CMPLXFOIL

MDO Lab

Oct 06, 2022

CONTENTS

CMPLXFOIL is a version of Mark Drela's XFOIL code with the GUI features removed. Gradient computation is implemented with the complex-step method. The Python interface is designed to be used with MACH-Aero tools for optimization. This package also includes some postprocessing tools.

CHAPTER

ONE

INSTALLATION

CMPLXFOIL is a Python wrapper for the Fortran-based XFOIL code. Building and installing the code is a multi-step process that is machine specific. This guide goes through the required steps to compile and install CMPLXFOIL.

1.1 Requirements

To compile the required XFOIL core components, a Fortran and a C compiler must be installed on your system. This can be GNU / GFortran or Intel, default configuration files are packaged for both types of compilers.

In addition to standard compilers, CMPLXFOIL requires the following dependencies:

Package	Version	Notes
Python	3.X.X	
NumPy	—	`conda install numpy` or `pip install numpy`
baseclasses	—	`pip install mdolab-baseclasses`
pyGeo	—	optional; required for `getTriangulatedMeshSurface` method
matplotlib	—	optional; required for `plotAirfoil` method (`pip install matplotlib`)
niceplots	—	optional; recommended for `plotAirfoil` method (`pip install niceplots`)

1.2 Build and Installation

Building CMPLXFOIL is handeled automatically by a set of Makefiles which are distributed with the code. These Makefiles require configuration files which specify machine-specific parameters, such as compiler locations and flags. Default configuration files for Linux GCC and Linux Intel are included in the `config/defaults` directory. Copy a configuration file to the main `config/` folder using the command below and modify its contents for your system and installation.

```
$ cp config/defaults/config.<version>.mk config/config.mk
```

Once the configuration file is adjusted as needed, CMPLXFOIL can be built by running `make` in the root directory:

```
$ make
```

This will compile both the real and complex versions of CMPLXFOIL, generating Python libraries which reference the XFOIL Fortran modules. These will be automatically copied to the `cmplxfoil/` directory.

Once the Python libraries are generated, install CMPLXFOIL by running pip install in the root directory:

```
$ pip install .
```

1.3 Verification

Tests are located in the `tests/` directory and can be run with the command:

```
$ testflo -v .
```

CHAPTER

TWO

OPTIMIZATION TUTORIAL

2.1 Introduction

This section describes a sample run script for airfoil optimization with CMPLXFOIL. It is very similar to the MACH-Aero single point airfoil tutorial. This example uses pyOptSparse's SLSQP optimizer because it comes with pyOptSparse, but SNOPT is recommended for more robustness, speed, and tunability.

The optimization problem solved in this script is

minimize

C_D

with respect to

4 upper surface shape variables (CST coefficients)

4 lower surface shape variables (CST coefficients)

subject to

$C_L = 0.5$

$V \geq 0.85V_0$

$R_{LE} \geq 0.85R_{LE,0}$

$t \geq 0.25t_0$

First upper surface CST coefficient = first lower surface CST coefficient

2.2 Getting set up

In addition to the required CMPLXFOIL packages, this script requires pygeo, multipoint, pyoptsparse, and mpi4py. The script can be found in CMPLXFOIL/examples/single_point.py.

2.3 Dissecting the optimization script

2.3.1 Import libraries

```
import os
import numpy as np
from mpi4py import MPI
from baseclasses import AeroProblem
```

(continues on next page)

(continued from previous page)

```
from pygeo import DVConstraints, DVGeometryCST
from pyoptsparse import Optimization, OPT
from multipoint import multiPointSparse
from cmplxfoil import CMPLXFOIL, AnimateAirfoilOpt
```

2.3.2 Specifying parameters for the optimization

These parameters define the flight condition and initial angle of attack for the optimization.

```
mycl = 0.5  # lift coefficient constraint
alpha = 0.0 if mycl == 0.0 else 1.0  # initial angle of attack (zero if the target cl is␣
↪zero)
mach = 0.1  # Mach number
Re = 1e6  # Reynolds number
T = 288.15  # 1976 US Standard Atmosphere temperature @ sea level (K)
```

2.3.3 Creating processor sets

Allocating sets of processors for different analyses can be helpful for multiple design points, but this is a single point optimization, so only one point is added.

```
MP = multiPointSparse(MPI.COMM_WORLD)
MP.addProcessorSet("cruise", nMembers=1, memberSizes=MPI.COMM_WORLD.size)
MP.createCommunicators()
```

