OpenKITE: User Manual

Petr Listov, Colin Jones

May 2, 2018

OpenKITE [Kite Identification Tracking & Estimation] is being developed at the Automatic Control laboratory EPFL as a part of the European Airborne Wind Energy Systems Optimization and Control (AWESCO). The program was originally aimed to test and validate optimal estimation and control algorithms for the indoor kite prototype built in the laboratory. It has is a number of routines for modeling, simulation as well as model-based nonlinear control, estimation and identification of rigid-wing kites. The software is aimed to reduce development time of both research-oriented and commercial AWE applications.

Modeling

OpenKITE [Kite Identification Tracking & Estimation] package contains implementation of several wind energy kite models:

- ✓ Rigid-wing single line kite [reference to future paper]
- ✓ Kinematic model ("tricycle on a sphere") [reference to Sanket]

Rigid-wing single line kite

The primary interest is the rigid-wing single-line kite model since it has very complex and highly-nonlinear dynamics. OpenKITE allows rather flexible configuration of the flying vehicle inertia and geometry, as well as some of the cable properties. For any particular kite the parametrization can be obtained using CAD and CFD analysis software. In our work we relied on the XFLR5 tool that is capable of performing aerodynamic and stability analysis of the wing for comparable low Reynolds numbers. The following parameters are necessary to specify for a kite simulation:

Geometry:

- b: wing span [m]
- c: mean aerodynamic chord (MAC)[m]
- AR: aspect ratio []
- S: wing surface area [m2]
- lam: taper ratio []
- St: horizontal tail surface area [m2]
- lt: tail level arm [m]
- Sf: fin surface area [m2]
- lf: fin level arm [m]
- Xac: aerodynamic centre [1/c]

Inertia:

- mass: [kg]
- Ixx: [kg * m2]
- Iyy: [kg * m2]

- Izz: [kg * m2]
- Ixz: [kg * m2]

Aerodynamics:

- aerodynamic:
- CL0: lift coefficient with zero angle of attack (AoA) []
- CLa_total: 5.483 total lift coefficient derivative with respect to AoA [1/rad]
- e_oswald: Oswald efficiency number []
- CD0_total: total zero lift drag coefficient []
- CYb: total side force coefficient [1/rad] (in BRF !!!)
- Cm0: zero AoA pitching moment
- Cma: pitching moment coefficient sensitivity wrt to AoA [1/rad]
- Cn0: zero yawing moment for symmetric aircrafts []
- Cnb: yawing moment sensitivity wrt to sideslip [1/rad]
- Cl0: rolling moment for symmetric aircrafts
- Clb: rolling moment sensitivity wrt to sideslip angle [1/rad]
- CLq: pitch-rate lift coefficient [1/rad]
- Cmq: pitch-rate moment coefficient [1/tad]
- CYr: yaw-rate side force effect []
- Cnr: yaw-rate yawing moment effect []
- Clr: yaw-rate rolling moment effect []
- CYp: roll-rate side force coefficient []
- Clp: roll-rate rolling moment coefficient []
- Cnp: roll-rate yawing moment coefficient []
- CLde: lift sensitivity to elevator deflection [1/rad]
- CYdr: side force sensitivity to rudder deflection [1/rad]
- Cmde: pitch moment sensitivity wrt to elevator deflection [1/rad]
- Cndr: yawing moment sensitivity wrt to rudder deflection [1/rad]
- Cldr: rolling moment sensitivity wrt to rudder deflection
- CDde: drag force sensitivity wrt elevator deflection

Tether:

- length: tether length [m]
- Ks: spring coefficient [N/m]
- Kd: damping coefficient [N m / s]
- rx: x-displacement of the tether mouting point wrt CoG [m]
- ry: y-displacement of the tether mouting point wrt CoG [m]
- rz: z-displacement of the tether mouting point wrt CoG [m]

Configuration file has YAML format, an example can be found in the "*data*" folder. There are two ways of using the modelling functionality. User may use the ROS integrated "black-box" simulator and access navigation data by subscribing to specific topic, or get direct access to the model through C++ API.

