# IMP
# (Interpolated Motion Planning)

User's Manual

Version E

ETEL

THIS PAGE IS INTENTIONALLY LEFT BLANK

# Table of contents

THIS PAGE IS INTENTIONALLY LEFT BLANK

## Record of revisions concerning the Interpolated Motion Planning:

| Document revisions | | |
|---|---|---|
| **Version** | **Date** | **Main modifications** |
| Ver A | 21.12.16 | First version (with IMP 1.00A) |
| Ver B | 09.11.17 | Updated version (with IMP 1.01A):<br>- Way point function updated (refer to §6.2.3)<br>- Trigger configuration updated (refer to §6.2.4)<br>- New function: Check conditions within interval (refer to §6.4.1)<br>- Save & load trajectory (refer to §6.6)<br>- Serviceability updated (refer to §6.8) |
| Ver C | 16.10.18 | Updated version (with IMP 1.02A, 1.03A):<br>- Geometry fitting updated (refer to §6.2.2)<br>- Traces acquisition updated (refer §6.2.4)<br>- Minor changes |
| Ver D | 06.09.19 | Updated version:<br>- Min-max box evaluation updated (refer to §6.4.2) |
| Ver E | 27.08.21 | Updated version (with IMP 1.04A):<br>- Update of supported Windows Operating Systems and development environment (refer to §3.).<br>- Minor changes |

## Documentation concerning the Interpolated Motion Planning:

| | |
|---|---|
| • **User's Manual** | **Interpolated Motion Planning principle of operation** |
| • HTML Reference Manual | IMP complete list of functions available |
| • EDI4 User's Manual | ETEL Device Interface principle of operation |

**Remark:** The HTML documentation is generated for each IMP release, being its most up-to-date reference manual.

# 1. Introduction

Nowadays market is pushing for trajectory generation applications demanding high performance with all kind of specifications. High trajectory precision, geometry fitting features and automatic transitions respecting different kinematic constraints (velocity, acceleration, jerk ...) are some of the most important requirements. Moreover, industrial applications need event management capabilities to execute specific actions at precise moments of the trajectory execution.

The Interpolated Motion Planning (IMP) library aims to provide a global solution regarding the aforementioned points.

**Remark:** The updates between two successive versions are highlighted with a modification stroke in the margin of the manual.

# 2. Definitions and acronyms

## 2.1 Definitions

- **Trajectory**: in the scope of the IMP development, the trajectory is a solution composed of:
  - Geometrical primitives,
  - Automatic transitions,
  - Kinematics constraints (this is established once for the whole trajectory).

In addition, it can also contain trigger actions. Once the trajectory has been defined using the previous elements, it can be compiled and finally executed.

- **PVT**: Position, Velocity and Time. Algorithm related to construct 3rd order Hermit splines:

$$x(t) = \frac{1}{6}jt^3 + \frac{1}{2}a_0t^2 + v_0t + x_0$$

  Extensively used in the motion planning field.

- **G-CODE**: this is the common name for the most widely used numerical control (NC) programming language. It is used mainly in computer-aided manufacturing to control automated machine tools. It defines a set of basic trajectory primitives to execute simple movements. In the case of ETEL's G-CODE commands, we can find circle arcs and line primitives.

- **Waypoint**: this is a passing point with vectorized kinematics constraints (position, velocity and acceleration vector) which must be placed only between automatic transition segments.

- **Bezier curve**: a Bezier curve is a parametric curve defined by a set of control points $p_0$ through $p_n$, where n is the order (n = 1 for linear, 2 for quadratic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

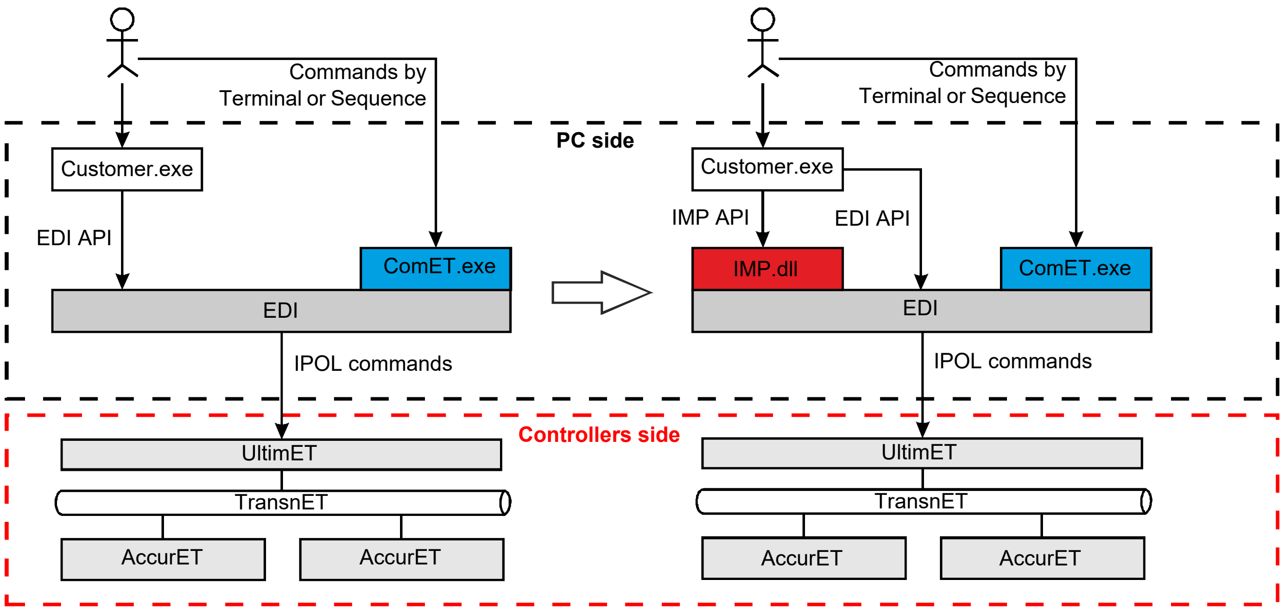
- **Workspace limits**: generally the user should fix the limits at the Min Software Limits (MINSL) and Max Software Limits (MAXSL) defined for each axis. However, one could imagine to reduce the space dimension for certain cases.

## 2.2 Acronyms

- **PVT**: Position Velocity and Time.
- **IMP**: Interpolated Motion Planning library.
- **API**: Application Programming Interface.
- **DLL**: Dynamic Link Library

# 3. System configuration

IMP is a Windows library working on a PC environment above ETEL's EDI middleware (which establishes the interface between the application and firmware levels) as depicted in the following figure.



The presented architecture requires specific configurations.

| PC configuration | |
|---|---|
| Machine requirements | • Minimum 2 GB of memory (very dependent on the quantity of primitives in the trajectory)<br>• Current generation of processors<br>• Windows 10 32/64-bit |
| Library & compilation requirements | • 32/64-bit compatible library<br>• Visual Studio 2019<br>• C++ library for a C++ customer application<br>• IMP is not a thread-safe library |
| EDI | • Version 4.25A or later<br>• Customer application must be linked to EDI dsa40.dll |

| Controllers configuration | |
|---|---|
| UltimET | • Firmware version 3.21A or later<br>• Only for UltimET PCI or PCIe with interpolation |
| AccurET | • Firmware version 3.21A or later |

**Remark:** IMP may run on other Windows operating system versions. However, the qualification of the IMP library has been realized under the conditions listed in the qualification environment table presented below. ETEL cannot guarantee full technical support for issues occurring under different operating system conditions.
ETEL does not support operating systems that are no longer supported by their respective vendors.

| Qualification environment | |
|---|---|
| Operating system | Windows 10 Enterprise 64-bit, version 1909 |
| Processor type | Intel ® Xeon® W-2125 CPU @ 4.00 GHz 4-core, ID 50654 |
| System memory (RAM) | 16 GB |
| Development environment | Visual Studio 2019 |
| Application type | 64-bit application, linked to EDI 4.25A |

# 4. What is IMP?

IMP is a trajectory generation and execution library optimized for ETEL products using interpolation features on UltimET and AccurET controllers.

The user will be able to define two types of elements:

- **Geometry Fitting**: geometrical element defined by the user and complying with parameters:
    - Shape:
        - Straight line primitive.
        - Circle arc primitive.
        - Elliptical arc primitive.
        - Bezier curve primitive.
        - Polynomial curve primitive.
    - Kinematic bounds:
        - Max. norm (absolute) geometry fitting error.
        - Max. norm (absolute) tangential velocity.
        - Max. geometry fitting error per axis.
        - Max. velocity per axis.
        - Max. acceleration per axis.
        - Max. jerk per axis.

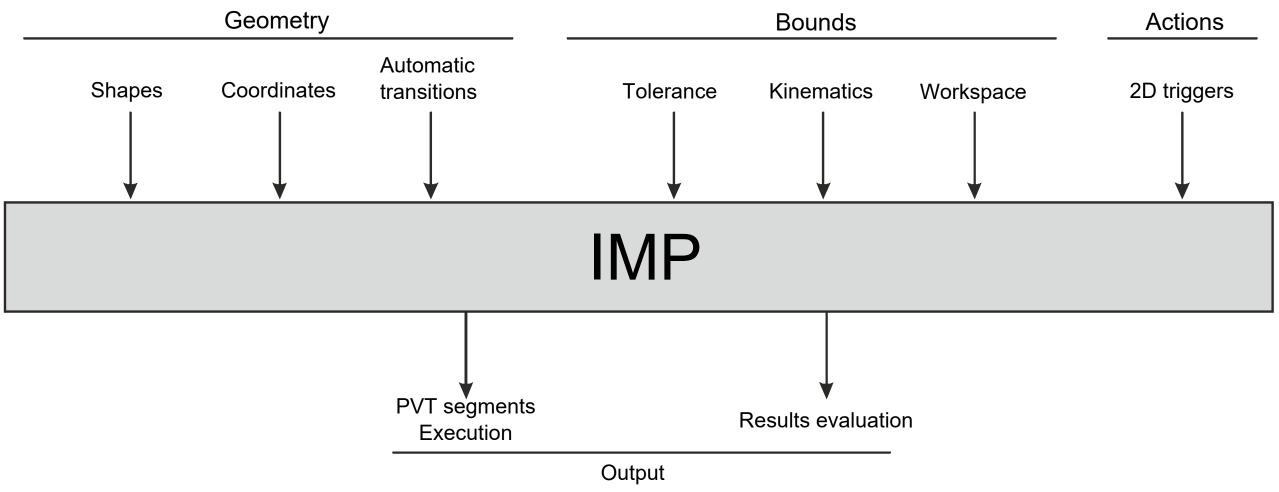
- **Automatic Transitions**: trajectory element automatically generated by the library between two points, complying with parameters:
    - Kinematic bounds:
        - Max. velocity per axis.
        - Max. acceleration per axis.
        - Max. jerk per axis.
    - Workspace limits.
    - Additionally, waypoints (the presence is optional) can be placed between automatic transitions complying with:
        - Passing point coordinates.
        - Velocity vector constraints at point.
        - Acceleration vector constraints at point.

So that, even though the automatic transitions have a geometry shape which is not managed by the user, he can impose checkpoints forcing the trajectory to pass by and respect vectorized kinematics constraints (velocity and acceleration).

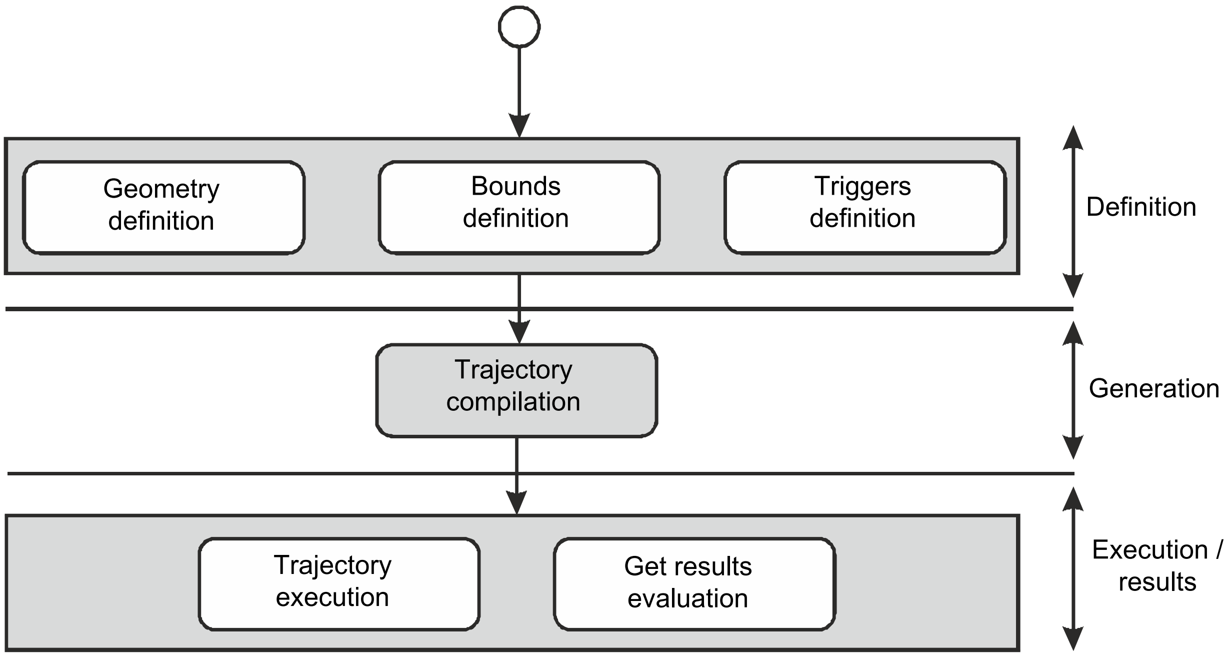
The final trajectory will be the result of a sequence of geometry fitting and automatic transitions segments (with or without waypoints presence in-between).

