Component-oriented acausal modeling of the dynamical systems in Python language on the example of the model of the sucker rod string

Volodymyr B Kopei Corresp., [1] , **Oleh R Onysko** [1] , **Vitalii G Panchuk** [1]

[1] Department of Computerized Mechanical Engineering, Ivano-Frankivsk National Technical University of Oil and Gas, Ivano-Frankivsk, Ukraine

Corresponding Author: Volodymyr B Kopei
Email address: volodymyr.kopey@nung.edu.ua

As a rule, the limitations of specialized modeling languages for acausal modeling of the complex dynamical systems are: limited applicability, poor interoperability with the third party software packages, the high cost of learning, the complexity of the implementation of hybrid modeling and modeling systems with the variable structure, the complexity of the modifications and improvements. In order to solve these problems, it is proposed to develop the easy-to-understand and to modify component-oriented acausal hybrid modeling system that is based on: (1) the general-purpose programming language Python, (2) the description of components by Python classes, (3) the description of components behavior by difference equations using declarative tools SymPy, (4) the event generation using Python imperative constructs, (5) composing and solving the system of algebraic equations in each discrete time point of the simulation. The classes that allow creating the models in Python without the need to study and apply specialized modeling languages are developed. These classes can also be used to automate the construction of the system of difference equations, describing the behavior of the model in a symbolic form. The basic set of mechanical components is developed — 1D translational components "mass", "spring-damper", "force". Using these components, the models of sucker rods string are developed and simulated. These simulation results are compared with the simulation results in Modelica language. The replacement of differential equations by difference equations allow simplifying the implementation of the hybrid modeling and the requirements for the modules for symbolic mathematics and for solving equations.

Component-oriented acausal modeling of the dynamical systems in Python language on the example of the model of the sucker rod string

Volodymyr Bohdanovych Kopei[1], Oleh Romanovych Onysko[1], Vitalii Georgievich Panchuk[1]

[1] Department of Computerized Mechanical Engineering, Ivano-Frankivsk National Technical University of Oil and Gas, Ivano-Frankivsk, Ukraine

Corresponding Author:
Volodymyr Kopei[1]
15 Karpatska Street, Ivano-Frankivsk, 76019, Ukraine
Email address: volodymyr.kopey@nung.edu.ua

Abstract

As a rule, the limitations of specialized modeling languages for acausal modeling of the complex dynamical systems are: limited applicability, poor interoperability with the third party software packages, the high cost of learning, the complexity of the implementation of hybrid modeling and modeling systems with the variable structure, the complexity of the modifications and improvements. In order to solve these problems, it is proposed to develop the easy-to-understand and to modify component-oriented acausal hybrid modeling system that is based on: (1) the general-purpose programming language Python, (2) the description of components by Python classes, (3) the description of components behavior by difference equations using declarative tools SymPy, (4) the event generation using Python imperative constructs, (5) composing and solving the system of algebraic equations in each discrete time point of the simulation. The classes that allow creating the models in Python without the need to study and apply specialized modeling languages are developed. These classes can also be used to automate the construction of the system of difference equations, describing the behavior of the model in a symbolic form. The basic set of mechanical components is developed — 1D translational components "mass", "spring-damper", "force". Using these components, the models of sucker rods string are developed and simulated. These simulation results are compared with the simulation results in Modelica language. The replacement of differential equations by difference equations allow simplifying the implementation of the hybrid modeling and the requirements for the modules for symbolic mathematics and for solving equations.

Introduction

As known, component-oriented simulation modeling is based on the separation of a complex system model into simple components. The component describes the mathematical model of the corresponding physical object (mass, spring, electrical resistance, hydraulic resistance, hydraulic motor, etc.), which is formulated as an algebraic, differential or difference equation. Components are connected with one another through ports (pins, flanges), which define a set of variables for the interaction between components (Elmqvist, 1978; Fritzson, 2015). Components and ports are stored in software libraries. Usually, it is possible to develop new components. The multi-domain modeling allows to use together of components which differ in the physical nature (mechanical, hydraulic, electric, etc.). The component-oriented modeling can be based on causal modeling or acausal modeling (Fritzson, 2015). In the first case, the component receives the signal x at the input, performs a certain mathematical operation $f(x)$ on it and returns the result y to the output. In this case, the modeling is realized by imperative programming by assigning the value of the expression $f(x)$ to the variable y. In the second case, the signal of the connected components can be transmitted in two directions. Such modeling is realized by declarative programming by solving the equation $y=f(x)$, where the unknown can be x or y. Here, the variables x and y are some physical quantities, and the equation $y=f(x)$ is the physical law that describes their relationship. It allows us to simplify the creation of the model, to focus on the physical formulation of the problem, but not on the algorithm for solving it. It is also possible to avoid errors that are typical for imperative programming.

Most often, the behavior of these models is described by the system of differential equations, which are solved by the finite difference method — numerical method based on the replacement of differential operators by difference schemes. As a result, the system of differential equations is replaced by the system of algebraic equations.

The solution of non-stationary problems by the finite difference method is the iterative process — at each iteration find the solution of the stationary problem for the given time point. Explicit and implicit difference schemes are used for this purpose. Explicit schemes immediately find unknown values, using information from previous iterations. Using of the implicit scheme requires the solution of a difference equation because unknown values can be in the right and left sides of the equation. The explicit Euler difference scheme is simple to implement, but it often has numerical instability and low accuracy. To improve accuracy and stability it is desirable to apply modified Euler methods, such as the Runge-Kutta method (Runge, 1895).

Statement of the problem

For the simulation of complex dynamic multi-domain systems such specialized modeling languages are developed: Dymola (Elmqvist, 1978), APMonitor (Hedengren et al., 2014), ASCEND (Piela, McKelvey & Westerberg, 1993), gPROMS (Barton & Pantelides, 1994), Modelica (Fritzson & Engelson, 1998) , MKL, Modelyze (Broman, 2010). Among them, Modelica is the most popular free language for component-oriented modeling of such systems. Its main features: free, object-oriented, declarative, focused on hybrid (continuous and discrete) component-oriented modeling of complex multi-domain physical systems, it supports the construction of hierarchical models, adapted for visual programming, widely used for research in various fields (Fritzson, 2015). Free Modelica Standard Library has about 1280 components. There are free and commercial simulation environments in Modelica language — OpenModelica, JModelica.org, Wolfram SystemModeler, SimulationX, MapleSim, Dymola, LMS Imagine.Lab AMESim.

