dot{x}_0)$ and physical parameters c, k, and m
   - Remember that $\gamma^2 < \omega_0^2$ for underdamped oscillators



$$x(0) = x_0, \qquad \dot{x}(0) = \dot{x}_0$$

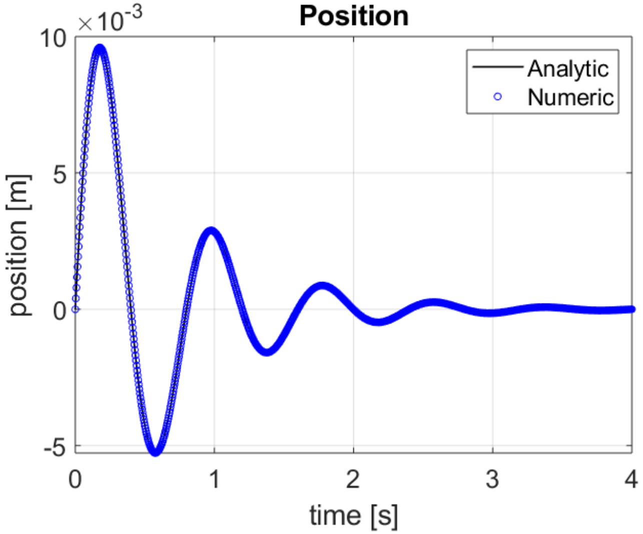
$$\begin{cases} x(t) = \mathrm{e}^{-\gamma t}\left[x_0 \cos\omega t + \dfrac{\dot{x}_0 + \gamma x_0}{\omega}\sin\omega t\right] \\ \dot{x}(t) = \mathrm{e}^{-\gamma t}\left[\dot{x}_0 \cos\omega t - \dfrac{\omega_0^2 x_0 + \gamma \dot{x}_0}{\omega}\sin\omega t\right] \end{cases}$$

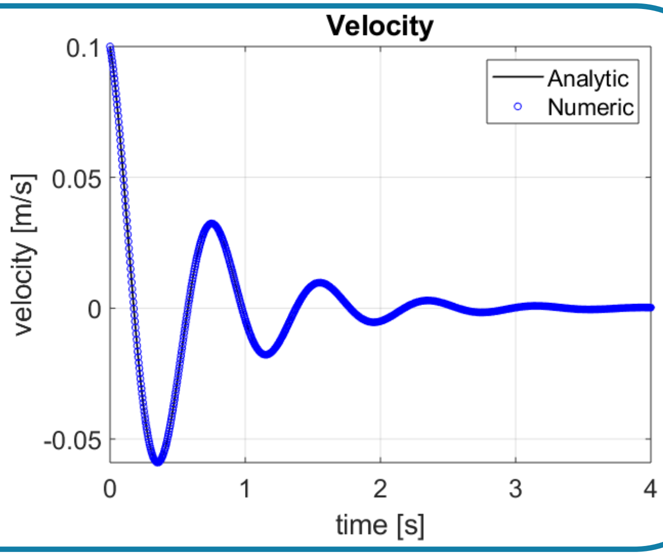
$$\gamma = \frac{c}{2m}, \qquad \omega_0 = \sqrt{k/m}, \qquad \omega = \sqrt{\omega_0^2 - \gamma^2}$$

Exercise 1: Damped harmonic oscillator

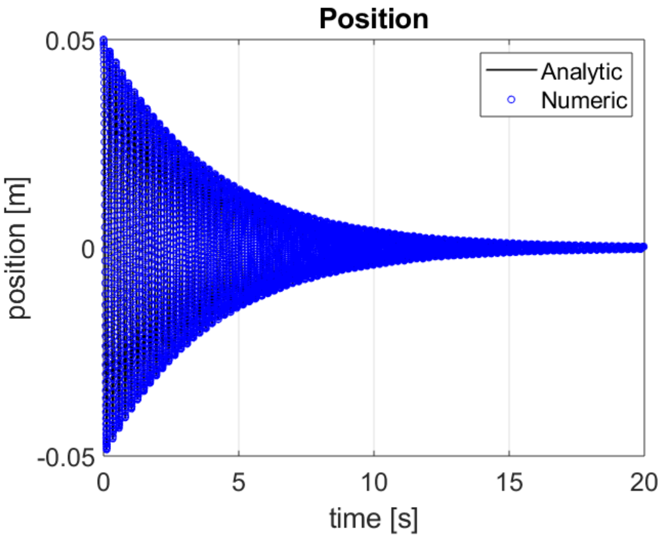
Sample solutions

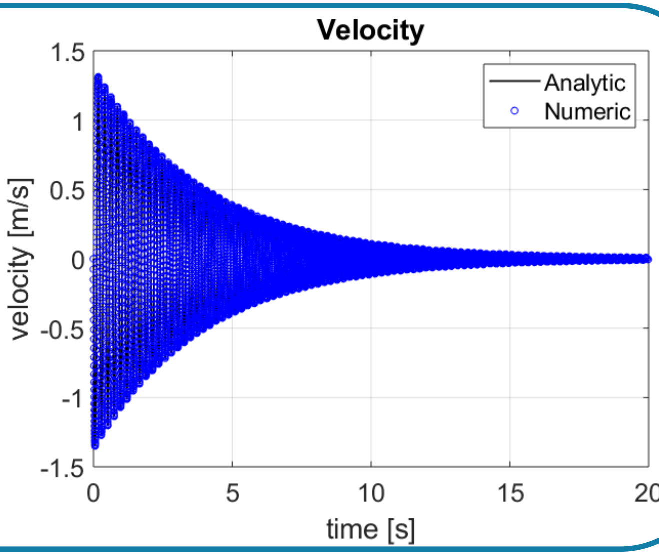
$m = 10$ [kg]
$k = 640$ [N/m]
$c = 30$ [kg/s]
$x_0 = 0$ [m]
$\dot{x}_0 = 0.1$ [m/s]





$m = 2$ [kg]
$k = 1.5$ [kN/m]
$c = 1$ [kg/s]
$x_0 = 0.05$ [m]
$\dot{x}_0 = 0$ [m/s]





Exercise 2: Orbital dynamics

Two-body problem

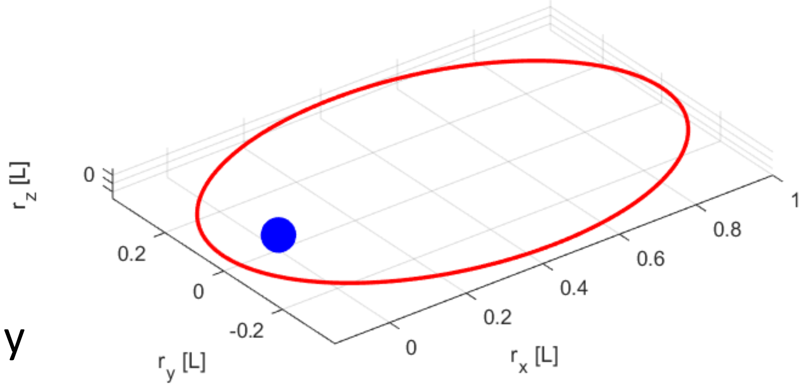
Exercise 2a: **Integrate numerically a Keplerian orbit (two-body problem)**

1. Implement the code to propagate the orbit[1]:
   - Identify the **states** of the system and the **physical parameters** involved
   - Write the *second-order ODE* describing dynamics
   - Reduce the problem to a *first-order ODE system*
   - Implement the `odefun` MATLAB function for this ODE system
   - Integrate numerically the system, choosing one of MATLAB's solvers and setting its options as needed.

