CoSApp: a Python library to create, simulate and design complex systems.

Étienne Lac[1¶], Guy De Spiegeleer[2], Adrien Delsalle[2], Frédéric Collonval[3], Duc-Trung Lê[4], and Mathias Malandain[5]

1 Safran Tech, Digital Sciences & Technologies Department, France **2** twiinIT, France **3** WebSciT, France **4** QuantStack, France **5** Inria centre at Rennes University, France ¶ Corresponding author

DOI: 10.21105/joss.06292

Software

- Review
- Repository
- Archive

Editor: Rohit Goswami

Reviewers:

- @jbussemaker
- @chenxinye

Submitted: 23 January 2024
Published: 29 February 2024

Summary

CoSApp, for *Collaborative System Approach*, is an object-oriented Python framework that allows domain experts and system architects to create, assemble, simulate and design complex systems. The API of CoSApp is focused on simplicity and explicit declaration of design problems. Special attention is given to modularity; a very flexible mechanism of solver assembly allows users to construct complex, customized simulation workflows. CoSApp handles steady-state simulation, as well as time-dependent dynamic systems, and multimode systems with event-based mode transitions.

Statement of need

The design of industrial products is usually a complex process, involving experts from multiple disciplines, and the interaction of many different components. Multidisciplinary Design Analysis and Optimization (MDAO) plays a crucial role in this process, by accounting for strong coupling between various components at early design stages, where the only way to assess the final product is to rely on physical simulation models. In this context, flexibility, rather than sheer performance, is key to assessing the value of new concepts and guiding design choices. Moreover, a clear separation between simulation models and resolution tools (solvers, optimizers, etc.) must be enforced to enable agile design processes.

CoSApp addresses these needs by providing a user-friendly and comprehensive framework for creating both stand-alone and composite computational models, and solving mathematical problems based on specific design criteria. In particular, constraints can be added interactively in CoSApp workflows, which offers several advantages:

1. Problem complexity can be increased gradually, to help global convergence by starting from a physically sound state of the system.
2. Engineering practices (referred to as *design methods* in CoSApp) can be coded within domain-specific models as a collection of algebraic problems that can be selectively activated in the context of a broader system.

Thanks to its inherent agility, CoSApp caters to domain experts involved in model development, as well as system architects focused on designing high-level assemblies that model systems of interest.

State of the field

There exist many MDAO frameworks. Among open-source, general-purpose libraries, three stand out in particular: OpenMDAO (Gray et al., 2019), GEMSEO (Gallard et al., 2018) and OpenModelica (Fritzson et al., 2020).

OpenMDAO and GEMSEO are Python frameworks focused on optimization. They adopt a *causal* approach, meaning that each model (referred to as *component* in OpenMDAO, and a *discipline* in GEMSEO) has a predefined input/output interface associated with a transfer function representing causality. Both packages offer a large choice of numerical methods to solve and optimize multidisciplinary systems. Noticeably, they can take advantage of system-level gradients, when available (either coded inside the models or computed by automatic differentiation), which allows them to handle large numbers of unknowns (tens of thousands). At present, CoSApp evaluates gradients using finite differences, which limits the size of problems it can tackle (a few hundreds of unknowns, typically, which proves to be sufficient in early design phases). Among other features, OpenMDAO, developed by NASA, is a powerful tool to optimize trajectories, whereas GEMSEO, at present, does not support dynamic systems. Although both packages offer a wealth of valuable features, neither, to our knowledge, supports multimode systems with event-based transitions. A benchmark between OpenMDAO, GEMSEO and CoSApp was published by Di Giuseppe et al. (2023).

OpenModelica is based on the Modelica language, which supports acausal models. Depending on boundary conditions, the computation of a given model requires a prior automatic causality analysis, generating C code compiled into an executable artifact. Defining non-causal models is a great advantage for developers, who do not have to worry about information fluxes, and need only provide implicit relations between model variables. Moreover, the causality analysis reduces the number of unknowns to its minimum, and the use of a compiled language yields faster execution of the direct model, compared to interpreted languages such as Python. A significant drawback of this approach, though, is the lack of control in the choice of design parameters or iterative variables required to break algebraic loops, which may lead to poorly converging, hard-to-debug systems.

