gILC user manual

Marnix Volckaert

May 6, 2012

Contents

1 Introduction

gILC is an open source software tool for the application of model based iterative learning control (ILC). The approach is easily adapted to work with a broad class of systems, including nonlinear systems, and provides excellent computational efficiency, even for very long data records. It is written in Python and can be installed in Linux and Windows environments. gILC makes use of 2 third-party open source software packages: IPOPT [1] is used for the solution of nonlinear programming (NLP) problems, and CasADi [2] is used for automatic differentiation.

1.1 License information

gILC is distributed under the LGPL license, and therefore users are free to use the code, including in commercial applications. The third-party software CasADi is also distributed under the same LGPL license, and its source code is included with gILC. The software package IPOPT is distributed under the similar EPL license. IPOPT requires the installation of external packages, of which the linear solver package is the most important. The version of IPOPT that is included with gILC (both for Windows and using the install script in Linux) uses the MUMPS linear solver, which is in the public domain. It is possible to build IPOPT from source in Linux using a linear solver of your choice, such as the HSL sparse linear solver. Since this solver is only free for academic users, it cannot be included with the gILC source code. Detailed instructions can be found in section 2.1.2.

2 Installation instructions

The gILC software can be installed both in Linux and Windows operating systems. Installation procedures are slightly different for both, because of the third party software packages. The gILC software itself is written in Python, and is equal for both Linux and Windows versions. Linux users can download the `gilc_linux.tar.gz` file, while Windows users can download the `gilc_windows.zip` file.

2.1 Installation in Linux

For easy installation in Linux, an install script has been created that first installs the prerequisite packages, if needed, using the advanced packaging tool `apt` (for Ubuntu or similar). This includes the package `coinor-libipopt-dev`, which also includes the MUMPS linear solver. gILC for Linux is bundled with the source code of CasADi, which will be built and installed automatically by the install script. Users who wish to build IPOPT from source (for example to include another linear solver, such as HSL), should not use the install script to build CasADi, as written in section 2.1.2.

2.1.1 Basic installation

The easiest way to install the gILC tool in Ubuntu is to use the `install_gilc.sh` script. The following instructions have been tested in version 11.04 and higher:

1. Download `gilc_linux.tar.gz` from the gILC website, and extract it.
2. Open a terminal and go into the extracted folder.
3. Type the following command: `./install_gilc.sh`

For the installation of prerequisite packages, the script will ask you for your root password. Feel free to inspect the `install_gilc.sh` script for more information about what packages are required, and what steps are taken in the installation of gILC.

2.1.2 Advanced installation: building IPOPT and CasADi from source

It is possible to build IPOPT from source, instead of using the `coinor-libipopt-dev` package. The main reason for this is to be able to use a different linear solver than MUMPS, for example the HSL routines (which are free for academic users). To use HSL, IPOPT needs the MA57 package, which can be requested for download from `www.hsl.rl.ac.uk/catalogue/ma57.xml`. These routines can benefit from the software package METIS, which is downloaded separately. These are the necessary steps for the advanced installation:

1. Download `gilc_linux.tar.gz` from the gILC website, and extract it.
2. Locate the files `ma57d.f, mc19d.f` and `deps.f` in the HSL package and copy them to the main `gILC/source` folder
3. Install prerequisite packages using the command:

   -
     ```
     sudo apt-get install wget subversion cmake mpich-bin gfortran libblas-dev \
     liblapack-dev python-dev python-numpy python-scipy swig1.3 --install-recommends
     ```

4. Open a terminal and go into the main gILC folder
5. Download the IPOPT source code, using the following commands:
   - `cd source`
   - `svn co https://projects.coin-or.org/svn/Ipopt/stable/3.10 ipopt`
6. Provide the necessary HSL routines, using the following command:
   - `cp mc19d.f ./ipopt/ThirdParty/HSL/mc19ad.f`
   - `cat ma57d.f deps.f > ./ipopt/ThirdParty/HSL/ma57ad.f`
7. Go into the folder for the METIS sourcecode, and download it by using:
   - `cd ./ipopt/Thirdparty/Metis`
   - `./get.Metis`
8. Create a `build` folder for IPOPT, and run the configure script, by using:
   - `cd ../..`
   - `mkdir build`
   - `cd build`
   - `../configure`
9. Make and install IPOPT:
   - `make`
   - `sudo make install`
10. Go into the build folder for CasADi, and use cmake, with the following commands:
    - `cd ../../casadi/build`
    - `cmake ..`
11. Build and install CasADi and its Python interface:
    - `make`
    - `make python`
    - `sudo make install_python`
12. Go into to the source folder and install the gILC script:
    - `cd ../..`
    - `sudo setup.py install`

If everything went well, you now have a successful gILC installation. Feel free to test some of the tutorials in the `gILC/Tutorials` folder

2.2 Installation in Windows

The Windows installation depends on a precompiled version of CasADi, that includes a precompiled IPOPT with MUMPS as linear solver. It is assumed that a working Python installation is present. An excellent source is `Python(x,y)`, which bundles everything you need to get started with Python in Windows. To install gILC, simply copy the contents of the `gILC\source` folder into the folder for external packages for the Python installation, for example in

