K06 FPV array

Introduction

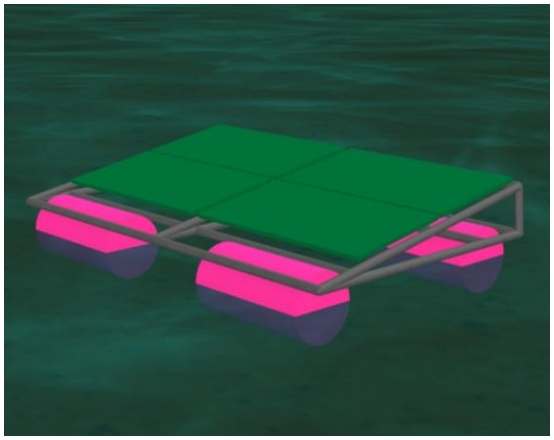
This example models a floating photovoltaic (FPV) array. Each raft is a rigid body that supports four solar panels, and there are 64 of these rafts in the array in an 8x8 modular rectangular configuration. The rafts are inter-connected using *constraint* objects. These make use of the double-sided connection feature each with one free rotational degree of freedom to model hinged connections between modules. When connected together, this forms a flexible structure on a larger scale.

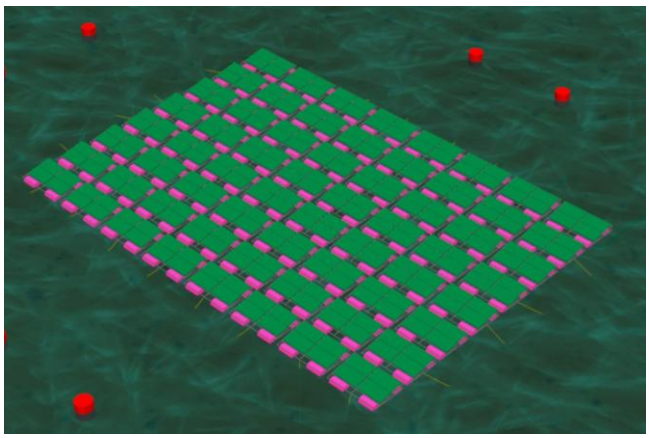
In addition to double-sided connections, this example shows how the OrcaFlex application programming Interface (OrcFxAPI) can be used to automate the array building process. Within this, we demonstrate how the `CreateClones` function can be used alongside `MoveObjects` to achieve the desired configuration. Whilst we include a Python script alongside this example, Python is not required to view or run the simulation. If you wish to recreate the model demonstrated here, the Python Interface: Installation help page provides further information on installing Python and the Python Interface to OrcaFlex.

Included alongside this document is:

- The OrcaFlex model *K06 FPV module.dat* containing a single modular raft,
- the Python script *K06 FPV script.py* used for the building of the array,
- a simulation file *K06 FPV array.sim* showing the dynamics of the array and mooring system under appropriate environmental conditions, and
- *K06 FPV results.wrk,* a workspace file which displays a few results.

Note: The values used in this example are arbitrary and it is always recommended that all model data is established and compared with real data you have available.





Building the model

We've chosen to model the type of FPV array made up of, what is referred to in DNVGL-RP-0584 (*Design, development and operation of FPV systems*) as, 'Modular Rafts'. These typically have a structural framework, to which the solar panels are attached, supported by separate buoyant members (which we have decided to call floats). This configuration allows the metal frame and connected panels to be kept clear of the water surface and allows both the frame and floats to be optimised for their respective functions.

The model containing a single modular raft *K06 FPV module.dat* was built through the OrcaFlex graphical user interface (GUI), using a combination of *lumped buoys*, *spar buoys*, and *line* objects to represent the rigid body. To simplify the model, we chose not to include floating maintenance walkways or any electrical components in our system.

Due to the array's modular design, our model for a single raft can be used as a source model from which we duplicate multiple rafts, before connecting these together to produce the array. We chose to automate this process using the OrcFxAPI and this is demonstrated in the Python script *K06 FPV script.py*. If you're unfamiliar with the Python interface to OrcaFlex, then an introduction can be found on the documentation page of the Orcina website. The OrcFxAPI help can also be accessed from the same page, or via the OrcaFlex Help menu.

