

# pyNLControl

pyNLControl is a package to solve general estimation and control problem (including non-linear problem). Further, it also provides different method for analysis of dynamic system. This package is based on `CasADi` for python (<https://web.casadi.org/>). This means problem should be formulated in `CasADi`.

## Requirements

- python >= 3.6 (might work on older version of python3, not tested)
- casadi>=3.5.5 and jinja2>=3.0.2 `pip install casadi jinja2`

## Installations

```
pip install pyNLControl
```

## Supported control and estimator

- Estimators: Kalman filter, Extended Kalman Filter, Unscented Kalman Filter and simple moving horizon estimators. Partial filter, advanced moving horizon estimator, etc will be added soon.
- Control: LQR only. Other controllers including nonlinear Model Predictive Control will be added soon
- Misc: Nonlinear observability analysis, Noise covariance identification will be added soon

## Module pynlcontrol.BasicUtils

### Functions

`Gen_Code(func, filename, dir='/', mex=False, printhelp=False, optim=False)` : Function to generate c code (casadi generated as well as interface) for casadi function.

Generates c and h file (from CasADi) as well as interface code. Using interface code, the generated codes can be integrated with other platform such as Simulink, PSIM, etc.

Parameters

-----

`func` : `casadi.Function`

CasADi function for which code needs to be generated. This function maps input to output.

`filename` : str

File name of the generated code. Note: Filename should not contain "\_Call" in it.

`dir` : str, optional

Directory where codes need to be generated. Defaults to current directory.

`mex` : bool, optional

Option if mex is required. Defaults to False.

`printhelp` : bool, optional

Option if information about input/output and its size are to be printed . If

mex is False, sfunction help is also printed. Defaults to False.

optim : bool, optional

Whether code is being generated for CasADi optimization problem. Defaults to False.

See Also

-----

Gen\_Test: Generates main function code which can be compiled and debugged.

Example

-----

```
>>> import casadi as ca
>>> import numpy
>>> x = ca.SX.sym('x')
>>> y = ca.SX.sym('y', 2)
>>> f1 = x + y[0]
>>> f2 = x + y[1]**2
>>> Func = ca.Function(
    'Func',
    [x, y],
    [f1, f2],
    ['x', 'y'],
    ['f1', 'f2'],
    )
>>> BasicUtils.Gen_Code(Func, 'GenCodeTest', dir='./', mex=False, printhelp=True)
x(1, 1), y(2, 1) -> f1(1, 1), f2(1, 1)
GenCodeTest.c
GenCodeTest_Call.c
#include "GenCodeTest.h"
#include "GenCodeTest_Call.h"
GenCodeTest_Call_Func(x, y, f1, f2);
```

Above code creates `GenCodeTest.c`, `GenCodeTest\_Call.c`, `GenCodeTest.h` and `GenCodeTest\_Call.h` that evaluates `f1` and `f2` from value of `x` and `y`. `x` and `y` in the above example should be declared as pointer.

Gen\_Test(headers, varsIn, sizeIn, varsOut, sizeOut, callFuncName, filename, dir='/') :  
Generates C code with main function. This code along with code generated by Gen\_Code function can be compiled to executable to check computation time or debug. It can also be used as example to compile on other target.

Parameters

-----

headers : list[str]

List of header files name along with extension `.h`. These files will be added in main file as #include "header".

varsIn : list[str]

List of name of input variables.

sizeIn : list[int]

List of size of input variables.

varsOut : list(str)

List of name of output variables.

sizeOut : list[int]

List of size of output variables.

callFuncName : list(str)

Name of the C function that needs to be called.

filename : str

Name of the generated C file (along with extension .c).

dir : str, optional

Directory where code needs to be generated. Defaults to current directory.

Example

```
>>> from pynlcontrol import BasicUtils
>>> import casadi as ca
>>> x = ca.SX.sym('x')
>>> y = ca.SX.sym('y', 2)
>>> f1 = x + y[0]
>>> f2 = x + y[1]**2
>>> Func = ca.Function(
    'Func',
    [x, y],
    [f1, f2],
    ['x', 'y'],
    ['f1', 'f2'],
)
>>> BasicUtils.Gen_Code(Func, 'GenCodeTest', dir='./', mex=False, printhelp=False)
>>> BasicUtils.Gen_Test(
    headers=['GenCodeTest.h', 'GenCodeTest_Call.h'],
    varsIn=['x', 'y'],
    sizeIn=[1, 2],
    varsOut=['f1', 'f2'],
    sizeOut=[1, 1],
    callFuncName='GenCodeTest_Call_Func',
    filename='MainTest.c',
    dir='./',
)
```

All the line except last one is same as from `Gen\_Code()` function. The last line generates filename with `MainTest.c` which can be used to test and debug the generated code.