```
C:\Python27\Lib\site-packages\
```

Caution: do not copy the `source` folder itself, but only put the content into the Python path.

3 Theoretical background

This section gives a brief overview of the theoretical background of the gILC algorithm. More background can be found in [3] and [4], [1]. Since the algorithm is a generalization of an existing linear approach called norm optimal ILC, this linear approach will be introduced first.

3.1 Linear norm optimal ILC

Conventional norm optimal ILC uses a linear update law of the form

$$\mathbf{u}_{i+1} = Q[\mathbf{u}_i + L(\mathbf{e}_i)], \tag{1}$$

with i the trial index, and $Q(\cdot)$ and $L(\cdot)$ a robustness and learning operator respectively. The tracking error $\mathbf{e}_i$ is defined as $\mathbf{e}_i = (\mathbf{r} - \mathbf{y}_i)$ with $\mathbf{r}$ the reference output. $Q(\cdot)$ and $L(\cdot)$ (or in matrix notation $\mathbf{Q}_{\text{ILC}}$ and $\mathbf{L}_{\text{ILC}}$) are designed in order to minimize a quadratic cost function, which is a trade off between the minimization of the tracking error, input effort, and input update rate, and has the following form [5]:

$$\begin{aligned} J_{i+1}(\mathbf{u}_{i+1}) = \mathbf{e}_{i+1}{}^{\mathrm{T}}\mathbf{Q}_u\mathbf{e}_{i+1} + \mathbf{u}_{i+1}{}^{\mathrm{T}}\mathbf{R}_u\mathbf{u}_{i+1} \\ + \Delta\mathbf{u}_i^{\mathrm{T}}\mathbf{S}_u\Delta\mathbf{u}_i, \end{aligned} \tag{2}$$

where $\Delta\mathbf{u}_i$ denotes the change in $\mathbf{u}$ from trial i to $i+1$, such that $\Delta\mathbf{u}_i = (\mathbf{u}_{i+1} - \mathbf{u}_i)$, $\mathbf{Q}_u$ is an $N \times N$ positive-definite matrix, and $\mathbf{R}_u$ and $\mathbf{S}_u$ are $N \times N$ positive-semidefinite matrices. Note that (2) depends on the unknown next trial tracking error $\mathbf{e}_{i+1}$, which needs to be approximated by using a model $\hat{P}(\mathbf{u})$ of the system $P(\mathbf{u})$, or in matrix form $\hat{\mathbf{P}} \approx \mathbf{P}$.

Since $\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta\mathbf{u}_i$, it follows that $\mathbf{y}_{i+1} = \mathbf{y}_i + \mathbf{P}\Delta\mathbf{u}_i$. Therefore the true value $\mathbf{e}_{i+1}$ can be written as $\mathbf{e}_{i+1} = \mathbf{r} - \mathbf{y}_i - \mathbf{P}\Delta\mathbf{u}_i$, and using $\hat{\mathbf{P}} \approx \mathbf{P}$ the error can be approximated by

$$\hat{\mathbf{e}}_{i+1} = \mathbf{r} - \mathbf{y}_i - \hat{\mathbf{P}}\Delta\mathbf{u}_i. \tag{3}$$

It can be shown that (3) is used as an approximation to $\mathbf{e}_{i+1}$ in the conventional norm optimal ILC to determine the matrices $\mathbf{Q}_{\text{ILC}}$ and $\mathbf{L}_{\text{ILC}}$. A further analysis of (3) shows that this approximation can be interpreted as an implicit model correction. Applying $\Delta\mathbf{u}_i = \mathbf{u}_{i+1} - \mathbf{u}_i$ to (3) and rearranging leads to:

$$\hat{\mathbf{e}}^{i+1} = \mathbf{r} - \underbrace{\left[\hat{\mathbf{P}}\mathbf{u}_{i+1} + (\mathbf{y}_i - \hat{\mathbf{P}}\mathbf{u}_i)\right]}_{\hat{\mathbf{y}}_{i+1}}. \tag{4}$$

The predicted next iteration output $\hat{\mathbf{y}}_{i+1}$ consists of the term $\hat{\mathbf{P}}\mathbf{u}_{i+1}$, which is the expected next iteration output based on the nominal model, corrected by the term $(\mathbf{y}_i - \hat{\mathbf{P}}\mathbf{u}_i)$, such that the next iteration tracking error is estimated better than using the nominal model only. The correction is only implicit, and in a fixed form.

3.2 gILC: a two-step approach

The gILC algorithm is based on a similar cost function (2), but it does not use a linear update law of the form (1). The main difference is in the approximation of the next trial's tracking error. The rearrangement of the predicted next iteration tracking error $\hat{\mathbf{e}}_{i+1}$ from (3) into (4) and the resulting interpretation are based on the LTI property of the model $\hat{P}$. However, the same type of approximation can also be made with nonlinear models if the correction term is made explicit. For example, the next iteration tracking error can be approximated as follows:

$$\hat{\mathbf{e}}_{i+1} = \mathbf{r} - \left[\hat{P}(\mathbf{u}_{i+1}) + \boldsymbol{\alpha}_i\right] \tag{5}$$

with $\boldsymbol{\alpha}_i$ a correction signal to be estimated after trial i. In this case the sum of the nominal model and the correction signal can be written as the corrected model $\hat{P}_c(\mathbf{u}, \boldsymbol{\alpha})$. It is clear that if $\hat{P}(\mathbf{u})$ is a linear model, and $\boldsymbol{\alpha}_i$ is estimated as $(\mathbf{y}_i - \hat{\mathbf{P}}\mathbf{u}_i)$, then $\hat{\mathbf{e}}_{i+1}$ is the same as for the conventional norm optimal ILC. However, explicitly estimating $\boldsymbol{\alpha}_i$ makes it possible to manipulate this signal, for example to constrain it, or to change its features in time- or frequency domain.

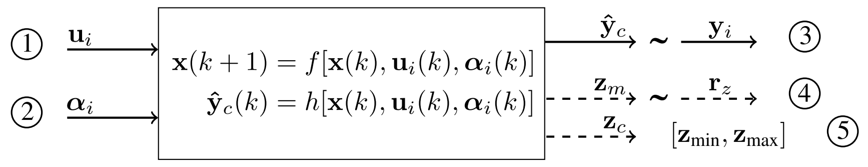
Furthermore, $\boldsymbol{\alpha}$ can enter the corrected model $\hat{P}_c(\mathbf{u},\boldsymbol{\alpha})$ in several ways. For (5) $\boldsymbol{\alpha}$ is additive to the model output, but other examples include $\hat{P}_c(\mathbf{u},\boldsymbol{\alpha}) = \hat{P}(\mathbf{u}+\boldsymbol{\alpha})$, or more complex forms. Regardless of this choice, $\hat{P}_c(\mathbf{u},\boldsymbol{\alpha})$ can in general be written as

$$\hat{P}_c : \begin{cases} \mathbf{x}(k+1) = f_c[\mathbf{x}(k),\mathbf{u}(k),\boldsymbol{\alpha}(k)] \\ \hat{\mathbf{y}}_c(k) = h_c[\mathbf{x}(k),\mathbf{u}(k),\boldsymbol{\alpha}(k)] \end{cases}, \tag{6}$$

and the signal $\boldsymbol{\alpha}$ can be considered to be an additional input of $\hat{P}_c$. Therefore the gILC algorithm consists of finding an optimal value of $\boldsymbol{\alpha}_i$ in an estimation step, followed by a control step to calculate $\mathbf{u}_{i+1}$. Both steps use a user defined corrected model $\hat{P}_c$.

3.2.1 Step 1: estimation

The aim of this step, after trial i, is to find a value for $\boldsymbol{\alpha}_i$, such that the modeled output $\hat{\mathbf{y}}_c$ follows the actual measurement $\mathbf{y}_i$ more closely. Since $\boldsymbol{\alpha}_i$ can be regarded as an additional input signal to the model $\hat{P}_c$, this is an optimal control problem. This optimal control problem is represented schematically in figure 1.



①
$\mathbf{u}_i$
②
$\boldsymbol{\alpha}_i$
$\mathbf{x}(k+1) = f[\mathbf{x}(k),\mathbf{u}_i(k),\boldsymbol{\alpha}_i(k)]$
$\hat{\mathbf{y}}_c(k) = h[\mathbf{x}(k),\mathbf{u}_i(k),\boldsymbol{\alpha}_i(k)]$
$\hat{\mathbf{y}}_c$ ~ $\mathbf{y}_i$ ③
$\mathbf{z}_m$ ~ $\mathbf{r}_z$ ④
$\mathbf{z}_c$ $[\mathbf{z}_{\text{min}},\mathbf{z}_{\text{max}}]$ ⑤

Figure 1: Schematic representation of the optimal control problem in step 1

This schematic representation consists of the following elements:

① **exogenous input**: this is the input signal that is known beforehand. For the estimation step, this is the previous trial's input signal $\mathbf{u}_i$

② **control input**: this is the input signal that is calculated by solving the optimal control problem, in this case the correction signal $\boldsymbol{\alpha}_i$.

③ **main objective**: the aim of the estimation step is to make the model output $\hat{\mathbf{y}}_c$ as close is possible to the measured output $\mathbf{y}_i$. Therefore the main objective function of the optimal control problem is:

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\alpha}_i}{\text{minimize}} \quad & J = \overbrace{(\mathbf{y}_i-\hat{\mathbf{y}}_c)^T\mathbf{Q}_y(\mathbf{y}_i-\hat{\mathbf{y}}_c)}^{J_1} \\ \text{subject to} \quad & g(\mathbf{x}) = 0 : \begin{cases} \mathbf{x}(0) = \mathbf{x}_{\text{init}} \\ \mathbf{x}(k+1) = f[\mathbf{x}(k),\mathbf{u}_i(k),\boldsymbol{\alpha}_i(k)] \end{cases} \end{aligned} \tag{7}$$

with $\mathbf{Q}_y$ a diagonal semidefinite matrix. Note that the states $\mathbf{x}$ are also