Overview

Systems and Ports

General assembly and execution semantics are coded within base classes, specialized for each model in derived classes. The basic bricks of CoSApp models are referred to as *systems*, represented by base class `System`. Systems are computational units with a defined input/output interface. Inputs and outputs are represented by collections of variables called *ports*. Like systems, custom ports, containing domain-specific variables, can be defined by specializing base class `Port`.

Systems may have an arbitrary number of input and output ports. Higher-level assemblies are created by connecting ports of different systems in the context of a parent system. Hence, a CoSApp system is organized as a hierarchical tree of sub-models, referred to as *child systems*. By design, child systems are always computed prior to parent systems, such that direct sub-systems are always up-to-date when the parent system is executed.

CoSApp systems can also be viewed as oriented graphs transforming input variables into output variables; upstream end nodes of such graphs are referred to as *free inputs*. One key feature of CoSApp is the ability to declare any free input as unknown, in order to solve inverse problems. Cyclic dependencies, if any, are automatically detected, and treated as inner constraints during system execution.

Drivers

Drivers are algorithmic objects that modify the state of a given system, according to a specific simulation intent. Solvers, optimizers and time integrators are typical examples of drivers. Drivers are attached, as external objects, to the system they are intended to modify. Original properties of drivers are:

1. **Any number of drivers** can be added to a single system. In this case, drivers are executed in sequence, each retrieving the owner system in the state determined by the previous driver.
2. Much like systems, drivers can be **organized as composite trees**, executed in a bottom-up fashion. Such assemblies can, for instance, allow one to solve an optimization problem at every time step of a dynamic simulation, or, conversely, to compute a time trajectory at every iteration of an optimization problem.
3. Drivers can be **attached at different levels** of a system tree. This property allows users to individually solve sub-parts of a system tree, in cases where this can help improve a complex convergence process for example.

Combining these properties allows the construction of complex simulation workflows, tailored to specific needs. CoSApp comes with a set of predefined drivers, some of which are discussed below. However, users can also define their own drivers, to implement custom algorithms in their simulation cases.

Intrinsic and Design Problems

Designing a system consists of calculating input variables that satisfy a set of constraints on the system. CoSApp distinguishes between constraints resulting from the inner structure of the system (imposed by physics, or by algebraic loops in the system tree), referred to as *intrinsic constraints*, and constraints imposed to meet specific requirements, referred to as *design constraints*. For example, current balance at the nodes of an electric circuit yields intrinsic constraints, that must be satisfied no matter what. In contrast, seeking a resistance value, say, that ensures a particular current in specific operating conditions, is a design problem, declared outside the model. An analogy can be drawn with object-oriented programming by noting that intrinsic constraints are characteristic of a class, while design problems are specific to individual instances of a class.

CoSApp solves both intrinsic and design problems jointly, as a single, aggregated algebraic problem. This avoids the use of two nested solvers, and guarantees that intrinsic constraints (in particular, coupling conditions) are only solved under desired design conditions, rather than at every iteration of an outer solver.

Main Features

Single- and Multi-point Design

In complex systems, parameters are designed under most critical constraints, that usually occur in different operating conditions. CoSApp offers a simple way of declaring such multi-point design problems.

Each design point is defined by specific environment conditions. While intrinsic constraints are enforced in every design point, specific design constraints can be declared on individual points. Unknowns can be declared as point-specific, or globally, for all points. In the former case, local values are determined for the targeted design points; in the latter case, a unique value of the unknown is sought, and used on all points. Geometrical parameters, typically, are generally regarded as global design unknowns, since their values are independent of operating conditions.

Optimization

Minimization or maximization of scalar quantities can be performed using a dedicated driver that encapsulates several algorithms from `scipy.optimize`. Both inequality and equality constraints may be specified. At present, multi-objective optimization is not supported.

Dynamic and Multimode Systems

CoSApp encompasses dynamic systems, containing variables that are implicitly known through their time derivative. Given initial conditions, the continuous trajectory of a dynamic system can be integrated numerically, using dedicated time drivers. Such simulations are referred to as *continuous-time* simulations.