As a rule, the limitations of such modeling languages are: limited applicability, poor interoperability with the third party software packages, the high cost of learning, the complexity of the modifications and improvements, the complexity of the implementation of hybrid modeling and modeling variable structure systems where the structure and number of equations can change at run-time (Fritzson, Broman & Cellier, 2008; Nikolić, 2016). Some problems can be solved by using interfaces to general-purpose languages (Akesson et al., 2010; Hedengren et al., 2014). But it is usually more difficult to learn a new language than to learn a component or library of a familiar programming language.

These problems are less common in modeling systems that are based on general-purpose programming languages: GEKKO (Beal et al., 2018), Ariadne (Benvenuti et al., 2014), SimuPy (Margolis, 2017), Sims.jl (Short, 2017), Modia.jl (Elmqvist, Henningsson & Otter, 2016), PyDSTool (Clewley et al., 2007), DAE Tools (Nikolić, 2016), Assimulo (Andersson, Führer & Åkesson, 2015). The implementation of such systems can be simplified if the difference equations are used to describe the model instead of differential equations. Many high-level general-purpose languages are suitable for implementing component-based modeling because they have convenient imperative and object-oriented constructions and allow declarative programming. The advantages of modeling systems based on general-purpose programming languages are described in detail in paper (Nikolić, 2016). Python language (Van Rossum & Drake, 1995) is a good choice mainly due to its features: multi-paradigm, object-oriented, intuitive with code readability and improved programmer's productivity, highly extensible, portable, open source, large community and extensive libraries as mathematical libraries SymPy and SciPy. SymPy is a Python library for symbolic mathematics (Meurer et al., 2017). SciPy is a fundamental library for scientific computing (Jones et al., 2001).

The purpose of this work is to develop of the easy-to-understand and to modify component-oriented acausal hybrid modeling system that is based on: (1) the use of general-purpose programming language Python, (2) the description of components by Python classes, (3) the description of components behavior by difference equations using declarative tools SymPy, (4) the event generation using Python imperative constructs, (5) composing and solving a system of algebraic equations in each discrete time point of the simulation. The principles of the system are described using the example of the model of the sucker rod string that is used in the oil industry to join together the surface and downhole components of a rod pumping system. Let's take a look the steel rod string, in which the length is 1500 m and sucker rod diameter is 19 mm. This column will have a mass of 3402 kg, a weight in the liquid of 29204 N, a spring constant of 39694 N/m, a damping constant of 1856 N·s/m. Liquid weight above the pump with a diameter of 38 mm will be 16688 N.

Model in Modelica language

First, we will simulate the free vibrations of the string using the Modelica language. We will develop the model of the simple mechanical translational oscillator, which consists of such components as `Mass`, `SpringDamper` and `Fixed` (Fig. 1). Component `SpringDamper` is designed to simulate the elastic-damper properties of the string. Component `Mass` simulates the inertial properties of the string. Component `Fixed` simulates the fixed point at the top of the string. The module code which describes this model is shown below (Listing S1). In order to simplify the model, these classes differ slightly from the corresponding classes of the standard Modelica library (Fritzson, 2015).

```
connector Flange // class-connector
  Real s; // variable (positions at the flange are equal)
  flow Real f; // variable (sum of forces at the flange is zero)
end Flange;

model Fixed // class-model
  parameter Real s0=0; // parameter (constant in time)
  Flange flange; // object of class Flange
equation // model equations
  flange.s = s0;
end Fixed;

partial model Transl // class-model
  Flange flange_a; // object of class Flange
  Flange flange_b; // object of class Flange
end Transl;

model Mass // class-model
  extends Transl; // inheritance of class Transl
  parameter Real m(min=0, start=1); // parameter
  Real s; // variable
  Real v(start=0); // variable with initial condition
  Real a(start=0); // variable with initial condition
equation // model equations
  v = der(s);
  a = der(v);
  m*a = flange_a.f + flange_b.f;
  flange_a.s = s;
  flange_b.s = s;
end Mass;

model SpringDamper // class-model
```

```
  extends Transl; // inheritance of class Transl
  parameter Real c(final min=0, start=1); // parameter
  parameter Real d(final min=0, start=1); // parameter
  Real s_rel(start=0); // variable
  Real v_rel(start=0); // variable
  Real f; // variable
equation // model equations
  f = c*s_rel+d*v_rel;
  s_rel = flange_b.s - flange_a.s;
  v_rel = der(s_rel);
  flange_b.f = f;
  flange_a.f = -f;
end SpringDamper;

model Oscillator // class-model
  Mass mass1(s(start=-1), v(start=0), m=3402.0); // object with
initial conditions
  SpringDamper spring1(c=39694.0, d=1856.0); // object
  Fixed fixed1(s0=0); // object
equation // additional equations
  // creates a system of equations (see Flange class)
  connect(fixed1.flange, spring1.flange_a);
  connect(spring1.flange_b, mass1.flange_a);
end Oscillator;
```

The Modelica language class describes the set of similar objects (components). The `Flange` class describes the concept of a mechanical flange. Its real-type variable `s` corresponds to the absolute position of the flange. Its value should be equal to the value of the variables `s` of the other flanges connected to this flange. The real-type variable `f` corresponds to the force on the flange. It is marked by the `flow` keyword, which means that the sum of all forces at the connection point is equal to zero. The `Fixed` class describes the concept of a fixed component with one flange, for example `fixed1` (Fig. 1). It has the real-type variable `s0`, which corresponds to the absolute position of the flange, and the object `flange` of the `Flange` class, designed to connect this component to others. The variable `s0` is marked by the `parameter` keyword, which means that it can be changed only at the start of the simulation. After the `equation` keyword, an equation describing the behavior of this component is declared — the `flange` object position must be equal to the `s0` value. The `Transl` class describes an abstract component that has two flanges — `flange_a` and `flange_b`. It is the base class for mechanical translational components with two flanges. The `Mass` class inherits the class `Transl` and describes the sliding mass with inertia. The example of such component is `mass1` (Fig. 1). The command `extends Transl` means inheriting members of the `Transl` class in such a way that they become members of the `Mass` class. That is, the `Mass` components will also have two flanges `flange_a` and `flange_b`. In addition, this class has the parameter `m` (mass) and variables `s` (position), `v` (speed), `a` (acceleration). Expression `start=0` is the default initial condition. After `equation` keyword the system of the differential and algebraic equations which describes behavior of this component is given. The keyword `der` means the derivative with respect to time t (v=ds/dt, a=dv/dt).

The class `SpringDamper` inherits the class `Transl` and describes the linear 1D translational spring and damper in parallel. The example of such component is `springDamper1` (Fig. 1). Class has the parameters `c` (spring constant), `d` (damping constant) and the variables `s_rel` (relative position), `v_rel` (relative speed), `f` (force at `flange_b`). After `equation` keyword the system of differential-algebraic equations of this component is given.

