GEKKO
DYNAMIC OPTIMIZATION

GEKKO Documentation

Release 1.3.3

Logan Beal, John Hedengren

Mar 29, 2026

CONTENTS

CHAPTER

ONE

OVERVIEW

GEKKO is a Python package for machine learning and optimization of mixed-integer and differential algebraic equations. It is coupled with large-scale solvers for linear, quadratic, nonlinear, and mixed integer programming (LP, QP, NLP, MILP, MINLP). Modes of operation include parameter regression, data reconciliation, real-time optimization, dynamic simulation, and nonlinear predictive control. GEKKO is an object-oriented Python library to facilitate local execution of APMonitor.

More of the backend details are available at *What does GEKKO do?* and in the GEKKO Journal Article. Example applications are available to get started with GEKKO.

CHAPTER

TWO

INSTALLATION

A pip package is available (see current download stats):

```
pip install gekko
```

Use the **—-user** option to install if there is a permission error because Python is installed for all users and the account lacks administrative priviledge. You can upgrade from the command line with the upgrade flag:

```
pip install --upgrade gekko
```

Another method is to install in a Jupyter notebook with **!pip install gekko** or with Python code, although this is not the preferred method:

```
try:
    from pip import main as pipmain
except:
    from pip._internal import main as pipmain
pipmain(['install','gekko'])
```

CHAPTER

THREE

PROJECT SUPPORT

There are GEKKO tutorials and documentation in:

- GitHub Repository (examples folder)
- Dynamic Optimization Course
- APMonitor Documentation
- GEKKO Documentation
- 18 Example Applications with Videos

For project specific help, search in the GEKKO topic tags on StackOverflow. If there isn't a similar solution, please consider posting a question with a Mimimal, Complete, and Verifiable example. If you give the question a GEKKO tag with [gekko], the subscribed community is alerted to your question.

CHAPTER

FOUR

CITING GEKKO

If you use GEKKO in your work, please cite the following paper:

> Beal, L.D.R., Hill, D., Martin, R.A., and Hedengren, J. D., GEKKO Optimization Suite, Processes, Volume 6, Number 8, 2018, doi: 10.3390/pr6080106.

The BibTeX entry is:

```
@article{beal2018gekko,
title={GEKKO Optimization Suite},
author={Beal, Logan and Hill, Daniel and Martin, R and Hedengren, John},
journal={Processes},
volume={6},
number={8},
pages={106},
year={2018},
doi={10.3390/pr6080106},
publisher={Multidisciplinary Digital Publishing Institute}}
```

CHAPTER

FIVE

CONTENTS

5.1 What does GEKKO do?

GEKKO is optimization software for mixed-integer and differential algebraic equations. It is coupled with large-scale solvers for linear, quadratic, nonlinear, and mixed integer programming (LP, QP, NLP, MILP, MINLP). Modes of operation include data reconciliation, real-time optimization, dynamic simulation, and nonlinear predictive control. The client or server is freely available with interfaces in MATLAB, Python, or from a web browser.

GEKKO is a high-level abstraction of mathematical optimization problems. Values in the models are defined by Constants, Parameters, and Variables. The values are related to each other by Intermediates or Equations. Objective functions are defined to maximize or minimize certain values. Objects are built-in collections of values (constants, parameters, and variables) and relationships (intermediates, equations, and objective functions). Objects can build upon other objects with object-oriented relationships.

The APMonitor executable on the back-end compiles a model to byte-code and performs model reduction based on analysis of the sparsity structure (incidence of variables in equations or objective function) of the model. For differential and algebraic equation systems, orthogonal collocation on finite elements is used to transcribe the problem into a purely algebraic system of equations. APMonitor has several modes of operation, adjustable with the imode parameter. The core of all modes is the nonlinear model. Each mode interacts with the nonlinear model to receive or provide information. The 9 modes of operation are:

1. Steady-state simulation (SS)
2. Model parameter update (MPU)
3. Real-time optimization (RTO)
4. Dynamic simulation (SIM)
5. Moving horizon estimation (EST)
6. Nonlinear control / dynamic optimization (CTL)
7. Sequential dynamic simulation (SQS)
8. Sequential dynamic estimation (SQE)
9. Sequential dynamic optimization (SQO)

Modes 1-3 are steady state modes with all derivatives set equal to zero. Modes 4-6 are dynamic modes where the differential equations define how the variables change with time. Modes 7-9 are the same as 4-6 except the solution is performed with a sequential versus a simultaneous approach. Each mode for simulation, estimation, and optimization has a steady state and dynamic option.

APMonitor provides the following to a Nonlinear Programming Solver (APOPT, BPOPT, IPOPT, MINOS, SNOPT) in sparse form:

- Variables with default values and constraints

- Objective function
- Equations
- Evaluation of equation residuals
- Sparsity structure
- Gradients (1st derivatives)
- Gradient of the equations
- Gradient of the objective function
- Hessian of the Lagrangian (2nd derivatives)
- 2nd Derivative of the equations
- 2nd Derivative of the objective function

Once the solution is complete, APMonitor writes the results in results.json that is loaded back into the python variables by GEKKO

When the system of equations does not converge, APMonitor produces a convergence report in 'infeasibilities.txt'. There are other levels of debugging that help expose the steps that APMonitor is taking to analyze or solve the problem. Setting *DIAGLEVEL* to higher levels (0-10) gives more output to the user. Setting *COLDSTART* to 2 decomposes the problem into irreducible sets of variables and equations to identify infeasible equations or properly initialize a model.

5.2 Quick Start Model Building

5.2.1 Model

Create a python model object:

```
from gekko import GEKKO
m = GEKKO([server], [name]):
```

5.2.2 Variable Types

GEKKO has eight types of variables, four of which have extra properties.

Constants, Parameters, Variables and Intermediates are the standard types. Constants and Parameters are fixed by the user, while Variables and Intermediates are degrees of freedom and are changed by the solver. All variable declarations return references to a new object.

Fixed Variables (FV), Manipulated Variables (MV), State Variables (SV) and Controlled Variables (CV) expand parameters and variables with extra attributes and features to facilitate dynamic optimization problem formulation and robustness for online use. These attributes are discussed in *MV Options* and *CV Options*.

All of these variable types have the optional argument 'name'. The name is used on the back-end to write the model file and is only useful if the user intends to manually use the model file later. Names are case-insensitive, must begin with a letter, and can only contain alphanumeric characters and underscores. If a name is not provided, one is automatically assigned a unique letter/number (c#/p#/v#/i#).

Constants

Define a Constant. There is no functional difference between using a GEKKO Constant, a python variable or a magic number in the Equations. However, the Constant can be provided a name to make the .apm model more clear:

```
c =  m.Const(value, [name]):
```

- Value must be provided and must be a scalar

Parameters

Parameters are capable of becoming MVs and FVs. Since GEKKO defines MVs and FVs directly, parameters just serve as constant values. However, Parameters (unlike Constants) can be (and usually are) arrays.:

```
p = m.Param([value], [name])
```

- The value can be a python scalar, python list of numpy array. If the value is a scalar, it will be used throughout the horizon.

Variable

Calculated by solver to meet constraints (Equations):

```
v = m.Var([value], [lb], [ub], [integer], [name]):
```

- *lb* and *ub* provide lower and upper variable bounds, respectively, to the solver.
- *integer* is a boolean that specifies an integer variable for mixed-integer solvers

Intermediates

Intermediates are a unique GEKKO variable type. Intermediates, and their associated equations, are like variables except their values and gradients are evaluated explicitly, rather than being solved implicitly by the optimizer. Intermediate variables essentially blend the benefits of sequential solver approaches into simultaneous methods.