Discontinuities, however, can be introduced with the occurrence of *events*, defined within a system. Events are triggerable objects that activate when a certain condition is detected in their owner system. Upon the occurrence of an event, systems may transition from one mode to another, possibly undergoing structural recomposition (new sub-system tree, new constraints, etc.)

A tailor-made algorithm tracks event occurrences (including event cascades and subsequent system transitions), and updates the system during continous-time phases between events. This algorithm is said to be *hybrid*, as it handles both continuous- and discrete-time (event-based) evolutions of the system.

Surrogate models

Surrogate models can be trained at any level in the system tree (including for the head system), using a response surface computed from a given Design of Experiment (DoE). When present, surrogate models supersede the original behaviour of the system. Several classes of surrogate models are available in CoSApp, but users can also define custom meta-models by providing their own implementation of a specific API.

Example

Many examples are available in online tutorials. In the following section, we illustrate multi-point design of a CoSApp model with a short example.

Multi-point design of a supersonic nozzle

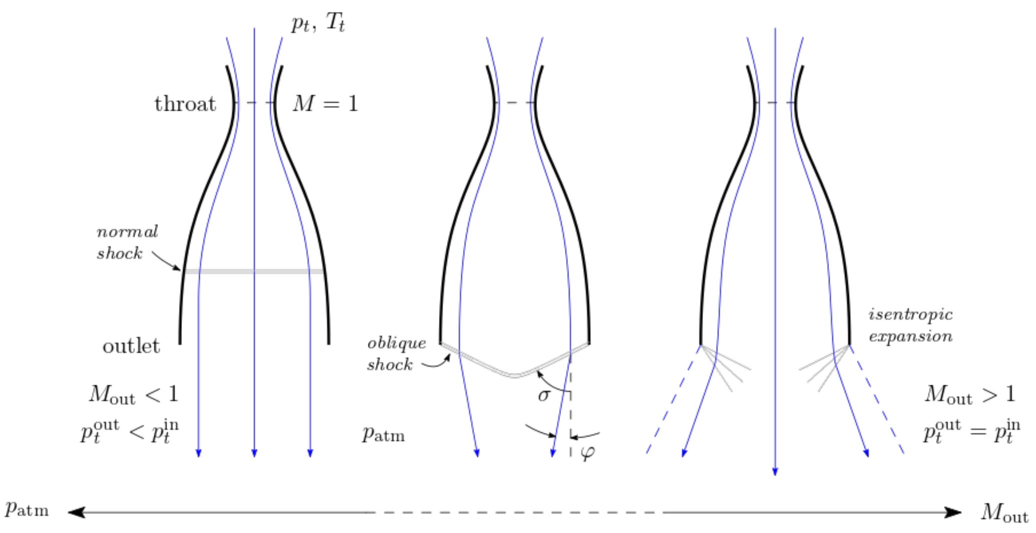

Figure 1: Schematic representation of possible flow conditions in a supersonic nozzle, depending on atmospheric pressure. Left: normal shock in the diverging part; middle: oblique shock at outlet; right: isentropic Prandtl-Meyer expansion at outlet. The subsonic flow regime is omitted.

Design phase

In the next code block, we consider a model named `NozzleAero` (not shown) which computes the flow regime (mass flowrate, Mach number, thrust, etc.) of a supersonic nozzle (Figure 1), given inlet flow conditions (stored in input port `flow_in`) and the atmospheric pressure at the outlet (stored as variable `p_atm`). In this example, two key parameters, namely nozzle throat area and exit section area, are jointly calculated to meet thrust targets at ground level ($p_{\text{atm}} = 1\,\text{bar}$) and in vacuum ($p_{\text{atm}} = 0$).

```python
from propulsion.systems import NozzleAero
from cosapp.drivers import NonLinearSolver, RunSingleCase

# Instantiate model
nozzle = NozzleAero('nozzle')

# Initialize input port `flow_in` and specific heat ratio
nozzle.flow_in.Pt = 115e5
nozzle.flow_in.Tt = 1500
nozzle.gamma = 1.3

# Add solver to the model
solver = nozzle.add_driver(NonLinearSolver('solver'))

# Define design points `ground` and `vacuum` as children of `solver`
ground = solver.add_child(RunSingleCase('ground'))
vacuum = solver.add_child(RunSingleCase('vacuum'))

# Define point-specific conditions
ground.set_values({'p_atm': 1e5})
vacuum.set_values({'p_atm': 0.0})

# Define design unknowns, common to all points
```

```python
solver.add_unknown(['throat.area', 'area_ratio'])

# Define point-specific constraints
ground.add_equation('thrust == 960e3')
vacuum.add_equation('thrust == 1340e3')

# Set initial guess
nozzle.throat.area = 1.0
nozzle.area_ratio = 25.0

# Solve
nozzle.run_drivers()

print(solver.problem)
```