The `Oscillator` class describes spring-mass system (Fig. 1). It contains three components `mass1`, `spring1`, `fixed1`, which are described by the classes `Mass`, `SpringDamper` and `Fixed`, respectively. The values of parameters and initial conditions of these components are shown in round brackets. The additional equations which are obtained from component connections are given after `equation` keyword. So, for example `connect(fixed1.flange, spring1.flange_a)` command connects the flanges of the `fixed1` and `spring1` components and creates the additional system of equations:

```
fixed1.flange.s = spring1.flange_a.s;
fixed1.flange.f = -spring1.flange_a.f
```

The model code can be prepared using any text editor or the Modelica Development Tooling (MDT) module (Pop et al., 2006) of the Eclipse development environment. Simulation of model requires the OpenModelica environment (Fritzson et al., 2005). To start calculations enter this in MDT console:

```
simulate(Oscillator, stopTime=10)
```

To plot the curve that describes the position of `mass1` component with time enter the following into the console:

```
plot(mass1.s)
```

Model in Python language

Description of components by Python-classes

Now we will develop the module `pycodyn` with similar components in Python (Listing S2). In addition, we will develop the `Force` class for simulating the external forces acting on the string. The behaviour of the components will be described by means of the difference equations. As a result, the system of components connected by flanges will be described by the system of the difference equations.

First, we'll import the sympy module and the standard mathematical module math. It is important to distinguish the functions of these modules.

```
from sympy import *
import math
```

Create the global variable dt (time step).

```
dt=0.1
```

If you only need to obtain the system of equations in a symbolic form, then this variable must be an instance of the Symbol class of the sympy module:

```
dt=Symbol('dt')
```

Translational1D is the basic class of mechanical 1D components that have translational motion. The constructor function __init__ is called when an object of this class is created and has two parameters — name of the component name and the dictionary of its attributes args. For component attribute naming, we use the following notation. The symbols x, v, a, f at the beginning of the name mean position, speed, acceleration and force, respectively. The symbol p at the end of the name means the value at time t-dt. The numerical index at the end corresponds to the flange number. To distinguish the variables of various components in the system, each of them begins with the name of the component followed by the symbol "_". For example, the name s1_x2p means the position of the second flange of the component s1 at time t-dt. The constructor for each name-value pair of the dictionary args (except name and self) creates SymPy variables. The symbolic variable of the Symbol class is created if its value is not known. The numeric variable of the Number class is created if its value is known. The self.eqs list contains the component equations, and the self.pins list contains the component flanges. Each equation is created using SymPy class Eq. Each flange is described by a dictionary whose keys are x, xp, f, and the values are the corresponding attributes of the component (see Mass, SpringDamper, Force classes). The pinEqs function returns a list of equations for the component flange that is connected to the flanges of the other components. It has the parameter pindex — the index of the flange (for example 0), and the parameter pins — the list of flanges of the other components. Always the positions of the mechanical 1D translational components on the flange are equal, and the sum of the forces on this flange is zero. For example, if the flange 2 of the component s1 is connected to the flange 1 of the component m1 then pinEqs function of the s1 component returns the list of equations [s1_x2==m1_x1, s1_x2p==m1_x1p, s1_f2==-m1_f1].

```
class Translational1D(object):
```

```
    def __init__(self, name, args):
        self.name=name # component name
        for k,v in args.iteritems(): # for each key-value pair
            if k in ['name','self']: continue # except name and self
            if v==None: # if value is None
                # create symbolic variable with name name+'_'+k
                self.__dict__[k]=Symbol(name+'_'+k)
            elif type(v) in [float,Float]: # if value is float
                self.__dict__[k]=Number(v) # create constant
        self.eqs=[] # equations list
        self.pins=[] # pins list

    def pinEqs(self,pindex,pins):
        eqs=[] # equation list of the flange
        f=Number(0) # sum of forces on flanges of other components
        for pin in pins: # for each flange of the other components
            # add equations describing the equality on the flange:
            # positions
            eqs.append(Eq(self.pins[pindex]['x'], pin['x']))
            # positions at time t-dt
            eqs.append(Eq(self.pins[pindex]['xp'], pin['xp']))
            f+=pin['f'] # add to the sum of forces
        # equality to zero the sum of forces on the flange
        eqs.append(Eq(self.pins[pindex]['f'], -f))
        return eqs
```

The class `Mass` describes the mass concentrated at a point, which has translational motion. It inherits `Translational1D` class. The constructor `__init__` calls the constructor of the base class `Translational1D` and send to it the parameters `name` and `locals()`. The latter is a dictionary of local variables `self`, `name`, `m`, `x`, `xp`, `v`, `vp`, `a`, `f1`, `f2`. The behavior of this component is described by a system of equations `self.eqs`. For example, for the component `m1`:

```
[m1_m*m1_a == m1_f1+m1_f2, m1_a == (m1_v- m1_vp)/dt,
m1_v == (m1_x-m1_xp)/dt]
```

A list of additional equations can be generated for each component flange using the function `pinEqs` described above. The first element of the `self.pins` list is the dictionary `dict(x=self.x, xp=self.xp, f=self.f1)` which means that the positions `x`, `xp` on the flange will be equal to the `self.x`, `self.xp` attributes of this component respectively, and the force `f` on the flange will be equal to the `self.f1` attribute. The same applies to the second element of the list.

```
class Mass(Translational1D):
    def __init__(self, name, m=1.0, x=None, xp=None, v=None, vp=None,
a=None, f1=None, f2=None):
        # base class constructor call
        Translational1D.__init__(self, name, locals())
        # system of equations
        self.eqs=[Eq(self.m*self.a, self.f1+self.f2),
                  Eq(self.a, (self.v-self.vp)/dt),
                  Eq(self.v, (self.x-self.xp)/dt)]
        # two flanges
        self.pins=[dict(x=self.x, xp=self.xp, f=self.f1),
dict(x=self.x, xp=self.xp, f=self.f2)]
```

The `SpringDamper` class describes the translational 1D spring and damper, which are connected in parallel. It inherits `Translational1D` class. In addition to the attributes described above, it has the following attributes: spring constant `c`, damping constant `d`, relative velocity between flanges `vrel`. The behavior of this component is described by a system of equations `self.eqs`. For example, for the component `s1`:

```
[s1_c*( s1_x2-s1_x1)+ s1_d*s1_vrel == s1_f2, -s1_f2 == s1_f1,
s1_vrel == (s1_x2-s1_x2p)/dt-(s1_x1-s1_x1p)/dt]
```