④ **additional objectives**: The user has the option to define any number of additional output signals $\mathbf{z}_m$. Such an output signal must be a function of $\mathbf{x}$, $\mathbf{u}_i$ and $\boldsymbol{\alpha}_i$, but does not need to be measurable in the system. A reference $\mathbf{r}_z$ and a diagonal weight matrix $\mathbf{Q}_z$ must be provided for each additional output. The objective function J is then augmented with a quadratic term, minimizing the difference of $\mathbf{z}_m$ and $\mathbf{r}_z$, such that the optimization problem becomes:

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\alpha}_i}{\text{minimize}} \quad & J = J_1 + (\mathbf{r}_z-\hat{\mathbf{z}}_m)^T\mathbf{Q}_z(\mathbf{r}_z-\mathbf{z}_m) \\ \text{subject to} \quad & g(\mathbf{x}) = 0 \end{aligned} \tag{8}$$

⑤ **constraints**: A second group of additional outputs, called $\mathbf{z}_c$, can be defined in order to add constraints to the optimal control problem. $\mathbf{z}_c$ must be a function of $\mathbf{x}$, $\mathbf{u}_i$ and $\boldsymbol{\alpha}_i$, and the lower and upper bounds $\mathbf{z}_{\text{min}}$ and $\mathbf{z}_{\text{max}}$ must be provided for each constrained output. The optimization problem then becomes:

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\alpha}_i}{\text{minimize}} \quad & J \\ \text{subject to} \quad & g(\mathbf{x}) = 0 \\ & \mathbf{z}_{\text{min}} \leq \mathbf{z}_c \leq \mathbf{z}_{\text{max}} \end{aligned} \tag{9}$$

The solution of the optimal control problem (7), (8) or (9) is the correction signal $\boldsymbol{\alpha}_i$, and the states that are associated with this solution. $\boldsymbol{\alpha}_i$ can then be used in the second step, to calculate $\mathbf{u}_{i+1}$

3.2.2 Step 2: control

After step 1 has resulted in an optimal model correction signal $\boldsymbol{\alpha}_i$, a second optimal control problem is formulated. The structure of this problem is identical with the first step. The second step it represented schematically in figure 2. The

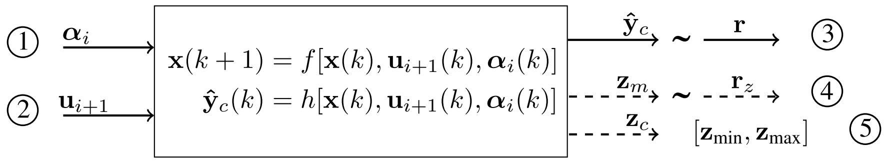

①
$\boldsymbol{\alpha}_i$
②
$\mathbf{u}_{i+1}$
$\mathbf{x}(k+1) = f[\mathbf{x}(k), \mathbf{u}_{i+1}(k), \boldsymbol{\alpha}_i(k)]$
$\hat{\mathbf{y}}_c(k) = h[\mathbf{x}(k), \mathbf{u}_{i+1}(k), \boldsymbol{\alpha}_i(k)]$
$\hat{\mathbf{y}}_c$
$\sim$
$\mathbf{r}$
③
$\mathbf{z}_m$
$\sim$
$\mathbf{r}_z$
④
$\mathbf{z}_c$
$[\mathbf{z}_{\text{min}}, \mathbf{z}_{\text{max}}]$
⑤

Figure 2: Schematic representation of the optimal control problem in step 1

problem is based on the following elements:

① **exogenous input**: in the control step this is the model correction signal, estimated in step 1

② **control input**: the input signal for the next trial

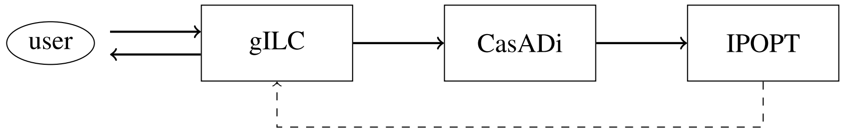
③ **main objective**: the main objective for the step 2 is to make the model output as close as possible to the reference. The optimization problem is similar to (7), but the objective function is $J_1 = (\mathbf{r} - \hat{\mathbf{y}}_c)^T \mathbf{Q}_y (\mathbf{r} - \hat{\mathbf{y}}_c)$. The weight matrix $\mathbf{Q}_y$ is not necessarily equal to the weight matrix used in step 1.

④ **additional objectives**: again the user has the option to augment the optimization problem, by defining additional output signals $\mathbf{z}_m$, and providing corresponding references $\mathbf{r}_z$ and diagonal weight matrices $\mathbf{Q}_z$.

⑤ **constraints**: similar to step 1, constraints can be added to the problem by defining constraint output functions $\mathbf{z}_c$ and providing lower and upper bounds $\mathbf{z}_{\text{min}}$ and $\mathbf{z}_{\text{max}}$.

3.2.3 Software structure

gILC consists of a Python script that interfaces between the two step approach to ILC described above, the underlying optimization problems and the solvers used to minimize those problems. The structure can be summarized in the following scheme:



The user provides the following information in order to use gILC:

1. Model equations: the functions f and h as a function of $\mathbf{x}$, $\mathbf{u}$ and $\boldsymbol{\alpha}$.
2. The reference output $\mathbf{r}$

3. The initial state $\mathbf{x}_{\text{init}}$
4. Additional objectives for the estimation step and/or control step (optional)
5. Constraints for the estimation step and/or control step (optional)