Running the script produces an array and saves the new model as a .dat file. Whilst building a mooring system is also possible through the API, we chose to return to the GUI and make use of the full capabilities and visualisation strengths of OrcaFlex to aid us in the building and design process. We demonstrate the mooring system and array dynamics in *K06 FPV array.sim*.

We model the array dynamics under an example environment with wind, waves and current applied. Full details of the environmental conditions applied are presented below in the 'Environment' section. It's important to note that we only consider waves with a long enough wavelength such that diffraction effects of the elements in our raft are insignificant. This approach assumes that Morison's equation applies and we therefore proceed with modelling the raft using *6D buoy* and *line* objects. Alternatively, in cases where diffraction effects dominate, a *vessel object* could be used for a potential-flow theory approach. It is ultimately down to the user to choose which modelling approach to take.

Modelling a single raft

In our model *K06 FPV module.dat*, the floats are modelled using *6D spar buoys*, named *FloatA*, *FloatB*, *FloatC* and *FloatD,* situated at the four corners of the module. Each buoy is modelled as a stack of co-axial cylinders, as explained on the Modelling, data and results | 6D buoys | Spar buoy and towed fish properties help page. Surface piercing is considered on a per-cylinder basis and used to calculate the hydrodynamic loads for each individual cylinder, which are then summed to obtain the total load on the buoy. We are therefore required to define the *drag* and *added mass* data on a per-cylinder basis. It would be possible to account for shielding between adjacent floats by setting the drag areas accordingly, but this is an effect we chose to ignore. Ultimately, the more cylinders used, the more accurately the hydrodynamic loads on the spar buoys can be captured. For the floats, we chose to use six of these cylinders. This is explained in more detail in the Buoy discretisation technical note published on the Orcina website and further detailed guidance on modelling a surface-piercing buoy can be found on the relevant help page.

The metal truss structure is modelled using multiple line objects, connected using line-to-line connections with infinite *end connection stiffness*. The line type, *Steel tubular,* used for these lines makes use of the *homogeneous pipe* line type category. This is useful for modelling pipes

constructed from a single homogeneous material and means that the pipe's structural properties are calculated from a given *Young's modulus*, *material density* and the *inner* and *outer diameters*. These lines are named for the corners which they connect, i.e. the line between the corners with *FloatB* and *FloatD* is called *BD*.

When building a structure which includes direct line-to-line connections, the option to *include torsion* on the line data form must be enabled so that moments can be transferred through the structure. It is also important to note that a line (or indeed any object) can only be connected to another line at an *arc length* where there is a line node.

Another particularly useful feature on the line data form is the option to set a line's *length and end orientations* to be *calculated from end positions*. This allows a user to easily create a line between two points whose length is the distance between those points and whose *line end orientations* are automatically set. Whilst we don't explicitly set this data in our example here, further details on setting line end orientations can be found in this tutorial video, available from the OrcaFlex tutorials page of the Orcina website.

Additionally, we've set the *seabed friction policy* for all lines in the structure to be *none,* as we do not expect them to contact the seabed. We can check that this has been set for all lines in our structure through the all objects data form. This data form is especially useful for when you have many objects in a model and want to easily check for consistency or apply changes to multiple objects at once. You can find this data form in the *Shared data* folder when the *model browser* is in groups view. If you open this now and select *Show: Other data* on the top left, make sure *lines* are selected as the *object type*, and navigate to the *statics* tab, you will see the *seabed friction policy* data for all lines in the model.

To model the PV panels, we use *6D lumped buoys*. As we are modelling thin plates, we assume that drag is only generated in the *z* direction. We therefore zero the drag coefficients in the *x* and *y* directions and assign the buoys' properties in the *z* direction following guidance found on the Modelling, data and results | 6D buoys | Hydrodynamic properties of a rectangular box help page. For simplicity, we assume that our panels will remain out of the water throughout the simulation. This means that we can choose not to include the effects of hydrodynamic added mass and we have zeroed the *Ca* and *Cm* values accordingly. As the panels are in air, the defined drag data will instead be used when accounting for wind loads on 6D buoy objects. This is discussed further in the 'Environment' section below.