Equations of motion:

$$\ddot{\mathbf{r}} + \frac{\mu}{r^3}\mathbf{r} = \mathbf{0}$$

μ: gravitational parameter of primary body



[1] **Orbit propagation**: prediction of the orbital characteristics of a body at some future date given the current orbital characteristics.

Exercise 2: Orbital dynamics

Two-body problem

Exercise 2a: **Integrate numerically a Keplerian orbit (two-body problem)**

2. Analyse the results for different initial conditions $(\mathbf{r}_0, \mathbf{v}_0)$:
   - Plot the orbit over 1 period T
     - Only elliptical (i.e. closed) orbits have a period. Hyperbolic and parabolic (i.e. open) orbits can also be computed, but they will never close.
   - Plot angular momentum vector $\mathbf{h}$ and eccentricity vector $\mathbf{e}$, and check that they remain constant in magnitude and direction
   - Check that $\mathbf{h}$ and $\mathbf{e}$ remain perpendicular (hint: plot the error of a suitable test condition)
   - Plot the specific energy ε, and check if it is constant in time
   - Plot the evolution of the radial and transversal components of the velocity

Exercise 2: Orbital dynamics

Hints

- Code your functions for a generic value of μ, so they can be reused
  - Earth's gravitational parameter: $\mu_\oplus = 398600 \text{ km}^3/\text{s}^2$
  - Sun's gravitational parameter: $\mu_\odot = 132712 \times 10^3 \text{ km}^3/\text{s}^2$
- Try different initial conditions $(\mathbf{r}_0, \mathbf{v}_0)$
  - To have a closed orbit (elliptical orbit), your energy must be $\varepsilon < 0$
- Some useful relations:

Orbital period	$T = 2\pi\sqrt{a^3/\mu}$	Specific energy	$\varepsilon = \frac{v^2}{2} - \frac{\mu}{r} = -\frac{\mu}{2a}$
Angular momentum	$\mathbf{h} = \mathbf{r} \times \mathbf{v}$	Eccentricity vector	$\mathbf{e} = \frac{1}{\mu}\mathbf{v} \times \mathbf{h} - \frac{\mathbf{r}}{r}$
True anomaly	$\cos f = \frac{\mathbf{r} \cdot \mathbf{e}}{re}$		

Exercise 2: Orbital dynamics

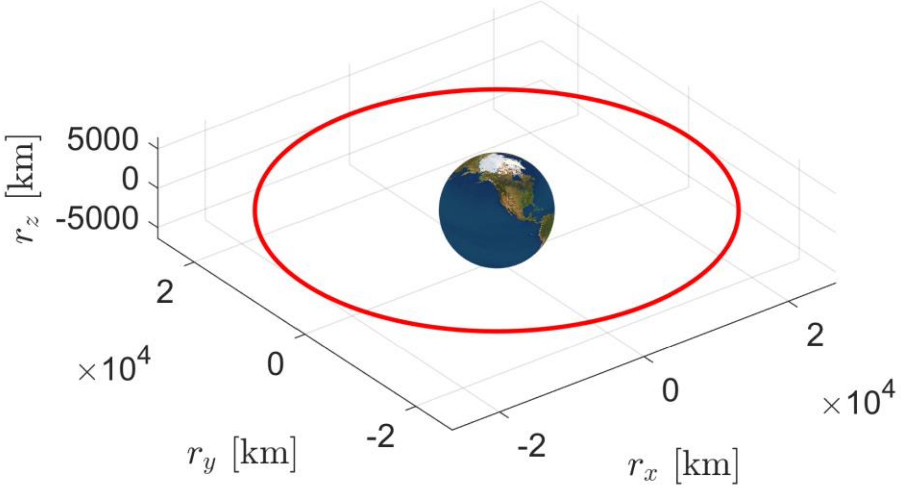
Sample solution: Quasi-circular Medium Earth Orbit

Parameters and initial condition

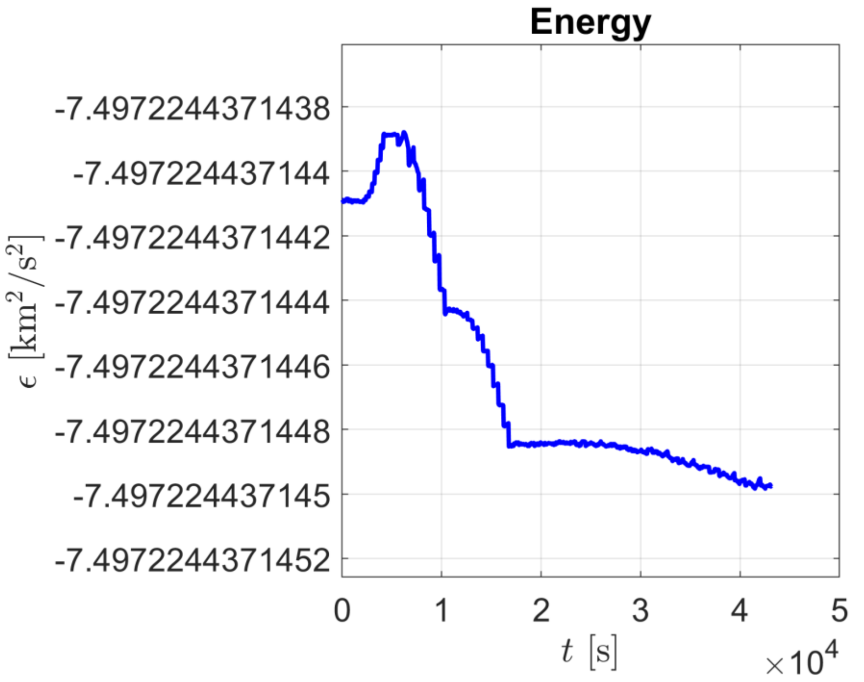
$$\mu_{\oplus} = 398600\ \mathrm{km^3/s^2}$$