The IMP library also provides event management capabilities. The user will be able to place customized triggers along the trajectory definition which represent specific actions launched during execution.

The IMP provides the execution of a PVT list complying with the corresponding inputs and the results evaluation to check if the user definitions and input constraints have been respected.



The general workflow of the IMP library is composed by 3 phases:
1: Definition.
2: Generation.
3: Execution & results.



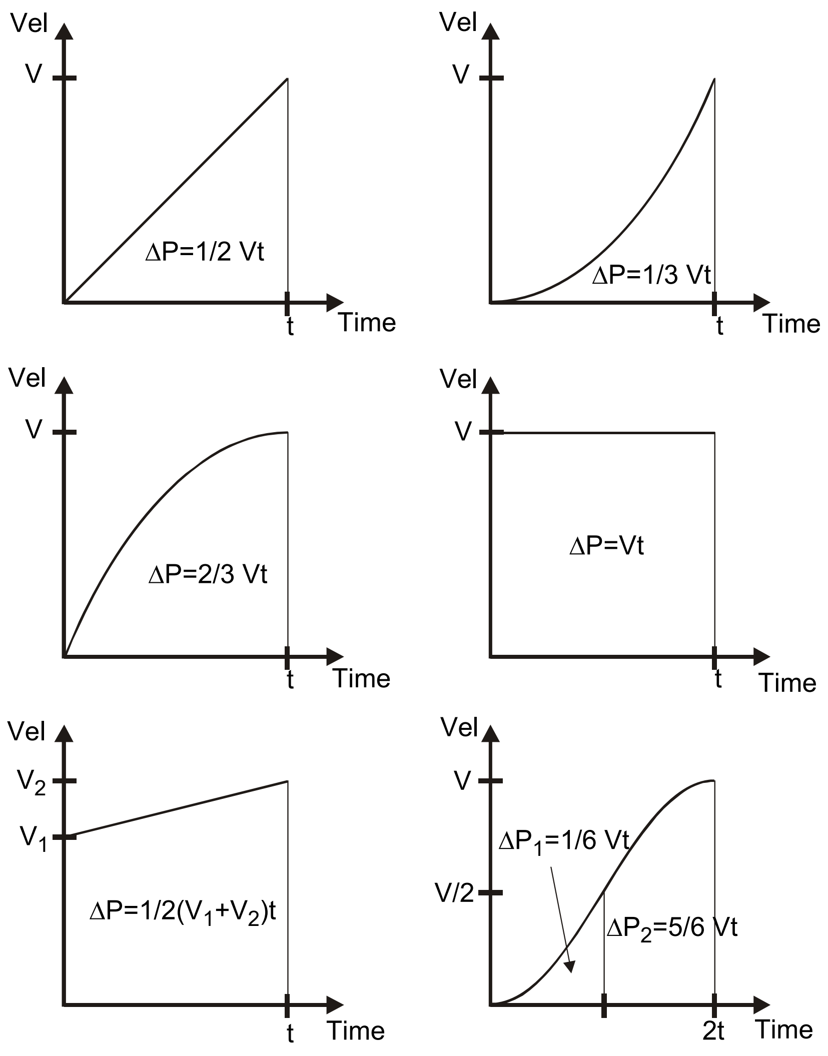
## 5. Why IMP?

In the current UltimET version, a mix of interpolation primitives (e.g. G-CODE + PVT…) for the same trajectory execution is not possible; so the user cannot take advantage of geometry fitting features (provided by G-CODE segments) and controlled-kinematics transitions (provided by PVT segments) at the same time. Moreover, at low level it is impossible to ensure that the kinematics boundaries are respected in a trajectory generation. IMP is a solution which represents a compilation of both requirements in the same trajectory accepting kinematics constraints.

In previous studies it has been proved that PVT segments between 1 to 3 ms can be suitable to satisfy geometry fitting for industrial applications. Moreover, PVT segments (3rd order polynomials) represent the most flexible primitive implemented in the UltimET interpolation module.
IMP takes advantage of this feature already present in the UltimET product. Once the trajectory is generated (compiled), all calculations will be translated to PVT interpolation primitives with a correct adapted sampling time.

For more detailed information; as mentioned in the UltimET User's Manual, we can define PVT segments as follows:

"The PVT movement describes a segment by giving its arrival point B (the starting point A is the current position), its speed vector at this point and the time to run this segment. The speed profile is then defined by the user who can execute the trajectory according to his needs. On the other hand, this mode requires more complex calculations from the user.
The algorithm calculating the trajectory between two points is based on a third-degree polynomial. A speed continuity is ensured thanks to the degree of the polynomial; on the other hand, the user has to manage the continuity of the acceleration and to limit the jerk when he wants to make a trajectory."



IMP also provides functionalities to place trigger actions along the trajectory. Even though triggers can be used independently from the IMP, this library integrates advanced functionalities as **offset triggers** (i.e. time delay or curvilinear distance displacement) or **smart triggers** (in mode and direction) which entail to set triggers into a trajectory context; thus these features are ONLY available under IMP use.

# 6. How IMP works?

In the following chapters, the use of the IMP is presented. The prototypes which describes the use of each functionality are mentioned for the corresponding features. The complete IMP API can be found in §10.. For more information about it, please, check the corresponding Doxygen (HTML) documentation.

## 6.1 Common structures

At different moments of the process, the user will need to create global structures to manipulate or exchange data during the execution of certain functions. IMP provides these structures with their corresponding management prototypes (create, get, set, delete):

- **IMP_TRAJECTORY**: trajectory definition.
- **TRIGGER_2D**: 2D Trigger definition.
- **EVAL_LIST:** evaluation list for results report.
- **TIME_STAMP_LIST:** list of time marks along the trajectory for results report.

The parameters used in the corresponding functions must be expressed in the following units:
Position: **[m]** or **turns [t]**

Velocity: **[m/s]** or **[t/s]**
Acceleration: **[m/s$^2$]** or **[t/s$^2$]**
Time: **[s]**

## 6.2 Trajectory definition

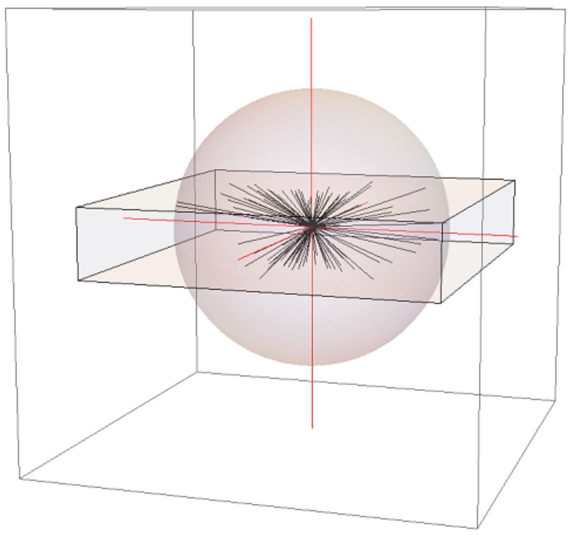
### 6.2.1 Trajectory bounds

The bounds (kinematics, tolerances, and workspace) are global to the whole trajectory. This function should be executed once per trajectory. There is no strict order between trajectory geometric calls (geometry fitting and automatic transitions) and bounds setting. We can execute one before the other or vice versa. However, both geometric and bounds definition must be executed before the trajectory generation (compilation).

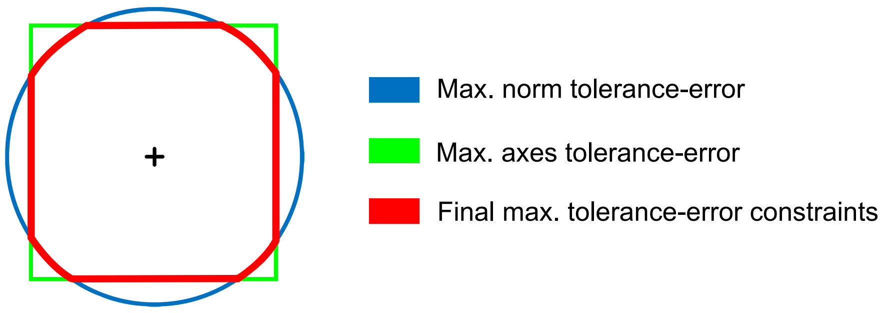
Using the function `imp_setTrajBounds`, the user can define the following bounds by parameter:

- **Workspace limits**: used only for the calculation of automatic transitions.
- **Max geometry fitting error bounds**: used only for the calculation of geometry fitting.
- **Max tangential velocity**: used only for the calculation of geometry fitting.
- **Max geometry fitting error per axis**: used in both calculations, geometry fitting and automatic transitions.
- **Max kinematics (speed, acceleration, jerk) per axis**: used in both calculations, geometry fitting and automatic transitions.

For geometry fitting calculation, there is a mix of constraints: norm constraints and axis constraints. In the following figure we depict a 3D example where both are combined (as it is the case for geometry fitting error or velocity). The constraints satisfaction must comply with the overlapping of both limits (rectangular prism for axes constraints and sphere for norm constraints):



In 2D this is the overlapping between a rectangle and a circle:


+
🟦 Max. norm tolerance-error
🟩 Max. axes tolerance-error
🟥 Final max. tolerance-error constraints

The automatic transitions only take into account the kinematics axes constraints. Thus, the free movement is not constrained by the tangential velocity imposed during the geometry fitting.

The parameters used in this function must be expressed in the following units:
Position: **[m]** or **[t]**
Velocity: **[m/s]** or **[t/s]**
Acceleration: **[m/s$^2$]** or **[t/s$^2$]**

### 6.2.2 Geometry fitting

The user can define trajectory primitives which will comply with certain geometry shapes, specific position tolerances and maximum kinematics thresholds. The current version of the IMP supports:

- Straight line primitives.
- Circle arc primitives.
- Elliptical arc primitives.
- Bezier curve primitives.
- Polynomial curve primitives.

Each one of the primitives will be inserted in the trajectory using `imp_add[specific_primitive]` functions; so that the user will be able to build a geometric sequence.

The user interface philosophy to define geometry primitives with the corresponding aforementioned functions is inherited from the machine tool domain; it is assumed that the last point of the current defined trajectory corresponds to the first point of the subsequent added primitive. It means the user must start defining the trajectory by calling `imp_setFirstTrajPoint`. If the user, for instance, continues by adding a line, just one target point will be passed as parameter assuming the first trajectory point (the last of the current trajectory) as the line start. The same concept is applied for all the geometry primitives.

These are the functions related to geometry fitting features:

- **`imp_setFirstTrajPoint`:** mandatory function call at the beginning of geometry definition. It fixes the origin of the trajectory.

- **`imp_addTrajLine`:** the target point is required by parameter. The line is defined between 2 points: the last trajectory point and the target one.

- **`imp_addTrajCircleArc`:** 2 points (p1, p2) and an angle are required by parameter. The circle shape is defined using 3 passing points: the last trajectory point, p1 and p2. None of the points (p1, p2) defines the final arc point, but only the circle shape constraints. The circle arc is defined by means of the angle in [radians]. The angle range is not limited to a single circle loop; it means that it can exceed 2π in value, so that the trajectory will continue revolving along the same circle till the specific angle is reached. There is no clockwise or counterclockwise meaning in circle direction; the sense of rotation is established as follows:
    - angle > 0 → the arc direction will be defined starting the rotation towards point p1.
    - angle < 0 → the arc direction will be defined starting the rotation towards point p2.

For numerical reasons, even if the intended arc is short (small angle), it is advisable to pass points evenly spread on the circle.

- **`imp_addTrajEllipticalArc`:** the elliptic arc is defined by means of 5 elements: the vector in the corresponding space which defines the semi-major axis, the vector for the semi-minor axis, the start angle φ, the offset angle α and the starting point of the ellipse p0 in the space. The point p0 is implicitly defined by the last point of the current trajectory; hence, the function will take the other 4 elements by parameters.