The gILC algorithm then uses CasADi to construct the optimization problem to be solved, and calls IPOPT to solve the problem. The results are then passed back to the user. After initialization of the algorithm (at which point the information listed above is passed), gILC provides 2 main functions: a function to solve the estimation step, to calculate $\boldsymbol{\alpha}_i$ based on $\mathbf{u}_i$ and $\mathbf{y}_i$, and a function to solve the control step, to calculate $\mathbf{u}_{i+1}$ based on $\boldsymbol{\alpha}_i$.

4 Tutorials

This section shows you how to use the gILC software by describing each command of a typical nonlinear ILC application. The tutorials gradually introduce extra functionality, but all are based on the example application described in section 4.1.

4.1 Example application

4.1.1 Simulated system and model

The system that is considered in the tutorials is shown in figure 3 and consists of a rod with a mass m of 0.5 kg, a moment of inertia I of 0.006 kgm^2 and a length $2L$ of 0.6 m (such that $L = 0.3$ m), attached to an actuated hinge on one end. A torque T can be applied to the rod in the hinge, which has a viscous damping coefficient c of 0.1 Ns/rad. The angle of the rod with respect to the vertical is θ.

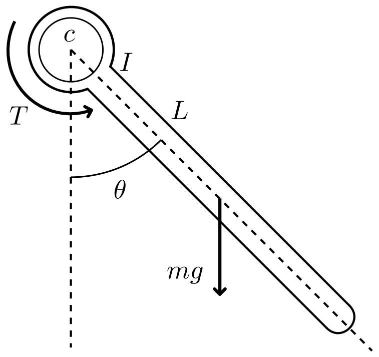


Figure 3: The example application is a rod attached to an actuated hinge

The purpose of the ILC algorithm is to find a feedforward control signal for T in order to raise the rod from the stable equilibrium $\theta = 0°$ to the unstable equilibrium $\theta = 180°$ with a constant rotational velocity, and return to $\theta = 0°$. It is assumed that the parameters m and c are not known exactly, to introduce a model plant mismatch. Due to the fact that $\theta = 180°$ is an unstable equilibrium and requires $T = 0$ N independent of m and c, the system is very sensitive to this model plant mismatch.

The dynamics of the rod are governed by the following equation:

$$(I + mL^2)\ddot{\theta} = T - gmL\sin(\theta) - c\dot{\theta} \tag{10}$$

Using T as input u, θ as output y, and $[\theta, \dot{\theta}]^T$ as state vector $\mathbf{x}$, the following state space model can be constructed:

$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} x_2 \\ \frac{1}{(I+mL^2)}[-gmL\sin(x_1) - cx_2 + u] \end{bmatrix} \\ y &= x_1 \end{aligned} \tag{11}$$

Discretized using a forward Euler approach with sample time T_s, this leads to:

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} x_1(k) + T_s x_2(k) \\ x_2(k) + \frac{T_s}{(I+mL^2)}\Big[-gmL\sin[x_1(k)] - cx_2(k) + u(k) \Big] \end{bmatrix} \\ y(k) = x_1(k) \tag{12}$$

This discrete-time state space model is used as simulated system and as model for the ILC algorithm. The parameters for the simulated system and the model are summarized in table 1.

	Plant	Model
m [kg]	0.5	0.495
c [Ns/rad]	0.1	0.105
I [kgm^2]	0.006	0.006
L [m]	0.3	0.3
$\mathbf{T_s}$ [s]	0.002	0.002

Table 1: Parameters of the simulated system and the model for the example application

4.1.2 Reference

The objective of the example application is to track a reference signal for the angle θ using gILC. An example reference signal is shown in figure 4.1.2. This reference signal has a trapezoidal velocity profile, and contains 2000 samples. A script called `a_create_reference.py` is included in order to generate this reference signal using a parameterized velocity profile, and to save it in a file. This file will then be read by each of the tutorial scripts to load the reference. After getting acquainted with gILC, feel free to experiment with different velocity profiles or reference signals.

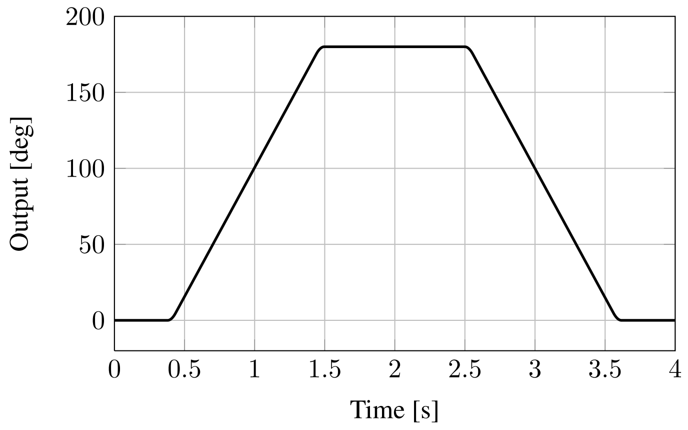


Figure 4: Reference output for the example application

4.1.3 Model correction

The provided model needs to be corrected by at least one term $\alpha(k)$. In this example application one correction term is used, additive to the input signal. This means that in the model (12), $u(k)$ is replaced by $[u(k) + \alpha(k)]$.

4.2 Tutorial 1: nonlinear ILC

The first tutorial can be found in the script