The function creates an intermediate variable *i* and sets it equal to argument *equation*:

```
i = m.Intermediate(equation,[name])
```

Equation must be an explicitly equality. Each intermediate equation is solved in order of declaration. All variable values used in the explicit equation come from either the previous iteration or an intermediate variable declared previously.

Fixed Variable

Fixed Variables (FV) inherit Parameters, but potentially add a degree of freedom and are always fixed throughout the horizon (i.e. they are not discretized in dynamic modes).:

```
f = m.FV([value], [lb], [ub], [integer], [name])
```

- *lb* and *ub* provide lower and upper variable bounds, respectively, to the solver.
- *integer* is a boolean that specifies an integer variable for mixed-integer solvers

Manipulated Variable

Manipulated Variables (MV) inherit FVs but are discretized throughout the horizon and have time-dependent attributes:

```
m = m.MV([value], [lb], [ub], [integer], [name])
```

- *lb* and *ub* provide lower and upper variable bounds, respectively, to the solver.
- *integer* is a boolean that specifies an integer variable for mixed-integer solvers

State Variable

State Variables (SV) inherit Variables with just a couple extra attributes:

```
s =  m.SV([value], [lb], [ub], [integer], [name])
```

Controlled Variable

Controlled Variables (CV) inherit SVs but potentially add an objective (such as reaching a setpoint in control applications or matching model and measured values in estimation):

```
c = m.CV([value], [lb], [ub], [integer], [name])
```

5.2.3 Equations

Equations are defined with the variables defined and python syntax:

```
m.Equation(equation)
```

For example, with variables `x`, `y` and `z`:

```
m.Equation(3*x == (y**2)/z)
```

Multiple equations can be defined at once if provided in an array or python list::
m.Equations(eqs)

Equations are all solved implicitly together.

5.2.4 Objectives

Objectives are defined like equations, except they must not be equality or inequality expressions. Objectives with *m.Obj()* are minimized (maximization is possible by multiplying the objective by -1) or by using the *m.Maximize()* function. It is best practice to use *m.Minimize()* or *m.Maximize()* for a more readable model:

```
m.Obj(obj)
m.Minimize(obj)
m.Maximize(obj)
```

5.2.5 Connections

Connections are processed after the parameters and variables are parsed, but before the initialization of the values. Connections are the merging of two variables or connecting specific nodes of a discretized variable. Once the variable is connected to another, the variable is only listed as an alias. Any other references to the connected value are referred to the principal variable (*var1*). The alias variable (*var2*) can be referenced in other parts of the model, but will not appear in the solution files.

```
m.Connection(var1,var2,pos1=None,pos2=None,node1='end',node2='end')
```

var1 must be a GEKKO variable, but *var2* can be a static value. If *pos1* or *pos2* is not *None*, the associated var must be a GEKKO variable and the position is the (0-indexed) time-discretized index of the variable.

5.2.6 Example

Here's an example script for solving problem HS71

```
from gekko import GEKKO

#Initialize Model
m = GEKKO()

#define parameter
eq = m.Param(value=40)

#initialize variables
x1,x2,x3,x4 = [m.Var(lb=1, ub=5) for i in range(4)]

#initial values
x1.value = 1
x2.value = 5
x3.value = 5
x4.value = 1

#Equations
m.Equation(x1*x2*x3*x4>=25)
m.Equation(x1**2+x2**2+x3**2+x4**2==eq)

#Objective
m.Minimize(x1*x4*(x1+x2+x3)+x3)

#Set global options
m.options.IMODE = 3 #steady state optimization

#Solve simulation
m.solve()

#Results
print('')
print('Results')
print('x1: ' + str(x1.value))
print('x2: ' + str(x2.value))
print('x3: ' + str(x3.value))
print('x4: ' + str(x4.value))
```

A more compact version of the same problem:

```
from gekko import GEKKO
import numpy as np
m = GEKKO()
x = m.Array(m.Var,4,value=1,lb=1,ub=5)
x1,x2,x3,x4 = x                 # rename variables
x2.value = 5; x3.value = 5      # change guess
m.Equation(np.prod(x)>=25)      # prod>=25
m.Equation(m.sum([xi**2 for xi in x])==40) # sum=40
m.Minimize(x1*x4*(x1+x2+x3)+x3) # objective
m.solve()
```

(continues on next page)

(continued from previous page)

```
print(x)
```

5.2.7 Diagnostics

The run directory *m._path* (with *m.path* alias) contains the model file *gk0_model.apm* and other files required to run the optimization problem either remotely (*m=GEKKO(remote=True)*) or locally (*m=GEKKO(remote=False)*). Use *m.open_folder()* to open the run directory. The run directory also contains diagnostic files such as *infeasibilities.txt* that is produced if the solver fails to find a solution. The default run directory can be changed:

```
from gekko import GEKKO
import numpy as np
import os
# create and change run directory
rd=r'.\RunDir'
if not os.path.isdir(os.path.abspath(rd)):
    os.mkdir(os.path.abspath(rd))
m = GEKKO(remote=False)          # solve locally
m.path = os.path.abspath(rd)     # change run directory
x = m.Array(m.Var,4,value=1,lb=1,ub=5)
x1,x2,x3,x4 = x                  # rename variables
x2.value = 5; x3.value = 5       # change guess
m.Equation(np.prod(x)>=25)       # prod>=25
m.Equation(m.sum([xi**2 for xi in x])==40) # sum=40
m.Minimize(x1*x4*(x1+x2+x3)+x3) # objective
m.solve(disp=False)
print(x)
```

The diagnostic level can be adjusted with *m.options.DIAGLEVEL* between 0 and 10. At level 0, there is minimal information reported that typically includes a summary of the problem and the solver output. At level 1, there are more information messages and timing information for the different parts of the program execution. At level 2, there are diagnostic files created at every major step of the program execution. A diagnostic level of >=2 slows down the application because of increased file input and output, validation steps, and reports on problem structure. Additional diagnostic files are created at level 4. The analytic 1st derivatives are verified with finite differences at level 5 and analytic 2nd derivatives are verified with finite differences at level 6. The DIAGLEVEL is also sent to the solver to indicate a desire for more verbose output as the level is increased. Some solvers do not support increased output as the diagnostic level is increased. A diagnostic level up to 10 is allowed.

5.2.8 Clean Up

Delete the temporary folder (*m.path*) and any files associated with the application with the command

```
m.cleanup()
```

Do not call the *m.cleanup()* function if the application requires another calls to *m.solve()* with updated inputs or objectives.

5.3 Modes

5.3.1 IMODE

`model.options.IMODE` defines the problem type. Each problem type treats variable classes differently and builds equations behind the scenes to meet the particular objective inherit to each problem type. The modes are:

	Simulation	Estimation	Control
Non-Dynamic	*1* Steady-State (SS)	*2* Steady-State (MPU)	*3* Steady-State (RTO)
Dynamic Simultaneous	*4* Simultaneous (SIM)	*5* Simultaneous (EST)	*6* Simultaneous (CTL)
Dynamic Sequential	*7* Sequential (SQS)	*8* Sequential (EST)	*9* Sequential (CTL)

5.3.2 Dynamics

Differential equations are specified by differentiation a variable with the *dt()* method. For example, velocity *v* is the derivative of position *x*:

```
m.Equation( v == x.dt() )
```

Discretization is determined by the model *time* attribute. For example, *m.time = [0,1,2,3]* will discretize all equations and variable at the 4 points specified. Time or space discretization is available with Gekko, but not both. If the model contains a partial differential equation, the discretization in the other dimensions is performed with Gekko array operations as shown in the hyperbolic and parabolic PDE Gekko examples.