A point on the ellipse is defined using the parametric angle $\varphi$ by means of the following equation:

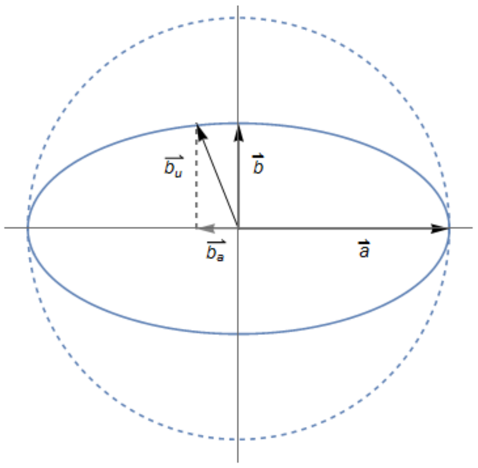
$$\vec{p}(\varphi) = \vec{a}\cos(\varphi) + \vec{b}\sin(\varphi) + \vec{c}$$

where $\vec{a}$ and $\vec{b}$ respectively define the semi-major and semi-minor axes of the ellipse. The vector $\vec{c}$ is the ellipse center. If the vectors $\vec{a}$ and $\vec{b}$ provided by the user are not perpendicular to each other, the vector $\vec{b}$ is replaced by:

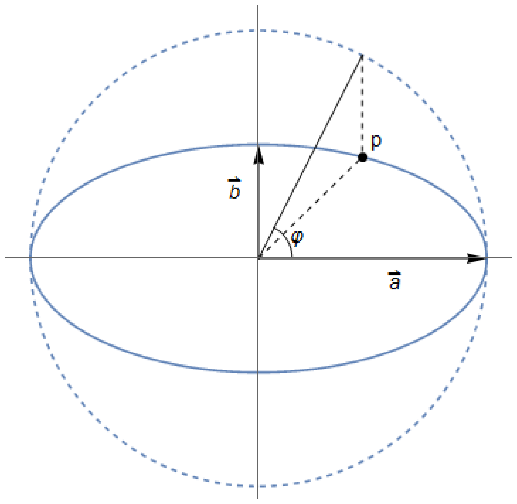
$$\vec{b} \to \vec{b} - \langle \vec{a}, \vec{b} \rangle \frac{\|\vec{b}\|}{\|\vec{a}\|}\vec{a}$$

i.e. the projection of $\vec{b}$ on $\vec{a}$ is removed from $\vec{b}$.

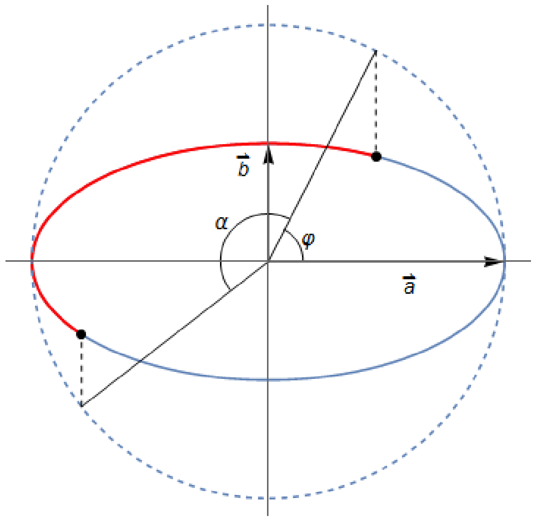
The user should provide semi-major and semi-minor axes a and b that are perpendicular to each other. If this is not the case, the user supplied semi-minor axis will be modified so as to become perpendicular to the semi-major axis. The figure below gives such an example where the user specified the axes as a and $b_u$. Since the provided axes are not perpendicular, the projection $b_a$ of $b_u$ on a is computed and subtracted from the supplied $b_u$, giving a corrected semi-minor axis b.



The point $\vec{p}$ is shown in the following figure, drawn in the plane defined by vectors $\vec{a}$ and $\vec{b}$. Note that angle between the line from the ellipse center to the point $\vec{p}$ and the semi-major axis is *not* equal to φ in general (which is the parametric angle, projected from the circle to the ellipse in the direction of the semi-minor axis following the corresponding parametric equation).



To define an elliptic arc, in addition to the vectors $\vec{a}$ , $\vec{b}$, the user needs to specify a starting parametric angle φ and a parametric arc-length α. The elliptic arc is hence defined by the parametric angle range $[\varphi, \varphi + \alpha]$ depicted as follows:



The direction of +α is given by the arc sense from the defined semi-major axis to the defined semi-minor axis.

Two important notes:

- The Euclidean length of the provided semi-major axis $\vec{a}$ *must* be larger or equal to the Euclidean length of the semi-minor axis $\vec{a}$ , i.e $\|\vec{a}\| \geq \|\vec{b}\|$
- It is recommended *not* to use ellipses to define circles. Computation time with ellipses is much larger than the computation time related to circles (about 10x).

- **`imp_addTrajBezier:`** the degree of the Bezier curve and the Bezier control points list are required by parameter. The last point of the current trajectory will be taken as the first Bezier control point added to the list passed by parameter. A polynomial is built using the input data as follows:

$$B(t) = \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k P_k,\ \text{degree} = n,\ \forall k P_k \in \text{control list}$$

- **`imp_addTrajPoly:`** polynomial coefficients, polynomial degree and independent variable range ($x_{max}$) are required by parameter. A polynomial is built using the input data as follows: (x)= ~~$c_0$~~ + $c_1x$ + $c_2x^2$ + ... + $c_nx^n$; degree = n; 0 > x > $x_{max}$. It is important to notice that the user must not pass the independent coefficient $c_0$ because it is defined by the last point of the current trajectory.

The parameters used in the corresponding functions must be expressed in the following units:
Position: **[m]** or **turns [t]**
Angle: **[rad]**

Polynomial or Bezier primitives are two different ways of defining the same thing: polynomial shapes (internally represented as Bezier curves). Directly defining them as Bezier primitive is numerically more stable.

The polynomial order (degree) for polynomial geometric definitions (poly or bezier primitives) can be defined up to 5.

Bezier segments with multiple first or last control points are forbidden. That is, if $P_0 \ldots P_n$ (n = degree) are the Bezier control points; the user must ensure that $P_0 \neq P_1$ and $P_{n-1} \neq P_n$.

In this way, since user-defined polynomial primitives are internally converted to Bezier segments, the user cannot provide any arbitrary values for the polynomial coefficients. The above condition translates to the following. Given the following polynomial equation for one axis:

$P(x)=c_0+c_1\ x+c_2x^2+\ldots+c_nx^n$; degree = n; $0<x<x_{max}$

the user must make sure that the polynomial has non-zero derivatives at x = 0 and x = $x_{max}$,
i.e. $P'(0) \neq 0$ and $P'(x_{max}) \neq 0$

**Example 1:**
The polynomial:

$P(x) = 3 + 4x^5$, $x_{max} = 1$

is forbidden since

$P'(x) = 20x^4 \rightarrow P'(0) = 0$ and $P'(1) = 20$

**Example 2:**
The polynomial

$P(x) = 7 - 8x + 4x^2$, $x_{max} = 1$

is forbidden since

$P'(x) = -8 + 8x \rightarrow P'(0) = -8$ and $P'(1) = 0$

In both examples, one of the polynomial end-points has zero derivative; therefore the polynomials cannot be used. Otherwise the following error would be displayed:

_checkControlPointMultiplicity. Defining Bezier geometric segments with multiple first or last control points is not accepted.

### 6.2.3 Automatic transitions

The user can define an automatic transition between 2 given points without the need of defining the shape between them. IMP will *try* to build a trajectory transition guaranteeing a certain level of smoothness (at least till 3rd derivative level, i.e. jerk) at the entry and at the out point depending on their corresponding trajectory connections. The automatic transitions can be added at the beginning or at the end of the whole trajectory. They can also be added in the middle, between two geometry fittings.
The resulting transition will comply with the tolerances, the workspaces limits and the axes kinematics constraints (not the tangential ones).

`imp_addTrajAutoTransition` function defines the automatic transitions. A target point is required as parameter. The transition will be established between the last point of the current trajectory and the defined target point.

The algorithm which evaluates the automatic transitions tries to find the best suitable connection between the start and target point complying with geometrical/kinematics constraints; however, the generation of a transition satisfying all the given constraints may be impossible or it may fail even though a feasible solution exists. An error will be generated in such cases. The error `IMP_ETRAJ_COMPILATION_WARNING` might be notified with the according explanation. In this case the trajectory has been compiled anyway (only some constraints have been exceeded and a description is displayed in the terminal) and the user can take the decision of executing it or not.

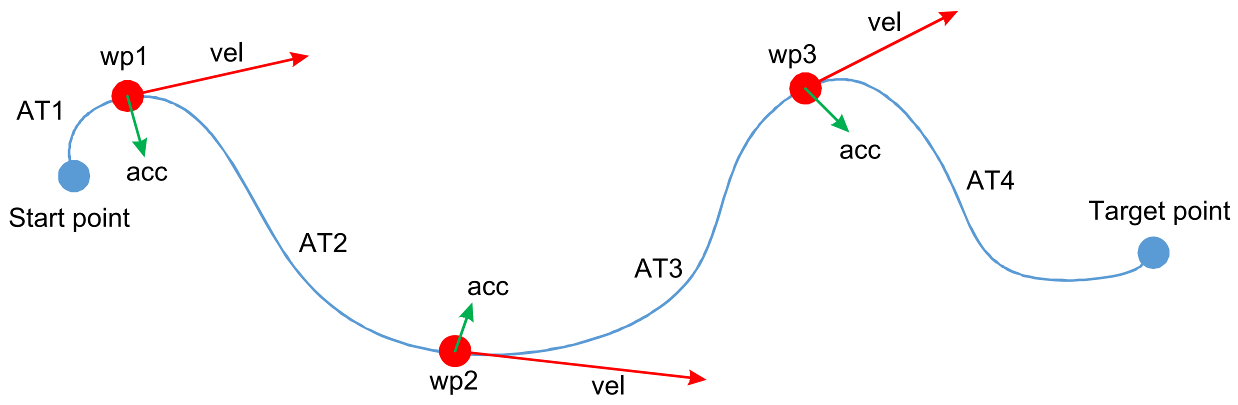
`imp_addWayPoint` function adds a waypoint to the last point of the current trajectory (which must correspond to the last point of an automatic transition). The waypoints must be placed between automatic transitions, with 2 exceptions: a waypoint can also be placed at the beginning (right after `imp_setFirstTrajPoint`) or at the end of a trajectory; there is no much geometrical sense on it, however it might be useful for the 2D triggers placement, as described in the triggers section. A speed and acceleration vector are passed as parameters to fix kinematic constraints at this last point. The waypoints are useful to impose passing points between these non-managed shape segments and to define kinematics constraints on them.

If an error in automatic transitions generation is produced, it is suggested to split the transition in two segments and place a "reasonable" waypoint between both automatic transitions and try again. The presence of a passing point can unblock the algorithm to find a feasible solution.

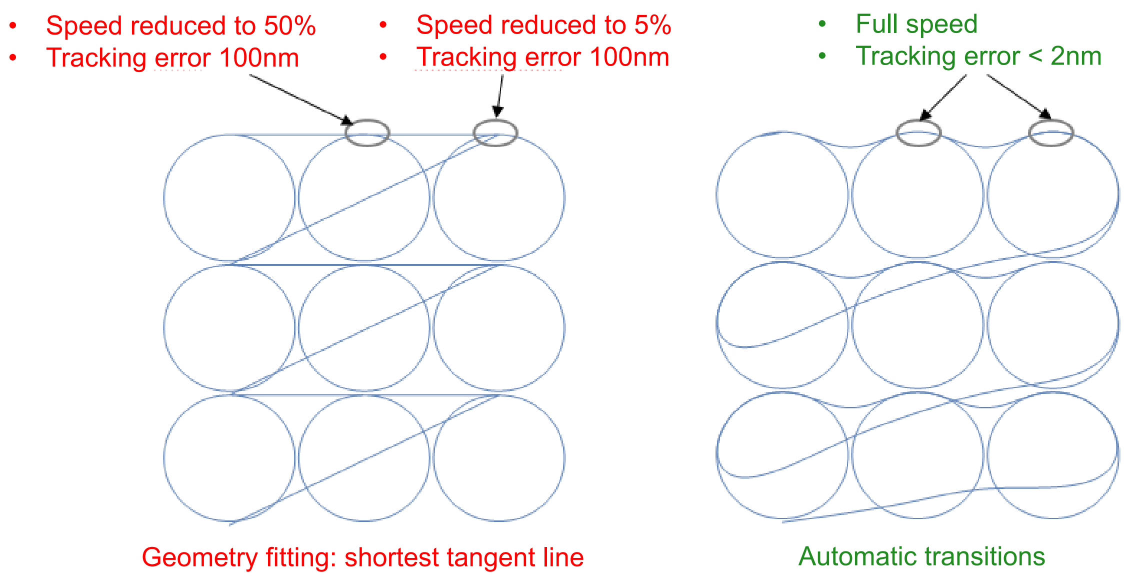
Automatic transitions are aimed to be placed between geometry fitting segments. The use of subsequent automatic transitions is not allowed; instead, as mentioned, the user can alternate automatic transitions with waypoints to link them.