2.3.4 Creating output directory

This section creates a directory in the run script's directory in which to save files from the optimization.

```
curDir = os.path.abspath(os.path.dirname(__file__))
outputDir = os.path.join(curDir, "output")

if not os.path.exists(outputDir):
    os.mkdir(outputDir)
```

2.3.5 CMPLXFOIL solver setup

The options tell the solver to write out chordwise aerodynamic data (`writeSliceFile`) and the airfoil coordinates (`writeCoordinates`) every time it is called. It also enables live plotting during the optimization (`plotAirfoil`). Finally, it specifies the output directory to save these files.

```
aeroOptions = {
    "writeSolution": True,
    "writeSliceFile": True,
    "writeCoordinates": True,
    "plotAirfoil": True,
```

(continues on next page)

(continued from previous page)

```
    "outputDirectory": outputDir,
}

# Create solver
CFDSolver = CMPLXFOIL(os.path.join(curDir, "naca0012.dat"), options=aeroOptions)
```

Other options allow the user to adjust the maximum iterations allowed to the XFOIL solver and to specify a location at which to trip the boundary layer.

2.3.6 Set the AeroProblem

We add angle of attack as a design variable (if the target lift coefficient is not zero) and set up the AeroProblem using given flow conditions.

```
ap = AeroProblem(
    name="fc",
    alpha=alpha if mycl != 0.0 else 0.0,
    mach=mach,
    reynolds=Re,
    reynoldsLength=1.0,
    T=T,
    areaRef=1.0,
    chordRef=1.0,
    evalFuncs=["cl", "cd"],
)
# Add angle of attack variable
if mycl != 0.0:
    ap.addDV("alpha", value=ap.alpha, lower=-10.0, upper=10.0, scale=1.0)
```

2.3.7 Geometric parametrization

This examples uses a class-shape transformation (CST) airfoil parameterization because it requires no additional files or other setup. Four CST parameters are added to the upper and lower surface (the class shape and chord length are other possible design variables through DVGeometryCST). The DVGeometryCST instance will set the initial design variables values by fitting them to the input dat file's geometry.

```
nCoeff = 4  # number of CST coefficients on each surface
DVGeo = DVGeometryCST(os.path.join(curDir, "naca0012.dat"), numCST=nCoeff)

DVGeo.addDV("upper_shape", dvType="upper", lowerBound=-0.1, upperBound=0.5)
DVGeo.addDV("lower_shape", dvType="lower", lowerBound=-0.5, upperBound=0.1)

# Add DVGeo object to CFD solver
CFDSolver.setDVGeo(DVGeo)
```

2.3.8 Geometric constraints

In this section, we add volume, thickness, and leading edge radius constraints. They are chosen to achieve practical airfoils and to guide the optimizer away from infeasible design, such as the upper and lower surfaces crossing over each other.

```
DVCon = DVConstraints()
DVCon.setDVGeo(DVGeo)
DVCon.setSurface(CFDSolver.getTriangulatedMeshSurface())

# Thickness, volume, and leading edge radius constraints
le = 0.0001
wingtipSpacing = 0.1
leList = [[le, 0, wingtipSpacing], [le, 0, 1.0 - wingtipSpacing]]
teList = [[1.0 - le, 0, wingtipSpacing], [1.0 - le, 0, 1.0 - wingtipSpacing]]
DVCon.addVolumeConstraint(leList, teList, 2, 100, lower=0.85, scaled=True)
DVCon.addThicknessConstraints2D(leList, teList, 2, 100, lower=0.25, scaled=True)
le = 0.01
leList = [[le, 0, wingtipSpacing], [le, 0, 1.0 - wingtipSpacing]]
DVCon.addLERadiusConstraints(leList, 2, axis=[0, 1, 0], chordDir=[-1, 0, 0], lower=0.85,␣
↪scaled=True)

fileName = os.path.join(outputDir, "constraints.dat")
DVCon.writeTecplot(fileName)
```