This component also has two flanges and it is possible to generate a list of additional equations using the `pinEqs` function.

```
class SpringDamper(Translational1D):
    def __init__(self, name, c=1.0, d=0.1, x1=None, x2=None, x1p=None,
x2p=None, vrel=None, f1=None, f2=None):
        Translational1D.__init__(self, name, locals())
        # system of equations
        self.eqs=[Eq(self.c*(self.x2-self.x1)+self.d*self.vrel,
self.f2), Eq(-self.f2, self.f1), Eq(self.vrel, (self.x2-self.x2p)/dt-
(self.x1-self.x1p)/dt)]
        # two flanges
        self.pins=[dict(x=self.x1, xp=self.x1p, f=self.f1),
dict(x=self.x2, xp=self.x2p, f=self.f2)]
```

The `Force` class describes a 1D force whose application point has translational motion. The value of the force `f` can be constant or variable. It inherits `Translational1D` class and has one flange.

```
class Force(Translational1D):
    def __init__(self,name,f=None,x=None,xp=None):
```

```
        Translational1D.__init__(self, name, locals())
        self.pins=[dict(x=self.x, xp=self.xp, f=-self.f)] # one flange
```

The class `System` describes the system of components connected by flanges. The constructor `__init__` gets two parameters — the list of components `els` and the list of additional equations `eqs`, which usually are created using `pinEqs` functions. The system components are stored in the `self.els` list and the `self.elsd` dictionary. The list `self.eqs` contains all system equations and is created by joining the equations of all components with additional equations `eqs`.

The function of this class `solve` solves a stationary problem. It returns the solution of a system of equations with conditions `ics` — a dictionary with known values of variables. To solve a system of equations, it can use the SymPy `solve` function, but its algorithm is very slow. It is possible to use fast algorithms for solving equations, for example, the function `scipy.optimize.root` from the SciPy library, which supports many effective methods for solving systems of equations. In this case, the call of the SymPy function `solve(eqs)` must be replaced with the call of the function `self.solveN(eqs)`, which adapts the system of equations for SciPy and solves it using `scipy.optimize.root`.

The function `solveDyn` solves a non-stationary problem. It receives three parameters — the dictionary with initial conditions `d`, the final time value `timeEnd` and the function `fnBC` that returns the dictionary to update the boundary conditions. First, the time variable `t` is assigned an initial value. In the `while` loop with the condition `t<timeEnd`, the following instructions are executed: the positions and velocities of the components in the previous steps `xp`, `x1p`, `x2p`, `vp` are assigned the values of the initial conditions `d`, the values of the boundary conditions are updated, the system of equations is solved by calling the `solve` function, solutions are assigned to the dictionary `d`, the results are saved, the time value increases by `dt`. After the loop is completed, the function returns the results as `T` and `Res` lists. These results can be represented in the form of plots using the `matplotlib` library.

```
class System(object):
    def __init__(self, els, eqs):
        self.els=els # components list
        self.elsd=dict([(e.name,e) for e in els]) # same, but dict.
        self.eqs=[] # list of system equations
        for e in self.els: # for each component
            self.eqs+=e.eqs # join with component equations
        self.eqs=self.eqs+eqs # join with additional equations

    def solveN(self, eqs): # solves the static problem
        # code is not shown here

    def solve(self, ics): # solves the dynamic problem
```

```
        eqs=[e.subs(ics) for e in self.eqs] # substitution of ics
        # discard all degenerate equations
        eqs=[e for e in eqs if e not in (True,False)]
        # solve the system of equations by:
        #sol=solve(eqs) # SymPy (slow)
        sol=self.solveN(eqs) # SciPy (faster)
        sol.update(ics) # update dictionary by dictionary ics
        return sol

    def solveDyn(self, d, timeEnd, fnBC):
        t=0.0 # time variable
        T=[] # list of time values
        Res=[] # list of results
        ics={} # dictionary with values of variables
        while t<timeEnd: # while t < final time value
            for e in self.els: # for each component
                # save positions and velocities
                if 'x' in e.__dict__:
                    ics.update({e.xp:d[e.x]})
                if 'x1' in e.__dict__:
                    ics.update({e.x1p:d[e.x1]})
                if 'x2' in e.__dict__:
                    ics.update({e.x2p:d[e.x2]})
                if 'v' in e.__dict__:
                    ics.update({e.vp:d[e.v]})
            ics.update(fnBC(self.elsd, d, t)) # update BC
            d=self.solve(ics) # solve the problem
            print t
            T.append(t)
            Res.append(d) # save results
            t+=dt # increase time value
            #if some_condition: # changing the system structure
            #    self.__init__(new_els, new_eqs)
        return T,Res
```

You can easily implement modeling of variable structure systems by overriding the `solveDyn` method and calling in it the constructor of the `System` class with new values of arguments `els`, `eqs`. Usually this call should occur after a certain condition.

Simulation of free vibrations of the sucker rod string

Let's perform the simulation of free vibrations of the sucker rod string (Fig. 1). In the separate module (Listing S3) we will create the components: spring-damper `s1` and mass `m1`. In round brackets there are the values of the attributes — the name and the known parameters values.

```
from pycodyn import *
s1=SpringDamper(name='s1', c=39694.0, d=1856.0)
m1=Mass(name='m1',m=3402.0)
```

Create the list of additional equations, formed by connecting the flanges of the components. Then create the object of the component system.

```
peqs=s1.pinEqs(1,[m1.pins[0]])
s=System(els=[s1,m1], eqs=peqs)
```

A list of the model equations can be printed using the command `print s.eqs`. To obtain equations only in the symbolic form, the numerical values of the constructors parameters `c`, `d`, `m` should be replaced by `None`:

```
[s1_c*(-s1_x1 + s1_x2) + s1_d*s1_vrel == s1_f2,
-s1_f2 == s1_f1,
s1_vrel == -(s1_x1 - s1_x1p)/dt + (s1_x2 - s1_x2p)/dt,
m1_a*m1_m == m1_f1 + m1_f2,
m1_a == (m1_v - m1_vp)/dt,
m1_v == (m1_x - m1_xp)/dt,
s1_x2 == m1_x, s1_x2p == m1_xp, s1_f2 == -m1_f1]
```

Let's solve the static problem — the column is stretched by 1 m.

```
ics={m1.x:-1.0,m1.v:0.0,m1.a:0.0,s1.x1:0.0,s1.x1p:0.0,m1.vp:0.0}
d=s.solve(ics)
```

The boundary conditions depend on the type of the problem. If this is the problem of free oscillations, then the position of the string top point `elsd['s1'].x1` and the force on the plunger `elsd['m1'].f2` are zero. Create the function to update the boundary conditions at time `t` for the `elsd` components. Then solve the dynamic problem — free vibrations of the string.

```
def fnBC(elsd, d, t):
    return {elsd['s1'].x1:0.0, elsd['s1'].x1p:0.0, elsd['m1'].f2:0.0}
T,R=s.solveDyn(d, timeEnd=10, fnBC=fnBC)
```

The comparison of oscillator simulation results for Python and Modelica is shown in Fig. 2. The differences are explained by the use of unequal difference schemes in the Python model and the Modelica solver. It is possible to improve the results in the Python model by using the more accurate but more complex difference schemes. For example, if the trapezoidal rule is used

(Listing S4, Listing S5), the second equation for the `Mass` should be `Eq((self.a+self.ap)/2, (self.v-self.vp)/dt)`.