The parameters used in these functions must be expressed in the following units:
Position: **[m]** or **[t]**
Velocity: **[m/s]** or **[t/s]**
Acceleration: **[m/s$^2$]** or **[t/s$^2$]**

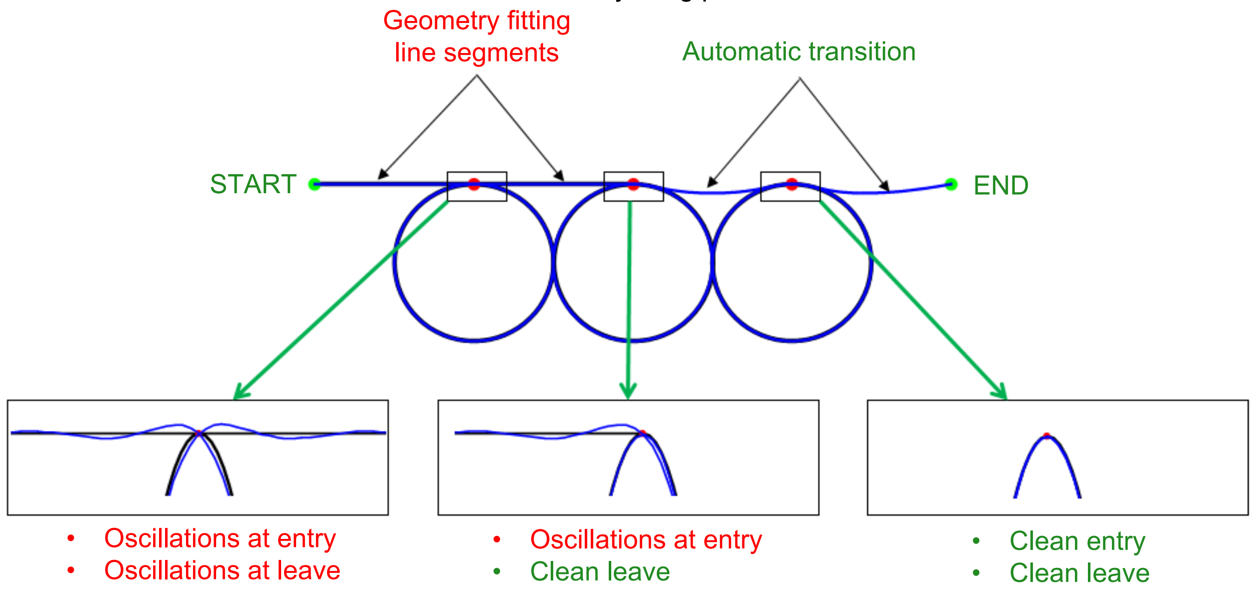
The following figure illustrates the final result using 4 automatic transitions (AT1, AT2, AT3, AT4) and 3 waypoints (wp1, wp2, wp3):



Here below there is an example of transitions between circles using lines versus automatic transitions segments. The advantage of the automatic transitions is very clear:

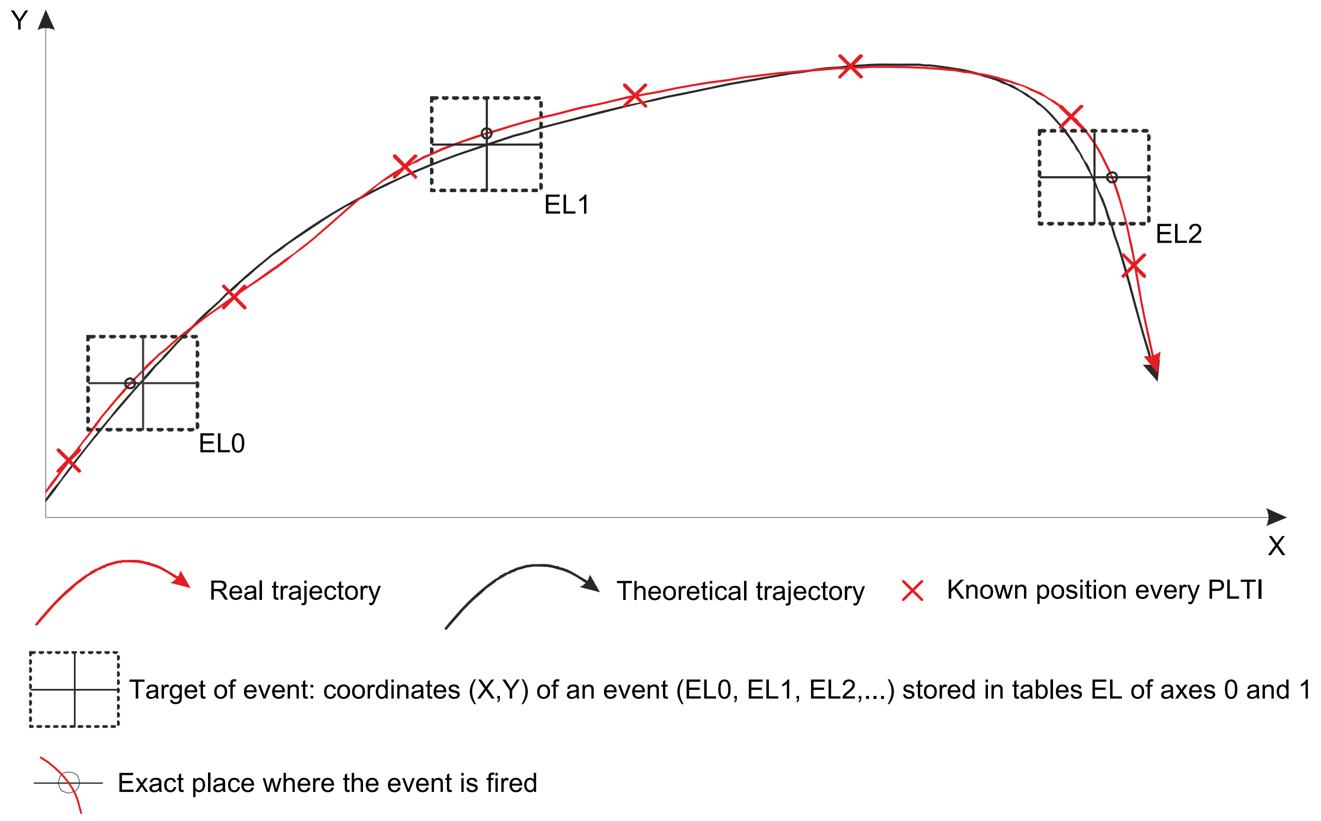


As announced, we can notice the smoothness at the joining points:



## 6.2.4 Triggers configuration

The 2D triggers functionality available on the AccurET controllers is used by the IMP and has been fully integrated in the library.


Y
X
EL0
EL1
EL2
Real trajectory
Theoretical trajectory
Known position every PLTI
Target of event: coordinates (X,Y) of an event (EL0, EL1, EL2,...) stored in tables EL of axes 0 and 1
Exact place where the event is fired

If the user is working with the IMP, it is not necessary to configure the 2D triggers in the AccurET by means of sequences or the commissioning software. IMP provides an API with suitable functions to carry out a proper 2D triggers set-up.

First, the user can create trigger structures using 2 functions:

- **`imp_newTrigger2D:`** this function creates a standard 2D trigger which will be added in the EL table. The corresponding configuration is defined by the required parameters:

    - **Threshold crossing direction**: defines the direction of movement for threshold crossing by x and y axes (2 dimensional parameter). This parameter can have the following values:
        - **`IMP_TRIGGER_BOTH_DIR`** can be either positive or negative crossing direction.
        - **`IMP_TRIGGER_POSITIVE_DIR`** positive crossing direction only.
        - **`IMP_TRIGGER_NEGATIVE_DIR`** negative crossing direction only.
        - **`IMP_TRIGGER_THRES_IGNORED`** axis threshold not taken into account. So the trigger is activated in the boundaries of the box.

    - **Threshold crossing mode**: the detection of the trigger is made using X OR/AND Y logic with the following values:
        - **`IMP_TRIGGER_X_OR_Y`** when X OR Y threshold crossing mode is selected, the event is fired at first threshold occurrence.
        - **`IMP_TRIGGER_X_AND_Y`** when X AND Y threshold crossing mode is selected, the event is fired at the second threshold crossed.
        - **`IMP_TRIGGER_X_ONLY`** when X only threshold crossing mode is selected, the event is fired only at X threshold crossed.
        - **`IMP_TRIGGER_Y_ONLY`** when Y only threshold crossing mode is selected, the event is fired only at Y threshold crossed.

    - **Reserve DOUT Mask**: the user must define which DOUT will be reserved for trigger function by means of a bit mask. The DOUT reservation will only be applied on the first axis.

    - **Reserve FDOUT Mask:** the user must define which FDOUT will be reserved for trigger function by means of a bit mask. The FDOUT reservation will only be applied on the first axis.

- **Combi DOUT Mask**: the user must define which outputs will be activated/deactivated when a programmed trigger is launched. To define these outputs, a 32-bit mask for DOUT must be defined by the user. Bits 0 to 15 of this mask are used to SET the 16 bits output (DOUT register) and bits 16 to 31 are used to RESET the 16 bits output (DOUT register). If the same output bit is SET and RESET, the output will be in a toggle state. The DOUT combi configuration will only be applied on the first axis.

- **Combi FDOUT Mask**: the user must define which outputs will be activated/deactivated when a programmed trigger is launched. To define these outputs, a 32-bit mask for FDOUT must be defined by the user. Bits 0 to 15 of this mask are used to SET the 16 bits output (FDOUT register) and bits 16 to 31 are used to RESET the 16 bits output (FDOUT register). If the same output bit is SET and RESET, the output will be in a toggle state. The FDOUT combi configuration will only be applied on the first axis.

- **`imp_newSmartTrigger2D`:** this function creates a smart 2D trigger which will be added in the EL table. The parameter "**Threshold crossing direction**" and "**Threshold crossing mode**" will be automatically set depending on the trajectory context where the trigger is placed. The output configuration parameters: "**Reserve DOUT Mask**", **"Reserve FDOUT Mask"**, **"Combi DOUT Mask"**, **"Combi FDOUT Mask"** must be defined by the user in the same way as for the standard triggers.

Once the triggers have been created with their specific configuration, they must be placed along the trajectory. When the user executes `imp_addTrigger2D`, the target point of the trigger will be placed by default at the initial point of the last segment definition added to the trajectory (if only the "set first point" has been executed, it will be placed at the first point of the trajectory).
A trigger definition right after an automatic transition is not allowed. The user has no control of the shape or position during an automatic transition that is why there is no sense to set a trigger within. Nevertheless, the user can set a trigger right after adding a waypoint (`imp_addWayPoint`) where the point coordinates are well defined; however, in this case, no offset distance can be set (otherwise an error will be produced).

Beside the target point, the user can also define by parameter:

- **Disable period:** after the current trigger activation and during this period, no other trigger will be enabled. If this deactivation exceeds the time between the current trigger and the next one (if any), no other trigger would be launched during trajectory execution (the EL table order would be broken); so in this case an error will be produced.

- **Time delay**: positive (post-trigger) or negative (pre-trigger) delay (in [s] units) to activate the trigger with respect to the target point. It is set to 0 by default.

- **Offset type**: the type of distance displacement that can be applied on the corresponding trigger:
  - **`IMP_TRIGGER_DIST_OFFSET`**, the trigger is re-placed adding the curvilinear distance (indicated by offset Distance) to the initial point of the last segment definition.
  - **`IMP_TRIGGER_RATIO_OFFSET`**, the trigger is re-placed with respect to a ratio (indicated by offset Distance) between [0.0, 1.0], the initial point and the last point of the last segment definition correspondingly.

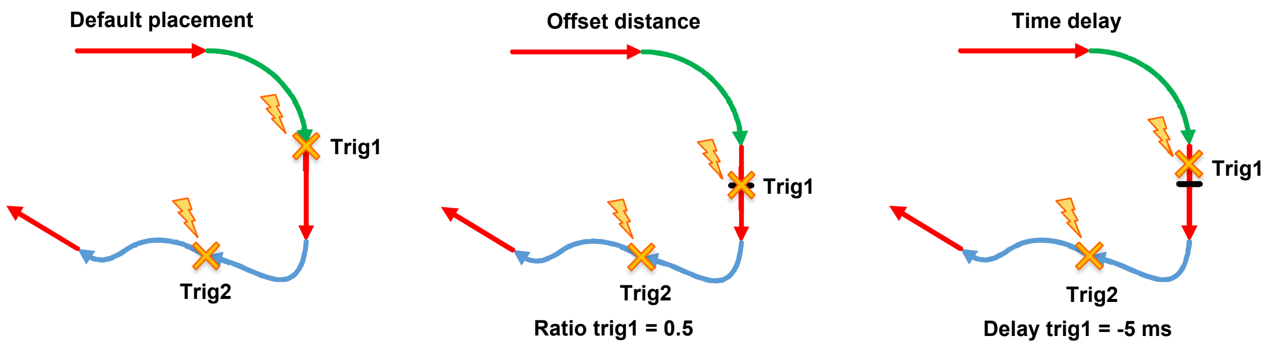
- **Offset distance**: the *curvilinear distance* or the *ratio distance* used to displace the corresponding trigger with respect to its default value (the initial point of the last segment definition). It is set to 0 if not defined by user.

If both, time delay and offset distance are defined, the calculation first takes into account the offset displacement; and once the point has been re-placed, the time delay with respect to the new position is computed.

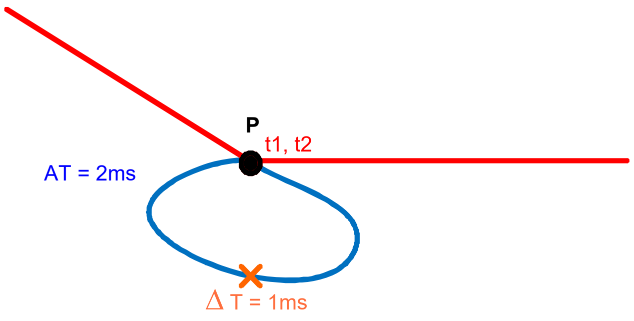
The offset displacement is always referred to the scope of the last segment definition (either for distance, either of course for ratio); however the time delay can cross the limits of the last segment definition.