Connected to the metal frame, we have a *constraint* object called *PanelTilt*, which is currently hidden from view. We can show this by selecting *PanelTilt* in the model browser and pressing *Ctrl + H*. This is fixed in all degrees of freedom and is simply used as a reference point to which the four 6D buoys representing the PV panels are connected. In our model these panels have a fixed inclination angle which will not change throughout a simulation. However, this inclination angle can be changed in the *reset state* by editing the constraint data form. If we open the *PanelTilt* data form, you will see the *azimuth* is currently set to 190°. This corresponds to a 10° tilt angle, measured anticlockwise from the horizontal. If we increase this value to 195° (180° + 15° tilt angle) and click *OK*, we see that the panels are now mounted at an angle 15° from the horizontal. Since the *length and end orientations* (for lines whose length would be affected by this change) are set to be *calculated from end positions*, this change to the raft configuration can be made easily. Also connected to this constraint is a *label* type shape object which will be useful later for indexing the position of a raft within the array.

Finally, we have a second constraint in this model called *temporary constraint*. Opening the constraint's data form, you can see that the in-frame is fixed in the global environment and the

out-frame is connected to the raft. On the *degrees* of freedom tab, we can see that movement of the out-frame is fixed in the *x*, y and *Rz* directions, which means that the raft can't float away or yaw. This allows us to perform a static analysis without the mooring system in place, so that we can determine the height the raft sits above the sea surface and verify that the static solution converges as expected. This is a useful application of *double-sided constraints*, as connection data of the objects in the raft doesn't have to be changed to achieve this.

Array building using OrcFxAPI

The Python script supplied with this example, *K06 FPV script.py*, demonstrates one approach to using the API to automate the model building process. Here, we have decided to prioritise clarity and readability over efficiency, and there will be numerous alternative approaches which could achieve the same outcome. Whilst each person's code will inevitably vary, the API functions, attributes, and methods will likely be the same, and these are documented in the OrcFxAPI Help.

Below we will discuss two of the functions used in slightly more detail, namely `CreateClones` and `MoveObjects`.

`CreateClones()`, documented here, allows you to clone multiple objects and preserve the connections between them, all from within the API. This is a function of the model object class and is called using `model.CreateClones([obj1, obj2, ...], model=None)`, the first argument being a list of the objects you wish to clone. Alternatively, instead of a list of objects, you can pass a group in as this first parameter and `CreateClones` will clone the group and all the objects within it, including objects nested within sub-groups. The function also allows you to pass a model as the (optional) second argument i.e. if you want to clone objects across into a different model. We demonstrate these functionalities in our example script.

`MoveObjects()` is a function of the OrcFxAPI Module which allows you to move and rotate a number of selected objects in a way equivalent to the move selected objects wizard, accessed from the *model browser* in the GUI. `MoveObjects(specification, points)` takes two parameters. The first, `specification`, is an instance of MoveObjectSpecification and specifies how to move the points. The second, `points`, is a sequence of instances of MoveObjectPoint containing the points which are to be moved. We demonstrate how this works in our script, but further details can be found on the respective help pages linked above.

The first thing we do in our script is import the `OrcFxAPI` module. This provides us access to the OrcaFlex programming interface. Second, we activate the `EnableBooleanDataType` policy. By default, boolean data in OrcaFlex (such as a checkbox that you tick in the GUI) is treated in the API as text data with possible values `Yes` and `No`. By including the given line of code in our script, we instead allow these to be treated as Python `bool` data. We do this purely for convenience.