`Integrate(odefun, method, Ts, x0, u0, *args)` : Function to integrate continuous-time ODE. It discretize the provided ODE function and gives value of state variables at next discrete time.

Parameters

-----

odefun : function

Python function with states as first and control input as second argument. Remaining argument could be anything required by state equations.

method : str

Method to integrate ODE. Supported methods are: 'FEuler', 'rk2', 'rk3', 'ssprk3', 'rk4', 'dormandprince'.

Ts : float

Step time to solve ODE

x0 : float or casadi.SX or numpy.1darray

Current states of the system

u0: float or casadi.SX or numpy.1darray

Current control input to the system

Returns

-----

float or casadi.SX or numpy.1darray

Next states of the system.

Example

-----

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def Fc(x, u):
    x1 = x[0]
    x2 = x[1]
    return np.array([(1-x2**2)*x1 - x2 + u, x1])
>>> T = 10
>>> Ts = 0.1
>>> t = np.arange(0, T+Ts, Ts)
>>> x = np.zeros((2, t.shape[0]))
>>> for k in range(t.shape[0]-1):
    u = 0 if t[k] < 1 else 1.0
    x[:,k+1] = Integrate(Fc, 'rk4', Ts, x[:,k], u)
>>> plt.plot(t, x[0,:])
>>> plt.plot(t, x[1,:])
>>> plt.xlabel('time (s)')
>>> plt.ylabel('$x_1$ and $x_2$')
>>> plt.show()
```

The above code integrates the ODE defined by function Fc from 0 to 10 s with step-time of 0.1 s using RK4 method.

`directSum(A)` : Direct sum of matrices in the list A.

Parameters

-----

A : list

List of matrices.

Returns

-----

casadi.SX.sym

Direct sum of all matrices in list A.

Example

-----

```
>>> import casadi as ca
>>> from pynlcontrol import BasicUtils
>>> import casadi as ca
>>> from pynlcontrol import BasicUtils
>>> A1 = ca.SX.sym('A1', 2, 2)
>>> A2 = ca.SX.sym('A2', 3, 3)
>>> A = [A1, A2]
>>> BasicUtils.directSum(A)
SX(@1=0,
[[A1_0, A1_2, @1, @1, @1],
 [A1_1, A1_3, @1, @1, @1],
```

```
[@1, @1, A2_0, A2_3, A2_6],
[@1, @1, A2_1, A2_4, A2_7],
[@1, @1, A2_2, A2_5, A2_8]])
```

Above code puts matrices `A1` and `A2` in the diagonal and fills zero elsewhere.

`nlp2GGN(z, J, g, lbg, ubg, p)` : Converts provided nonlinear programming into quadratic form using generalized Gauss-Newton method.

#### Parameters

-----

```
z : casadi.SX
    Vector of unknown variables of optimization problem
J : casadi.SX
    Object function of the optimization problem
g : casadi.SX
    Vector of constraints function
lbg : casadi.SX
    Vector of lower limits on constraint function
ubg : casadi.SX
    Vector of upper limits on constraint function
p : casadi.SX
    Vector of input to the optimization problem
```

#### Returns

-----

```
dict
    Dictionary of optimization problem.
keywords
    x: Vector of decision variables
    f: New quadratic cost function
    g: New constraint function
    lbg: Lower limits on constraint function
    ubg: Upper limits on constraint function
```

## Module `pynlcontrol.Estimators`

### Functions

`EKF(nX, nU, nY, F, H, Qw, Rv, Ts, Integrator='rk4')` : Function to implement Extended Kalman filter (EKF).

#### Parameters

-----