$$\mathbf{r}_0 = [\,26578.137, 0, 0\,]\ \mathrm{km}$$

$$\mathbf{v}_0 = [\,0, 3.873, 0\,]\ \mathrm{km/s}$$



Orbit representation



Specific energy

Exercise 2: Orbital dynamics

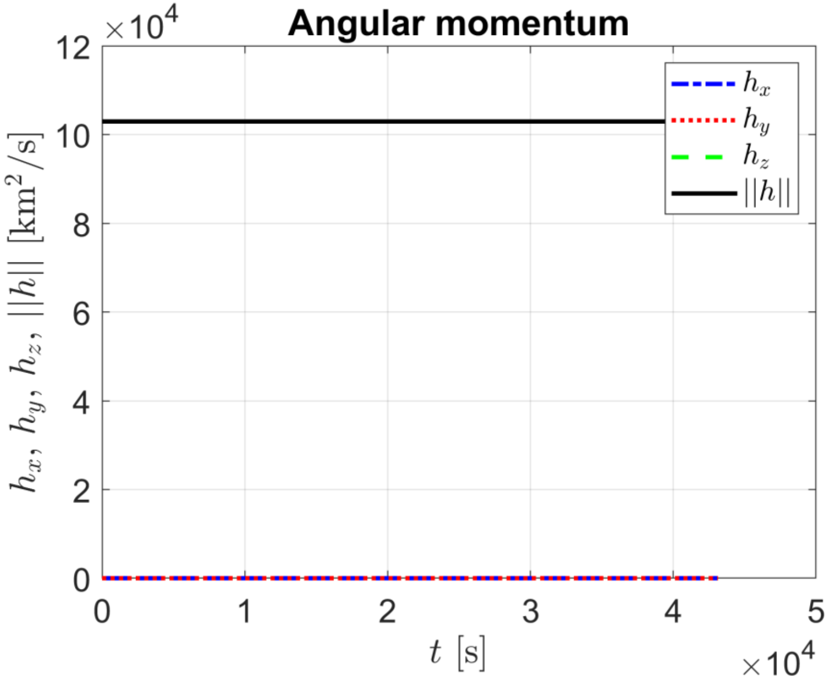
Sample solution: Quasi-circular Medium Earth Orbit

Parameters and initial condition

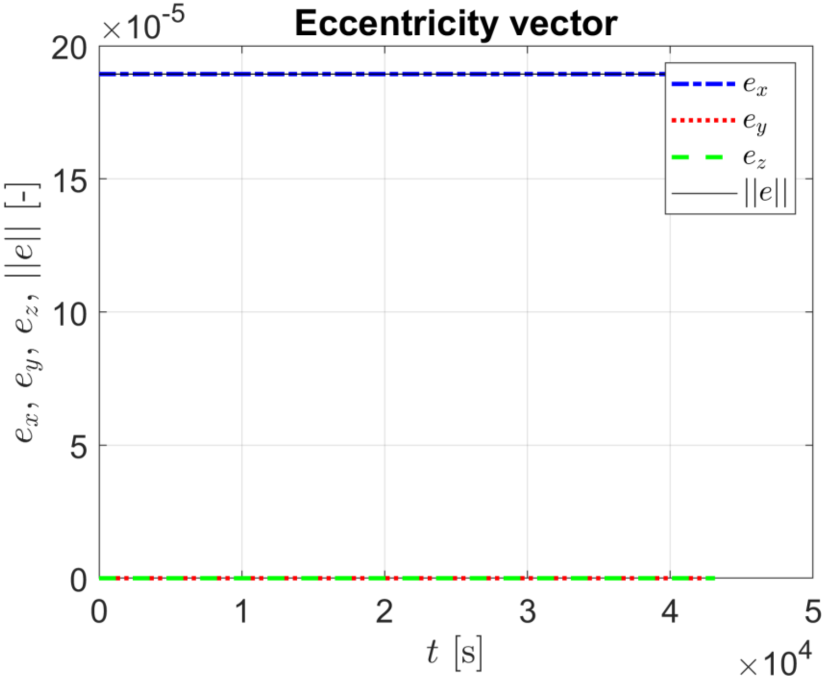
$$\mu_{\oplus} = 398600\ \mathrm{km}^3/\mathrm{s}^2$$

$$\mathbf{r}_0 = [\,26578.137, 0, 0\,]\ \mathrm{km}$$

$$\mathbf{v}_0 = [\,0, 3.873, 0\,]\ \mathrm{km/s}$$



Angular momentum



Eccentricity vector

Exercise 2: Orbital dynamics

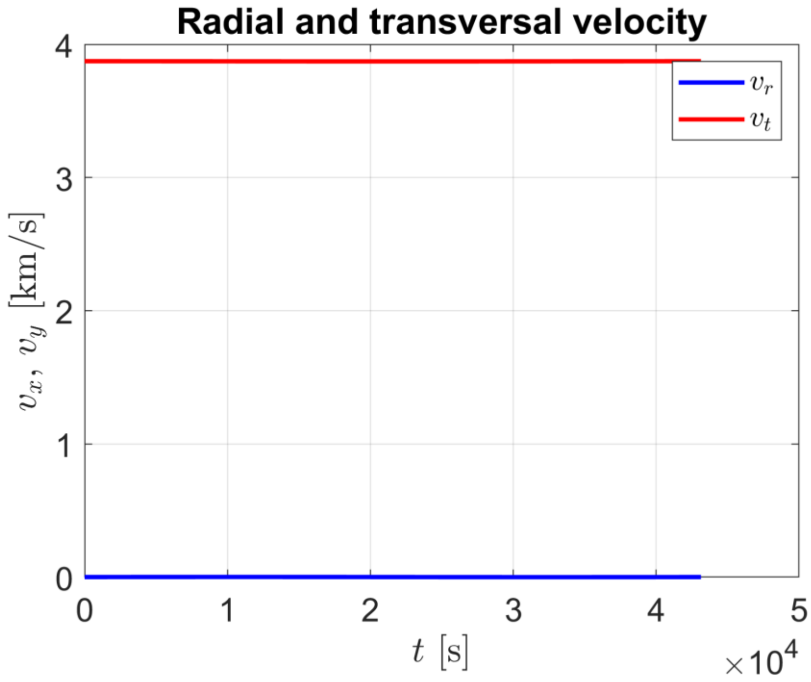
Sample solution: Quasi-circular Medium Earth Orbit

Parameters and initial condition

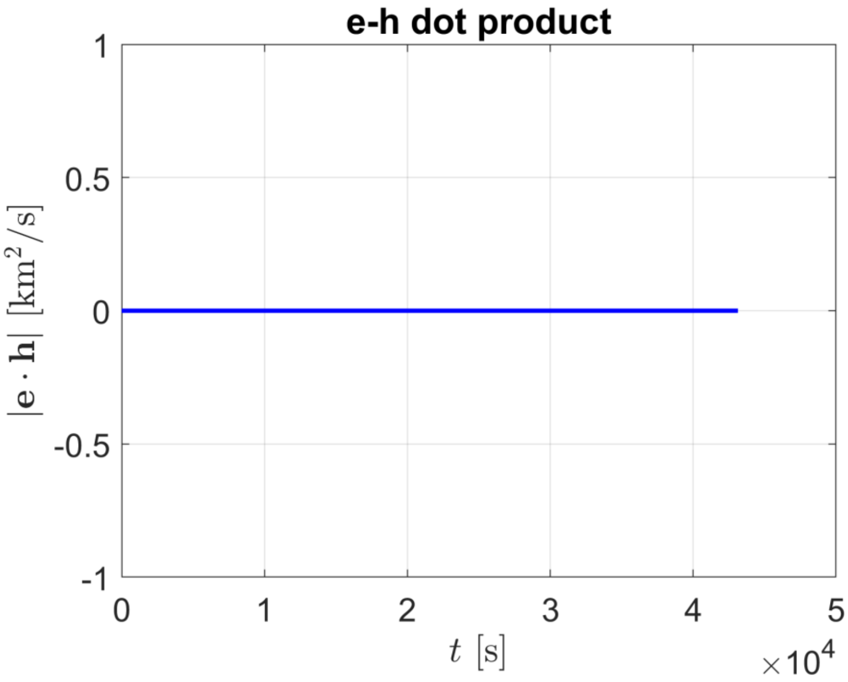
$$\mu_{\oplus} = 398600 \text{ km}^3/\text{s}^2$$