Simultaneous methods use orthogonal collocation on finite elements to implicitly solve the differential and algebraic equation (DAE) system. Non-simulation simultaneous methods (modes 5 and 6) simultaneously optimize the objective and implicitly calculate the model/constraints. Simultaneous methods tend to perform better for problems with many degrees of freedom.

Sequential methods separate the NLP optimizer and the DAE simulator. Sequential methods satisfy the differential and algebraic equations, even when the solver is unable to find a feasible optimal solution.

Non-Dynamic modes sets all differential terms to zero to calculate steady-state conditions.

5.3.3 Simulation

Steady-state simulation (IMODE=1) solves the given equations when all time-derivative terms set to zero. Dynamic simulation (IMODE=4,7) is either solved simultaneous (IMODE=4) or sequentially (IMODE=7). Both modes give the same solution but the sequential mode solves one time step and then time-shifts to solve the next time step. The simultaneous mode solves all time steps with one solve. Successful simulation of a model within GEKKO helps initialize and and facilitates the transition from model development/simulation to optimization. In all simulation modes (IMODE=1,4,7), the number of equations must equal the number of variables.

5.3.4 Estimation

MPU

Model Parameter Update (IMODE=2) is parameter estimation for non-dynamic data when the process is at steady-state. The same model instance is used for all data point sets that are rows in the data file. The purpose of MPU is to fit large data sets to the model and update parameters to match the predicted outcome with the measured outcome. .. This mode implements the special variable types as follows:

FV

Fixed variables are the same across all instances of the model that are calculated for each data row.

STATUS adds one degree of freedom for the optimizer, i.e. a parameter to adjust for fitting the predicted outcome to measured values.

FSTATUS allows a *MEAS* value to provide an initial guess (when *STATUS=1*) or a fixed measurement (when *STATUS=0*).

MV

Manipulated variables are like FVs, but can change with each data row, either calculated by the optimizer (*STATUS=1*) or specified by the user (*STATUS=0*).

STATUS adds one degree of freedom for each data row for the optimizer, i.e. an adjustable parameter that changes with each data row.

FSTATUS allows a *MEAS* value to provide an initial guess (when *STATUS=1*) or a fixed measurement (when *STATUS=0*).

CV

Controlled variables may include measurements that are aligned to model predicted values. A controlled variable in estimation mode has objective function terms (squared or l1-norm error equations) built-in to facilitate the alignment.

If *FSTATUS* is on (*FSTATUS=1*), an objective function is added to minimize the model prediction to the measurements. The error is either squared or absolute depending on if *m.options.EV_TYPE* is 2 or 1, respectively. *FSTATUS* enables receiving measurements through the *MEAS* attribute.

If *m.options.EV_TYPE = 1*, *CV.MEAS_GAP=v* will provide a dead-band of size *v* around the measurement to avoid fitting to measurement noise.

STATUS is ignored in MPU. Example applications with parameter regression and oil price regression demonstrate MPU mode.

MHE

Moving Horizon Estimation (IMODE=5,8) is for dynamic estimation, both for states and parameter regression. The horizon to match is the discretized time horizon of the model *m.time*. *m.time* should be discretized at regular intervals. New measurements are added at the end of the horizon (e.g. *m.time[-1]*) and the oldest measurements (e.g. *m.time[0]*) are dropped off.

timeshift enables automatic shifting of all variables and parameters with each new solve of a model. The frequency of new measurements should match the discretization of *m.time*.

FV

Fixed variables are fixed through the horizon.

STATUS adds one degree of freedom for the optimizer, i.e. a fixed parameter for fit.

FSTATUS allows a *MEAS* value to provide an initial guess (when *STATUS=1*) or a fixed measurement (when *STATUS=0*).

MV

Manipulated variables are like FVs, but discretized with time.

STATUS adds one degree of freedom for each time point for the optimizer, i.e. a dynamic parameter for fit.

FSTATUS allows a *MEAS* value to provide an initial guess (when *STATUS=1*) or a fixed measurement (when *STATUS=0*).

CV

Controlled variables may include measurements that are aligned to model predicted values. A controlled variable in estimation mode has objective function terms (squared or l1-norm error equations) built-in to facilitate the alignment.

If *FSTATUS* is on (*FSTATUS=1*), an objective function is added to minimize the model prediction to the measurements. The error is either squared or absolute depending on if *m.options.EV_TYPE* is 2 or 1, respectively. *FSTATUS* enables receiving measurements through the *MEAS* attribute.

If *m.options.EV_TYPE = 1*, *CV.MEAS_GAP=v* will provide a dead-band of size *v* around the measurement to avoid fitting to measurement noise.

STATUS is ignored in MHE. Example *IMODE=5* application:

```
from gekko import GEKKO

t_data = [0, 0.1, 0.2, 0.4, 0.8, 1]
x_data = [2.0,  1.6,  1.2, 0.7,  0.3,  0.15]

m = GEKKO(remote=False)
m.time = t_data
x = m.CV(value=x_data); x.FSTATUS = 1  # fit to measurement
k = m.FV(); k.STATUS = 1               # adjustable parameter
m.Equation(x.dt()== -k * x)            # differential equation

m.options.IMODE = 5   # dynamic estimation
m.options.NODES = 5   # collocation nodes
m.solve(disp=False)   # display solver output
k = k.value[0]

import numpy as np
import matplotlib.pyplot as plt  # plot solution
plt.plot(m.time,x.value,'bo',\
         label='Predicted (k='+str(np.round(k,2))+')')
plt.plot(m.time,x_data,'rx',label='Measured')
# plot exact solution
t = np.linspace(0,1); xe = 2*np.exp(-k*t)
plt.plot(t,xe,'k:',label='Exact Solution')
plt.legend()
plt.xlabel('Time'), plt.ylabel('Value')
plt.show()
```

5.3.5 Control

RTO

Real-Time Optimization (RTO) is a steady-state mode that allows decision variables (*FV* or *MV* types with *STATUS=1*) or additional variables in excess of the number of equations. An objective function guides the selection of the additional variables to select the optimal feasible solution. RTO is the default mode for Gekko if *m.options.IMODE* is not specified.

MPC

Model Predictive Control (MPC) is implemented with *IMODE=6* as a simultaneous solution or with *IMODE=9* as a sequential shooting method.

Controlled variables (*CV*) have a reference trajectory or set point target range as the objective. When *STATUS=1* for a CV, the objective includes a minimization between model predictions and the setpoint.

If *m.options.CV_TYPE=1*, the objective is an l1-norm (absolute error) with a dead-band. The setpoint range should be specified with *SPHI* and *SPLO*. If *m.options.CV_TYPE=2*, the objective is an l2-norm (squared error). The setpoint should be specified with *SP*.