C++ API of the simulator

Listing (1) shows how the kite model should instantiated using C++ API. It is possible to obtain numerical and symbolic expressions of the right-hand side (RHS) ODE describing kite dynamics, Jacobian and Integrator, that can be used in estimator and controller design.

```
#include "openkite/kite.h"

/** load kite properties */
std::string kite_config_file = "path_to_config_file";
KiteProperties kite_props = kite_utils::LoadProperties(kite_config_file);

/** (optional) specify properties of the built-in integrator */
AlgorithmProperties algo_props;
algo_props.Integrator = RK4;

/** instantiate kite model */
KiteDynamics kite(kite_props, algo_props);

/** get RHS of the ode and Jacobian */
Function ode = kite.getNumericDynamics();
Function jac = kite.getNumericJacobian();
```

Listing 1: Instantiation of the kite model

It is possible then use on the pre-implemented ODE solvers to simulate the system, or to implement a custom integration method. Listing (3) demonstrates this functionality.

ROS kite simulator

Kite simulator can be accessed smiply be calling the command:

```
roslaunch openkite simulator.launch
```

User may edit the launch file to specify the initial point of the simulation:

```
<rosparam param="init_state"> [vx,vy,vz, wx,wy,wz, x,y,z, q,qx,qy,qz]</↩
    rosparam>
```

Configuration file describing the wing and tether properties:

```
<param name="kite_params" value="path_to_configuration_file" />
```

To setup 3D visualisation it is necessary to run RViz from ROS distribution prior to simulator:

```
rosrun rviz rviz
```

```
#include "openkite/kite."h
#include "openkite/integrator."h"

. . ".

KiteDynamics kite(kite_props, algo_props);
Function ode = kite.getNumericDynamics();

    /** compare three ode solvers */
Dict opts;
opts["tf"]         = 1.0;
opts["poly_order"] = 21;
opts["tol"]        = 1e-4;

/**Create solvers */
/** Runge-Kutte 4ord method without step size control */
opts["method"]     = IntType::RK4;
ODESolver rk4_solver(ode, opts);

/** BDF based integrator from CVODES package *
opts["method"] = IntType::CVODES;
ODESolver cvodes_solver(ode, opts);

/** Chebyshev collocation method: uses Newton method with line search */
opts["method"] = IntType::CHEBYCHEV;
ODESolver chebychev_solver(ode, opts);

/** Pseudospektral Chebyshev collocation: uses IPOPT to solve constraints */
double tf = 1.0;
PSODESolver<10,4,13,3>ps_solver(ode, tf);

/** solve a problem */
DM rk4_sol, cheb_sol, cv_sol, ps_sol;

DM init_state = DM::vertcat({vx, vy, vz, wx, wy, wz, x, y, z, q, qx, qy, qz});
DM control = DM::vertcat({throttle, elevator, rudder, aileron});

rk4_sol  = rk4_solver.solve(init_state, control, tf);
cv_sol   = cvodes_solver.solve(init_state, control, tf);
cheb_sol = chebychev_solver.solve(init_state, control, tf);
ps_sol = ps_solver.solve(init_state, control, FULL);
```

Listing 2: OpenKITE integrators

Published topics:

+ *kite_state* (sensor_msgs::MultiDOFJointState) : navigation data

Subscribed to topics:

+ *kite_control* (openkite::aircraft_controls) : control input

Parameters:

+ *simulation_rate* (double : default 50 Hz) : simulator sampling frequency

Transport delay

AWE systems are often subject to considerable delays due to imperfect communication link between a flying vehicle and ground station. The effect becomes more important if the control and estimation algorithms are running on the ground. The transport delay node is introduced to simulate these delays. Launch the node by:

```
rosrun openkite transport_delay
```

Published topics:

+ *delayed_control* (openkite::aircraft_controls) : delayed control input

Subscribed to topics:

+ *kite_control* (openkite::aircraft_controls) : control input

Parameters:

+ *delay* (double : default 20 ms) : transport delay
+ *deviation* (double : default 5 ms) : uncertainty about transport delay (modeled by uniform distribution)
+ *rate* (double : default 50 Hz) : publishing rate of the node

Other functionalities and scripts

The toolbox also includes several modules tailored to the specific small-scale prototype, and numerous scripts (Matlab and Pyhton) for telemetry and data logs analysis.