$$\mathbf{r}_0 = [\,26578.137, 0, 0\,] \text{ km}$$

$$\mathbf{v}_0 = [\,0, 3.873, 0\,] \text{ km/s}$$



Radial and transversal velocity



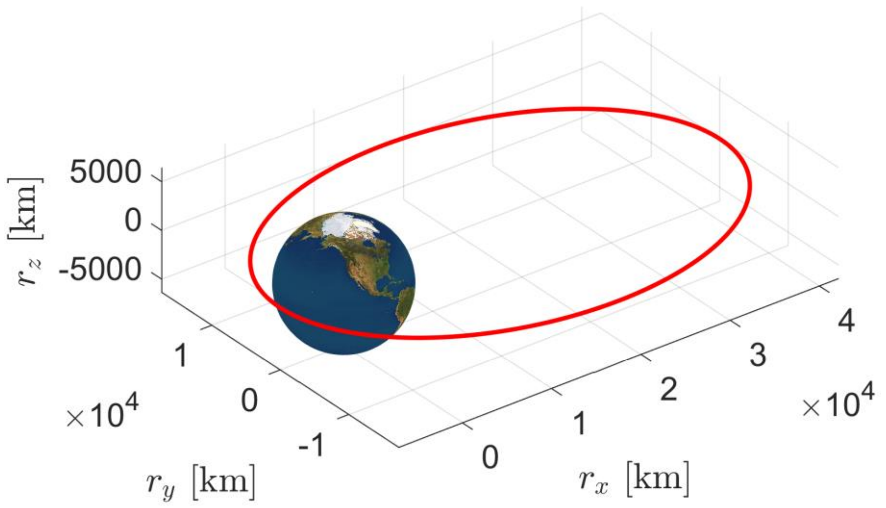
Perpendicularity condition for $\mathbf{e}$ and $\mathbf{h}$

Exercise 2: Orbital dynamics

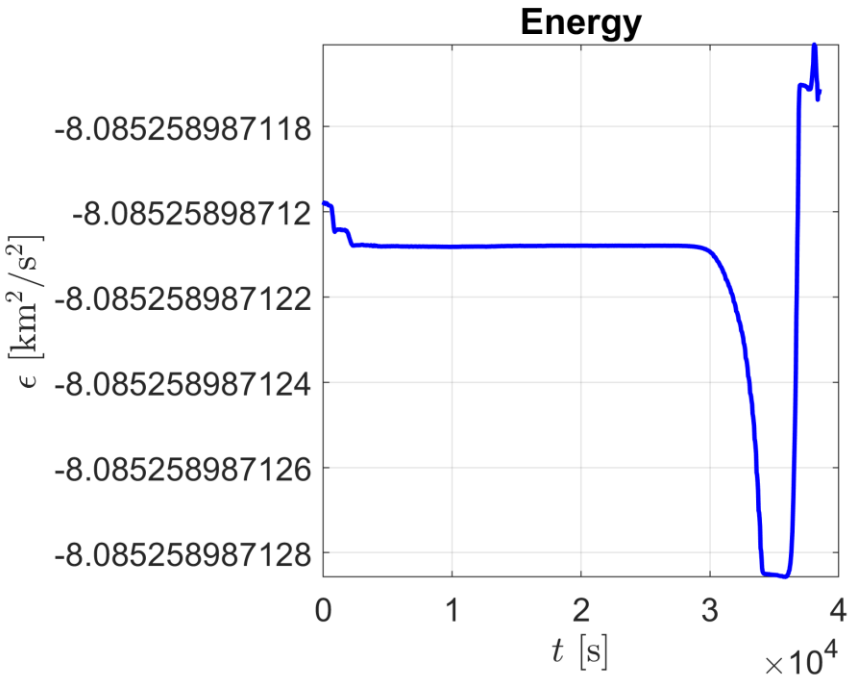
Sample solution: Geostationary Transfer Orbit (GTO)

Parameters and initial condition

$$\mu_\oplus = 398600 \text{ km}^3/\text{s}^2$$
$$\mathbf{r}_0 = [-7128.137, 0, 0\,] \text{ km}$$
$$\mathbf{v}_0 = [\,0, -9.781, 0\,] \text{ km/s}$$



Orbit representation



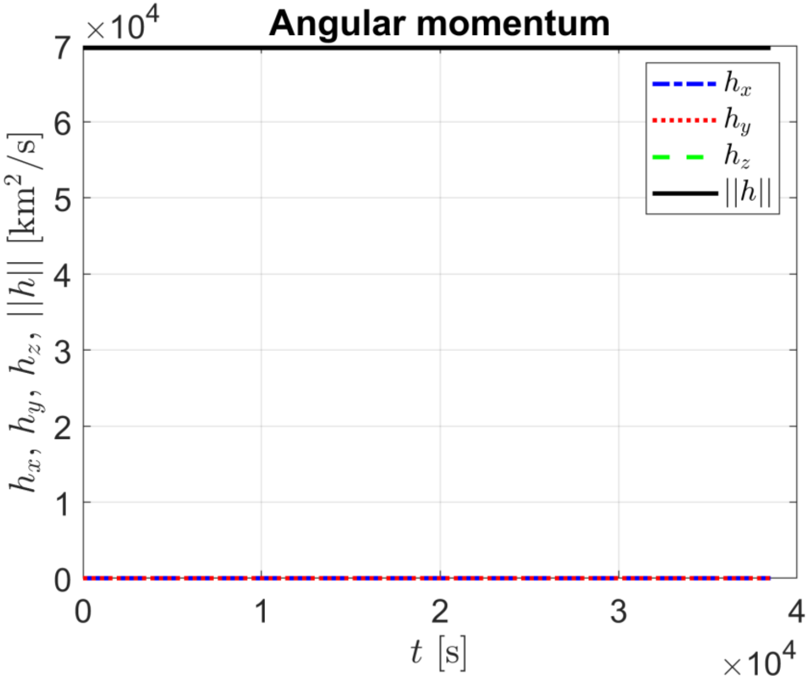
Specific energy

Exercise 2: Orbital dynamics

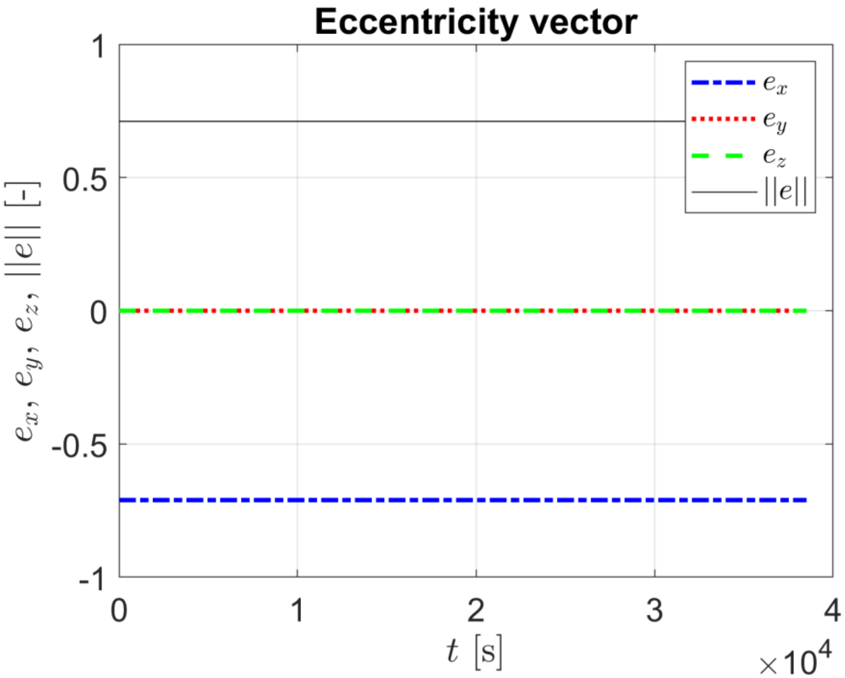
Sample solution: Geostationary Transfer Orbit (GTO)