Simulation of the pumping process by the two-section string

Now in the new module (Listing S6) we will create the model of the sucker rod string, which contains two sections. The model of each section consists of three 1D mechanical translational components: `SpringDamper`, `Mass` and `Force` (Fig. 3). The `SpringDamper` component is designed to simulate the elastic-damper properties of the string section, the `Mass` component simulates the inertial properties of the section, and the `Force` component simulates the section weight in the fluid and other external forces acting on the section.

Assign values to the variable of sections weights `fs` and the variable of liquid weight above the plunger `fr`.

```
from pycodyn import *
fs=(-14602.0, -14602.0)
fr=-16688.0
```

Let's create the components: the spring-damper of the first section `s1`, the mass of the first section `m1`, the weight of the first section `f1`, the spring-damper of the second section `s2`, the mass of the second section `m2`, the weight of the second section with the weight of the liquid `f2`.

```
s1=SpringDamper(name='s1', c=79388.0, d=3712.0)
m1=Mass(name='m1',m=1701.0)
f1=Force(name='f1', f=fs[0])
s2=SpringDamper(name='s2', c=79388.0, d=3712.0)
m2=Mass(name='m2', m=1701.0)
f2=Force(name='f2', f=fs[1]+fr)
```

Form the list of the additional equations of the string model, formed by connecting of the components flanges. And create the object of the component system (string model).

```
peqs=s1.pinEqs(1,[m1.pins[0]])
peqs+=m1.pinEqs(1,[s2.pins[0],f1.pins[0]])
peqs+=s2.pinEqs(1,[m2.pins[0]])
peqs+=m2.pinEqs(1,[f2.pins[0]])
s=System(els=[s1,m1,s2,m2,f1,f2], eqs=peqs)
```

The complete list of equations for this system `s.eqs` in the SymPy format:

```
[s1_c*(-s1_x1 + s1_x2) + s1_d*s1_vrel == s1_f2,
-s1_f2 == s1_f1,
s1_vrel == -(s1_x1 - s1_x1p)/dt + (s1_x2 - s1_x2p)/dt,
```

```
m1_a*m1_m == m1_f1 + m1_f2,
m1_a == (m1_v - m1_vp)/dt,
m1_v == (m1_x - m1_xp)/dt,
s2_c*(-s2_x1 + s2_x2) + s2_d*s2_vrel == s2_f2,
-s2_f2 == s2_f1,
s2_vrel == -(s2_x1 - s2_x1p)/dt + (s2_x2 - s2_x2p)/dt,
m2_a*m2_m == m2_f1 + m2_f2,
m2_a == (m2_v - m2_vp)/dt,
m2_v == (m2_x - m2_xp)/dt,
s1_x2 == m1_x, s1_x2p == m1_xp,
s1_f2 == -m1_f1, m1_x == s2_x1,
m1_xp == s2_x1p, m1_x == f1_x,
m1_xp == f1_xp, m1_f2 == f1_f - s2_f1,
s2_x2 == m2_x, s2_x2p == m2_xp,
s2_f2 == -m2_f1, m2_x == f2_x,
m2_xp == f2_xp, m2_f2 == f2_f]
```

Let's solve the static problem — the string under the maximum static loads.

```
ics={m1.v:0.0, m1.a:0.0, m2.v:0.0, m2.a:0.0}
ics.update({s1.x1:0.0, s1.x1p:0.0})
d=s.solve(ics)
```

Dictionary `d` contains the results. To display the position value for the bottom point of the second section, enter the command `print d[m2.x]`. We get the result -0.972. This is the elongation value of the string under maximum load. Let's solve the dynamic problem — the upper point has a harmonic motion. The stroke length of the upper point is 3 m, the number of double strokes per minute is 6.5. The `motion` function describes the harmonic motion of the upper point and returns its position at time `t`.

```
def motion(t):
    A=3.0/2 # amplitude
    n=6.5/60 # frequency
    return A*math.sin(2*math.pi*n*t) # position
```

The `force` function returns the value of the force on the pump plunger `F`, depending on the value of its speed `v`. If the speed is less than zero (downstroke of the string), the function returns the weight value of the second section. Otherwise, the function returns the sum of the second section weight and the liquid weight above the plunger. This function should be smoothed when the sign of the velocity changes, for example, using the hyperbolic tangent function `math.tanh`.

```
def force(v):
    F=fs[1] # weight of the second section
    if v>0: # if upperstroke
        F+=fr # increase the force by value of the fluid weight
    return F*math.tanh(abs(v)/0.01) # smoothing near the point v=0
```

Create the function to update the boundary conditions at time `t` for `elsd` components. Here `d` is the dictionary of the results calculated in the previous step. Then solve the problem.

```
def fnBC(elsd, d, t):
    return {elsd['s1'].x1:motion(t), elsd['f2'].f:force(d[m2.v])}
T,R=s.solveDyn(d, timeEnd=2*60/6.5, fnBC=fnBC)
```

The results (Fig. 4) correspond to practical dynamometer cards obtained on real wells. The simulation of the variable structure system (the breakage of the sucker rod string) is implemented in Listing S7. This is done by overriding the `solveDyn` method. The simulation results are shown in Fig. 5.

Conclusions

The Python-classes that allow creating the models in Python without the need to study and apply specialized modeling languages are developed. These classes can also be used to automate the construction of the system of difference equations, describing the behavior of the model, in a symbolic form. To fully describe the behavior of the model, these equations must be supplemented with initial and boundary conditions, which are described by certain functions (`motion`, `force`, `fnBC`). These functions may contain any imperative code and it simplifies integration with the third party software packages. Composing and solving the system of algebraic equations at each discrete time point of the simulation using SymPy and SciPy is quite slow, but it makes easier to implement variable structure systems modeling. For example, by changing the values of system attributes in the `solveDyn` function. The replacement of differential equations by difference equations allows simplifying the implementation of the hybrid modeling and the requirements for the modules for symbolic mathematics and for solving equations. However, the problem in the form of difference equations is usually more difficult to formulate. In the future it is planned to extend the set of the components, optimize the algorithm for solving equations and develop support for hierarchical models and the tools for building models using block diagrams. The source code is available on the GitHub (https://github.com/vkopey/pycodyn).