Path following controller

OpenKITE contains an implementation of a Path Following Nonlinear Model Predictive Controller (NMPFC) based on the pseudospectral discretization. An example of the controller design is described in the example below.

The control algorithm is currently under active development and will be released soon as an independent general real-time model predictive control and trajectory optimization tool.

```
#include "kiteNMPF.h"

/** create kite model */
KiteDynamics kite = KiteDynamics(kite_props, algo_props);

/** define geometric curve to follow */
SX x = SX::sym("x");
double radius   = 2.72;
double altitude = 2.00;
SX Path = SX::vertcat(SXVector{radius * cos(x), radius * sin(x), altitude});
Function path = Function("path", {x}, {Path});

/** instantiate controller */
KiteNMPF controller(kite, path);

/** set controller constraints */
DM lbu = DM::vertcat({T_min, El_min, Rud_min, Ail_min});
DM ubu = DM::vertcat({T_max, El_max, Rus_max, Ail_max});

controller.setLBU(lbu);
controller.setUBU(ubu);

/** set state constraints that controller should respect */
DM lbx = DM::vertcat({vx_min, vy_min, vz_min, wx_min, wy_min, wz_min, x_min, x_min, x_min,
                      qw_min, qx_min, qy_min, qz_min});

DM ubx = DM::vertcat({vx_max, vy_max, vz_max, wx_max, wy_max, wz_max, x_max, x_max, x_max,
                      qw_max, qx_max, qy_max, qz_max});

controller.setLBX(lbx);
controller.setUBX(ubx);

/** set reference velocity */
DM vel_ref = 7.0;
controller.setReferenceVelocity(vel_ref);

/** create NLP : necessary for constraints initialisation */
controller.createNLP();

. . .

/** compute control signal in the loop */
while(true)
{
    controller.computeControl(state_estimate);

    DM opt_traj = controller.getOptimalTrajetory();
    DM opt_ctrl = controller.getOptimalControl();
}
```

Listing 3: Controller design example

State estimation

OpenKITE contains the Extended Kalman Filter (EKF) approach for the kite state estimation. The implementation, however is tailored to the indoor setup at the moment where it's possible to measure the vehicle pose. The use is:

```
rosrun openkite ekf_node
```

Published topics:

+ *kite_state* (geometry_msgs::PoseStamped) : kite pose

Subscribed to topics:

+ *kite_pose* (sensor_msgs::MultiDOFJointState) : kite pose

+ *kite_control* (openkite::aircraft_controls) : control input

Parameters:

+ *kite_params* (string) : a path to file containing kite parameters

+ *rate* (double : default 50 Hz) : sampling rate of the node

The CasADi and ROS are rather heavy dependencies and are not often suitable for real-time embedded applications. Therefore, another framework was developed specifically targeting these limitations. The project YAKF (https://github.com/LA-EPFL/yakf) is suitable for the low-memory, real-time embedded applications and deals with a more general estimation formulation of the form:

$$\begin{aligned} \dot{x} &= f(x,u) + w,\ w \sim \mathcal{N}(0, \Sigma_w) \\ y &= h(x,u) + v,\ v \sim \mathcal{N}(0, \Sigma_v) \end{aligned} \tag{1}$$

The documentation can be found on the project's webpage.

Parameter identification

It is possible to solve an optimal nonlinear parameter estimation problem of the form:

$$\begin{aligned} J = &\int_{t_0}^{t_f} \|y(\tau) - h(x(\tau))\|_Q^2 \, d\tau \\ &\text{subject to: } \forall \tau \in [t_0, t_f] : \\ &\dot{x}(\tau) = f(x(\tau), u_k(\tau), \lambda),\ x(t_0) = x_0 \end{aligned} \tag{2}$$

using pseudopsetral discretization. An example can be found in *kite_identification_test.cpp* file.