Parameters and initial condition

$$\mu_\oplus = 398600 \text{ km}^3/\text{s}^2$$
$$\mathbf{r}_0 = [-7128.137, 0, 0\,] \text{ km}$$
$$\mathbf{v}_0 = [\,0, -9.781, 0\,] \text{ km/s}$$



Angular momentum



Eccentricity vector

Exercise 2: Orbital dynamics

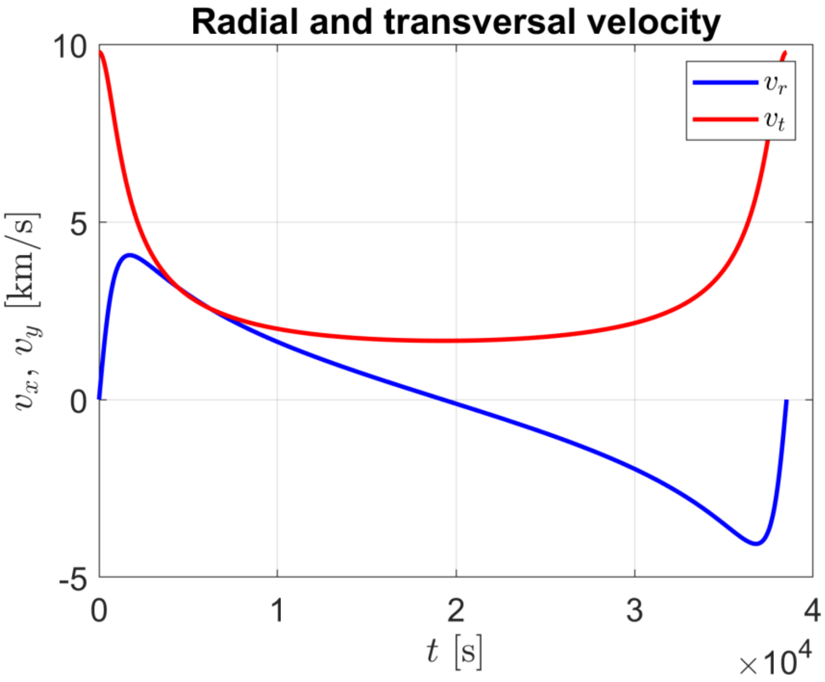
Sample solution: Geostationary Transfer Orbit (GTO)

Parameters and initial condition

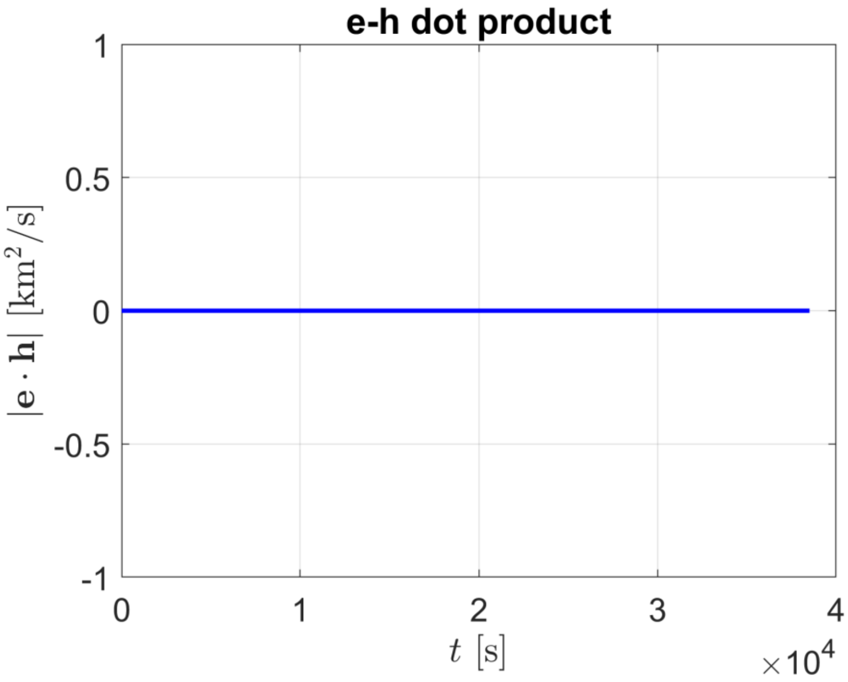
$$\mu_\oplus = 398600\ \mathrm{km^3/s^2}$$

$$\mathbf{r}_0 = [-7128.137, 0, 0\,]\ \mathrm{km}$$

$$\mathbf{v}_0 = [\,0, -9.781, 0\,]\ \mathrm{km/s}$$



Radial and transversal velocity



Perpendicularity condition for **e** and **h**

Perturbed 2BP

Non-spherical gravity fields and other perturbations

- The 2BP only accounts for **2 bodies with spherical gravity fields:**
  - In practice, the two bodies **will not be homogeneous spheres**
  - Many other perturbations will be present, depending on the orbital region: atmospheric drag, solar radiation pressure, gravitational attraction from other bodies, etc.
  - Spacecraft may perform manoeuvres using their propulsion systems

- Non-sphericity of Earth:
  - The Earth bulges out at the equator due to centrifugal forces, taking the form of an **oblate spheroid**
  - **Zonal variations**: the oblateness causes the gravitational field to depend not only on distance, but also on **latitude**. These **zonal variations** of the gravitational field can be expressed as a **series**, being the major contribution the **second zonal harmonic J_2**

Perturbed 2BP

Earth's oblateness – Effect of J_2

In Cartesian formulation ($\mathbf{r}$ and $\mathbf{v}$ in inertial reference frame), the perturbation due to the **second zonal harmonic J_2** can be expressed as:

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} + \mathbf{a}_{J_2}$$

$$\mathbf{a}_{J_2} = \frac{3}{2}\frac{J_2 \mu R_e^2}{r^4}\left[\frac{x}{r}\left(5\frac{z^2}{r^2} - 1\right)\hat{\mathbf{i}} + \frac{y}{r}\left(5\frac{z^2}{r^2} - 1\right)\hat{\mathbf{j}} + \frac{z}{r}\left(5\frac{z^2}{r^2} - 3\right)\hat{\mathbf{k}}\right]$$

where (values of R_e and J_2 taken from [2]):