References

Åkesson J, Årzén K-E, Gäfvert M, Bergdahl T, Tummescheit H. 2010. Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problems. *Computers and Chemical Engineering*, **34**(11): 1737-1749

DOI: 10.1016/j.compchemeng.2009.11.011
Andersson C, Führer C, Åkesson J. 2015. Assimulo: A unified framework for ODE solvers. *Mathematics and Computers in Simulation* **116**:26-43 DOI: 10.1016/j.matcom.2015.04.007.
Barton PI, Pantelides CC. 1993. gPROMS–a combined discrete/continuous modelling environment for chemical processing systems. *Simulation Series* **25**:25-34
Beal LDR, Hill D, Martin RA, Hedengren JD. 2018. GEKKO Optimization Suite. *Processes* **6**(8) DOI: 10.3390/pr6080106.
Benvenuti L, Bresolin D, Collins P, Ferrari A, Geretti L, Villa T. 2014. Assume–guarantee verification of nonlinear hybrid systems with Ariadne. *Int. J. Robust Nonlinear Control* **24**:699-724 DOI: 10.1002/rnc.2914
Broman D. 2010. Meta-Languages and Semantics for Equation-Based Modeling and Simulation. PhD thesis, Thesis No 1333. Department of Computer and Information Science, Linköping University, Sweden.
Clewley RH, Sherwood WE, LaMar MD, Guckenheimer JM. 2007. PyDSTool, a software environment for dynamical systems modeling. *Available at http://pydstool.sourceforge.net* (accessed 16 March 2019).
Elmqvist H. 1978. A Structured Model Language for Large Continuous Systems. Department of Automatic Control, Lund Institute of Technology (LTH).
Elmqvist H, Henningsson T, Otter M. 2016. Systems Modeling and Programming in a Unified Environment Based on Julia. In: Margaria T., Steffen B. (eds) *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications. ISoLA 2016. Lecture Notes in Computer Science* **9953**. Cham: Springer DOI: 10.1007/978-3-319-47169-3_15
Fritzson P, Engelson V. 1998. Modelica—a unified object-oriented language for system modeling and simulation. In: Jul E, ed. *ECOOP'98—Object-Oriented Programming. Lecture Notes in Computer Science* 1445:67-90, Berlin Heidelberg: Springer.
Fritzson P, Aronsson P, Lundvall H, Nyström K, Pop A, Saldamli L, Broman D. 2005. The OpenModelica modeling, simulation, and development environment. In: *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005)*, Trondheim, Norway, October 13-14, 2005.
Fritzson P., Broman D., Cellier F. 2009. Equation-Based Object-Oriented Languages and Tools. In: Eugster P, ed. *Object-Oriented Technology. ECOOP 2008 Workshop Reader. ECOOP 2008. Lecture Notes in Computer Science* **5475**. Berlin, Heidelberg: Springer, 18-29 DOI: 10.1007/978-3-642-02047-6_3.
Fritzson PA. 2015. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach.* 2nd edition. Wiley-IEEE Press.
Hedengren JD, Shishavan RA, Powell KM, Edgar TF. 2014. Nonlinear modeling, estimation and predictive control in APMonitor. *Computers and Chemical Engineering*. **70**:133-148 DOI: 10.1016/j.compchemeng.2014.04.013.
Jones E, Oliphant E, Peterson P, et al. 2001. SciPy: Open Source Scientific Tools for Python.

Available at http://www.scipy.org (accessed 16 March 2019).

Margolis B. 2017. SimuPy: A Python framework for modeling and simulating dynamical systems. *The Journal of Open Source Software* **2**(17):396 DOI: 10.21105/joss.00396.

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3:e103 DOI: 10.7717/peerj-cs.103.

Nikolić DD. 2016. DAE Tools: equation-based object-oriented modelling, simulation and optimisation software. *PeerJ Computer Science* **2**:e54 DOI: 10.7717/peerj-cs.54

Piela P., McKelvey R., Westerberg A. 1993. An Introduction to the ASCEND Modeling System: Its Language and Interactive Environment. *Journal of Management Information Systems*. **9**:91-122 DOI: 10.1080/07421222.1992.11517969.

Pop A, Fritzson P, Remar A, Jagudin E, Akhvlediani D. 2006. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In: Kral C and Haumer A, ed. *Proceedings of the 5th International Modelica Conference*, Vienna, Austria, September 2006.

Runge C. 1895. *Math. Ann.* **46**:167 DOI: 10.1007/BF01446807.

Short T. 2017. Equation-based modeling and simulations in Julia. *Available at https://github.com/tshort/Sims.jl* (accessed 16 March 2019).

Van Rossum G, Drake JrFL. 1995. *Python reference manual*. Amsterdam: Centrum voor Wiskunde en Informatica.

Figure 1 (on next page)

Block diagram of the oscillator model.

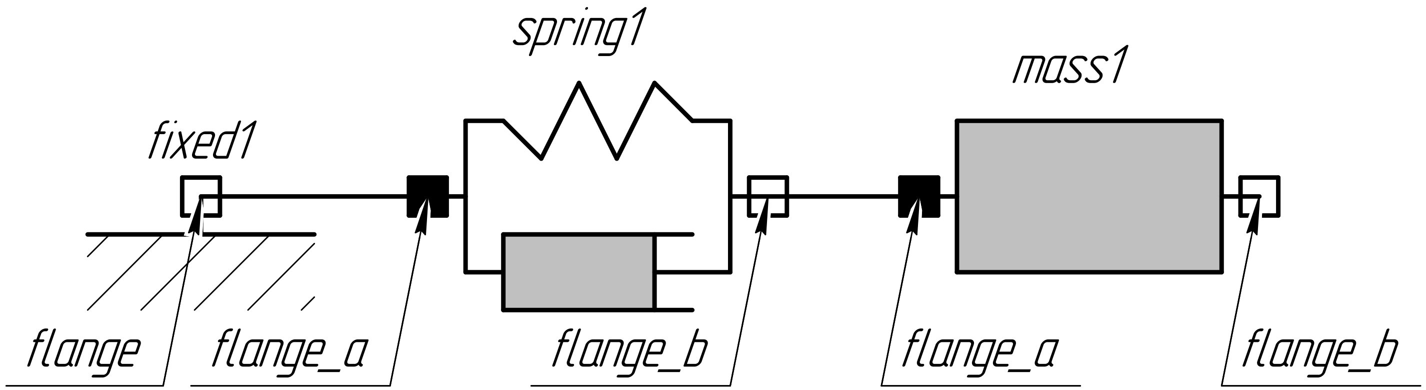

spring1
mass1
fixed1
flange
flange_a
flange_b
flange_a
flange_b

Figure 2 (on next page)

Plunger position (x) during free oscillation of the string.