In our source model, *K06 FPV module.dat* we make use of object tags to define the names of the objects that we wish to connect our constraints to. This information is recorded on the *tags* page of the *general* data form, and we access this using the Python script. On the same *tags* page, we also define a reference object, `refObj`, and the prefix ("Float") of the float objects which will be used for the move operation. This allows us to move the four floats (to which all other objects are connected) and the reference object only. Using tags means that we don't have to hardcode this information into our script.

We rename the objects in the source model, to prevent any name clashing, and then move them to the desired position in the global environment. We then clone this module from the `sourceModel` into the `newModel` and then, if there is a module to the left create a row constraint, and if there is a module above create a column constraint. The modules are named with a row-

column suffix, e.g. *Raft3-4* is in row three and column four of the array. The connections between these modules are named such that the row constraint connecting *Raft3-3* and *Raft3-4* is named *3-3/4* and a column constraint between *Raft5-7* and *Raft 6-7* is called *5/6-7*.

We note here that double-sided connections are not only necessary to circumvent a circular dependency issue but are more easily added into the FPV system than single-sided connection constraints would be. This is because we only need to change the constraint's *in-frame* and *out-frame* connection data, whereas previously, trying to replicate the same system with a single-sided constraint would require connecting the constraint to a parent object, and then connecting the second object to that constraint. This would create a chain of connected objects that, when long, can be hard to keep track of.

Mooring System modelling

Now open *K06 FPV array.sim*. The FPV array is moored using a buoy-catenary system with 12 buoyancy modules around the perimeter of the array. These are connected to the array by two polyester rope bridle lines, named *MooringA* and *MooringB*. The buoys are anchored to the seabed by a third line, *MooringC*; a polyester rope with a section of chain near its anchored end. Using buoyancy modules in the mooring helps support the weight of the mooring chains and reduces the load on the perimeter rafts in the array.

The buoyancy modules are modelled as *6D spar buoys*, allowing us to model the geometry and hydrodynamics of these in more detail. Data for these buoys is set in a similar way as for the raft floats (described earlier), following the Modelling, data and results | 6D buoys | Modelling a surface-piercing buoy guidance.

The line type properties for the *Polyester rope* were generated by the *line type wizard*, using the *special category* 'Rope/wire', with 'Polyester (8-strand Multiplait)' selected as the *construction*. Similarly, the *Studlink chain* line type data was generated by selecting 'Chain' as the *special category* and then 'Studlink' as the *link type*. Further details about the line type wizard and these options are documented here: Modelling, data and results | Lines | Line type wizard. We've added a small non-zero bending stiffness to these line types to help with convergence.

Each of the mooring lines have 0.5 m long segments across their sections. This value was chosen arbitrarily in an attempt to capture behaviour like compression in the mooring lines whilst also keeping simulation run times relatively low. In general, if a line has too few segments, the line's modelled response may not represent the actual behaviour of the real system. However, if a line has many very short, segments, then this can lead to longer simulation run times. We would usually recommend carrying out line segmentation sensitivity checks to focus on achieving a balance between results accuracy and computational expense.

Environment

In this model, we chose to replicate a possible sheltered, "near-shore" environment and so the water *depth* has been set to 20m. In certain cases, it may be necessary to consider irregular waves as part of the design process. However, for the sake of demonstration, we chose to model design waves as they are often easier to set up and quicker to run. For our simulation, we use *dean stream* waves of *height* H=0.5m and *period* T=5s.

We have considered a *current speed* of 0.5 m/s and adjusted the *current options* to *ramped*. This means that the effects of the current are excluded from the *static analysis*, and the current is instead ramped to its full value during the *build-up period*. The *build-up period* is by default the first stage of a simulation (Stage 0) and is indicated by negative time i.e. any time before 0.0s is the

build-up period and any time after is the normal simulation period. Details on how ramping is applied can be found in Theory | Dynamic analysis | Ramping. Sometimes, as is the case with this model, we find that excluding the current from the static analysis helps to make convergence more robust.