- $R_e = 6378.137$ km is Earth's equatorial radius
- $J_2 = 0.00108263$
- $\mathbf{r} = x\,\hat{\mathbf{i}} + y\,\hat{\mathbf{j}} + z\,\hat{\mathbf{k}}$

[2] https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html

Exercise 2: Orbital dynamics

Effect of J_2

Exercise 2b: Earth orbit propagation with J_2

1. Modify the function from **Exercise 2a** to include also the effect of J_2
   - Add physical parameters J_2 and R_e as inputs to the function
   - Remember that the value of $\mathbf{a}_{J_2}$ has to be recomputed at each time step

2. Repeat the analysis in point **2.** of **Exercise 2a**, for initial conditions corresponding to Earth-bound elliptical orbits
   - Plot together and compare the results with and without J_2
   - Are all the conservation principles for the 2BP still valid for the orbit propagation including J_2?
   - Propagate for a 1-year time span and check how the differences between both models accumulate in time

Data

$$\mu_\oplus = 398600 \text{ km}^3/\text{s}^2 \qquad R_e = 6378.137 \text{ km} \qquad J_2 = 0.00108263$$

ROOT FINDING

Root-finding

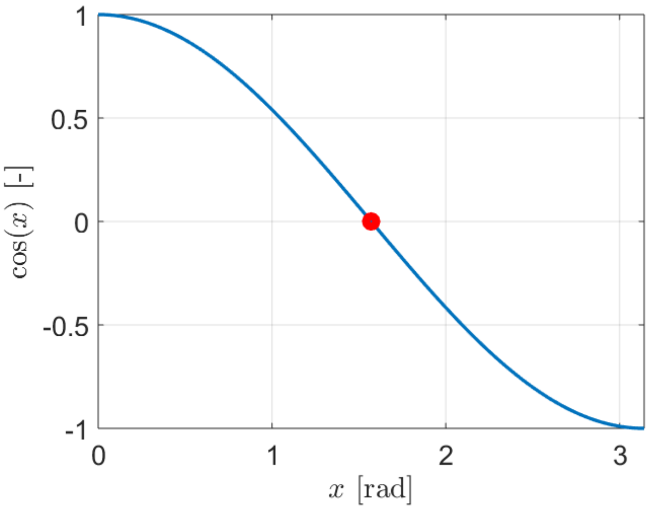
A basic problem in engineering

A root (or zero) of a function $\boldsymbol{F}(\boldsymbol{x})$ is a number $\boldsymbol{x_0}$ such that:

$$\boldsymbol{F}(\boldsymbol{x_0}) = \boldsymbol{0}$$

$\boldsymbol{F}$ and $\boldsymbol{x_0}$ can be scalars or vectors of the same dimension

- Root-finding is a very common problem in engineering
  - A classic Orbital Mechanics example is **solving Kepler's equation**
- There are many root-finding algorithms:
  - Bisection
  - Secant
  - Regula falsi
  - Newton's method
  - etc



Root-finding

MATLAB’s root-finding algorithms

- **`fsolve`**
  - Can solve systems of non-linear equations
  - Includes several algorithms to choose from
  - Very robust and versatile
- **`fzero`**
  - Works only with scalar equations
  - Combines bisection, secant, and inverse quadratic interpolation methods
  - Can be faster than **`fsolve`**, especially if bounds for $\boldsymbol{x_0}$ are provided

Read the documentation pages on **`fsolve`** and **`fzero`** to learn how to use them!

Both require as inputs the function to be solved and an initial guess for the solution

- **Anonymous functions** can be helpful

Exercise 3: Kepler’s equation

Short recap of the two-body problem

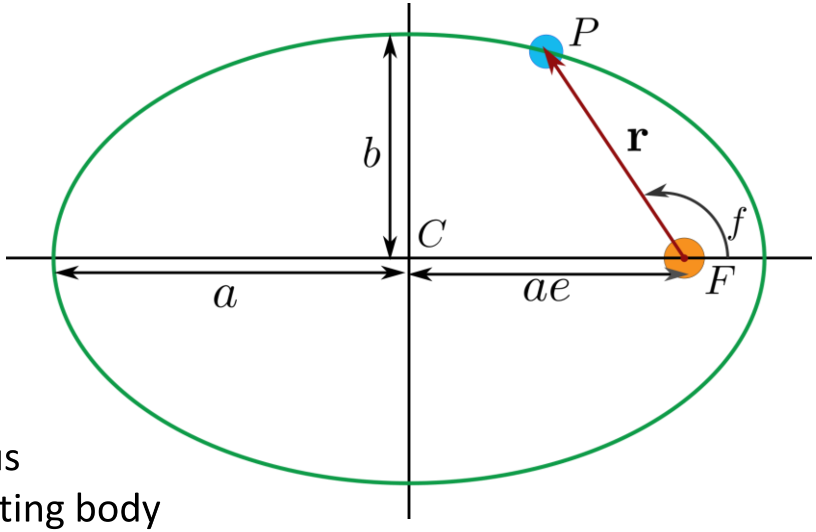
- **Two-body problem (2BP)**: Motion of two bodies subjected only to their mutual gravitational attraction
  - The bodies are points or homogeneous spheres
  - **First Kepler’s law**: In an inertial frame centred on one of the bodies (**the primary**), the **other body** describes a **conic** (ellipse, hyperbola, or parabola) with the **primary in one of the foci**

Equations of motion:

$$\ddot{\mathbf{r}} + \frac{\mu}{r^3}\mathbf{r} = \mathbf{0}$$

Polar equation of the orbit:

$$r = \frac{p}{1 + e\cos f}$$



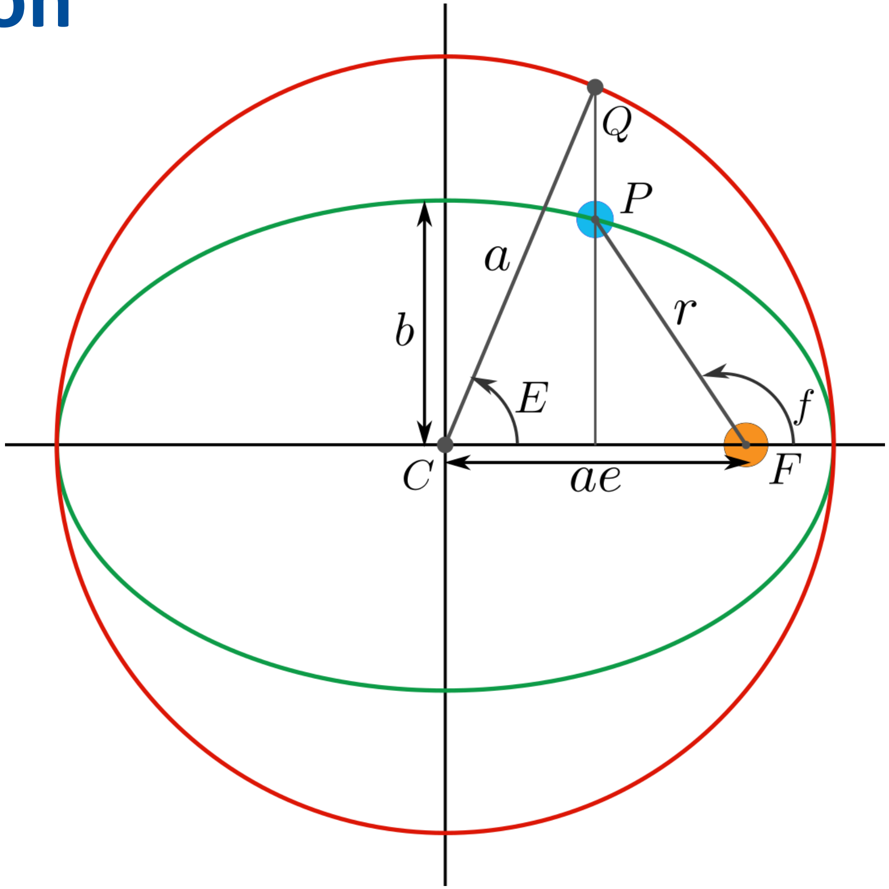
$\mathbf{r}$**:** Position vector of the secondary body
μ: gravitational parameter of primary body
p: semilatus rectum of the conic
e: eccentricity of the conic
f: true anomaly

