green: call function
orange: create class instance
red: add codes for k

Start

S = **ControlSystem** (-> ControlSystem.__init__())

S.dyn_sys = **DynamicalSystem**(-> DynamicalSystem.__init__())

S.eqs = **CollocationSystem**(-> DynamicalSystem.__init__())

f = sys.f_sym(sp.symbols(x,u,par))
Df = f.Jacobi()

S.eqs._ff_vectorized = sym2num_vectorized(f,x,u,par)
S.eqs._Df_vectorized = sym2num_vectorized(Df,x,u,par)

S.eqs.trajectories = **Trajectory** (-> Trajectory.__init__())

S.eqs.trajectories._chain, *S.eqs.trajectories*._eqind = auxiliary.find_integrator_chains(sys)

S.set_param()

x, u, k = S.solve() — — *S.eqs.trajectories.x, S.eqs.trajectories.u, S.park[0]*

S._iterate()

S.eqs.trajectories.*init_splines()*

splines[xx/uu] = Spline (-> Spline.__init__())

splines[ss]._coeffs = sp.symarray('c' + ss,...)

splines[ss].make_steady()

make_steady()

(get:) splines._dep_array, *splines*.dep_array_abs, *splines*._indep_coeffs

S.indep_coeffs[ss] = splines[ss]._indep_coeffs
***S*.indep_coeffs['z_par'] = np.array([sp.symbols('k')])**
S.splines = splines

S.eqs.get_guess()

1st. Round: (if without initial value by user:) free_coeffs_all = np.hstack(*S.eqs*.trajectories.indep_coeffs.values())
guess = 0.1 * np.ones(free_coeffs_all.size)
(if with initial value by user:) free_coeffs_guess = *S.eqs*.trajectories.splines[ss].interpolate()
guess = np.hstack((guess, free_coeffs_guess))
guess[-1] = self._parameters['z_par']

from 2.nd round: guess = *S.eqs*.trajectories.splines[k].interpolate(s_old.f, ...), k: x and u
S.eqs.guess = np.hstack((guess, *S.eqs*.sol[-1]))

S.eqs.guess = guess

C =*S*.eqs.build()

Mx, Mx_abs, Mdx, Mdx_abs, Mu, Mu_abs, Mp, Mp_abs =
S.eqs._build_dependence_matrices(indic)

Return Mx, Mx_abs, Mdx, Mdx_abs, Mu, Mu_abs, Mp, Mp_abs

mx, mx_abs/ mdx, mdx_abs = x_fnc[xx].im_self.get_dependence_vectors(p, d=dorder_fx/d=dorder_dfx)
mu, mu_abs = u_fnc[uu].im_self.get_dependence_vectors(p, d=dorder_fu)
mp, mp_abs = self.get_dependence_vectors_p(p)

Mx[k, i:j] = mx, ...

DdX = Mdx[take_indices, :]
DXUP = np.vstack((Mx[n_states * i : n_states * (i+1)].toarray(), Mu[n_inputs * i : n_inputs * (i+1)].toarray(), Mp[1 * i : 1 * (i+1)].toarray())),
i =0: ncpts-1

ff_vec = *S.eqs*._ff_vectorized, Df_vec = *S.eqs*._Df_vectorized

sol, par = *S.eqs*.solve(C.G, C.DG)

Return S.eqs.sol, par

S.eqs.solver = Solver (-> Solver.__init__())

S.eqs.sol, par =
S.eqs.solver.solve()

Return S.eqs.solver.sol, S.eqs.solver..par

S.eqs.solver.leven()

x = self.x0 (x0 = S.eqs.guess)

Fx = *S.eqs.solver*.F(x)

Return G

X = Mx.dot(x)[:,None] + Mx_abs, U = Mu.dot(c)[:,None] + Mu_abs, P = Mp.dot(c)[:,None] + Mp_abs
F = ff_vec(X, U, P).ravel(order='F').take(take_indices, axis=0)[:,None]
dX = Mdx.dot(c)[:,None] + Mdx_abs

G = F - dX

DFx = *S.eqs.solver*.DF(x)

Return DG

DF_blocks = Df_vec(X,U,P).transpose([2,0,1])
DF_csr = sparse.block_diag(DF_blocks, format='csr').dot(DXUP)

DG = DF_csr - DdX

(Suppose: J = DFx, F = Fx, h = s) Levenberg Method:
R = || F(x)||²₂ - || F(x+h)||²₂ , R~ = || F(x)||²₂ - || F(x) + F´(x)*h||²₂, ρ = R / R~
μ_new = 2*μ_old (ρ <= 0.2); μ_new = μ_old (0.2 < ρ < 0.8); μ_new = 0.5*μ_old (ρ >= 0.8)
Solve h in equation: ($J^T J + \mu^2$) *h = - J^T * F

S.eqs.solver.sol = x+h, *S.eqs.solver*.par = np.array([sol[-1]])

S.eqs.trajectories.set_coeffs(sol)

S.eqs.trajectories.coeffs_sol[k] = sol_bak[i:j]

subs[k] = sol[:i]
S.eqs.trajectories.splines[k].set_coefficients(free_coeffs=subs[k])

S.simulate(par)

S = Simulator (-> Simulator.__init__)

S.solver = ode(S.rhs)

Return dx

dx = S.ff(x, u, pt)

S.sim_data = S.simulate(par)

Return [S.t, S.xt, S.ut]

t, y = *S*.calcStep()

Return t, x

x = list(*S*.solver.integrate(*S*.solver.t+*S*.dt))
t = round(*S*.solver.t, 5), S.t.append(t)

S.xt.append(x), *S*.ut.append(S.u(t))

***S*.sim_data.append(par)**

S.check_accuracy()

S.reached_accuracy = (maxH < ierr) and (max(err) < eps) and (par_k > 0.0)

NO

reached_accuracy
or
max_round

YES

return S.eqs.trajectories.x,
S.eqs.trajectories.u, S.park[0]

x, u, k = S.solve()