Result:

```
Unknowns [2]
  throat.area = 0.059317698181515305
  area_ratio = 67.61637505773945
Equations [2]
  ground[thrust == 960e3] := -1.6763806343078613e-08
  vacuum[thrust == 1340e3] := 0.0
```

where the number following symbol := indicates the residue (*left* - *right*) of equation *left* == *right*.

Off-design study

```python
import numpy

# Purge drivers
nozzle.drivers.clear()

for nozzle.p_atm in numpy.logspace(5, 2, 11):
    nozzle.run_drivers()  # update model
    print(
        f"p_atm = {nozzle.p_atm * 1e-5:.3f} bar",
        f"thrust = {nozzle.thrust * 1e-3:.1f} kN",
        f"M_out = {nozzle.M_out:.2f}",
        f"{nozzle.status}",
        sep=",\t",
    )
```

Result:

```
p_atm = 1.000 bar,  thrust = 960.0 kN,   M_out = 2.65,  Oblique shock (M=5.39)
p_atm = 0.501 bar,  thrust = 1094.6 kN,  M_out = 3.66,  Oblique shock (M=5.39)
p_atm = 0.251 bar,  thrust = 1158.3 kN,  M_out = 4.44,  Oblique shock (M=5.39)
p_atm = 0.126 bar,  thrust = 1192.6 kN,  M_out = 5.04,  Oblique shock (M=5.39)
p_atm = 0.063 bar,  thrust = 1215.8 kN,  M_out = 5.57,  Isentropic expansion
p_atm = 0.032 bar,  thrust = 1234.9 kN,  M_out = 6.13,  Isentropic expansion
p_atm = 0.016 bar,  thrust = 1250.9 kN,  M_out = 6.72,  Isentropic expansion
p_atm = 0.008 bar,  thrust = 1264.4 kN,  M_out = 7.36,  Isentropic expansion
p_atm = 0.004 bar,  thrust = 1275.8 kN,  M_out = 8.04,  Isentropic expansion
p_atm = 0.002 bar,  thrust = 1285.5 kN,  M_out = 8.77,  Isentropic expansion
p_atm = 0.001 bar,  thrust = 1293.7 kN,  M_out = 9.56,  Isentropic expansion
```

References

Di Giuseppe, R., Delbecq, S., Budinger, V., & Pauvert, V. (2023). An exploratory study of open-source frameworks for MDAO. *AeroBest 2023, ECCOMAS, Lisbon, 19–21 July*. ISBN: 978-989-53599-4-3

Fritzson, P., Pop, A., Abdelhak, K., Ashgar, A., Bachmann, B., Braun, W., Bouskela, D., Braun, R., Buffoni, L., Casella, F., Castro, R., Franke, R., Fritzson, D., Gebremedhin, M., Heuermann, A., Lie, B., Mengist, A., Mikelsons, L., Moudgalya, K., … Östlund, P. (2020). The OpenModelica integrated environment for modeling, simulation, and model-based development. *Modeling, Identification and Control*, *41*(4), 241–295. https://doi.org/10.4173/mic.2020.4.1

Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., & Gazaix, A. (2018). GEMS: A Python library for automation of multidisciplinary design optimization process generation. *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. https://doi.org/10.2514/6.2018-0657

Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., & Naylor, B. A. (2019). OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, *59*(4), 1075–1104. https://doi.org/10.1007/s00158-019-02211-z