F: focus
P: orbiting body
C: centre of the conic
a: semimajor axis
b: semiminor axis

Exercise 3: Kepler's equation

Angles in the 2BP

- **True anomaly $\boldsymbol{f}$**: angular position of body P, measured from the direction of periapsis.
- **Eccentric anomaly $\boldsymbol{E}$**: angular position of point Q, which is the projection (perpendicular to the semimajor axis) of P onto a **circle** with radius equal to the semimajor axis of the ellipse and same centre C.
  The relation between f and E is purely geometrical.



- **Mean anomaly $\boldsymbol{M}$** (not in the figure): angular position of a *fictitious body* orbiting the **circular orbit**, with constant angular velocity and the same period as the real orbit.
  M is measured over the same circle as E, but they differ because E does not have constant velocity (it follows P).

Exercise 3: Kepler's equation

Kepler's equation: time law for the 2BP

Kepler's equation provides a relation between E and M, depending on the orbit's eccentricity e. For elliptical orbits (i.e. with $e < 1$):

$$M = E - e \sin E \ .$$

$M - M_0 = E - E_0 - e(\sin E - \sin E_0)$

$\hookrightarrow = n(t - t_0)$

M is related to time t through the mean motion n (M_0 is the mean anomaly at reference time t_0):

$$M = M_0 + n(t - t_0) = M_0 + \sqrt{\frac{\mu}{a^3}}(t - t_0) \ ,$$

if $t_0 = t_p$ (TIME OF PERIGEE PASSAGE) $\Rightarrow M_0$ (& E_0) $= 0$

E is related to true anomaly f (geometrical relations):

$$\cos f = \frac{\cos E - e}{1 - e \cos E} \ , \quad \boxed{\tan \frac{f}{2} = \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2} \ .}$$

$y/x = \tan E$

$y = x \tan \frac{E}{2}$

$= \tan \frac{ho}{2}$ $\quad = \sqrt{\frac{1+e}{1-e}}$

Therefore, **Kepler's equation gives us the position of the orbiting body (true anomaly) as an implicit function of time (time law for the Keplerian orbit).**

Exercise 3: Kepler's equation

Handling multiple revolutions

- If $M < 0$ or $M \geq 2\pi$ radians, we may want to account for the **number of revolutions**
  - This is straightforward, as a complete revolution in M also corresponds to a complete revolution in E and f

- A simple algorithm to do this:
  1. Reduce M to $\overline{M} \in [0{,}2\pi]$ rad plus a whole number of revolutions $k \in \mathbb{Z}$, so that $M = \overline{M} + k2\pi$
     - **Hint:** Take a look at functions like `floor`, `ceil`, `fix`, `mod`, `wrapTo2Pi`, `wrapToPi`.
  2. Solve Kepler's equation for $\overline{M}$
  3. Compute the corresponding true anomaly $\overline{f} \in [0{,}2\pi]$ rad
  4. Add the whole number of revolutions, $f = \overline{f} + k2\pi$

Exercise 3: Kepler's equation

Implement a solver for Kepler's equation

Exercise 3a: **Implement a solver for Kepler's equation**

- **Inputs:**
  - Time t
  - eccentricity e
  - semimajor axis a
  - gravitational parameter of the primary μ
  - reference initial time t_0 and true anomaly f_0

from $f_0 \rightarrow E_0$

- **Outputs**:
  - true anomaly f

$$n(t-t_0) = E - e\sin E - (E_0 - e\sin E_0)$$

- Try and compare using both `fsolve` and `fzero`
  - You can also implement your own root-finding algorithms (e.g. Newton)
- Be very careful with the ranges of the angles to prevent discontinuities

Exercise 3: Kepler’s equation

Hints

- Read the documentation pages for `fsolve` and `fzero`
  - You may need to set the tolerance (error) for the solution
- You can use an anonymous function to pass the function to be solved to `fsolve` and `fzero`
- A good initial guess for the root is [1]: $E_{\text{guess}} = M + \dfrac{e \sin M}{1 - \sin(M+e) + \sin M}$
- You can add additional inputs as you see fit
  - It is possible to call a function without passing all the inputs. You can check the number of inputs passed using `nargin`. This way, you can program a function to have some inputs as ‘optional’. Example:

```
% Seventh input is the tolerance ‘tol’, treated as optional
if nargin<7
    tol = 1e-6; % Default value used if input not given
end
```

if nargin < 5 → f_0 & $t_0 = \emptyset$ (assume reference from $\underline{r}_p$)

[1] Battin, R., *An Introduction to the Mathematics and Methods of Astrodynamics*, AIAA Education Series, 1999

Exercise 3: Kepler's equation

Plot the evolution of true anomaly with time

Exercise 3b: Plot $f(t)$ for different orbits

1. Create a function that computes the true anomaly for an orbit with fixed e, a, and μ, for an N-points array of times covering k periods of the orbit
   - Hint: This new function can reuse the function you created in **Exercise 3a**
   - Hint: You can create arrays of uniformly distributed points in MATLAB using the `colon` operator or the `linspace` function
   - Hint: The period of an orbit is $T = 2\pi/n$, where n is the mean motion
2. Compute $f(t)$ for the following values:
   - $a = 7000 \text{ km}$
   - $\mu = 398600 \text{ km}^3/\text{s}^2$ (Earth's gravitational parameter)
   - $f_0(t_0 = 0) = 0 \text{ rad}$
   - $k = 2, N \geq 100$
   - Six different eccentricities: $e = 0, 0.2, 0.4, 0.6, 0.8, 0.95$