```
b_tutorial1.py
```

and demonstrates how to use gILC to find a torque profile to track the reference without any additional objectives or constraints. The script `b_tutorial1.py` also contains commands used for the simulation of the system and for plotting, but since these commands are not gILC specific, they will not be described here.

4.2.1 Initialization of gILC

In order to use the gILC package, it must be imported at the beginning of the script, by using the command

```
from gilc import *
```

All interaction with gILC occurs through an object, typically called `ilc`, which needs to be created by the command

```
ilc = gilcClass()
```

Using this `ilc` object, the first settings to be defined are the problem dimensions:

`ilc.nu = 1`	Number of inputs
`ilc.na = 1`	Number of model correction terms
`ilc.nx = 2`	Total number of states (system states + additional states)
`ilc.nxa = 0`	Number of additional states
`ilc.ny = 1`	Number of outputs

For some applications, the state vector is augmented with additional states. This is for example the case when a constraint or penalty term requires knowledge of the control signal at a time instant different from k, such as $\mathbf{u}(k-1)$. However, such additional states do not require an initial value (since this value can be unknown), which is required for the system states. To make this distinction, `ilc.nx` and `ilc.nxa` are passed separately.

The next step is to provide the model equations. First the state equation is defined in a function:

```
def f(xk,uk,ak):
    I = 0.006
    m = 0.495
    L = 0.3
    c = 0.105
    Ts = 1/500.0
    xkp1 = [xk[0]+Ts*xk[1],\
            xk[1]+(Ts/(I+m*L**2))*(-c*xk[1]-9.81*m*L*sin(xk[0])+(uk[0]+ak[0])),\
            ]
    return xkp1
```

Note that the function `f(xk,uk,ak)` is a function of the state vector $\mathbf{x}(k)$, the input vector $\mathbf{u}(k)$, and the correction vector $\boldsymbol{\alpha}(k)$. All three dependencies must be included in the function definition. However, they do not need to be present in the function itself, for example if the correction only occurs in the model output equation. Note that `xk`, `uk` and `ak` must have an index, even if their dimension is 1 (for example, use `uk[0]` for the first and only input, instead of `uk`). The output equation is defined in another function:

```
def h(xk,uk,ak):
    y = [xk[0], \
        ]
    return y
```

Again the function definition must include all three dependancies, even if they are not present in the actual function.

gILC also requires the initial state of the system (the state of the system at the start of the trial). As is common in ILC, it is assumed that the system is brought to the same initial state for each trial. This state, which for the example application is the zero vector, is passed to the algorithm by the command:

```
ilc.xinit = [0.0,0.0]
```

Now that all necessary information is passed to the algorithm, the simulation can start. The two main functions of the gILC algorithm are called `ilc.estimation` and `ilc.control` respectively. These functions are described in the following section.

4.2.2 Estimation and control functions

The purpose of the estimation function is to find the optimal value of $\boldsymbol{\alpha}_i$, based on $\mathbf{u}_i$ and $\mathbf{y}_i$ of the trial i. The function is called with the following command:

```
a_i, xa_i = ilc.estimation(u_i,y_i)
```

After completing the estimation step, the control step is called to find the next trial's input signal $\mathbf{u}_{i+1}$, based on the correction signal $\boldsymbol{\alpha}_i$. This step is performed using:

```
u_i, xu_i = ilc.control(a_i)
```

Note that the states are also given as output of both the estimation and control functions. Before starting the simulation, the input signal for the first trial is found by initializing the model correction term to zero and calling the control function.

4.2.3 Additional settings

The commands described in 4.2.1 are necessary commands, and gILC will send an error if they are not provided. There are however a number of settings for which a default value is used if they are not provided, but which can be overwritten if necessary.

1. **Initial guess of optimization variables**. The optimization variables are the states and correction terms (for the estimation step), or the states and input (for the control step). The default is to initialize all variables at zero. However, it is possible to initialize at another value. In that case, an initial value must be provided during the initialization of gILC, using the following commands:

   ```
   ilc.est.xguess = zeros((ilc.nx,N))
   ilc.est.alphaguess = zeros((ilc.na,N))
   ```

   for the estimation step, and

   ```
   ilc.con.xguess = zeros((ilc.nx,N))
   ilc.con.uguess = zeros((ilc.nu,N))
   ```

   for the control step. Providing an initial guess for one step doesn't require you to provide a guess for the other step. However, if only part of the initial guess is needed (such as an initial guess of the states but not the input in the control step), then both parts of that step have to be initialized explicitly, even if a zero value is used. The initial guess can be updated at any time during the operation of the algorithm. A typical example is to use the solution for the states that is found in the estimation step, called `xa_i` in the tutorial, as an initial guess for the states in the control step, using:

   ```
   ilc.con.xguess = xa_i
   ```

2. **Weight of the residual**. The diagonal weight matrix for the main objective in both the estimation and control optimization problems is $\mathbf{Q}_y$. This matrix has a default value of $1e8\mathrm{I}$, for both steps, and therefore does not need to be provided explicitly. The default value can be overwritten using the following commands:

   ```
   ilc.est.Qy = 1e4*ones((ilc.ny,N))
   ilc.con.Qy = 1e4*ones((ilc.ny,N))
   ```

4.3 Tutorial 2: adding constraints

The second tutorial builds on tutorial 1, and demonstrates how to add constraints to the gILC algorithm. It was shown in section 3 that constraints and additional objectives are handled by defining additional outputs $\mathbf{z}$, namely $\mathbf{z}_c$ for constraints, with $\mathbf{z}_c(k)$ a function of $\mathbf{x}(k)$, $\mathbf{u}(k)$ and $\boldsymbol{\alpha}(k)$. Upper and lower bounds for each time instant k can then be put on this output. The script