2.3.9 Optimization callback functions

This section defines callback functions that are used by the optimizer to get objective, constraint, and derivative information. See the MACH-Aero aerodynamic optimization tutorial for more information.

```
def cruiseFuncs(x):
    print(x)
    # Set design vars
    DVGeo.setDesignVars(x)
    ap.setDesignVars(x)
    # Run CFD
    CFDSolver(ap)
    # Evaluate functions
    funcs = {}
    DVCon.evalFunctions(funcs)
    CFDSolver.evalFunctions(ap, funcs)
    CFDSolver.checkSolutionFailure(ap, funcs)
    if MPI.COMM_WORLD.rank == 0:
        print("functions:")
        for key, val in funcs.items():
            if key == "DVCon1_thickness_constraints_0":
                continue
            print(f"    {key}: {val}")
    return funcs


def cruiseFuncsSens(x, funcs):
```

(continues on next page)

(continued from previous page)

```
    funcsSens = {}
    DVCon.evalFunctionsSens(funcsSens)
    CFDSolver.evalFunctionsSens(ap, funcsSens)
    CFDSolver.checkAdjointFailure(ap, funcsSens)
    print("function sensitivities:")
    evalFunc = ["fc_cd", "fc_cl", "fail"]
    for var in evalFunc:
        print(f"    {var}: {funcsSens[var]}")
    return funcsSens


def objCon(funcs, printOK):
    # Assemble the objective and any additional constraints:
    funcs["obj"] = funcs[ap["cd"]]
    funcs["cl_con_" + ap.name] = funcs[ap["cl"]] - mycl
    if printOK:
        print("funcs in obj:", funcs)
    return funcs
```

2.3.10 Optimization problem

This section sets up the optimization problem by adding the necessary design variables and constraints. An additional constraint for this problem enforces that the first upper and lower surface CST coefficients are equal. This is to maintain continuity on the leading edge. It also prints out some useful information about the optimization problem setup. See the MACH-Aero aerodynamic optimization tutorial for more information.

```
# Create optimization problem
optProb = Optimization("opt", MP.obj)

# Add objective
optProb.addObj("obj", scale=1e4)

# Add variables from the AeroProblem
ap.addVariablesPyOpt(optProb)

# Add DVGeo variables
DVGeo.addVariablesPyOpt(optProb)

# Add constraints
DVCon.addConstraintsPyOpt(optProb)

# Add cl constraint
optProb.addCon("cl_con_" + ap.name, lower=0.0, upper=0.0, scale=1.0)

# Enforce first upper and lower CST coefficients to add to zero
# to maintain continuity at the leading edge
jac = np.zeros((1, nCoeff), dtype=float)
jac[0, 0] = 1.0
optProb.addCon(
```

(continues on next page)

(continued from previous page)

```
    "first_cst_coeff_match",
    lower=0.0,
    upper=0.0,
    linear=True,
    wrt=["upper_shape", "lower_shape"],
    jac={"upper_shape": jac, "lower_shape": jac},
)

# The MP object needs the 'obj' and 'sens' function for each proc set,
# the optimization problem and what the objcon function is:
MP.setProcSetObjFunc("cruise", cruiseFuncs)
MP.setProcSetSensFunc("cruise", cruiseFuncsSens)
MP.setObjCon(objCon)
MP.setOptProb(optProb)
optProb.printSparsity()
optProb.getDVConIndex()
```

2.3.11 Run optimization

Run the optimization using pyOptSparse's SLSQP optimizer and print the solution.

```
# Run optimization
optOptions = {"IFILE": os.path.join(outputDir, "SLSQP.out")}
opt = OPT("SLSQP", options=optOptions)
sol = opt(optProb, MP.sens, storeHistory=os.path.join(outputDir, "opt.hst"))
if MPI.COMM_WORLD.rank == 0:
    print(sol)
```

2.3.12 Postprocessing

Finally, we save the final figure and use the built-in animation utility to create an optimization movie.

```
# Save the final figure
CFDSolver.airfoilAxs[1].legend(["Original", "Optimized"], labelcolor="linecolor")
CFDSolver.airfoilFig.savefig(os.path.join(outputDir, "OptFoil.pdf"))

# Animate the optimization
AnimateAirfoilOpt(outputDir, "fc").animate(
    outputFileName=os.path.join(outputDir, "OptFoil"), fps=10, dpi=300, extra_args=["-
↪vcodec", "libx264"]
)
```

2.4 Run it yourself!

To run the script, use the following command:

```
$ python single_point.py
```

In the output directory, it should create the following animation after the optimization completes:

CHAPTER

THREE

OPTIONS

maxIters: int = 100

The maximum iterations for XFOIL solver. By default 100.

writeCoordinates: bool = True

If True, it will write airfoil coordinates to dat file when `writeSolution` is called. By default True.

writeSliceFile: bool = True

If True, it will save chordwise data in a pickle file when `writeSolution` is called. By default True.

writeSolution: bool = False

If True, it will call `writeSolution` when the solver is called with an AeroProblem. By default False).

plotAirfoil: bool = False

If True, it will show airfoil plot with cp and cf data when `writeSolution` is called. By default False.

outputDirectory: str = .