Example MPC (*IMODE=6*) application:

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

m = GEKKO()
m.time = np.linspace(0,20,41)

# Parameters
mass = 500
b = m.Param(value=50)
K = m.Param(value=0.8)

# Manipulated variable
p = m.MV(value=0, lb=0, ub=100)
p.STATUS = 1  # allow optimizer to change
p.DCOST = 0.1 # smooth out gas pedal movement
p.DMAX = 20   # slow down change of gas pedal

# Controlled Variable
v = m.CV(value=0)
v.STATUS = 1  # add the SP to the objective
m.options.CV_TYPE = 2 # squared error
v.SP = 40     # set point
v.TR_INIT = 1 # set point trajectory
v.TAU = 5     # time constant of trajectory

# Process model
m.Equation(mass*v.dt() == -v*b + K*b*p)

m.options.IMODE = 6 # control
m.solve(disp=False)

# get additional solution information
import json
with open(m.path+'//results.json') as f:
    results = json.load(f)

plt.figure()
plt.subplot(2,1,1)
plt.plot(m.time,p.value,'b-',label='MV Optimized')
plt.legend()
plt.ylabel('Input')
plt.subplot(2,1,2)
plt.plot(m.time,results['v1.tr'],'k-',label='Reference Trajectory')
plt.plot(m.time,v.value,'r--',label='CV Response')
plt.ylabel('Output')
plt.xlabel('Time')
plt.legend(loc='best')
plt.show()
```

The other setpoint options include *TAU*, *TIER*, *TR_INIT*, *TR_OPEN*, *WSP*, *WSPHI*, and *WSPLO*.

5.4 Global Options

The following is a list of global model attributes. Each attribute is available through the model attributes *options*, eg:

```
from gekko import GEKKO
m = GEKKO()
m.options.IMODE = 3
```

GEKKO only allows input options to be written, while all options can be read.

This is a complete list of the configuration parameters. Each section includes an indication of the variable type (Integer or Real), the default value, data flow, and description.

5.4.1 APPINFO

Type: Integer, Output

Default Value: 0

Description: Application information: 0=good, error otherwise

Explanation: APPINFO is an error code when the solution is not successful. The meaning of an error code is different for each solver that is selected. The one common code is a value of 0 that always indicates a successful solution. APPINFO is a more detailed error report than APPSTATUS that only gives a binary indication of application success.

5.4.2 APPINFOCHG

Type: Integer, Output

Default Value: 0

Description: Application information change (new-old): 0=no change

Explanation: APPINFOCHG is the difference between the prior value of APPINFO and the most current value of APPINFO. The difference is useful in determining when an application solution may have changed due to several factors as reported by the error code when APPINFOCHG is non-zero.

5.4.3 APPSTATUS

Type: Integer, Output

Default Value: 1

Description: Application status: 1=good, 0=bad

Explanation: APPSTATUS is the overall health monitoring for an application. An application may be unsuccessful for a number of reasons and the APPSTATUS changes to 0 (bad) if it is not able to provide a solution. The APPSTATUS is 1 (good) when the solver converges to a successful solution and there are no errors in reporting the solution.

5.4.4 AUTO_COLD

Type: Integer, Input

Default Value: 0

Description: Automatically cold start model after a specified number of bad cycles 0 = do not auto coldstart 1+ = cold start after specified number of bad cycles as recorded by BAD_CYCLES

Explanation: AUTO_COLD is the number of consecutive bad cycles to wait to attempt a COLDSTART as an initialization mode for an application. It is by default at 0 (no action with bad cycles) but can be set to a higher integer level (1+) as a trigger to initiate a coldstart. When the AUTO_COLD limit is reached, the COLDSTART flag is set to 1

(ON) and it remains on until there is a successful solution. Once a successful solution is found, COLDSTART and BAD_CYCLES are set to zero.

5.4.5 BAD_CYCLES

Type: Integer, Input/Output

Default Value: 0

Description: Counter on number of bad cycles 0 = initial value 1+ = number of consecutive unsuccessful solution attempts

Explanation: BAD_CYCLES is a counter that increments when the application fails to produce a successful solution. When BAD_CYCLES reaches the AUTO_COLD limit, a COLDSTART is initiated. Once a successful solution is found, COLDSTART and BAD_CYCLES are set to zero.

5.4.6 BNDS_CHK

Type: Integer, Input

Default Value: 1

Description: Bounds checking: 1=ON, 0=OFF

Explanation: BNDS_CHK validates a measurement with VLHI, VLLO, and VDVL properties. When BNDS_CHK is OFF, there is no checking of the validity limits. Although an application may have specific validity limits set, it is occasionally desirable to take off the checks to observe raw data input to the model without error detection. All measurement validation actions are reported in the text file dbs_read.rpt. BNDS_CHK, FRZE_CHK, and MEAS_CHK are options regarding data cleansing before it enters the applications. When a measurement is bad, the LSTVAL is restored as the measured value or else LSTVAL+VDVL or LSTVAL-VDVL. The VDVL shift depends on whether VLACTION is 0 (keep LSTVAL) or else is 1 (step by VDVL towards measurement).

5.4.7 COLDSTART

Type: Integer, Input/Output

Default Value: 0

Description: Cold start model: 0=warm start, 1=cold start, 2=decompose problem

Explanation: COLDSTART is an initialization mode for an application. It is by default at 0 (no initialization) but can be set to higher levels (1 or 2) to handle cases where initialization of the problem is a key enabler for a successful solution or to reset an application after a failed solution or a period of inactivity. The initialization values are typically taken from a prior solution as stored in t0 files on the server. When COLDSTART>=1, the t0 files are ignored and the initialization values are taken from the file, the DBS file, and the CSV file in that order. A value of 1 indicates that all FV, MV, and CV STATUS values are turned off (0) temporarily to reduce the number of degrees of freedom and achieve a feasible, yet suboptimal solution. A value of 2 is a more aggressive initialization where the degrees of freedom are turned off and the sparsity of the problem is analyzed to create a sequence of lower block triangular segments. These segments are solved sequentially and an error is reported if a particular block is infeasible. Therefore, COLDSTART=2 is one method to help identify equations or variables bounds that may be causing an infeasible solution. Once the application is successfully solved with COLDSTART>=1, the value of COLDSTART is automatically reset to 0 for the next solution.

5.4.8 CSV_READ

Type: Integer, Input

Default Value: 2

Description: CSV read: 0=Off, 1=Batch, 2=Sequential

Explanation: CSV_READ indicates whether a comma separated value data file should be used by the application as part of initializing a problem and loading data. The name of the CSV file is the same as the name of the model file.

Values assigned to GEKKO variables and the model times are primarily communicated to APMonitor through the csv file. If CSV_READ=0, only scalar values of parameters and variables are read through the .apm file. If CSV_READ=1, all parameter values are also read in. If CSV_READ=2, all parameter and (initial) variable values are loaded.

Only values defined by the user since model initialization or the last solve are written to the csv. All other values are loaded back from the APM internal .t0 database files and timeshifted according to the TIME_SHIFT option. If a measurement (.MEAS) is defined, the associated variable value is not written to the csv. Also, if a variable value is fixed with the fix() command, the fixed value is written in the csv file.

5.4.9 CSV_WRITE

Type: Integer, Input

Default Value: 0

Description: CSV write: 0=Off, 1=Write results.csv, 2=Write results_all.csv

Explanation: CSV_WRITE is an option that controls how much information is written to results files. When it is 0 (OFF), no result files are written. This may be desirable to further reduce the time associated with solution post-processing. When CSV_WRITE=1, a file results.csv is written with all of the constants, parameters, and variables. The data is written in a row oriented form with each row header as the name and subsequent column values are the values at each of the time points requested in the input data file. The values from results.csv are automatically loaded back to the GEKKO variables' attribute *value*. When CSV_WRITE=2, another file named results_all.csv is produced that contains not only the endpoints but also the intermediate collocation nodes of the solution. This data file is saved in the model path, visible by *print(m._path)*.

5.4.10 CTRLMODE

Type: Integer, Output

Default Value: 1

Description: Control mode: 0=terminate, 1=simulate, 2=predict, 3=control

Explanation: The CTRLMODE is the actual controller mode implemented by the application and is an output after the application has completed each cycle. The requested control mode (REQCTRLMODE) is set as an input to the desired level of control but sometimes the CTRLMODE is not able to match the request because of a failed solution, a critical MV is OFF, or other checks with the application. A CTRLMODE level of 0 indicates that the program did not run due to a request to terminate. A CTRLMODE level of 1 (cold mode) indicates that the program was run as a simulator with all STATUS values turned off on FVs, MVs, and CVs. A CTRLMODE level of 2 (warm mode) indicates that the application calculates control actions but only after the second cycle. This mode is commonly used to observe anticipated control actions before the controller is activated to level 3. The CTRLMODE level of 3 means that the controller is ON and implementing changes to the process.

5.4.11 CTRL_HOR

Type: Integer, Input or Output (with CSV read)

Default Value: 1

Description: Control horizon: Horizon length where MVs can be adjusted by the solver

Explanation: CTRL_HOR is the control horizon in the beginning portion of the time horizon where MV changes are allowed. There is also a PRED_HOR (prediction horizon) that is greater than or equal to CTRL_HOR. As opposed to CTRL_HOR, no manipulated variable movement is allowed in the prediction horizon. The individual size of time steps of the control horizon are set with CTRL_TIME. The individual size of time steps of the prediction horizon are set with PRED_TIME. The PRED_TIME is generally larger than the CTRL_TIME to give a section of the overall horizon that

predicts steady state arrival to the set point. When CSV_READ is on and the horizon time points are specified from the data file, the CTRL_HOR and PRED_HOR are set equal to the number of time steps present in the data file.

5.4.12 CTRL_TIME

Type: Real, Input or Output (with CSV read)

Default Value: 60

Description: Time for each step in the control horizon

Explanation: CTRL_TIME is the cycle time of the controller for real-time applications. The clock time trigger for initiating successive controller solutions should be synchronized with CTRL_TIME to avoid dynamic mismatch between the internal simulation time and the process measurements. When CSV_READ is on and the horizon time points are specified from the data file, the CTRL_TIME and PRED_TIME are set equal to the time increment of the first time step.

5.4.13 CTRL_UNITS

Type: Integer, Input

Default Value: 1

Description: Model time units (1=sec, 2=min, 3=hrs, 4=days, 5=yrs)