Exercise 3: Kepler's equation

Plot the evolution of true anomaly with time

Exercise 3b: Plot $f(t)$ for different orbits

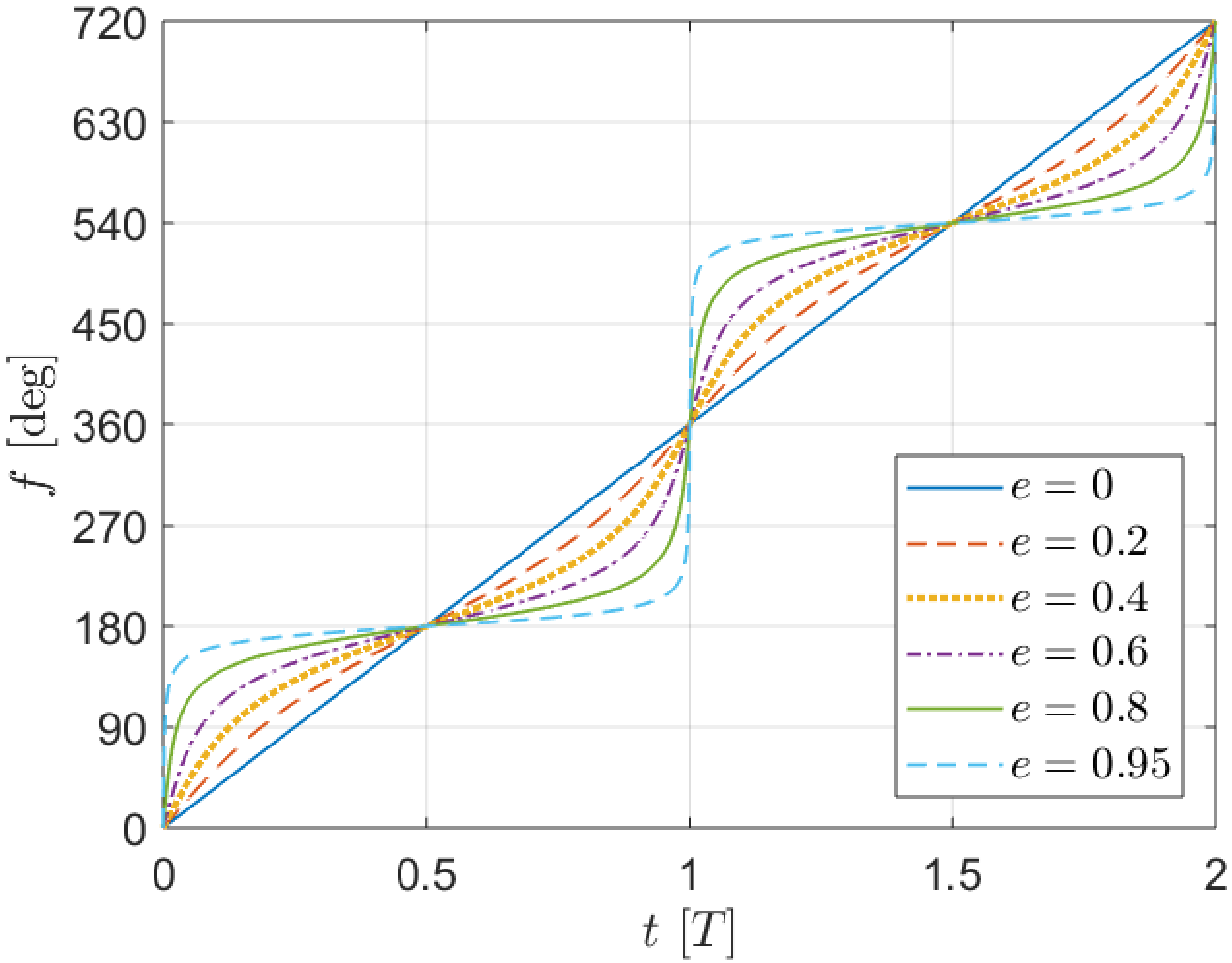
3. Plot $f(t; e)$ for all the cases in **2.** in a single 2D plot, using the `plot` command
   - Remember to set the axes and other properties of the plot
   - Hint: You can plot several lines at the same time with a single `plot` command, or you can use `hold on` to add them one by one

4. Plot $f(t; e)$ as a 3D surface using the `surf` command
   - Represent time in the x axis, eccentricity in the y axis, and true anomaly in the z axis. Use the **documentation center** to check the correct way of passing these inputs to `surf`
   - You can compute $f(t; e)$ for additional values of e to get a smoother plot

5. Compare with the results obtained from the numerical propagation of the 2BP in **Exercise 2**

Exercise 3: Kepler’s equation

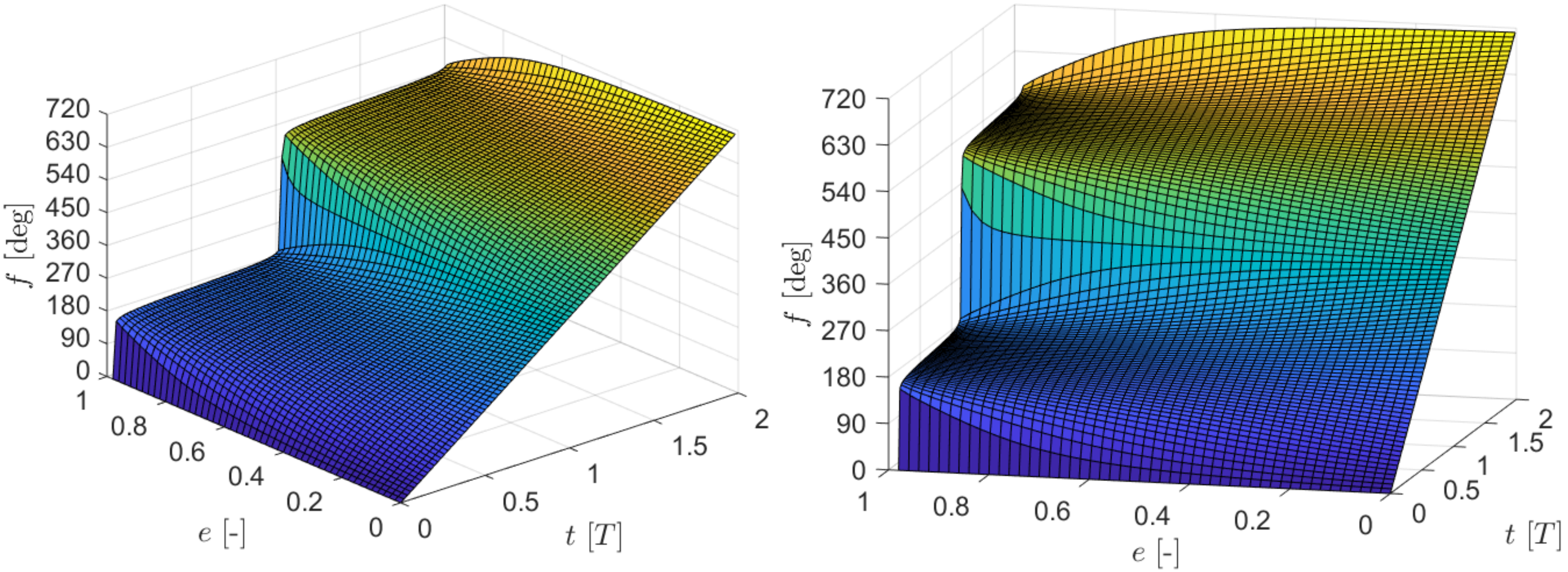
Solution: 2D plot



Evolution of true anomaly with time for $a = 7000\ \mathrm{km}$, $\mu = 398600\ \mathrm{km}^3/\mathrm{s}^2$, and different eccentricities

Exercise 3: Kepler's equation

Solution: surface plot



Evolution of true anomaly with time and eccentricity for $a = 7000$ km and $\mu = 398600$ km^3/s^2