(▬) Euler method with time step dt=0.1 s; (—) Euler method with time step dt=0.01 s; (---) Trapezoidal rule with time step dt=0.1 s; (....) Runge–Kutta method, order 4 (Modelica-model).

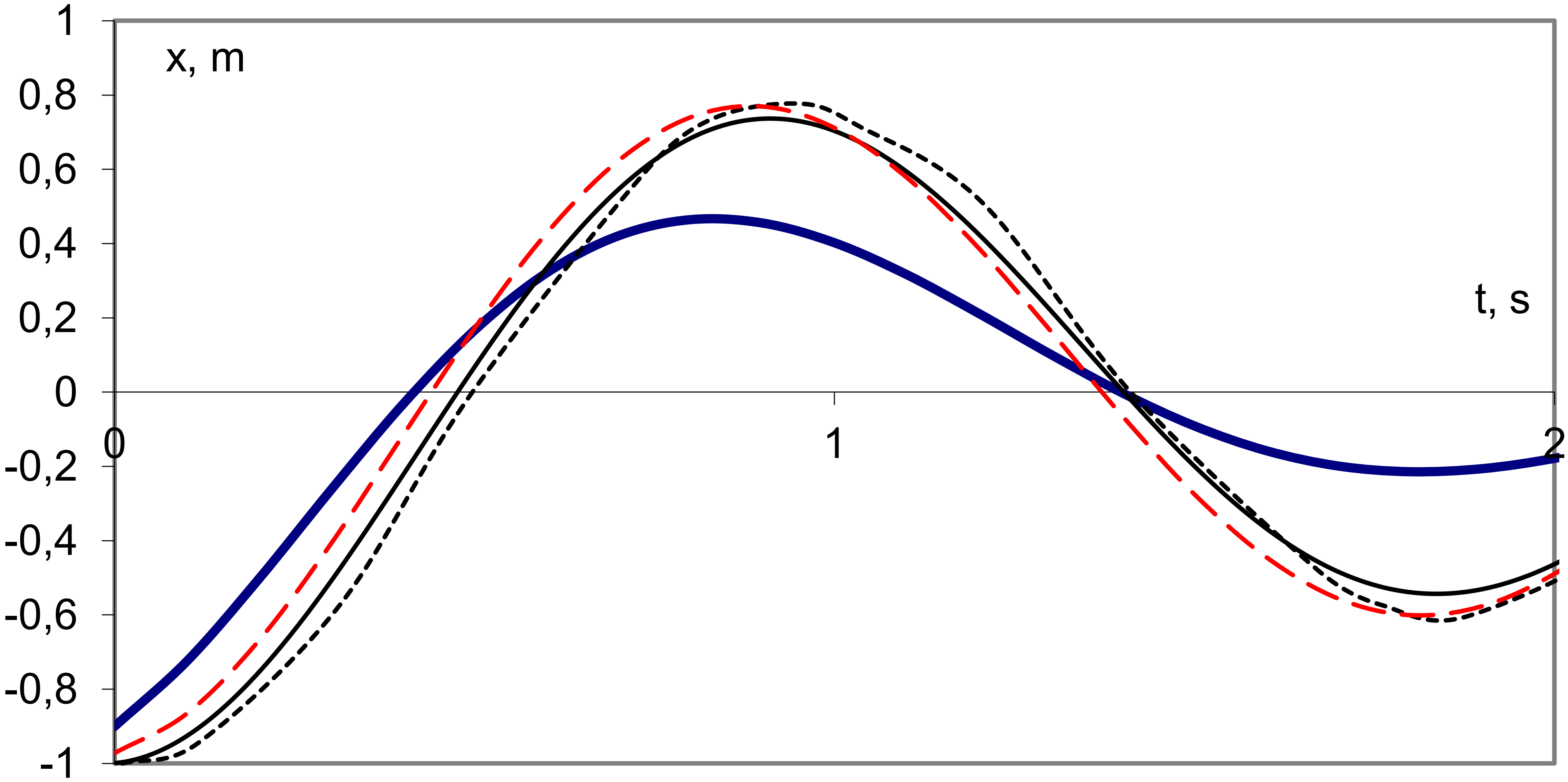

1
0,8
0,6
0,4
0,2
0
-0,2
-0,4
-0,6
-0,8
-1
x, m
t, s
0
1
2

Figure 3 (on next page)

Block diagram of the model with two sections.

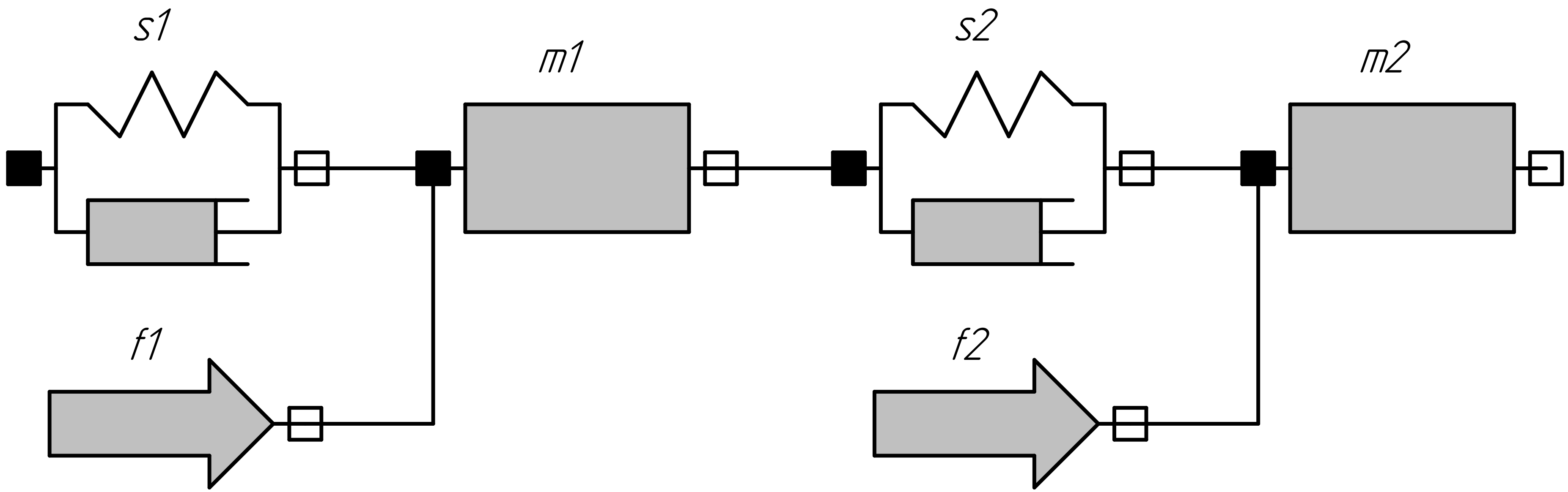

s1
m1
s2
m2
f1
f2

Figure 4(on next page)

Simulation results – the wellhead (at the top) and plunger (at the bottom) dynamometer cards.