The directory to save the output files. By default the current directory.

numberSolutions: bool = True

If True, will add call counter to output file names. By default True.

xTrip: ndarray = numpy.full

The boundary layer trip location specified as a two-element numpy array where the first element is the chordwise location at which to trip the upper surface and the second is the chordwise location at which to trip the lower surface. By default will not trip the boundary layer and instead use XFOIL's transition model.

CHAPTER

FOUR

CMPLXFOIL API

4.1 CMPLXFOIL

class cmplxfoil.**CMPLXFOIL**(**args*, ***kwargs*)

CMPLXFOIL Class Initialization

Parameters

fileName
[str] Filename of DAT file to read in

options
[dict of option-value pairs, optional] Options for the solver. Available options can be found in the Options section of the documentation or the options.yaml file in the docs directory.

debug
[bool, optional] Set this flag to true when debugging with a symbolic debugger. The MExt module deletes the copied .so file when not required which causes issues debugging, by default False

checkAdjointFailure(*aeroProblem*, *funcsSens*)

Pass through to checkSolutionFailure to maintain the same interface as ADflow.

This checks if the primal solve fails and can be called when the sensitivity is being evaluated (either through FD or CS).

Parameters

aeroProblem
[pyAero_problem class] The aerodynamic problem to to get the solution for

funcsSens
[dict] Dictionary into which the functions are saved.

checkSolutionFailure(*aeroProblem*, *funcs*)

Take in a an aeroProblem and check for failure.

Then append the fail flag in funcs. Information regarding whether or not the last analysis with the aeroProblem was sucessful is included. This information is included as "funcs['fail']". If the 'fail' entry already exits in the dictionary the following operation is performed:

funcs['fail'] = funcs['fail'] or <did this problem fail>

In other words, if any one problem fails, the funcs['fail'] entry will be True. This information can then be used directly in multiPointSparse. For direct interface with pyOptSparse the fail flag needs to be returned separately from the funcs.

Parameters

aeroProblem
[pyAero_problem class] The aerodynamic problem to get the solution for

funcs
[dict] Dictionary into which the functions are saved.

`computeJacobianVectorProductFwd`(*xDvDot=None*, *xSDot=None*, *mode='CS'*, *h=None*)

This the main Python gateway for producing forward mode jacobian vector products. They are computed using either the complex step or finite difference method. This function is not generally called by the user but rather internally or from another solver. A DVGeo object must be set for this routine.

Parameters

xDvDot
[dict] Perturbation on the design variables

xSDot
[numpy array] Perturbation on the surface

mode
[str ["FD" or "CS"]] Specifies how the jacobian vector products will be computed

h
[float] Step sized used when the mode is "FD" or "CS" (must be complex if mode = "CS"), by default 1e-6 for FD and 1e-200j for CS

Returns

dict
Jacobian vector product of evalFuncs given perturbation

`evalFunctions`(*aeroProblem*, *funcs*, *evalFuncs=None*, *ignoreMissing=False*)

This is the main routine for returning useful information from CMPLXFOIL. The functions corresponding to the strings in `evalFuncs` are evaluated and updated into the provided dictionary.

Parameters

aeroProblem
[`AeroProblem` instance] Aero problem from which to pull evalFuncs and flight conditions.

funcs
[dict] Dictionary into which the functions are saved.

evalFuncs
[iterable object containing strings] If not none, use these functions to evaluate.

ignoreMissing
[bool] Flag to supress checking for a valid function. Please use this option with caution.

`evalFunctionsSens`(*aeroProblem*, *funcsSens*, *evalFuncs=None*, *mode='CS'*, *h=None*)

Evaluate the sensitivity of the desired functions given in iterable object, 'evalFuncs' and add them to the dictionary 'funcSens'. The keys in the 'funcsSens' dictionary will be have an `<ap.name>_` prepended to them.