In the following figure we can observe the segment insertion order:
1) Add Line
2) Add Circle arc
3) Add Line

4) Add Trigger (Offset ratio = 0.5; time delay -5 ms)
5) Add Automatic transition
6) Add waypoint
7) Add Trigger
8) Add Automatic transition
9) Add Line



The following figure depicts a double triggering issue which can be solved by using the "disable period" feature.



In this example, the user has defined 2 triggers (t1, t2) at the same point P. The first trigger t1 disables a laser cutting tool, then an automatic transition is executed, and finally the second time passing by P the laser tool is reactivated. However an issue can occur here, t1 and t2 might be launched at the same time or one right after the other before executing the automatic transition, so an undesired behavior of laser enable/disable at the same moment can happen. The "disable period" parameter serves to avoid this kind of problems.
If the IMP library detects 2 overlapping triggers, it will inform the user about it at the moment of trajectory compilation, suggesting to use "disable period" feature and indicating the execution time between both in order to have an idea about which value could be set. In the example above, the user could set a "disable period" of 1 ms for t1, so after it is launched no other trigger will be activated during 1 ms. The internal mechanism developed for this functionality is a "hidden" trigger definition (with no output interfaces) inserted between t1 and t2 and placed at the coordinates corresponding to the Δt disable period parameter on the automatic transition. The EL table order must be respected, so t1 and t2 will not be consecutive anymore, thus the described double triggering issue will not be produced.

There are certain trigger configurations that are common to **all** the triggers present in the trajectory. The function `imp_setTrigger2DConfig` only needs to be called once for a trajectory and must be called before the trajectory execution. Trigger missed event capabilities can be set (they are optional) with this function. 7 parameters are required:

- **Geometry fitting error tolerance**: x and y tolerances for the trigger box detection.

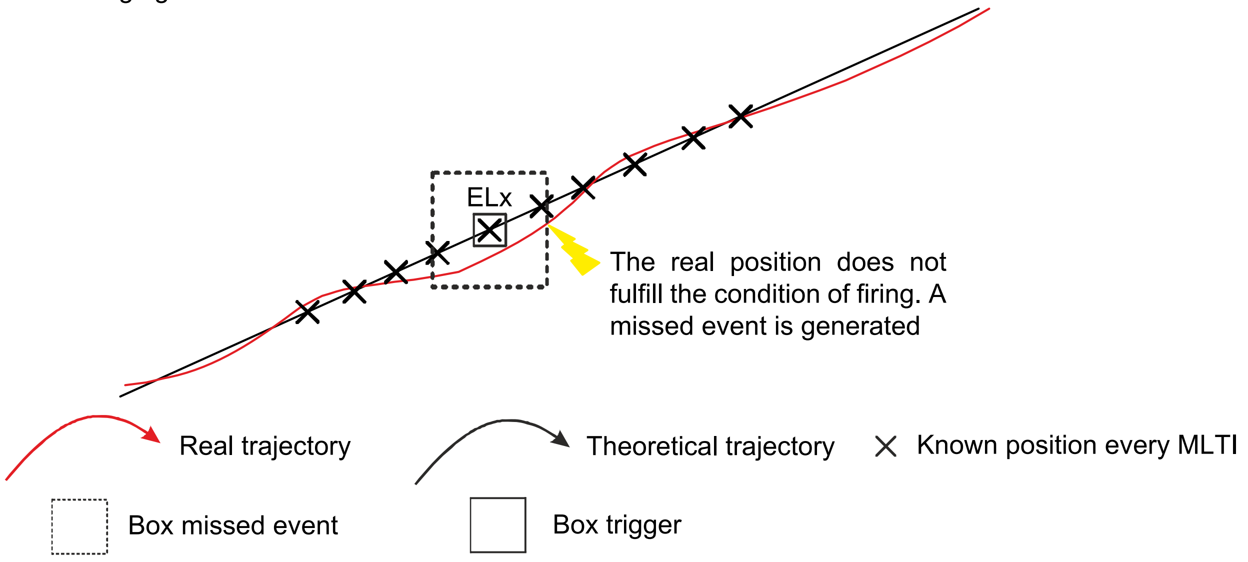
- **Position type**: the type of position where the trigger is launched:
    - **IMP_TRIGGER_REAL_POS,** the trigger activation is based on the real position.
    - **IMP_TRIGGER_THEOR_POS,** the trigger activation is based on the theoretical position.

- **Missed event action/feedback**: *optional parameter* which indicates the feedback generated when a missed event is produced [default value set to IMP_TRIGGER_NO_MISSED_EVENT]:
    - **IMP_TRIGGER_NO_MISSED_EVENT,** the missed event feature is disabled.
    - **IMP_TRIGGER_MISSED_EVENT_ERROR,** the missed event generates an error in the corresponding controller.

- **IMP_TRIGGER_MISSED_EVENT_WARNING,** the missed event generates a warning in the corresponding controller.

- **Missed event tolerance**: *optional parameter* which indicates the x and y tolerances for the missed event box detection.
- **Missed event timeout**: *optional parameter* which indicates the maximum allowed elapsed time since crossing the missed event box until reaching the trigger detection. If no trigger is detected, when this time is consumed, the corresponding missed event feedback will be generated.
- **Position RTV slot idx**: *optional parameter* which indicates the RTV slot index used to transfer the position information between controllers, in case the trigger positions are coming from different controllers. The type of register will be ML+[slot_idx]. The range of possible values is from 450 to 457 in a controller. [Default value = 450].
- **Missed event RTV slot idx**: *optional parameter* which indicates the RTV slot index used to transfer the missed event position information between controllers, in case the trigger positions are coming from different controllers. The type of register will be ML+[slot_idx]. The range of possible values is from 450 to 457 in a controller. [Default value = 451].

The following figure illustrates the missed event feature:


ELx
The real position does not fulfill the condition of firing. A missed event is generated
Real trajectory
Theoretical trajectory
Known position every MLTI
Box missed event
Box trigger

All the triggers defined in IMP are considered with box-enabled capability.

For the moment, there is no Pulse Generator configuration for 2D triggers in the scope of the IMP. The pulse generation should be managed by the user application once the trigger is detected.

Continuous trigger feature is naturally integrated in the IMP library. The user can define as many triggers as he wants without taking care of any limitation or buffer mechanism. This functionality remains transparent for the user.

For more detailed information about 2D triggers, please refer to the AccurET Operation & Software Manual in the corresponding section.

## 6.3 Trajectory generation

Once the user has added all the trajectory definitions, the trajectory is ready to be calculated satisfying the corresponding bounds. The function `imp_compileTrajectory` will compile all the primitives and generate the PVT list to be used later.

The trajectory generation algorithm first calculates the geometry fitting pieces and then the automatic transitions which connect them. The algorithm will do its best to reach the maximum kinematics bounds; however, depending on the geometry, the distances and the kinematics values, the trajectory might not always be feasible.

The user may need to satisfy certain kinematics conditions, for instance he may want to fix constant velocity at some sections of the trajectory (e.g. along the sequence: line, arc circle, line). So, he could set the max tangential velocity to this constant speed; nevertheless, as mentioned, there is no guarantee that this condition is satisfied; either because it is physically unfeasible or because the algorithm does not succeed in finding the existing feasible one. Kinematics results can be checked by means of the min-max report functionality.

It may happen, under certain conditions, that the resulting trajectory violates some of the defined bounds because the algorithm could not find a better-adjusted solution. In these cases, the `imp_compileTrajectory` function will return the error `IMP_ETRAJ_COMPILATION_WARNING = -2698`. Even though an error is produced, for this specific case, the trajectory will have been generated/compiled and will be ready to be executed if the user chooses to do so (after checking the error code or the error message). For all the other cases where the `imp_compileTrajectory` returns an error code, there will be no trajectory ready to be executed.

## 6.4 Evaluation and results

### 6.4.1 Check conditions within interval

The function `imp_checkTrajIntervalCondition` gets a boolean information about the fulfillment of some bounds/kinematics evaluation within a defined range (min/max value) between two trajectory points previously tagged with a time mark. So the user will know that all the trajectory sections between these points comply, yes or not, with the indicated min/max value limits. This function can only be executed once the trajectory has been compiled. Some parameters must be defined:

- **Mark label 1 & mark label 2:** these points will define the trajectory limits for bounds evaluation.
- **Evaluation axis:** in case it is an axis evaluation, otherwise (norm evaluation) this parameter will not be taken into account.
- **Evaluation type:**
  - **`IMP_AXIS_POS_EVAL_TYPE`**: min/max position values for a specific axis.
  - **`IMP_AXIS_VEL_EVAL_TYPE`**: min/max velocity values for a specific axis.
  - **`IMP_AXIS_ACC_EVAL_TYPE`**: min/max acceleration values for a specific axis.
  - **`IMP_AXIS_JRK_EVAL_TYPE`**: min/max jerk values for a specific axis.
  - **`IMP_NORM_VEL_EVAL_TYPE`**: min/max tangential velocity values (axis not taken into account).
  - **`IMP_NORM_ACC_EVAL_TYPE`**: min/max tangential acceleration values (axis not taken into account).
  - **`IMP_NORM_JRK_EVAL_TYPE`**: min/max tangential jerk values (axis not taken into account).

Geometry fitting error evaluation types are not accepted here because the limits are already fulfilled by default during compilation using the user bounds constraints.

- **Min/max limit value:** the limit thresholds which define the range within which the evaluation type can be satisfied (true return value) or not (false return value).
- **Condition satisfied:** returned boolean result which responds to the respect of the corresponding min/max range limits between the corresponding mark points.

With this function the user can get a direct and concise bounds compliance information from a section of the corresponding trajectory.

### 6.4.2 Min-Max box evaluation

The user needs a way to evaluate the results after trajectory compilation:
- Shape correctness.
- Trajectory within workspace limits.
- Geometry fitting error tolerances.
- Kinematics constraints satisfaction.

The solution provided by the IMP library is a min-max box evaluation of the corresponding values of interest by axis or by norm. The sequence of boxes will represent a list of min/max values at 30 ms sampling time along the whole trajectory.

The function `imp_getMinMaxBoxEvaluation` is in charge of returning these results to the user. The following parameters are required:

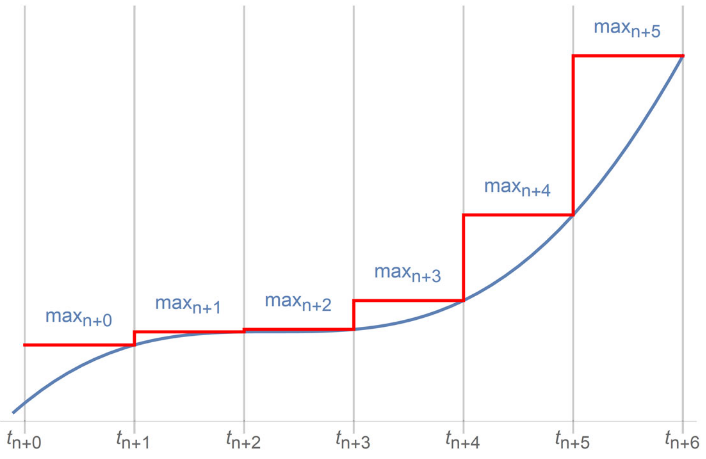
- **Axis**: the dimension on which the min/max evaluation is carried out. This is equivalent to the index of the axis in the interpolation group (not the axis number in the AccurET). This parameter will not be taken into account for norm values evaluation.

- **Evaluation type**:
    - **`IMP_AXIS_POS_EVAL_TYPE:`** min/max position values for a specific axis.
    - **`IMP_AXIS_VEL_EVAL_TYPE:`** min/max velocity values for a specific axis.
    - **`IMP_AXIS_ACC_EVAL_TYPE:`** min/max acceleration values for a specific axis.
    - **`IMP_AXIS_JRK_EVAL_TYPE:`** min/max jerk values for a specific axis.
    - **`IMP_AXIS_TRACK_ERROR_EVAL_TYPE:`** min/max geometry fitting error values for a specific axis.
    - **`IMP_NORM_TRACK_ERROR_EVAL_TYPE:`** min/max norm geometry fitting error values (since it is the norm, no axis is needed, so the axis parameter will not be taken into account).
    - **`IMP_NORM_VEL_EVAL_TYPE:`** min/max tangential velocity values (since it is the norm, no axis is needed, so the axis parameter will not be taken into account).