Explanation: CTRL_UNITS are the time units of the model. This option does not affect the solution but does affect the x-axis of the web plots. The time displayed on the web plots is shown according to the HIST_UNITS option but scaled from the model units as specified by CTRL_UNITS. The valid options are 1=sec, 2=min, 3=hrs, 4=days, and 5=yrs. If CTRL_UNITS=1 and HIST_UNITS=2 then the model is in seconds and the web plots have model predictions and measurements that are displayed in minutes.

5.4.14 CV_TYPE

Type: Integer, Input

Default Value: 1

Description: Controlled variable error model type: 1=linear, 2=squared, 3=ref traj

Explanation: CV_TYPE is a selection of the type of objective function to be used for variables defines as controlled variables (CVs) in control (IMODE=6 or 9) applications. The options are to use a linear penalty from a dead-band trajectory (CV_TYPE=1), squared error from a reference trajectory (CV_TYPE=2), or an experimental reference trajectory type (CV_TYPE=3).

5.4.15 CV_WGT_SLOPE

Type: Real, Input

Default Value: 0.0

Description: Slope for weight on future CV error (e.g. [+] favors steady state)

Explanation: CV_WGT_SLOPE is how the controlled variable WSPHI, WSPLO, or WSP change with time over the control horizon. This option is used to favor either near-term attainment of the setpoint or long-term (future) steady state tracking. It is normally set to zero but can be adjusted to be positive to increase the weighting for future time points or negative to decrease the weighting for future time points.

5.4.16 CV_WGT_START

Type: Integer, Input

Default Value: 0

Description: Start interval for controlled variable error model weights

Explanation: CV_WGT_START is the time step that the controlled variables WSPHI, WSPLO, or WSP start. Before this time step, there is no penalty so that there are no near term objectives to reach a setpoint or trajectory. This option is used to consider long-term (future) steady state tracking while ignoring the near-term path to achieve the final target. It is normally set to zero but can be adjusted to be any positive integer between 0 and the number of time steps in the horizon.

5.4.17 CYCLECOUNT

Type: Integer, Input with Output (+1)

Default Value: 0

Description: Cycle count, increments every cycle

Explanation: CYCLECOUNT is a counter that records the number of cycles of the application. It is both an input and output value because the CYCLECOUNT value can be reset. The CYCLECOUNT may be reset when it is desirable to record when a certain event occurs such as a set point change or when the application is re-initialized.

5.4.18 DBS_LEVEL

Type: Integer, Input

Default Value: 1

Description: Database level limited to a subset of options 0=Basic, Limited Options 1=All Options

Explanation: DBS_LEVEL is an input option to control what is written to the DBS (database) file. When DBS_LEVEL=0, only a subset of basic parameters are written to the DBS file. This is used for compatibility with some industrial control systems that only support a subset of parameters. The basic global parameters are the following:

APPINFO, APPINFOCHG, APPSTATUS, COLDSTART, CYCLECOUNT, DIAGLEVEL, ITERATIONS, OBJFCNVAL, REQCTRLMODE, SOLVESTATUS, SOLVETIME

Local parameters that are designated as basic types are only used when DBS_LEVEL=0:

BIAS, COST, CRITICAL, DCOST, DMAX, DMAXHI, DMAXLO, FSTATUS, LOWER, LSTVAL, MEAS, MODEL, NEWVAL, NXTVAL, PSTATUS, REQONCTRL, SP, SPHI, SPLO, STATUS, TAU, UPPER, VDVL, VLHI, VLLO, WSPHI, WSPLO

5.4.19 DBS_READ

Type: Integer, Input

Default Value: 1

Description: Database read: 0=OFF, 1={Name = Value, Status, Units}, 2={Name,Value}

Explanation: DBS_READ specifies when to read the input DBS files with 0=OFF, 1=Read values with Units, and 2=Read values without Units. There are several text file database (DBS) files that are read at the start of an application for both configuring the problem as well as reading in measurements. The DBS files are skipped when DBS_READ=0 except for header parameters. The DBS files are read in the order of defaults.dbs, {problem name}.dbs, measurements.dbs (source is replay.csv when REPLAY>=1), and overrides.dbs. The DBS files may be need to be skipped to re-initialize an application without feedback from the process.

5.4.20 DBS_WRITE

Type: Integer, Input

Default Value: 1

Description: Database write: 0=OFF, 1={Name = Value, Status, Units}, 2={Name,Value}

Explanation: DBS_WRITE specifies when to write the output DBS files with 0=OFF, 1=Write values with Units, and 2=Write values without Units. It may be desirable to turn off the DBS file writing when the DBS file sends feedback to the process and the application is under development.

5.4.21 DIAGLEVEL

Type: Integer, Input

Default Value: 0

Description: Diagnostic level: 0=none, 1=messages, 2=file checkpoints, 4=diagnostic files, 5=check 1st deriv

Explanation: DIAGLEVEL is the diagnostic level for an application. With higher levels, it is used to give increasingly greater detail about the model compilation, validation, and traceability of information. At level 0, there is minimal information reported that typically includes a summary of the problem and the solver output. At level 1, there are more information messages and timing information for the different parts of the program execution. At level 2, there are diagnostic files created at every major step of the program execution. A diagnostic level of >=2 slows down the application because of increased file input and output, validation steps, and reports on problem structure. Additional diagnostic files are created at level 4. The analytic 1st derivatives are verified with finite differences at level 5 and analytic 2nd derivatives are verified with finite differences at level 6. The DIAGLEVEL is also sent to the solver to indicate a desire for more verbose output as the level is increased. Some solvers do not support increased output as the diagnostic level is increased. A diagnostic level up to 10 is allowed.

5.4.22 EV_TYPE

Type: Integer, Input

Default Value: 1

Description: Estimated variable error model type: 1=linear, 2=squared, 3=approximate linear

Explanation: EV_TYPE applies a specific objective function. Linear is an l1-norm, or in other words the solver minimizes the sum of the absolute value of the difference between the CV and the set point. Squared is an l2-norm or sum squared error (SSE), or in other words the solver minimizes the sum of the squared difference between the CV and the set point. l1-norm can be useful when noise or measurement error is expected because it better rejects those. Option 3 is not typically used as an approximate absolute value function that uses a nonlinear function instead of slack variables.

5.4.23 EV_WGT_SLOPE

Type: Real, Input

Default Value: 0.0

Description: Slope for weight on more current EV error (e.g. favor near-term matching)

Explanation: EV_WGT_SLOPE is how the weight on measurement error (WMEAS) and prior model difference (WMODEL) change with time over the estimation horizon. This option is typically used to favor alignment of the most recent data. It is normally set to zero but can be adjusted to be positive to increase the weighting for more recent data points.

5.4.24 FILTER

Type: Floating Point, Input

Default Value: 1.0

Description: Measurement first-order filter: (0-1)

Explanation: FILTER determines how much of the raw measurement is used to update the value of MEAS. A filter of 0 indicates that the measurement should not be used in updating the MEAS value. The FILTER parameter applies to all inputs into the model from the CSV (data) file and also from the DBS (database) file. FILTER at 1.0 uses all of the measurement and ignores any prior measurement value and 0.5 uses half of each.

CSV (data) file

Model = Model * (1-FILTER) + Measured * FILTER

DBS (database) file

MEAS = MEAS * FILTER + LSTVAL * (1-FILTER)

The FSTATUS parameter is used to adjust the fractional update from new measurements, but this only applies to a single entity at a time. FILTER applies globally to all inputs. Both FSTATUS and FILTER are useful to control the flow of information into the model. It is sometimes desirable to set FILTER or FSTATUS to a low value (close to 0) when the solver is not able to find a solution because of big input changes. A drawback of a filter on data is that raw inputs are not used in the application and it takes several cycles to reach true input values.

5.4.25 FRZE_CHK

Type: Integer, Input

Default Value: 1

Description: Frozen measurement checking 1=ON 0=OFF

Explanation: FRZE_CHK is a checking mechanism to ensure that a measurement has variation and is not frozen at the same value for repeat cycles of the application. If the measurement does not change, it is marked as bad and not used in the application. FRZE_CHK is useful for certain types of measurements that are known to periodically fail and stay at a constant value, such as gas chromatographs. FRZE_CHK can also detect when the application is cycling at a faster rate than the measurement device can deliver a new measurement. FRZE_CHK may be undesirable if the sensor resolution gives a false indication of a frozen measurement when it is actually a steady signal and poor sensor resolution. When FRZE_CHK is OFF, there is no checking of the difference from the prior measurement. BNDS_CHK, FRZE_CHK, and MEAS_CHK are options regarding data cleansing before it enters the applications.

5.4.26 HIST_HOR

Type: Integer, Input

Default Value: 0

Description: History horizon in web plot displays: Integer >= 0

Explanation: HIST_HOR is the number of historical data points to display in the web browser plots. The history horizon values are appended to hst files stored on the server. When HIST_HOR is very large, it can slow down the rendering of the plots or expand the display horizon so much that it can be difficult to distinguish near-term movement. The HIST_HOR does not affect the solution, only the display of prior results. The history always trails off the back of the horizon for simulation, estimation, or control modes. For simulation and control, the newest history points are the initial conditions from the prior cycle. For estimation, the newest history points are also from the initial conditions. Initial conditions from estimation problems are not the current time but from the earliest part of the moving horizon that is time shifted off of the estimation horizon.