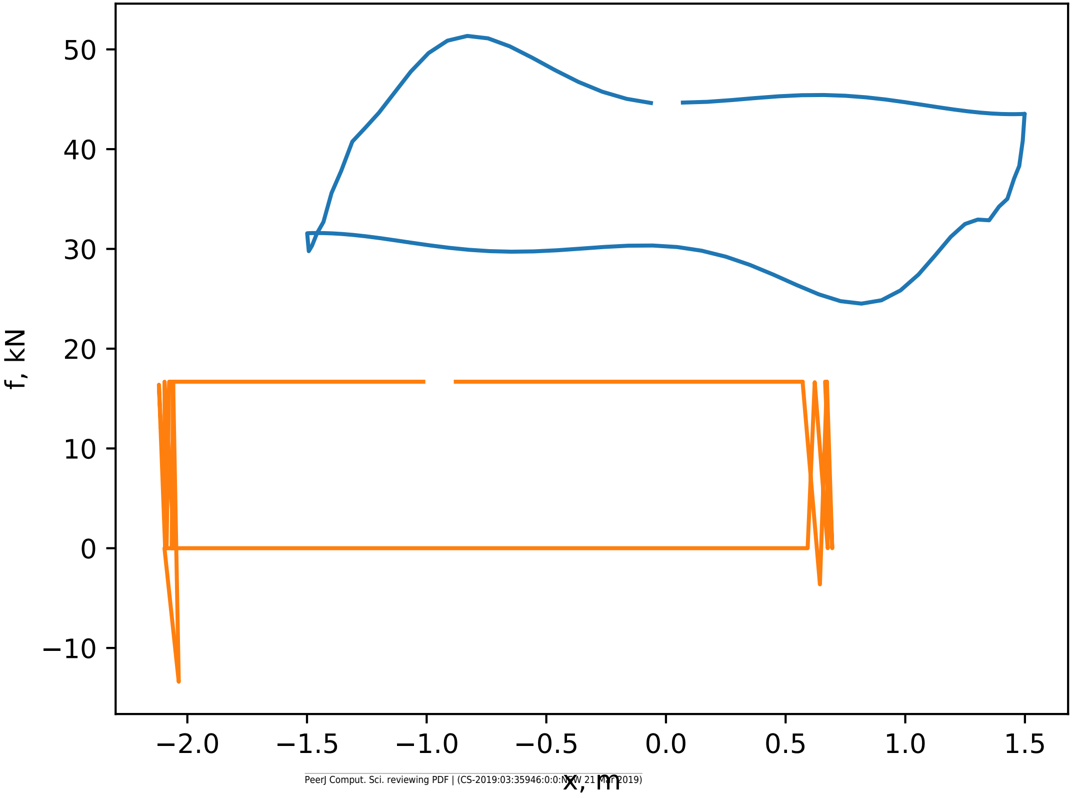
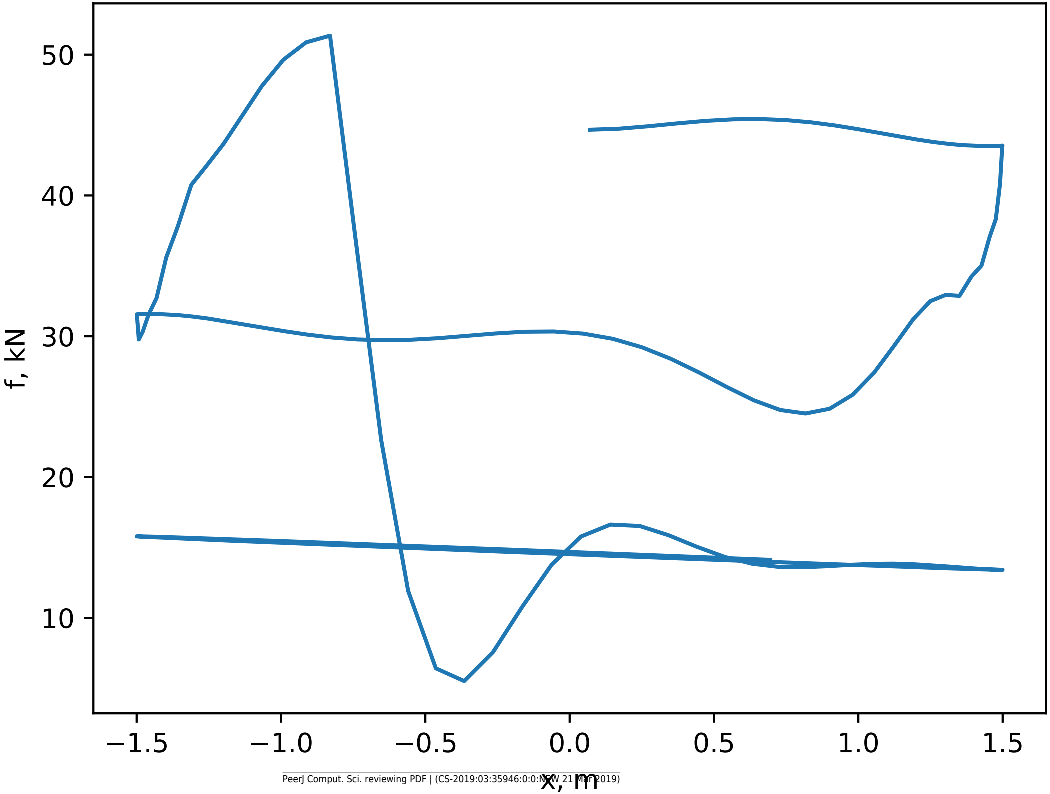

50
40
30
20
10
0
−10
f, kN
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x, m

Figure 5 (on next page)

The simulation of the breakage of the sucker rod string (wellhead dynamometer card).


50
40
30
20
10
f, kN
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x, m