- **Return values: EVAL_LIST pointers (3 arguments):**
    - EVAL_LIST * min: list of min values (N samples)
    - EVAL_LIST * max: list of max values (N samples)
    - EVAL_LIST * time: list of time values (N+1 samples)

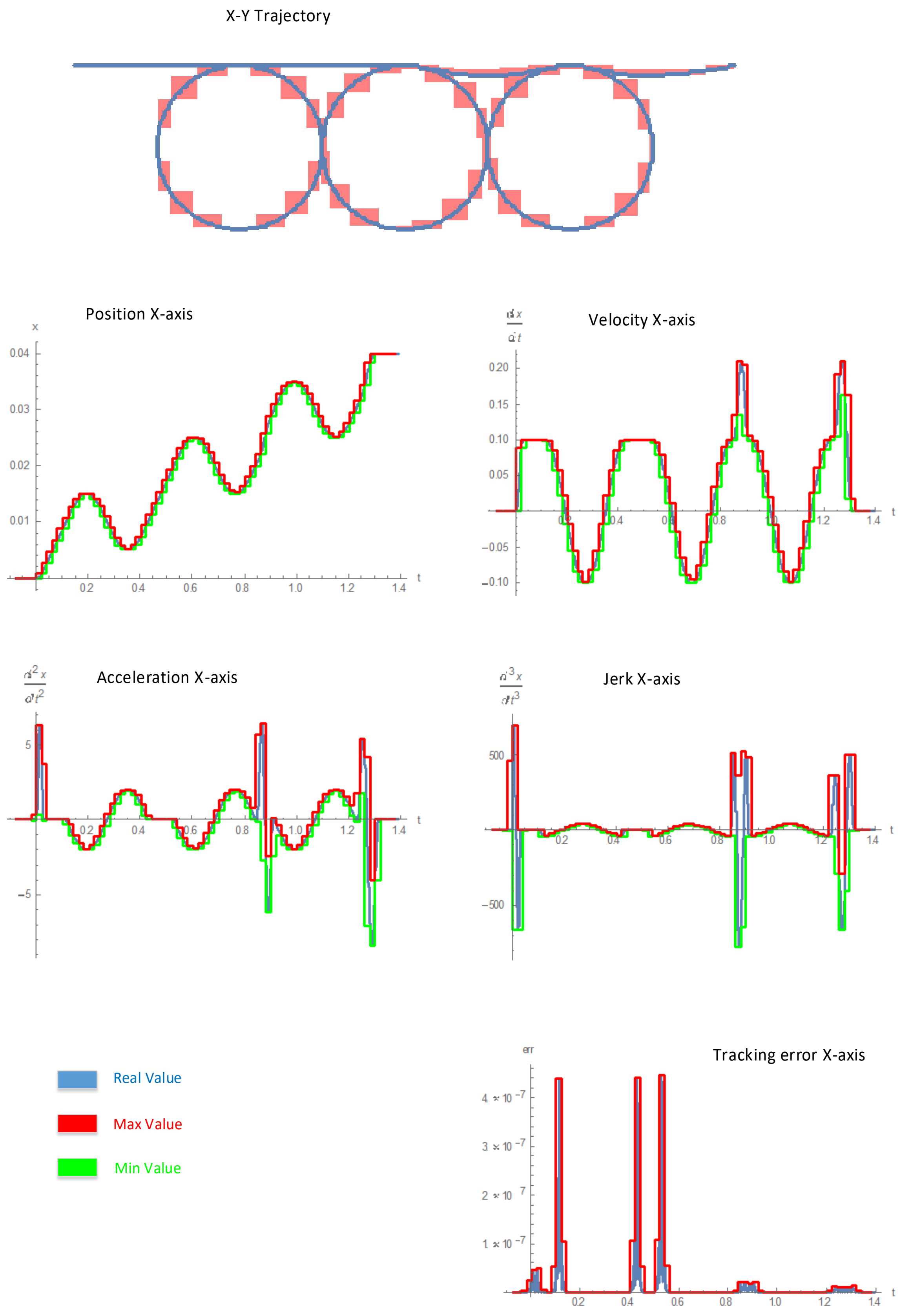
As result, the user will get 3 lists: min and max values for the corresponding axis and evaluation type; and the resulting sampling time list for each box; this is currently constant at 30 ms per box, but in the future, one could imagine variable box sampling depending on the type of movement. With min/max box evaluation the user will be able to check if all the input constraints are respected or not.

The following figure shows the returned list values (on the example of max-boxes). The time samples give the box-boundaries. Therefore, the list of time samples is always one sample longer than the value lists.



With min/max box evaluation the user will be able to check if all the input constraints are respected or not.

In the following figure, some graphics constructed from min/max evaluation data are shown:



## 6.4.3 Time stamp list

The user can combine the min/max box evaluation with some markers along the trajectory; so that he can establish some time marks to help him finding a reference in the previous graphics analysis.

`imp_addTimeMark` will work as described for trigger placement, i.e. the target point of the mark will be placed

by default at the initial point of the last segment definition added to the trajectory or on a previous added waypoint. The same parameters described for `imp_addTrigger2D` are valid as well. In addition an extra parameter is needed:
**Mark label**: this is the label to identify the corresponding time mark. It must be a positive integer.

The same limitation as explained for triggers is also present for the time marks: the use of this function is restricted only after geometry fitting or waypoints definitions.

Finally the user can get all the time stamping report using the following functions:
`imp_getTimeStampList` which returns a `TIME_STAMP_LIST` structure to subsequently use `imp_getTimeStampValue` for the corresponding time marks, returning input user label and time value for a specific index of the list.

### 6.4.4 Min/Max statistics evaluation

The functions `imp_getMinMaxStatsEvaluation` and `imp_getMinGeomFittingVel` provide to the user min/max statistics about the same evaluation types defined in min/max box evaluation section, so he can get a rough summary of the trajectory statistics values about:

- **Evaluation type**:
    - **`IMP_AXIS_POS_EVAL_TYPE`:** min/max position values for a specific axis.
    - **`IMP_AXIS_VEL_EVAL_TYPE`:** min/max velocity values for a specific axis.
    - **`IMP_AXIS_ACC_EVAL_TYPE`:** min/max acceleration values for a specific axis.
    - **`IMP_AXIS_JRK_EVAL_TYPE`:** min/max jerk values for a specific axis.
    - **`IMP_AXIS_TRACK_ERROR_EVAL_TYPE`:** min/max geometry fitting error values for a specific axis.
    - **`IMP_NORM_TRACK_ERROR_EVAL_TYPE`:** min/max norm geometry fitting error values (since it is the norm, no axis is needed, so the axis parameter will not be taken into account).
    - **`IMP_NORM_VEL_EVAL_TYPE`:** min/max tangential velocity values (since it is the norm, no axis is needed, so the axis parameter will not be taken into account).

Finally, the function `imp_getMinGeomFittingVel` provides the minimum velocity for geometry fitting segments. Two additional parameters are returned: `minVelT0` and `minVelT1` which delimit the time interval where the minimum velocity has been detected.

## 6.5 Trajectory execution

Once the user has defined the geometry (geometry fitting and/or automatic transitions), the trajectory bounds and the triggers (optional), the trajectory must be compiled before being executed.

The trajectory execution is directly linked to EDI in order to manage the controllers connected to the system. So parameters such as the UltimET device, interpolation group, or trigger group (types defined by the DSA DLL) are required. The trigger group MUST contain 2 axes (according to the 2 dimensions of 2D triggers). It is possible these 2 axes do not belong to the same controller. It is also possible a combined presence of gantry configuration and 2D triggers; in this case, the axes of the trigger group MUST be defined in different controllers (triggers cannot be defined in the same controller where both gantry axes are placed because they represent the same direction/dimension), additionally only the master gantry axis can be part of the trigger group.

The order of appearance of the corresponding axes inside the trigger group must be equal to the coordinates order of the added target points and threshold crossing direction parameters.

The function `imp_executeTrajectory` is responsible for executing the PVT list generated during trajectory generation/compilation. Moreover, it manages the triggers configuration and activation.

An ***ILINE** command is executed to make the approach to the first trajectory point. The values of velocity, acceleration and jerk for this movement are defined by the max bounds of the corresponding trajectory.

The execution module manages all the interpolation functions:
- `dsa_ipol_begin_s`
- `dsa_ipol_set_abs_mode_s`

- `dsa_ipol_end_s`
- `...`

So the user does not need to take care of them.

`imp_executeTrajectory` has an optional parameter, offset, which indicates a displacement for each axis in the PVT list execution and in the trigger placement.

The position-velocity-time resolution with which the execution module sends the PVT commands to the UltimET corresponds to:

| | |
|---|---|
| *K522 = 1; | // Ufpi multiplication factor |
| *K523 = -9; | // Ufpi ten power for position → nm |
| *K524 = -6; | // Ufsi ten power for speed → µm/s |
| *K525 = -6; | // Ufai ten power for acceleration → µm/s$^2$ |
| *K526 = -3; | // Ufti ten power for time → ms |

The previous values are internally set by default. *K523, *K524 and *K526 are specifically used for PVT segments. Trajectory execution on gantry configuration is supported.

## 6.6 Save and load trajectory

The trajectory generation is an offline step in the whole process which can take some time to get the compiled trajectory ready to be executed. As compilation is a time consuming step, a user may want a list of well-known trajectories ready to be executed (perhaps just changing an offset parameter) without having always to compile them from the beginning.

The function `imp_saveTrajCompilation` saves in a provided path all the relevant information needed to execute the trajectory. So that, the final trajectory coming from the geometrical segments, bounds and kinematics definition; and finally the 2D triggers definition. Time marks will not be backup. A trajectory can be saved always after a previous compilation, otherwise an error will be produced. The generated file will be encrypted to comply with security conditions and to assure integrity control (protection against eventual modifications).

Only the function `imp_loadTrajCompilation` will be able to decrypt and read the trajectory/triggers information previously saved. All the data will be loaded in an IMP_TRAJECTORY structure which will be overwritten in case it contains any previous information. Since the resulting trajectory is not the consequence of a previous compilation but a previous load, no evaluation results can be extracted from it. Moreover, new trajectory definition calls (add geometrical segments, automatic transitions, bounds definition, trigger definition …) will not be supported after loading; the corresponding trajectory structure will have to be cleared to start a new process. Only trajectory execution will be supported after loading.

The save and load functions allow the user to create a pre-compiled trajectories repository ready to be executed without wasting time with the recompilation process.

## 6.7 Memory management

The IMP library provides functions to create (new) and destroy (delete) the common structures. It is important that customer application destroys all the created entities before exiting the program, otherwise memory leak issues could arise.

It may be useful, in a customer application program, to work with the same trajectory structure in an iterative generation-execution process, so the user does not need to continuously create and destroy the same entity. To achieve that, the user can use the function `imp_clearTrajectory` to reuse the same trajectory structure for the next generation-execution period.

## 6.8 Serviceability

### 6.8.1 Error diagnosis

For any function of the IMP library, there is a returned error code. If the error is not 0, something wrong occurred during the execution of the corresponding call. The function `imp_diag()` will print in the standard output the diagnosis report of the last error produced during execution. So, it is advisable to use the following operations sequence:

```
int error;
if(error = imp_function(...))
{
     imp_diag();
     // error treatment (e.g. switch-case strategy)
     return error;
}
```

List of IMP errors:

| | |
|---|---|
| `IMP_EASSERT` | Assert violation: Pre-Conditions, Invariants or Post-Conditions not respected. |
| `IMP_ETRAJ_COMPILATION_WARNING` | Trajectory ready but some condition is violated. |
| `IMP_ETRAJ_COMPILATION_ERROR` | Trajectory compilation failed. |
| `IMP_EGEOMETRY_DEF` | Wrong geometry definition. |
| `IMP_ESYSTEM_CONFIG` | Wrong devices configuration. |
| `IMP_ECOMPATIBILITY` | The current IMP library version is not compatible with this feature. |
| `IMP_EWRONG_DEVICE` | Invalid device. |
| `IMP_EDEVICE_IN_ERROR` | Device in error. |
| `IMP_EWRONG_OPS_SEQUENCE` | Invalid sequence of operations. |
| `IMP_ETRIGGER_CONFIG` | Incorrect trigger configuration. |
| `IMP_EPVT_DEF` | Wrong PVT segment definition. |
| `IMP_ELOGIC` | Internal logic error. |
| `IMP_EDOMAIN` | Domain error. |
| `IMP_EOUT_OF_RANGE` | Out of range error. |
| `IMP_ELENGTH` | Container length error. |
| `IMP_EINVALID_ARGUMENT` | Invalid argument. |
| `IMP_ETHREAD_MANAGEMENT` | Thread configuration error. |
| `IMP_EIN_OUT` | Input/output operation failed. |

In case an error is produced, a stack trace file called "**imp_stackTrace.dmp**" will be automatically created, saving the stack trace symbols and memory addresses at this moment. This file can be sent to ETEL for further analysis about the corresponding issue.

### 6.8.2 Versioning

The function `imp_getVersion` returns the current IMP library version with a Microsoft format coming from a longword of the form (HEX) "<major><minor><micro><reserved>". Each time an error is produced this IMP library version is printed. Additionally, this function can be used for user logging purposes.

### 6.8.3 Logging

A **log file imp.log** is generated in the corresponding computer %TEMP% folder for the last IMP execution. The log file specifies the date, the library version (IMP and EDI) and a list of entries indicating entry date, entry type (information, warning or error), IMP module and entry description.