5.4.27 HIST_UNITS

Type: Integer, Input

Default Value: 0

Description: History time units on plots only (0=same as CTRL_UNITS, 1=sec, 2=min, 3=hrs, 4=days, 5=yrs)

Explanation: HIST_UNITS are the plots displayed in the web browser. This option does not affect the solution but does affect the x-axis of the web plots. The time displayed on the web plots is shown according to the HIST_UNITS option but scaled from the model units as specified by CTRL_UNITS. The valid options are 1=sec, 2=min, 3=hrs, 4=days, and 5=yrs. If CTRL_UNITS=1 and HIST_UNITS=2 then the model is in seconds and the web plots have model predictions and measurements that are displayed in minutes.

5.4.28 ICD_CALC

Type: Integer, Input

Default Value: 0

Description: Specifications for initial condition differentials (MHE only): 0=OFF, 1=ON

Explanation: ICD_CALC is an option for the initial conditions that are associated with differential variables in estimation applications (IMODE=5 or 8). When ICD_CALC=1 (ON), the algebraic variables remain fixed at current values but the differential variables become adjustable by the solver. Any value in the time horizon can be fixed or calculated by setting FIXED or CALCULATED in a CONNECTIONS section of the model. The ICD_CALC option is typically used when estimation problems have uncertain initial conditions, especially during initialization. Leaving ICD_CALC=1 (ON), may reduce the predictive capability of a model to check for conservation of mass or energy because the differential equations are not enforced for the initial conditions.

5.4.29 IMODE

Type: Integer, Input

Default Value: 3

Description: Model solution mode: 1=ss, 2=mpu, 3=rto, 4=sim, 5=est, 6=ctl

Explanation: IMODE sets the mode or type of model solution. IMODE=1-3 uses steady state models, meaning that all differential variables are set to zero and there are no model dynamics. Options 4-9 calculate the dynamics with either simulation, estimation, or control. There are three modes each for steady state (IMODE=1-3), simultaneous method to compute the dynamics (IMODE=4-6), and sequential method to compute the dynamics (IMODE=7-9). The first option in each set is simulation (IMODE=1, 4, and 7) where the problem must have the same number of variables and equations and optimization is not allowed. The second option in each set is estimation where model parameters such as FVs and MVs or initial conditions are adjusted to match measured values (IMODE=2, 5, and 8). The third option in each set is optimization or control where controlled variables are driven to a desired target value or an objective function is either maximized or minimized (IMODE=3, 6, 9).

5.4.30 ITERATIONS

Type: Integer, Output

Default Value: 1

Description: Iterations for solution: >=1

Explanation: ITERATIONS are the number of major iterations required to find a solution. If the number of iterations reaches MAX_ITER, the unfinished solution is returned with an error message. The number of iterations is typically in the range of 3-100 for most problems although this number can be higher for large-scale or complex systems.

5.4.31 LINEAR

Type: Integer, Input

Default Value: 0

Description: 0 - Nonlinear problem 1 - Linear problem

Explanation: Linear programming (LP) or mixed integer linear programming (MILP) problems can typically be solved much faster than a comparably sized nonlinear programming problem. The LINEAR option indicates whether the problem is linear. This allows the modeling language to set options in certain nonlinear solvers that improve the efficiency of the solution. All of the solvers built into APMonitor are nonlinear programming (NLP) or mixed-integer nonlinear programming (MINLP) solvers. Some of the solvers have simplifications to handle linear problems as well. This option does not override the choice of solver with SOLVER but does communicate to the solver that the problem has linear structure.

5.4.32 MAX_ITER

Type: Integer, Input

Default Value: 100

Description: Maximum iteration: >=1

Explanation: MAX_ITER is the maximum number of major iterations for solution by the solver. If this number is reached, the result cannot be trusted because the equation convergence or objective minimization does not satisfy the Karush Kuhn Tucker conditions for optimality. Reaching the maximum number of iterations can happen when the problem is large, difficult, highly nonlinear or if the problem is infeasible. Increasing MAX_ITER for infeasible problems will not lead to a feasible solution but can help detect the infeasible conditions.

5.4.33 MAX_MEMORY

Type: Integer, Input

Default Value: 4

Description: Maximum memory utilized during model compilation with range 10^2 to 10^10

Explanation: Computer memory is allocated at run-time based on the size of the problem but there are particular parts during the model compilation that need an upper limit on model complexity. MAX_MEMORY=4 allows up to 10^4 = 10,000 sparsity elements per equation but can go up to 10^10. There are rare cases when more sparsity elements are needed than the default of 10,000. In these cases, an error message states that the amount of memory should be increased with the option MAX_MEMORY. A good practice is to increase MAX_MEMORY by an order of magnitude (+1) until the error is not encountered. Increasing MAX_MEMORY requires more Random Access Memory (RAM) when the model is compiled at the start of each optimization problem.

5.4.34 MAX_TIME

Type: Real, Input

Default Value: 1.0e20

Description: Maximum run time in seconds

Explanation: MAX_TIME is the maximum amount of clock time in seconds that the solver should continue. Solutions forced to terminate early by this constraint do not satisfy the Karush Kuhn Tucker conditions for optimality. Even with this constraint, the application may fail to terminate at the required time because there are limited checks within the solvers. If a solver is stuck in a single major iteration when the time limit is reached, the program will terminate once that major iteration is completed. A supervisory application at the operating system level should generally be used to terminate applications that exceed a maximum desired amount of clock or CPU time.

5.4.35 MEAS_CHK

Type: Integer, Input

Default Value: 1

Description: Measurement checking: 1=ON, 0=OFF

Explanation: MEAS_CHK indicates whether to check the validity of measurements before they are used in an application. When MEAS_CHK=0 (OFF), there is no checking of the measurement. Although an application may have specific validation limits, it is occasionally desirable to take off the checks to observe raw data input to the model without error detection. All measurement validation actions are reported in the text file dbs_read.rpt. BNDS_CHK, FRZE_CHK, and MEAS_CHK are options regarding data cleansing before it enters the applications. When a measurement is bad, the LSTVAL is restored as the measured value or else LSTVAL+VDVL or LSTVAL-VDVL. The VDVL shift depends on whether VLACTION is 0 (keep LSTVAL) or else is 1 (step by VDVL towards measurement).

5.4.36 MV_DCOST_SLOPE

Type: Real, Input

Default Value: 0.1

Description: Slope for penalization on future MV moves (i.e. reduces controller procrastination)

Explanation: MV_DCOST_SLOPE implements a linear increase in movement penalty (DCOST). The increase in DCOST favors near term movement in the manipulated variable. One issue with a deadband trajectory is a phenomena called controller procrastination where the optimal solution delays a move because it gives an equal objective function to wait one or more cycles. This causes the controller to be stuck in a state of inaction. Favoring movement on the first step of the controller avoids this delay in implementing the needed changes.

5.4.37 MV_STEP_HOR

Type: Integer, Input

Default Value: 1 (for MV_STEP_HOR) or 0 (for MV(#).MV_STEP_HOR)

Description: Step length for manipulated variables: 0 uses MV_STEP_HOR as default

Explanation: MV_STEP_HOR is the horizon length between each allowable movement of the manipulated variables. There are cases where the MV should not move every time step but should be constrained to move only a certain multiple of the collocation time step. With MV_STEP_HOR = 2, the manipulated variable is allowed to move on the first step and every other step thereafter. MV_STEP_HOR = 5, the manipulated variable is allowed to move on the first step and every 5th step thereafter. There is also a parameter MV_STEP_HOR that is used as a global configuration for all MVs when the individual MV option is set to 0.

5.4.38 MV_TYPE

Type: Integer, Input

Default Value: 0

Description: Manipulated variable type: 0=zero order hold, 1=linear

Explanation: MV_TYPE specifies either a zero order hold (0) or a first order linear (1) interpolation between the MV endpoints. When the MV_STEP_HOR is two or greater, the MV_TYPE is applied only to each segment where adjustments are allowed. The MV segment is otherwise equal to the prior time segment. The MV_TYPE only influences the solution when the number of NODES is between 3 and 6. It is not important when NODES=2 because there are no interpolation nodes between the endpoints.

5.4.39 NODES

Type: Integer, Input

Default Value: 3

Description: Nodes in each horizon step

Explanation: NODES are the number of collocation points in the span of each time segment. For dynamic problems, the time segments are linked together into a time horizon. Successive endpoints of the time segments are merged to form a chain of model predictions. Increasing the number of nodes will generally improve the solution accuracy but also increase the problem size and computation time. Solution accuracy can also be improved by adding more time segments.

5.4.40 OBJFCNVAL

Type: Real, Output

Default Value: 0.0

Description: Objective function value

Explanation: OBJFCNVAL is the objective function value reported by the solver. All objectives are converted to a minimization form before solution is attempted. Any maximization terms are multiplied by -1 to convert to an equivalent minimization form. For maximization problems, the objective function should be multiplied by -1 after retrieving OBJFCNVAL. The objective function may include multiple terms, not just a single objective. OBJFCNVAL is a summation of all objectives.