Parameters

funcsSens
[dict] Dictionary into which the function derivatives are saved

evalFuncs
[iterable object containing strings] The additional functions the user wants returned that are not already defined in the aeroProblem

mode
[str ["FD" or "CS"]] Specifies how the jacobian vector products will be computed

h
[float] Step sized used when the mode is "FD" or "CS" (must be complex if mode = "CS")

`getCoordinates()`

Return the current airfoil coordinates

Returns

coords
[ndarray] Airfoil coordinates with each column being (x, y, z) where z is a dummy value.

`getTriangulatedMeshSurface(offsetDist=1.0)`

This function returns a pyGeo surface. The intent is to use this for DVConstraints.

Note: This method requires the pyGeo library

Parameters

offsetDist
[float] Distance to extrude airfoil (same units as airfoil coordinates)

Returns

pyGeo surface
Extruded airfoil surface

`plotAirfoil(fileName=None, showPlot=True)`

Plots the current airfoil and returns the figure.

Parameters

fileName
[str, optional] FileName to save to, if none specified it will show the plot with plt.show()

showPlot
[bool, optional] Pop open the plot, by default True

Returns

matplotlib figure
Figure with airfoil plotted to it

list of matplotlib axes
List of matplotlib axes for CP and airfoil plots (in that order)

`setAeroProblem(aeroProblem)`

Sets the aeroProblem to by used by CMPLXFOIL.

Parameters

aeroProblem
[`AeroProblem` instance] Aero problem to set (gives flight conditions)

`setCoordinates`(*coords*)

Update the airfoil coordinates and associated point sets.

Parameters

coords
[ndarray] New airfoil coordinates (either 2 or 3 columns)

`setCoordinatesComplex`(*coords*)

Update the complex airfoil coordinates and associated point sets.

Parameters

coords
[ndarray] New airfoil coordinates (either 2 or 3 columns)

`setDVGeo`(*DVGeo*, *pointSetKwargs=None*)

Set the DVGeometry object that will manipulate 'geometry' in this object. Note that CMPLXFOIL does not **strictly** need a DVGeometry object, but if optimization with geometric changes is desired, then it is required.

Parameters

DVGeo
[A DVGeometry object.] Object responsible for manipulating the constraints that this object is responsible for.

pointSetKwargs
[dict] Keyword arguments to be passed to the DVGeo addPointSet call. Useful for DVGeometryMulti, specifying FFD projection tolerances, etc

`solveCL`(*aeroProblem*, *CLStar*, *alpha0=None*, *alphaBound=None*, *delta=0.5*, *tol=0.001*, *CLalphaGuess=None*, *maxIter=20*, *useNewton=False*)

Find the angle of attack that gives a target lift coefficient.

Parameters

aeroProblem
[pyAero_problem class] The aerodynamic problem to solve

CLStar
[float] The desired CL

alpha0
[float, optional] Initial guess for secant search (deg). If None, use the value in the aeroProblem, by default None

alphaBound
[float, tuple, list, optional] Bounds for angle of attack, if scalar then value is treated as a +- bound, by default None, in which case limit is +/-15 deg

delta
[float, optional] Initial step direction for secant search, by default 0.5

tol
[float, optional] Desired tolerance for CL, by default 1e-3

CLalphaGuess
[float, optional] The user can provide an estimate for the lift curve slope in order to accelerate convergence. If the user supplies a value to this option, it will not use the delta value anymore to select the angle of attack of the second run. The value should be in 1/deg., by default None

maxIter
[int, optional] Maximum number of iterations, by default 20

useNewton
[bool, optional] If True, Newton's method will be used where the dCL/dAlpha is computed using complex-step, otherwise the secant method is used, by default False

Returns

None, but the correct alpha is stored in the aeroProblem

`writeCoordinates`(*fileName*)

Write dat file with the current coordinates.

Parameters

fileName
[str] File name for saved dat file (".dat" will be automatically appended).