```
c_tutorial2
```

demonstrates this by constraining the input signal to the interval $[-2, 2]$ Nm. The first step is to define the constrained output function during initialization:

```
def hzc(xk,uk,ak):
    zc = [uk[0]]
    return zc
```

Note that, similar to the functions `f` and `h`, `hzc` must be a function all the variables `xk`,`uk`,`ak`, even if they do not appear in the function itself, and again all variables need an index. The command written above merely defines the function, so the next step is to assign the function to either the estimation step or the control step, in this case by the command:

```
ilc.con.hzc = hzc
```

Now that the constrained output is defined for the control step, the upper and lower bounds are passed using the following commands:

```
ilc.con.zmin = -2*ones((1,N))
ilc.con.zmax = 2*ones((1,N))
```

These commands are written in the initialization of the gILC algorithm. However, it is possible to change the bounds during each trial, by simply assigning new values to `zmin` and `zmax`.

4.4 Tutorial 3: adding additional objectives - dealing with measurement noise

The third tutorial demonstrates how to add additional objectives to the optimization functions of the estimation and control optimization problems. It was shown in section 3 that an additional objective can be seen as a penalty or regularization term to the objective function, minimizing the difference between $\mathbf{z}_m$ and a given reference $\mathbf{r}_z$, in a least squares sense and weighted by a diagonal matrix $\mathbf{Q}_z$. By setting the reference for the defined output function to zero, the function itself can be minimized. The tutorial is a further extension of tutorial 2, and can be found in the following script:

```
d_tutorial3
```

For this tutorial, the effect of measurement noise is investigated. Therefore a Gaussian distributed random noise term with a standard deviation of 0.005 rad is added to the output of the simulated model (12). Measurement noise in ILC typically leads to an input signal with large contributions at high frequencies, which is often countered by applying low pass filtering to the calculated input signal. In gILC, the effect of measurement noise can be minimized by adding a regularization term in the control step, to minimize the change of the input signal with respect to the previous trial's input signal. The first step is defining the additional output function:

```
def hzm(xk,uk,ak):
    zm = [uk[0]]
    return zm
```

and assigning this function to the control step:

```
ilc.con.hzm = hzm
```

The reference will be set to the previous trial's input signal during each trial, but needs to be initialized first:

```
ilc.con.rz = zeros((1,N))
```

Since the regularization should only be applied after the first trial, the diagonal of the weight matrix is initialized at zero:

```
    ilc.con.Qz = 0*ones((1,N))
```

After the first trial has been evaluated, the reference and the weight for the additional objective are updated by the commands:

```
        ilc.con.rz = u_i
        ilc.con.Qz = 1e7*ones((1,N))
```

References

[1] IPOPT. A software package for large-scale nonlinear optimization. `http://projects.coin-or.org/Ipopt`, May 2012.

[2] CasADi. Implementing automatic differentiation by means of a hybrid symbolic/numeric approach. `http://sourceforge.net/apps/trac/casadi/`, May 2012.

[3] Marnix Volckaert, Jan Swevers, and Moritz Diehl. A two step optimization based iterative learning control algorithm. *ASME Dynamic Systems and Control Conference*, 2010.

[4] Marnix Volckaert, Moritz Diehl, and Jan Swevers. Iterative learning control for nonlinear systems with input constraints and discontinuously changing dynamics. *American Control Conference*, pages 3035–3040, 2011.

[5] S. Gunnarsson and M. Norrlof. On the design of ILC algorithms using optimization. *Automatica*, 37(12):2011–2016, Jan 2001.