Here is an example:

| IMP library version: 1.01A and EDI library version: 4.17beta4x | | | |
|---|---|---|---|
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | Exploring devices for trajectory execution... |
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | UltimET type: EU-LCP-0-0-0000-00; UltimET firmware version: 3.14beta2 |
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | Axis Index: 0; AccurET type: EA-P2M-400-10/20A-0001-01; AccurET firmware version: 3.13beta5x |
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | Axis Index: 1; AccurET type: EA-P2M-400-10/20A-0001-01; AccurET firmware version: 3.13beta5x |
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | Initializing trajectory execution |
| Wed Nov 1 16:53:11 2017 | Information | Trajectory Execution module | Start trajectory execution. |
| Wed Nov 1 16:53:13 2017 | Information | Trajectory Execution module | End trajectory execution. |

## 6.8.4 Trace acquisition

In a general way, the user can carry out acquisitions either by means of the Scope tool from the Commissioning software ComET or by means of the EDI API to obtain traces for the purpose of monitoring and troubleshooting.

In the context of the IMP trajectory execution, there is a reference triggered by an UltimET register to get exactly the beginning of the movement (it does not take into account the approach phase because it does not make part of the planned trajectory). This is an interpolation trajectory mark which is enabled in *K540 register.

Since *K540 is an UltimET register, ETNE mechanism (not ETND) must be set-up to provide multiple connection interface in order to trigger on it (via the Scope tool). In this way, IMP gets the TransnET communication with the PCI interface, but additionally opens a port for simultaneous Commissioning tool connection. This method requires to add some code in the customer application during device initialization/ configuration:

```
if (error = dsa_get_etb_bus(*ultimet, &etb))
{
     DSA_EXT_DIAG(err, *ultimet);
     return err;
}

if (error = etne_start_custom(etb, ETNE_PORT, "ETNE IMP"))
{
     DSA_EXT_DIAG(err, *ultimet);
     return err;
}
```

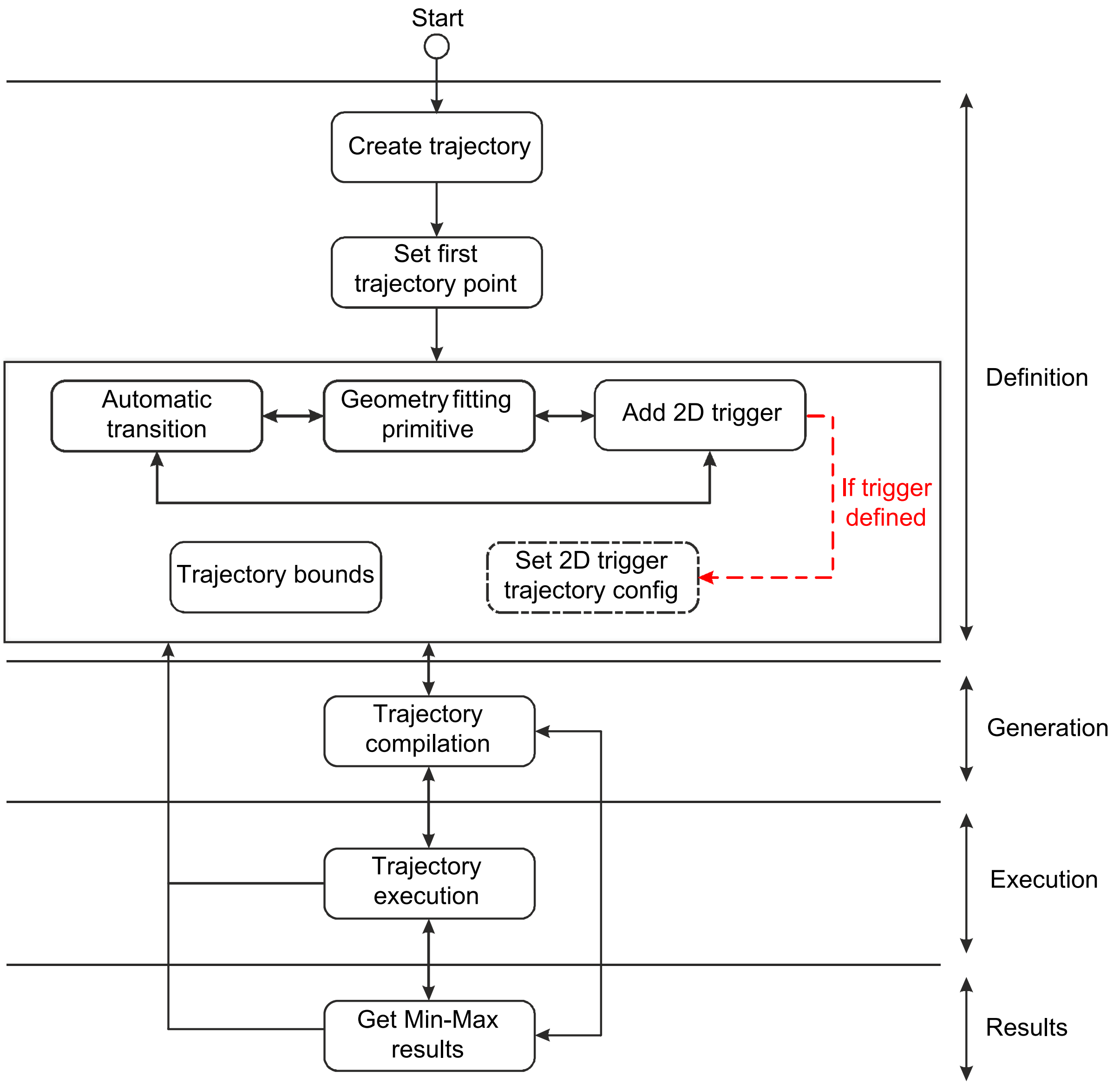
`ETNE_PORT` must be customized depending on the rest of the commissioning ports configuration. For more information about ETNE procedure, please contact ETEL support.

It is important to notice that both, trace acquisition (via ETNE) and trigger features, are using the streaming mechanism to upload data, so the combined use of these functionalities can produce some errors. The following recommendations should be considered:

- It is not recommended to use "Continuous acquisition" with the IMP trajectory execution containing 2D triggers. In this case, only "Single acquisition" launched before trajectory execution is recommended (except of the case exposed in the next point).
- If there is a high quantity (> 512) of 2D triggers very close to each other in the user application, parallel acquisition should not be used at all; otherwise, the balance of firmware resources to enable the acquisition and manage the trigger functionality at the same time could cause a bad management of the trigger events, producing a warning or an error in the AccurET about the management of the EL table.

## 6.9 Execution pipeline

The following figure represents the state diagram which describes the possible transitions between operations:



# 7. IMP constraints

- The library is limited to a 4 axes configuration. This is the axes limit for PVT commands.
- The trigger features are limited to 2D configuration, so a projection will be taken into account with respect to the resulting trajectory during its execution.
- The number of trigger output combinations between DOUT/FDOUT within a trajectory is limited to 16.
- The IMP library is not intended to work under application multi-threading architectures (e.g. separate threads for trajectory generation and trajectory execution).

## 8. IMP important remarks

- IMP is a compounded solution with a generation and an execution module. The generation part has been designed to be an offline process with potentially intensive use of memory and processing time; thus this module is not intended to work in embedded systems. Therefore, concurrent trajectory generation-execution loop in a real time mode is not supported.
- PVT brake: If the PVT list execution is interrupted for some reason, at the end of the impacted PVT primitive, a brake command (IBRK) will stop the movement within the time defined in parameter *K561 of the UltimET. This guarantees certain safety protection in case the PVT segment list finishes in an abrupt way with certain motion dynamics.
- For the moment, the geometry fitting generation does not take into account the workspace limits. It is the user's responsibility to ensure that the defined geometry remains within the workspace. However, the library will check against the workspace limits for the automatic transitions (whose shape is a priori unknown by the user).
- The sampling time of PVT segments will be fixed to 3 ms. This sampling could change in the future.
- The bounds (kinematics, tolerances, and workspace) are global to the whole trajectory. Thus, the user cannot define different bounds for different trajectory sections.
- Trajectory generation algorithm will do its best to reach the maximum kinematics. However, depending on the geometry, the distances and the kinematics values along the trajectory, it might not be always possible.
- UltimET frequency during interpolation is fixed at 10 kHz.

## 9. IMP modules

*IMP package* currently contains 6 ETEL libraries which correspond to different modules and levels in the IMP architecture:

- **imp10.dll:** this is the interface module. It means the access point to the library with C-like format which returns an error code if something wrong happened. This module is the kernel of the IMP library. It has been built as a wrapper of the object oriented architecture. It calls all the other modules, converts and arranges the data between them. **The user only needs to include and link this specific module.**

- **trajgen10.dll**: this is the trajectory generation module. It is in charge of building the geometry fitting trajectories and the automatic transitions complying with the corresponding kinematics constraints, error tolerance and workspace limits (geometry generator and trajectory planner in the figure below). It is also in charge of placing the offset triggers according to the corresponding time delay or distance displacement within the trajectory. Finally it will provide the suitable interpolation target list, in our case, as defined, PVT primitive list (trajectory interpolator in the figure below).

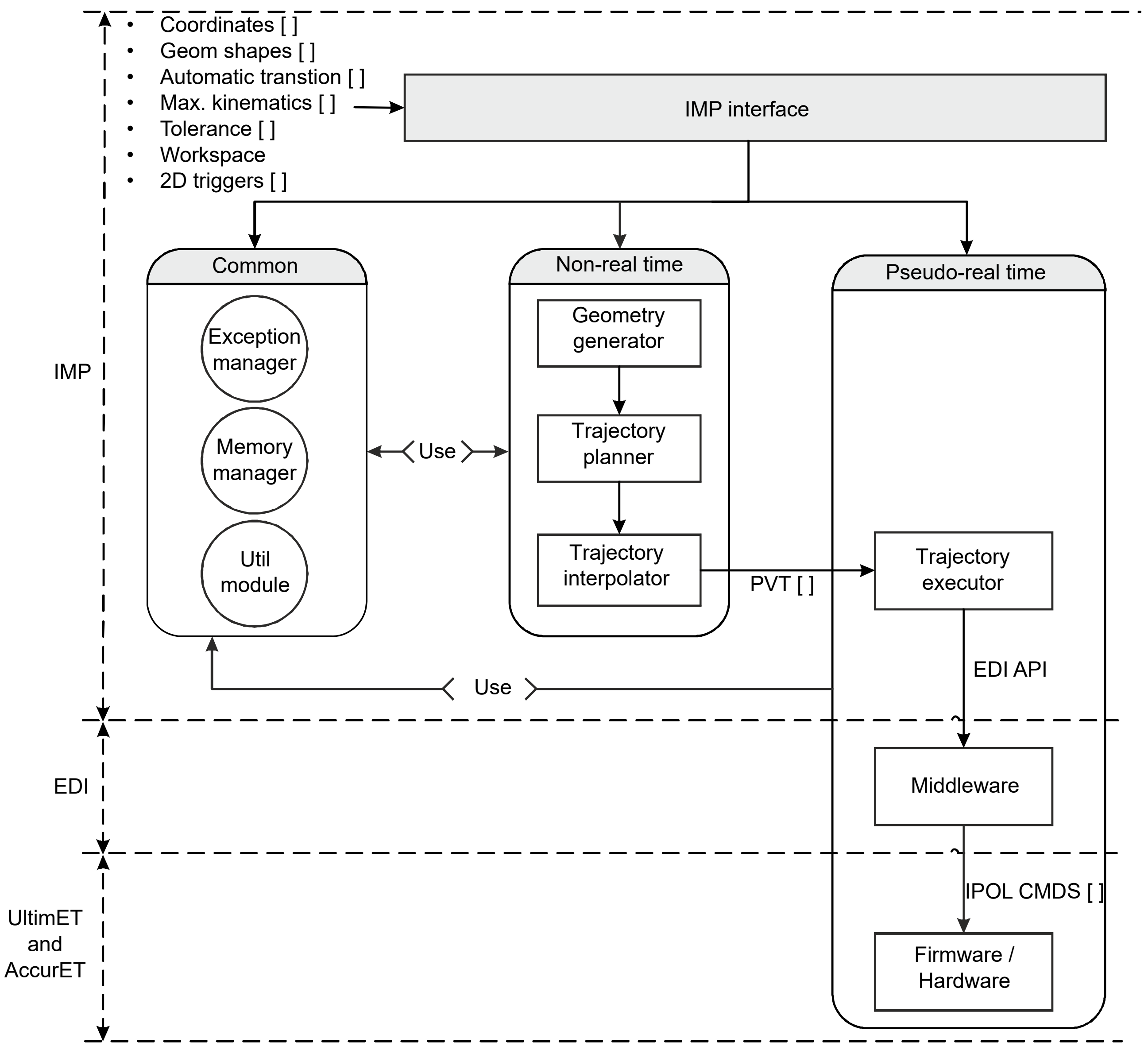
- **trajexe10.dll**: this is the trajectory execution module. It is in charge of the communication with the controllers through the EDI middleware. It executes UltimET and AccurET initialization for interpolation mode. It manages the interpolation buffer sent to the UltimET for the PVT list execution and the trigger action configuration and launch. To achieve these objectives, this module is executed in a priority thread for a "real-time service".

- **excmgmt10.dll**: this is the exception manager module. It is in charge of managing the trace and report of exceptions produced during the execution.

- **mmgmt10.dll**: this is the memory manager module. The library is currently using a memory pool to optimize and control the memory allocation during the whole process. The main objective of this module is to prevent memory fragmentation during a long (or indefinite) execution time.

- **util10.dll**: this is a common data module. It is in charge of providing common structures and global data types to exchange information between the different modules.