`writeSlice`(*fileName*)

Write pickle file containing the sliceData dictionary. The data can be accessed using the AeroProblem name as the key. Within that is a dictionary containing

- Pressure coefficient data on the upper surface
  - `"cp_visc_upper"`: viscous CP on the airfoil's upper surface
  - `"cp_invisc_upper"`: inviscid CP on the airfoil's upper surface
  - `"x_upper"`: x coordinates of the upper surface CP data
  - `"y_upper"`: y coordinates of the upper surface CP data
- Pressure coefficient data on the lower surface
  - `"cp_visc_lower"`: viscous CP on the airfoil's lower surface
  - `"cp_invisc_lower"`: inviscid CP on the airfoil's lower surface
  - `"x_lower"`: x coordinates of the lower surface CP data
  - `"y_lower"`: y coordinates of the lower surface CP data
- Skin friction coefficient data on the upper surface
  - `"cf_upper"`: skin friction coefficient on the upper surface
  - `"x_cf_upper"`: x coordinates of upper surface skin friction coefficient
  - `"y_cf_upper"`: y coordinates of upper surface skin friction coefficient
- Skin friction coefficient data on the lower surface
  - `"cf_lower"`: skin friction coefficient on the lower surface
  - `"x_cf_lower"`: x coordinates of lower surface skin friction coefficient
  - `"y_cf_lower"`: y coordinates of lower surface skin friction coefficient

Parameters

fileName
[str] File name for saved pkl file (".pkl" will be automatically appended).

writeSolution(*outputDir=None*, *baseName=None*, *number=None*)

This is a generic shell function that potentially writes the various output files. The intent is that the user or calling program can call this file and CMPLXFOIL write all the files that the user has defined. It is recommended that this function is used along with the associated logical flags in the options to determine the desired writing procedure.

Parameters

outputDir
[str] Use the supplied output directory

baseName
[str] Use this supplied string for the base filename. Typically only used from an external solver.

number
[int] Use the user supplied number to index solution. Again, only typically used from an external solver.

4.2 AnimateAirfoilOpt

class cmplxfoil.**AnimateAirfoilOpt**(*dirName*, *APName*)

Class for generating animations of airfoil optimization

Initialize the object with the directory and AeroProblem name. This object assumes that the files are accessible under the name <dirName>/<APName>_<iteration number>.<dat or pkl> and that BOTH dat (airfoil shape) and pkl (chordwise data) files are available.

Parameters

dirName
[str] Name of directory that contains the airfoil optimization data files

APName
[str] Name of the AeroProblem to be animated.

animate(*outputFileName='airfoil_opt'*, *ext='mp4'*, ***animKwargs*)

Generate an animation of an optimization.

Parameters

outputFileName
[str, optional] Movie filename to save to with no extension (default "airfoil_opt")

ext
[str, optional] Extension for animation ("mp4" and "gif" are useful ones)

animKwargs
[optional] Additional keyword arguments to be passed to matplotlib's FuncAnimation save method

CHAPTER

FIVE

CITATION

If you use CMPLXFOIL, please cite the following paper for XFOIL:

- Drela M. (1989) XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils. In: Mueller T.J. (eds) Low Reynolds Number Aerodynamics. Lecture Notes in Engineering, vol 54. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-84010-4_1

```
@InProceedings{XFOIL,
    author    = Mark Drela,
    editor    = Thomas J. Mueller,
    title     = {XFOIL}: {A}n {A}nalysis and {D}esign {S}ystem for {L}ow {R}eynolds {N}
↪umber {A}irfoils,
    booktitle = {L}ow {R}eynolds {N}umber {A}erodynamics,
    year      = 1989,
    publisher = {S}pringer {B}erlin {H}eidelberg,
    address   = {Berlin, Heidelberg},
    pages     = 1--12,
    isbn      = 978-3-642-84010-4,
    doi       = 10.1007/978-3-642-84010-4_1
}
```

CHAPTER

SIX

DISCLAIMER

XFOIL's derivatives are known to be unreliable due to its transition model. Because this is a result of the numerical simulation approach, using a complex step derivative computation method still faces these problems.

Adler, Gray, and Martins describe this phenomenon in their paper entitled *To CFD or not to CFD? Comparing RANS and viscous panel methods for airfoil shape optimization*. They observe that these inaccurate derivatives can create local optima. To visualize the behavior, they evaluate drag along a line in the design space between two optima. The result is shown below. Every dip corresponds to the transition location moving across one panel. Increasing the number of panels helps (CMPLXFOIL uses 284 panels by default), but does not fix the problem.

INDEX