We account for the effects of wind loading in our OrcaFlex model by choosing to *include wind loads on 6D buoys* on the *Wind* tab of the *Environment* data form. This is selected by default, but unlike the default wave data, we must set our own *wind speed* and *direction*. We chose a *constant wind type* which requires us to specify wind *speed* and *direction*, which we chose to be 10 m/s at a 175° heading. This wind direction is indicated in our model visually by a solid white arrow drawn on the *view axes*, but this can be adjusted on the Environment | Drawing page. We also decided to ramp the wind *from zero*, and the ramping occurs in the same way as above. Further information about wind modelling can be found on the Modelling, data and results | Environment | Wind data help page.

Results

Now open the workspace *K06 FPV results.wrk*.

Explanations of each of the available constraints results can be found on the Modelling, data and results | Constraints | Results help page. These results are coordinate-system dependent, and so they are referred to as being either relative to global (labelled with upper-case *X*, *Y*, *Z* or *GX*, *GY*, *GZ*) or object-relative (labelled with lower-case *x*, *y*, *z* or *Lx*, *Ly*, *Lz*). You can find further details about this on the Theory | Coordinate systems help page.

In this example, we focus on a few local results for the constraint between *Raft1-3* and *Raft1-4*, named *1-3/4*. Opening the constraint's data form, we can see that the constraint in-frame is connected to *BD1-3* (on *Raft1-3*) and the constraint out-frame is connected to *AC1-4* (on *Raft1-4*).

On the *degrees of freedom* tab, we can see that the constraint is free in *Ry* and fixed in all other degrees of freedom (DOF). This means that the out-frame can rotate freely about the constraint's in-frame *y-axis*.

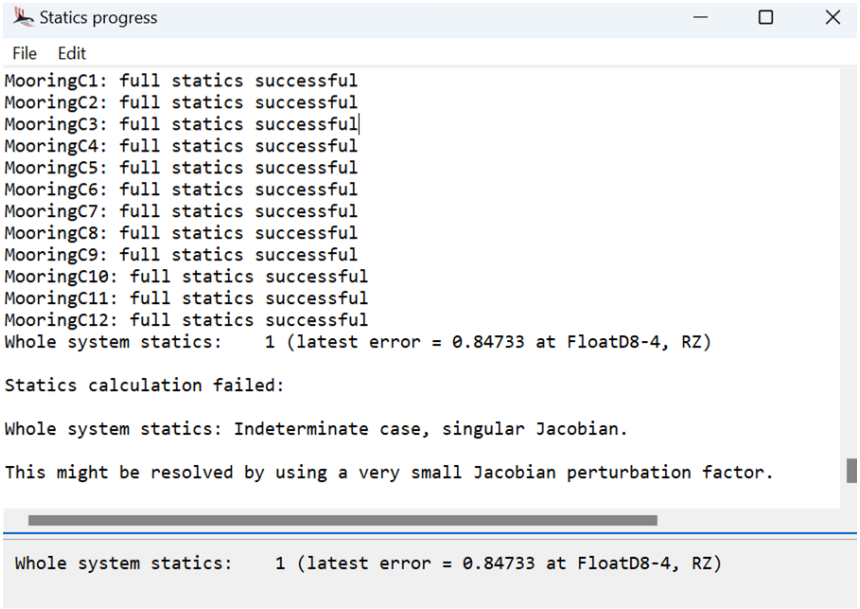
Let's first consider displacement and connection forces in the local x-direction, *Lx*. The *x* result gives the position of the *out-frame* with respect to the axes of the *in-frame* and so, because this DOF is fixed, this should be reported as 0 at all log sample times (this is not to be confused with *out-frame X* or *in-frame X* which give the position in global coordinates and will change throughout the simulation). As our constraints have *double-sided connections*, OrcaFlex makes use of the indirect solution method and we therefore see that these results fluctuate slightly around zero, but still remain within the tolerance of the analysis. Because the constraint is *fixed* in this DOF, a force is applied to the out-frame, by the in-frame, to hold the out-frame rigidly in position. This is the force required to hold the *Raft1-4* subsystem onto the *Raft1-3* subsystem. This is given by the *out-frame connection Lx force* result.