```
nX: (int)
    Number of state variables
nU: (int)
    Number of control inputs
ny: (int)
    Number of measurement outputs
F: (function)
```

Function that returns right-hand side of state differential equation. Input arguments to F should be states and inputs respectively.

H: (function)  
 Function that returns measurement variable from state and input variables. Input arguments to H should be states and inputs respectively.

Qw: (numpy.2darray or casadi.SX array)  
 Process noise covariance matrix

Rv: (numpy.2darray or casadi.SX array)  
 Measurement noise covariance matrix

Ts: (float)  
 Sample time of the Kalman filter.

Integrator: (str, optional)  
 Integration method. Defaults to 'rk4'. For list of supported integrator, please see documentation of function `Integrate()`.

Returns

-----

tuple:

Tuple of Input, Output, Input name and Output name. Inputs are u, y, xp, Pp and output are xhat and Phat. Input and output are casadi symbolics (`casadi.SX`).

u: Current input to the system

y: Current measurement of the system

xp: State estimate from previous discrete time

Pp: Covariance estimate from previous discrete time (reshaped to column matrix)

xhat: State estimate at current discrete time

Phat: Covariance estimate at current discrete time (reshaped to column matrix)

These inputs and outputs can be mapped using `casadi.Function` which can further be code generated.

KF(A, B, C, D, Qw, Rv, Ts, Integrator='rk4') : Function to implement Kalman filter (KF).

Parameters

-----

A: (numpy.2darray or casadi.SX array)

Continuous-time state matrix of the system

B: (numpy.2darray or casadi.SX array)

Continuous-time input matrix of the system

C: (numpy.2darray or casadi.SX array)

Continuous-time measurement matrix of the system

D: (numpy.2darray or casadi.SX array)

Continuous time output matrix coefficient of input

Qw: (numpy.2darray or casadi.SX array)

Process noise covariance matrix

Rv: (numpy.2darray or casadi.SX array)

Measurement noise covariance matrix

Ts: (float)

Sample time of KF

Integrator: (str, optional)

Integrator to be used for discretization. Defaults to 'rk4'.

Returns

-----

tuple

Tuple of Input, Output, Input name and Output name. Inputs are u, y, xp, Pp and output are xhat and Phat. Input and output are casadi symbolics (`casadi.SX`).

u: Current input to the system  
y: Current measurement of the system  
xp: State estimate from previous discrete time  
Pp: Covariance estimate from previous discrete time (reshaped to column matrix)  
xhat: State estimate at current discrete time  
Phat: Covariance estimate at current discrete time (reshaped to column matrix)

These inputs and outputs can be mapped using `casadi.Function` which can further be code generated.

#### Example