- Third party libraries: **libcrypto-1_1, libssl-1_1.dll**: OpenSSL libraries for encryption purpose.


Coordinates [ ]
Geom shapes [ ]
Automatic transtion [ ]
Max. kinematics [ ]
Tolerance [ ]
Workspace
2D triggers [ ]
IMP interface
IMP
Common
Exception manager
Memory manager
Util module
Use
Non-real time
Geometry generator
Trajectory planner
Trajectory interpolator
PVT [ ]
Pseudo-real time
Trajectory executor
Use
EDI API
EDI
Middleware
IPOL CMDS [ ]
UltimET and AccurET
Firmware / Hardware

# 10. Appendix A: *prototypes signature*

```
// Trigger2D Struct
//
_DLL_IMP_API int imp_newTrigger2D(const int thrCrossDirection[IMP_TRIG_DIM],
const int thrCrossMode, const unsigned int reserveDOUTMask, const unsigned int
reserveFDOUTMask, const unsigned int combiDOUTMask, const unsigned int
combiFDOUTMask, TRIGGER_2D** trigger);
_DLL_IMP_API int imp_newSmartTrigger2D(const unsigned int reserveDOUTMask, const
unsigned int reserveFDOUTMask, const unsigned int combiDOUTMask, const unsigned
int combiFDOUTMask, TRIGGER_2D** trigger);
_DLL_IMP_API int imp_deleteTrigger2D(TRIGGER_2D** trigger);

// EvaluationList Struct
//
_DLL_IMP_API int imp_newEvaluationList(EVAL_LIST** evalList);
_DLL_IMP_API int imp_deleteEvaluationList(EVAL_LIST** evalList);
_DLL_IMP_API int imp_getEvaluationListSize(const EVAL_LIST* evalList, unsigned
int& size);
_DLL_IMP_API int imp_getEvaluationValue(const EVAL_LIST* evalList, const unsigned
int idx, double& value);

// TimeStamp Struct
//
_DLL_IMP_API int imp_newTimeStampList(TIME_STAMP_LIST** timeList);
_DLL_IMP_API int imp_deleteTimeStampList(TIME_STAMP_LIST** timeList);
_DLL_IMP_API int imp_getTimeStampListSize(const TIME_STAMP_LIST* timeList,
unsigned int& size);
_DLL_IMP_API int imp_getTimeStampValue(const TIME_STAMP_LIST* timeList, const
unsigned int idx, unsigned int& label, double& value);

// ImpTrajectory Struct
//
_DLL_IMP_API int imp_newTrajectory(const unsigned int dimension, IMP_TRAJECTORY**
trajectory);
_DLL_IMP_API int imp_deleteTrajectory(IMP_TRAJECTORY** trajectory);
_DLL_IMP_API int imp_clearTrajectory(IMP_TRAJECTORY* trajectory);

_DLL_IMP_API int imp_setFirstTrajPoint(IMP_TRAJECTORY* trajectory, const double
firstPoint[IMP_MAX_DIM]);
_DLL_IMP_API int imp_addTrajLine(IMP_TRAJECTORY* trajectory, const double
nextPoint[IMP_MAX_DIM]);
_DLL_IMP_API int imp_addTrajCircleArc(IMP_TRAJECTORY* trajectory, const double
circleP1[IMP_MAX_DIM], const double circleP2[IMP_MAX_DIM], const double angle);
_DLL_IMP_API int imp_addTrajEllipticArc(IMP_TRAJECTORY* trajectory, const double
semiMajorAxis[IMP_MAX_DIM], const double semiMinorAxis[IMP_MAX_DIM], const double
startAngle, const double offsetAngle);
_DLL_IMP_API int imp_addTrajPoly(IMP_TRAJECTORY* trajectory, const int degree,
const double poly[IMP_MAX_POLY_COEFFS][IMP_MAX_DIM], const double xMax);
_DLL_IMP_API int imp_addTrajBezier(IMP_TRAJECTORY* trajectory, const int degree,
const double ctrlPts[IMP_MAX_CTRL_POINTS][IMP_MAX_DIM]);

_DLL_IMP_API int imp_addTrajAutoTransition(IMP_TRAJECTORY* trajectory, const
double targetPoint[IMP_MAX_DIM]);
_DLL_IMP_API int imp_addWayPoint(IMP_TRAJECTORY* trajectory, const double
velocity[IMP_MAX_DIM], const double acceleration[IMP_MAX_DIM]);

_DLL_IMP_API int imp_setTrajBounds(IMP_TRAJECTORY* trajectory, const double
minWorkSpaceLimits[IMP_MAX_DIM], const double maxWorkSpaceLimits[IMP_MAX_DIM],
const double maxGeomFittingErrorNorm, const double maxTrajSpeedNorm,
const double maxGeomFittingErrorAxes[IMP_MAX_DIM], const double
maxTrajSpeedAxes[IMP_MAX_DIM], const double maxTrajAccAxes[IMP_MAX_DIM], const
double maxTrajJerkAxes[IMP_MAX_DIM]);

_DLL_IMP_API int imp_setTrigger2DConfig(IMP_TRAJECTORY* trajectory, const double
triggerTrackingErrorTolerance[IMP_MAX_DIM], const int positionType, const int
missedEventFdbck = IMP_TRIGGER_NO_MISSED_EVENT, const double
missedEventTolerance[IMP_MAX_DIM] = nullptr, const double missedEventTimeout =
0.0, const unsigned int posRTVSlotIdx = IMP_TRIGGER_REG_POS_SLOT_IDX, const
unsigned int missedEventRTVSlotIdx = IMP_TRIGGER_MISSED_EVENT_SLOT_IDX);

_DLL_IMP_API int imp_addTrigger2D(IMP_TRAJECTORY* trajectory, const TRIGGER_2D*
trigger, const double disablePeriod = 0.0, const double timeDelay = 0.0, const
unsigned int offsetType = IMP_DIST_OFFSET, const double offsetDistance = 0.0);
_DLL_IMP_API int imp_addTimeMark(IMP_TRAJECTORY* trajectory, const unsigned int
markLabel, const double timeDelay = 0.0, const unsigned int offsetType =
IMP_DIST_OFFSET, const double offsetDistance = 0.0);

_DLL_IMP_API int imp_compileTrajectory(IMP_TRAJECTORY* trajectory);
_DLL_IMP_API int imp_executeTrajectory(const IMP_TRAJECTORY* trajectory,
DSA_DSMAX* ultimet, DSA_IPOL_GROUP* ipolGroup, DSA_DEVICE_GROUP* triggerGroup =
nullptr, const double offset[IMP_MAX_DIM] = nullptr);

_DLL_IMP_API int imp_saveTrajCompilation(const IMP_TRAJECTORY* trajectory, const
char* filePath);
_DLL_IMP_API int imp_loadTrajCompilation(const IMP_TRAJECTORY* trajectory, const
char* filePath);

_DLL_IMP_API int imp_getLastPoint(const IMP_TRAJECTORY* trajectory, double
lastPoint[IMP_MAX_DIM]);

_DLL_IMP_API int imp_checkTrajIntervalCondition(const IMP_TRAJECTORY* trajectory,
const unsigned int markLabel1, const unsigned int markLabel2, const unsigned int
axis, const int evaluationType, const double minValue, const double maxValue,
bool& isConditionSatisfied);
_DLL_IMP_API int imp_getMinMaxBoxEvaluation(const IMP_TRAJECTORY* trajectory,
const unsigned int axis, const int evaluationType, EVAL_LIST* minValueList,
EVAL_LIST* maxValueList, EVAL_LIST* timeValueList);
_DLL_IMP_API int imp_getMinMaxStatsEvaluation(const IMP_TRAJECTORY* trajectory,
const unsigned int axis, const int evaluationType, double& minValue, double&
maxValue);
_DLL_IMP_API int imp_getMinGeomFittingVel(const IMP_TRAJECTORY* trajectory,
double& minVelValue, double& minVelT0, double& minVelT1);
_DLL_IMP_API int imp_getTimeStampList(const IMP_TRAJECTORY* trajectory,
TIME_STAMP_LIST* timeList);

_DLL_IMP_API void imp_diag();
_DLL_IMP_API void imp_getVersion(char* impVersion);
```

# 11. Service and support

For any inquiry regarding technical, commercial and service information relating to ETEL S.A. products, please contact your ETEL S.A. representative:

| HEADQUARTER / SWITZERLAND | BELGIUM | CHINA |
|---|---|---|
| **ETEL S.A.**<br>**Zone industrielle**<br>**CH-2112 Môtiers**<br>**Phone: +41 (0)32 862 01 00**<br>**E-mail: etel@etel.ch**<br>**http://www.etel.ch** | **HEIDENHAIN nv/sa**<br>Pamelse Klei 47<br>1760 Roosdaal<br>Phone: +32 54 34 31 58<br>E-mail: sales@heidenhain.be | **DR. JOHANNES HEIDENHAIN (CHINA) Co., Ltd**<br>No. 6, Tian Wei San Jie, Area A,<br>Beijing Tianzhu Airport, Industrial Zone<br>Shunyi District, Beijing 101312<br>Phone: +86 400 619 6060<br>E-mail: sales@heidenhain.com.cn |
| **CZECH Republic** | **FRANCE** | **GERMANY** |
| **HEIDENHAIN s.r.o.**<br>Dolnomecholupská 12b<br>102 00 Praha 10 - Hostivar<br>Phone: +420 272 658 131<br>E-mail:heidenhain@heidenhain.cz | **HEIDENHAIN FRANCE SARL**<br>2 avenue de la cristallerie<br>92310 Sèvres<br>Phone: +33 (0)1 41 14 30 09<br>E-mail: sales@heidenhain.fr | **DR. JOHANNES HEIDENHAIN GmbH**<br>Technisches Büro Südwest II<br>Verkauf ETEL S.A.<br>Schillgasse 14<br>78661 Dietingen<br>Phone: +49 (0)741 17453-0<br>E-mail: tbsw.etel@heidenhain.de |
| **GREAT-BRITAIN** | **ISRAEL (Representative)** | **ITALY** |
| **HEIDENHAIN (GB) Ltd.**<br>200 London Road, Burgess Hill,<br>West Sussex RH 15 9RD<br>Phone: +44 (0)1444 247711<br>E-mail: sales@heidenhain.co.uk | **MEDITAL COMOTECH Ltd.**<br>36 Shacham St.,<br>P.O.B 7772, Petach Tikva<br>Israel 4951729<br>Phone: +972 3 923 3323<br>E-mail: comotech@medital.co.il | **ETEL S.A.**<br>Piazza della Repubblica 11<br>28050 Pombia<br>Phone: +39 0321 958 965<br>E-mail: etel@etelsa.it |
| **JAPAN** | **KOREA** | **SINGAPORE** |
| **HEIDENHAIN K.K.**<br>Hulic Kojimachi Bldg. 9F<br>3-2 Kojimachi, Chiyoda-ku<br>Tokyo - 102-0083<br>Phone: +81 3 3234 7781<br>E-mail: sales@heidenhain.co.jp | **HEIDENHAIN KOREA Ltd.**<br>75, Jeonpa-ro 24beon-gil,<br>Manan-gu, Anyang-si<br>Gyeonggi-do, 14087, Korea<br>Phone: + 82 31-380-5304<br>E-mail: etelsales@heidenhain.co.kr | **HEIDENHAIN PACIFIC PTE. LTD**<br>51 Ubi Crescent,<br>Singapore 408593<br>Phone: +65 6749 3238<br>E-mail: info@heidenhain.com.sg |
| **SPAIN (Representative)** | **SWEDEN** | **SWITZERLAND** |
| **Farresa Electronica, S.A.**<br>C/ Les Corts, 36 bajos<br>ES-08028 Barcelona<br>Phone: +34 93 409 24 91<br>E-mail: farresa@farresa.es | **HEIDENHAIN Scandinavia AB**<br>Storsätragränd 5<br>127 39 Skärholmen<br>Phone: +468 531 93 350<br>E-mail: sales@heidenhain.se | **HEIDENHAIN (SCHWEIZ) AG**<br>Vieristrasse 14<br>CH-8603 Schwerzenbach<br>Phone: +41 (0)44 806 27 27<br>E-mail: verkauf@heidenhain.ch |
| **TAIWAN** | **THE NETHERLANDS** | **UNITED STATES** |
| **HEIDENHAIN CO., LTD.**<br>No. 29, 33rd road,<br>Taichung Industrial Park<br>Taichung 40768, Taiwan, R.O.C.<br>Phone: +886 4 2358 8977<br>E-mail: info@heidenhain.tw | **HEIDENHAIN NEDERLAND B.V.**<br>Copernicuslaan 34<br>6716 BM Ede<br>Phone: +31 (0)318 581800<br>E-mail: verkoop@heidenhain.nl | **HEIDENHAIN CORPORATION**<br>333 E. State Parkway<br>Schaumburg, IL 60173<br>Phone: +1 847 490 1191<br>E-mail: info@heidenhain.com |

The technical hotline, based in ETEL S.A.'s headquarters, can be reached by:

- Phone: +41 (0)32 862 01 12.
- Fax: +41 (0)32 862 01 01.
- E-mail: support@etel.ch.

Please refer to your corresponding ETEL S.A. representative for more information about the technical documentation. ETEL S.A. organizes training courses for customers on request, including theoretical presentations of our products and practical demonstrations at our facilities.