We can see the rotational movement in our nominated *free* DOF on the time history graph of *Ry*, which gives the orientation of the out-frame relative to the in-frame. We can see that this oscillates between ±2° throughout the simulation. The *out-frame connection Ly moment* would be the y component of the total moment applied to the out frame by the in-frame to maintain a rigid connection between the out and in frames. As our constraint has 0 *rotational stiffness* and the frames can move freely in this DOF, this moment should be 0. Again, due to the constraint's indirect solution method, we can see that there are some very small fluctuations about this value, but otherwise, this result is as expected.

The above explanation and observations apply to the results of all constraint objects. However, what is the difference when dealing with constraints that have a double-sided connection? Selecting the *in-frame connection force* gives us the magnitude of the force applied to the in-frame of the constraint *1-3/4* by the object to which the in-frame is connected, *BD1-3*. But what if we want to know the force applied to the out-frame of the constraint by the out-frame's parent object, *AC1-4*? In our case, the force on the out-frame by *AC1-4* is equal and in the opposite direction to the *out-frame connection force*. However, this is only the case because we have no *applied loads* or other *child objects* connected to the constraint in question. In general, the most reliable way to obtain this result would be to add a dummy constraint into the system, as suggested and further explained in the note near the bottom of the Constraints | Results help page.

Indeterminate systems

For this model, when performing our initial static analysis, the whole system statics calculation failed to converge due an 'indeterminate case, singular Jacobian' error. An indeterminate (or underdetermined) case occurs when a model typically has some coordinates, or some combination of coordinates, that add nothing to the physical system, but for which the OrcaFlex solver still has to find a solution. This results in a singular Jacobian matrix, which is explained further in Modelling, data and results | General data | Jacobians. This indeterminacy is reported by the *statics progress* window, as seen below.



In our case, the model was initially indeterminate due to the configuration of the mooring system. The mooring lines were connected to the axially symmetric cylindrical mooring buoys at the centre of their base, directly below the buoys' *centre of mass*. This meant that all directions that the buoy could rotate about its *local z-axis* were equally likely, and any rotation about this axis would not affect the forces it experiences. That's to say, the angle of rotation of the buoy is effectively redundant. There are therefore multiple valid solutions, and the OrcaFlex solver had no way of deciding which one to choose. This positional indeterminacy could be resolved by connecting the mooring lines to the buoy at a slight offset. Making this change solves our singular matrix issue and allows the static analysis to run successfully.

Instead of identifying the issue and resolving it directly, as we did above, adding a very small non-zero Jacobian perturbation factor (JPF) such as 10^{-18} would also allow the static analysis to run. We can do this through the *Jacobians* tab on the *general* data form. This would be perfectly fine, as we

do not care about the rotation of our mooring buoys, and simply want to look at results for our solar array. However, if we did want to extract the rotational orientation results of our mooring buoys, then we must be aware that this result will be just one of the many valid solutions. Note that using a non-zero JPF doesn't affect the accuracy of the solution, but instead indicates that the results may already be inaccurate because the solver doesn't have enough information. Adding a small non-zero JPF simply supplies a little extra information required to allow the model to run anyway.

The same applies to models with *indirect constraints*. These can often have indeterminacies that occur due to redundant degrees of freedom perceived by the solver, either in physical space or in its nondynamical Lagrange multipliers. It can often be tricky to identify which coordinates or which constraint degrees of freedom, are causing the system to be indeterminate, but this is necessary to move forward with the analysis. If you decide that these coordinates or degrees of freedom are not relevant to the results that you wish to extract, then adding a small non-zero JPF is a valid and recommended way to allow your model to run. However, if the redundant degrees of freedom will affect the results that you're reporting, then these results may not be the only valid solution – and therefore may be perceived as inaccurate when comparing with a real-world system that has no such indeterminacies. There is a more in-depth discussion of this in terms of connection load ambiguities in the Modelling, data and results | Constraints | Solution method help page.

In our example, we have no such issues, and our model runs with the JPF at its default value of 0. Whilst this can often be the case, it is nevertheless important to be aware of common issues surrounding indeterminant connection loads with *indirect constraints*.