```

-----
>>> from pynlcontrol import Estimator, BasicUtils
>>> import casadi as ca
>>> Q11 = ca.SX.sym('Q11')
>>> Q22 = ca.SX.sym('Q22')
>>> Q33 = ca.SX.sym('Q33')
>>> Q = BasicUtils.directSum([Q11, Q22, Q33])
>>> R22 = ca.SX.sym('R22')
>>> R = BasicUtils.directSum([R11, R22])
>>> A = ca.SX([[ -0.4, 0.1, -2], [0, -0.3, 4], [1, 0, 0]])
>>> B = ca.SX([[1, 1], [0, 1], [1, 0]])
>>> C = ca.SX([[1, 0, 0], [0, 1, 0]])
>>> D = ca.SX([[0, 0], [0, 0]])
>>> In, Out, InName, OutName = Estimator.KF(A, B, C, D, Q, R, 0.1)
>>> KF_func = ca.Function('KF_func', In + [Q11, Q22, Q33, R11, R22], Out, InName +
['Q11', 'Q22', 'Q33', 'R11', 'R22'], OutName)
>>> BasicUtils.Gen_Code(KF_func, 'KF_code', printhelp=True)
u(2, 1), y(2, 1), xhatp(3, 1), Pkp(9, 1), Q11(1, 1), Q22(1, 1), Q33(1, 1), R11(1,
1), R22(1, 1) -> xhat(3, 1), Phat(9, 1)
KF_code.c
KF_code_Call.c
#include "KF_code.h"
#include "KF_code_Call.h"
KF_code_Call_Func(u, y, xhatp, Pkp, Q11, Q22, Q33, R11, R22, xhat, Phat);

```

Running above code generates C-codes for KF implementation. Implementation using Simulink can be found in example folder.

UKF(nX, nU, nY, F, H, Qw, Rv, Ts, PCoeff=None, Wm=None, Wc=None, alpha=0.001, beta=2.0, kappa=0.0, Integrator='rk4') : Function to implement Unscented Kalman filter (UKF).

If either of PCoeff or Wm or Wc is None, it calculates those values with alpha=1e-3, Beta=2 and kappa=0. To use manual weights, specify PCoeff, Wm and Wc. Otherwise, use alpha, beta and kappa parameters to set those values.

#### Parameters

-----

nX: (int)  
Number of state variables

`nU: (int)`  
 Number of control inputs  
`nY: (int)`  
 Number of measurement outputs  
`F: (function)`  
 Function that returns right-hand side of state differential equation. Input arguments to F should be states and inputs respectively.  
`H: (function)`  
 Function that returns measurement variable from state and input variables. Input arguments to H should be states and inputs respectively.  
`Qw: (numpy.2darray or casadi.SX array)`  
 Process noise covariance matrix  
`Rv: (numpy.2darray or casadi.SX array)`  
 Measurement noise covariance matrix  
`Ts: (float)`  
 Sample time of the Kalman filter.  
`PCoeff: (float)`  
 Coefficient of covariance matrix (inside square root term) when calculating sigma points. Defaults to None  
`Wm: (list, optional)`  
 List of weights for mean calculation. Defaults to None.  
`Wc: (list, optional)`  
 List of weights for covariance calculation. Defaults to None.  
`alpha: (float, optional)`  
 Value of alpha parameter. Defaults to 1.0e-3.  
`beta: (float, optional)`  
 Value of beta parameter. Defaults to 2.0.  
`kappa: (float, optional)`  
 Value of kappa parameter. Defaults to 0.0.  
`Integrator: (str, optional)`  
 Integration method. Defaults to 'rk4'. For list of supported integrator, please see documentation of function `Integrate`.

#### Returns

-----

tuple:

Tuple of Input, Output, Input name and Output name. Inputs are u, y, xp, Pp and output are xhat and Phat. Input and output are casadi symbolics (`casadi.SX`).

u: Current input to the system  
 y: Current measurement of the system  
 xp: State estimate from previous discrete time  
 Pp: Covariance estimate from previous discrete time (reshaped to column matrix)  
 xhat: State estimate at current discrete time  
 Phat: Covariance estimate at current discrete time (reshaped to column matrix)

These inputs and outputs can be mapped using `casadi.Function` which can further be code generated.

`simpleMHE(nX, nU, nY, nP, N, Fc, Hc, Wp, Wm, Ts, pLower=[], pUpper=[], arrival=False, GGN=False, Integrator='rk4', Options=None)` : Function to generate simple MHE code using grqp solver. For use with other advanced solver, see MHE class.



## Parameters

-----

nX: (int)  
Number of state variables.

nU: (int)  
number of control input.

nY: (int)  
Number of measurement variables.

nP: (int)  
Number of parameter to be estimated. nP=0 while performing state estimation only.

N: (int)  
Horizon length.

Fc: (function)  
Function that returns right hand side of state equation.

Hc: (function)  
Function that returns right hand side of measurement equation.

Wp: (float or casadi.SX array or numpy.2darray)  
Weight for process noise term. It is  $Q_w^{-1/2}$  where  $Q_w$  is process noise covariance.

Wm: (float or casadi.SX array or numpy.2darray)  
Weight for measurement noise term. It is  $R_v^{-1/2}$  where  $R_v$  is measurement noise covariance.

Ts: (float): Sample time for MHE

pLower: (list, optional)  
List of lower limits of unknown parameters. Defaults to [].

pUpper: (list, optional)  
List of upper limits of unknown parameters. Defaults to [].

arrival: (bool, optional)  
Whether to include arrival cost. Defaults to False.

GGN: (bool, optional)  
Whether to use GGN. Use this option only when optimization problem is nonlinear. Defaults to False.

Integrator: (str, optional)  
Integration method. See ``BasicUtils.Integrate()`` function. Defaults to 'rk4'.

Options: (dict, optional)  
Option for ``qrqp`` solver. Defaults to None.

## Returns

-----

tuple:  
Tuple of Input, Output, Input name and Output name. Input and output are list of casadi symbolics (``casadi.SX``).  
Input should be control input and measurement data of past horizon length  
Output are all value of decision variable, estimations of parameter, estimates of states and cost function.

# Module pynlcontrol.Controller

## Functions

LQR(A, B, C, D, Q, R, Qt, Ts, horizon=inf, reftrack=False, NMAX=1000, tol=1e-05, Integrator='rk4') : Function to implement discrete-time linear quadratic regulator (LQR).

## Parameters

-----

A : numpy.2darray or casadi.SX.array  
Continuous time state matrix

B : numpy.2darray or casadi.SX.array  
Continuous time input matrix

C : numpy.2darray or casadi.SX.array  
Continuous time output matrix

D : numpy.2darray or casadi.SX.array  
Continuous time output matrix coefficient of input

Q : numpy.2darray or casadi.SX.array  
Weight to penalize control error

R : numpy.2darray or casadi.SX.array  
Weight to penalize control effort

Qt : numpy.2darray or casadi.SX.array  
Weight of terminal cost to penalize control error

Ts : float  
Sample time of controller

horizon : int, optional  
Horizon length of LQR. Defaults to inf for infinite horizon LQR problem.

reftrack : bool, optional  
Whether problem is reference tracking. Defaults to False.

NMAX : int, optional  
Maximum iteration for solving matrix Ricatti equation. Defaults to 1000.

tol : float, optional  
Tolerance for solution of matrix Ricatti equation. Defaults to 1e-5.

Integrator : str, optional  
Integrator to be used for discretization. Defaults to 'rk4'.

## Returns

-----

tuple

Tuple of Input, Output, Input name and Output name. Inputs are x or [x, r] (depending upon the problem is reference tracking or not) and output are u and K. Input and output are casadi symbolics (`casadi.SX`). These inputs are and outputs can be mapped using `casadi.Function` which can further be code generated.

## Example

-----

```
>>> from pynlcontrol import Controller, BasicUtils
>>> import casadi as ca
>>> Q11 = ca.SX.sym('Q11')
>>> Q22 = ca.SX.sym('Q22')
>>> Q33 = ca.SX.sym('Q33')
>>> Q = BasicUtils.directSum([Q11, Q22, Q33])
>>> R11 = ca.SX.sym('R11')
>>> R22 = ca.SX.sym('R22')
>>> R = BasicUtils.directSum([R11, R22])
>>> A = ca.SX([[ -0.4, 0.1, -2], [0, -0.3, 4], [1, 0, 0]])
>>> B = ca.SX([[1, 1], [0, 1], [1, 0]])
>>> C = ca.SX([[1, 0, 0], [0, 1, 0]])
>>> D = ca.SX([[0, 0], [0, 0]])
>>> In, Out, InName, OutName = Controller.LQR(A=A, B=B, C=C, D=D, Q=Q, R=R, Qt=Q,
Ts=0.1, horizon=10, reftrack=True)
>>> lqr_func = ca.Function('lqr_func', In + [Q11, Q22, Q33, R11, R22], Out, InName
+ ['Q11', 'Q22', 'Q33', 'R11', 'R22'], OutName)
>>> BasicUtils.Gen_Code(lqr_func, 'lqr_code', printhelp=True)
```

```

x(3, 1), ref(2, 1), Q11(1, 1), Q22(1, 1), Q33(1, 1), R11(1, 1), R22(1, 1) -> u(2,
1), K(6, 1)
lqr_code.c
lqr_code_Call.c
#include "lqr_code.h"
#include "lqr_code_Call.h"
lqr_code_Call_Func(x, ref, Q11, Q22, Q33, R11, R22, u, K);

```

Running above code generates C-codes for LQR implementation. Implementation using Simulink can be found in example folder.

# Module pynlcontrol.QPInterface

## Classes

qpOASES(H, h, p=None, A=None, lbA=None, ubA=None, lbx=None, ubx=None) : Class to create interface to qpOASES solver.

It can be used to generate to C/C++ code to solve given quadratic optimization problem.

### Parameters

-----

H: (casadi.SX)  
Hessian matrix of cost function

h: (casadi.SX)  
Linear coefficient vector of cost function

p: (list, optional)  
List of input parameters to optimization problem. Defaults to None.

A: (casadi.SX, optional)  
Constraint matrix. Defaults to None.

lbA: (casadi.SX, optional)  
Lower bound on constraint matrix. Defaults to None.

ubA: (casadi.SX, optional)  
Upper bound on constraint matrix. Defaults to None.

lbx: (casadi.SX, optional)  
Lower bound on decision variables. Defaults to None.

ubx: (casadi.SX, optional)  
Upper bound on decision variables. Defaults to None.

### Returns

-----

None

### Example

-----

Consider optimization problem:

```

min (x1-a)^2 + (x2-b)^2
s.t. 2x1+3x2 <= 3
      0 <= x1-x2 <= 10

```

Solver for this optimization problem can be generated as

```
>>> import casadi as ca
```

```

>>> from pynlcontrol import QPInterface
>>> x1 = ca.SX.sym('x1')
>>> x2 = ca.SX.sym('x2')
>>> a = ca.SX.sym('a')
>>> b = ca.SX.sym('b')
>>> J = (x1-a)**2 + (x2-b)**2
>>> H, h, _ = ca.quadratic_coeff(J, ca.vertcat(x1, x2))
>>> g = ca.vertcat(2*x1+3*x2, x1-x2), ca.vertcat(x1, x2)
>>> lbg = ca.vertcat(-ca.inf, 0)
>>> ubg = ca.vertcat(3, 10)
>>> A, c = ca.linear_coeff(ca.vertcat(2*x1+3*x2, x1-x2), ca.vertcat(x1, x2))
>>> lbA = lbg - c
>>> ubA = ubg - c
>>> qp = QPInterface.qpOASES(H, h, A=A, lbA=lbA, ubA=ubA, p=[a, b])
>>> qp.exportCode('test1', dir='Test1_Exported', printsfun=True, mex=False,
TestCode=False, Options={'max_iter': 5})
(xGuess[2x1], a[1x1], b[1x1])->(xOpt_VAL[2x1], Obj_VAL[1x1])
Test1_Exported/test1.c
Test1_Exported/test1_EVAL_CODE.c
Test1_Exported/test1_EVAL_CODE_Call.c
Test1_Exported/BLASReplacement.cpp
Test1_Exported/Bounds.cpp
Test1_Exported/Constraints.cpp
Test1_Exported/Flipper.cpp
Test1_Exported/Indexlist.cpp
Test1_Exported/Matrices.cpp
Test1_Exported/MessageHandling.cpp
Test1_Exported/Options.cpp
Test1_Exported/QQPinterface.cpp
Test1_Exported/QProblem.cpp
Test1_Exported/QProblemB.cpp
Test1_Exported/SolutionAnalysis.cpp
Test1_Exported/SparseSolver.cpp
Test1_Exported/SQProblem.cpp
Test1_Exported/SQProblemSchur.cpp
Test1_Exported/SubjectTo.cpp
Test1_Exported/Utils.cpp
#include "Test1_Exported/test1.h"
#include "Test1_Exported/test1_EVAL_CODE.h"
#include "Test1_Exported/test1_EVAL_CODE_Call.h"
// Try this first:
test1_Call(xGuess, a, b, xOpt_VAL, Obj_VAL);
// If previous does not work, try this:
double *xGuess_TEMP = (double *)xGuess;
double *a_TEMP = (double *)a;
double *b_TEMP = (double *)b;
test1_Call(xGuess_TEMP, a_TEMP, b_TEMP, xOpt_VAL, Obj_VAL);

```

Running above code generates C/C++ code that can be integrated into other platform. Simulink example is given. Printed output gives name and location of C/C++ files, header files and function call.

### Methods

```

`exportCode(self, filename, dir='/', mex=False, printsfun=False, Options=None,
TestCode=False)`
: Method of class qpOASES to export the code that solves quadratic programming.

```

#### Parameters

-----

filename: (str)  
    Filename for exported code.  
dir: (str, optional)  
    Directory where code is to be exported. Defaults to '/'.  
mex: (bool, optional)  
    Whether mex interface is required. Defaults to False.  
printsfun: (bool, optional)  
    Whether MATLAB s-function interface is to be implemented. Defaults to False.  
Options: (dict, optional)  
    Options for qpOASES solver. Defaults to None.  
TestCode: (bool, optional)  
    Whether main function is required. Useful while testing and debugging. Defaults to False.

#### Returns

-----

None

``exportEvalCode(self, funcname, dir='/', options=None)``

: Method of class qpOASES to generate C code to evaluate H, h, A, lbA, ubA, lbx, ubx.

#### Parameters

-----

funcname: (str)  
    Function to be named that evaluates H, h, A, lbA, ubA, lbx and ubx.  
dir: (str, optional)  
    Directory where codes are to be exported. Defaults to '/'.  
options: (dict, optional)  
    Options for code generation. Same option as `casadi.Function.generate()` function. Defaults to None.

#### Returns

-----

None