5.4.41 OTOL

Type: Real, Input

Default Value: 1.0e-6

Description: Objective function tolerance for successful solution

Explanation: OTOL is the relative objective function tolerance for reporting a successful solution. A lower value of OTOL, such as 1e-8, will give a more precise answer but at the expense of more iterations. The default of 1e-6 is generally sufficient for most problems. However, there are times when there are multiple objectives and higher precision is required to fully resolve minor objectives. OTOL and RTOL (relative tolerance on the equations) should generally be adjusted together.

5.4.42 PRED_HOR

Type: Integer, Input or Output (with CSV read)

Default Value: 1.0

Description: Prediction horizon: Total horizon, including control horizon

Explanation: PRED_HOR is the prediction horizon that includes the control horizon and any additional points to track towards steady state. The PRED_HOR must be greater than or equal to CTRL_HOR (control horizon). As opposed to CTRL_HOR, no manipulated variable movement is allowed in the prediction horizon. The individual size of time steps of the prediction horizon beyond the control horizon are set with PRED_TIME. The PRED_TIME is generally larger than the CTRL_TIME to give a section of the overall horizon that predicts steady state arrival to the set point. When CSV_READ is on and the horizon time points are specified from the data file, the CTRL_HOR and PRED_HOR are set equal to the number of time steps present in the data file.

5.4.43 PRED_TIME

Type: Real, Input or Output (with CSV read)

Default Value: 60.0

Description: Time for each step in the horizon

Explanation: PRED_TIME is the prediction time of a controller beyond the control horizon. PRED_TIME is typically set to a larger value than CTRL_TIME to reach steady state conditions but also have fine resolution for near term movement of MVs. When CSV_READ is on and the horizon time points are specified from the data file, the CTRL_TIME and PRED_TIME are set equal to the time increment of the first time step.

5.4.44 REDUCE

Type: Integer, Input

Default Value: 0

Description: Number of pre-processing cycles to identify equations or variables to eliminate

Explanation: REDUCE is the number of cycles of pre-solve analysis before sending the problem to a solver. The analysis eliminates variables and equations with the following characteristics:

- variables that don't belong to an equation or objective
- equations with assignments such as x=2.5 (set to value and fix)
  - identified with a single non-zero in Jacobian row
  - set variable equal to zero, evaluate residual
  - then set x = -resid/jac (when abs(jac)>tolerance)
  - check that upper or lower limits are not violated
- equations that connect two variables y = z (merge)
  - set variables equal to zero, evaluate residual
  - if abs(residual)~0 and abs(jac_1 - jac_2) ~ 0, merge
- independent blocks of linear equations (not yet implemented)
  - perform Lower Block Triangularization (LBT)
  - analyze independent blocks
  - if equation blocks are linear then solve and fix

If no variables are left (all reduced) then APMonitor reports a successful solution without sending the problem to a solver. When no variables or equations are eliminated, the remainder of the REDUCE cycles are skipped. REDUCE has the potential to present a more dense and smaller problem to the solver but may also require more pre-processing time that is more efficiently utilized by the solver.

5.4.45 REPLAY

Type: Integer, Input

Default Value: 0

Description: Row counter for data in replay.csv

Explanation: REPLAY is a row indicator that is incremented each cycle when REPLAY >= 1. When replay mode is activated, data from successive rows in replay.csv are inserted into the file measurements.dbs. The application solves and then REPLAY is incremented by 1. On the next cycle, measurements.dbs is populated from the next row and the

cycle repeats. REPLAY is an efficient way of loading large amounts of data into an application without needing to load each measurement individually.

5.4.46 REQCTRLMODE

Type: Integer, Input

Default Value: 3

Description: Requested control mode: 1=simulate, 2=predict, 3=control

Explanation: REQCTRLMODE is the requested controller mode as an input for the application. The requested control mode (REQCTRLMODE) is set as an input to the desired level of control but sometimes the CTRLMODE is not able to match the request because of a failed solution, a critical MV is OFF, or other checks with the application. REQCTRLMODE level of 0 indicates that the program should not run and the program terminates without attempting a solution. REQCTRLMODE level of 1 (cold mode) indicates that the program should be run as a simulator with all STATUS values turned off on FVs, MVs, and CVs. REQCTRLMODE level of 2 (warm mode) indicates that the application should calculate control actions but only after the second cycle. This mode is commonly used to observe anticipated control actions before the controller is activated to level 3. REQCTRLMODE level of 3 means that the controller should be turned ON and implement changes to the process.

5.4.47 RTOL

Type: Real, Input

Default Value: 1.0e-6

Description: Equation solution tolerance

Explanation: RTOL is the relative inequality or equality equation tolerance for reporting a successful solution. A lower value of RTOL, such as 1e-8, will give a more precise answer but at the expense of more iterations. The default of 1e-6 is generally sufficient for most problems. However, there are times when the equation solution should be reported more precisely. Making RTOL too small may cause a bad solution to be reported because it surpasses the computer precision. RTOL and OTOL (relative tolerance for the objective function) should generally be adjusted together.

5.4.48 SCALING

Type: Integer, Input

Default Value: 1

Description: Variable and Equation Scaling: 0=Off, 1=On (Automatic), 2=On (Manual)

Explanation: SCALING is an option to adjust variables by constants to make starting values equal to 1.0. Scaling of variables and equations generally improves solver convergence. Automatic and Internal scaling strategies are often implemented within solvers as well. The purpose of scaling is to avoid very large or very small values that may cause numerical problems with inverting matrices. Poor scaling may lead to ill-conditioned matrix inversions in finding a search direction. The difference between the largest and smallest eigenvalues should be within 12 orders of magnitude to avoid numerical problems. Scaling can be turned OFF (0) or ON (1). With SCALING=1, the scaling is taken from the initial guess values in the file. If the absolute value is less than one, no scaling is applied. With SCALING=2, the scaling is set for each variable individually.

5.4.49 SENSITIVITY

Type: Integer, Input

Default Value: 1

Description: Sensitivity Analysis: 0=Off, 1=On

Explanation: SENSITIVITY determines whether a sensitivity analysis is performed as a post-processing step. The sensitivity analysis results are accessed either through the web-interface with the SENS tab or by retrieving the sensitivity.txt or sensitivity.htm files. The sensitivity is a measure of how the FV and MV values influence the objective function, SV, and CV final values. Another file, sensitivity_all.txt contains sensitivities for all of the dependent variable values with respect to the FV and MV values.

5.4.50 SEQUENTIAL

Type: Integer, Input

Default Value: 0

Description: Sequential solution method 0=Off 1=On

Explanation: SEQUENTIAL determines whether a the solution is attempted with a simultaneous (0) or sequential (1) approach. The sequential solution method solves independent decision variables in an outer loop from the system equations. This approach attempts to maintain feasibility from iteration to iteration but can significantly slow down the overall solution time. The SEQUENTIAL option can be used with any problem that has degrees of freedom such as IMODE=2,3,5,6. When IMODE=7-9 (Sequential Dynamic Simulation, Estimation, and Control), the SEQUENTIAL option is automatically elevated to 1 and placed back to the default of 0 when the solution is completed.

5.4.51 SOLVER

Type: Integer*, Input

Default Value: 3

Description: Solver options: 0 = Benchmark All Solvers, 1-5 = Available Solvers Depending on License

Explanation: SOLVER selects the solver to use in an attempt to find a solution. There are free solvers: 1: APOPT, 2: BPOPT, 3: IPOPT distributed with the public version of the software. There are additional solvers that are not included with the public version and require a commercial license. IPOPT is generally the best for problems with large numbers of degrees of freedom or when starting without a good initial guess. BPOPT has been found to be the best for systems biology applications. APOPT is generally the best when warm-starting from a prior solution or when the number of degrees of freedom (Number of Variables - Number of Equations) is less than 2000. APOPT is also the only solver that handles Mixed Integer problems. Use option 0 to compare all available solvers. Some solvers and solver options are not available with *remote=False* (default) due to licensing requirements. There is additional information on solver options.

* SOLVER can also be declared with a string (required for solver_extension module). It is converted to integer prior to passing to APMonitor.

5.4.52 SOLVER_EXTENSION

Type: Integer*, Input

Default Value: 0

Description: Solver extension module, 0=Off, 1=AMPLPY, 2=PYOMO

Explanation: Set SOLVER_EXTENSION to 1 (AMPLPY) or 2 (PYOMO) to use the solver extension module to solve, allowing access to a wider range of solvers. For more information see the solver extension module.

* SOLVER_EXTENSION also allows both the strings `AMPLPY` and `PYOMO` instead of the corresponding number.

5.4.53 SOLVESTATUS

Type: Integer, Output

Default Value: 1

Description: Solution solve status: 1=good

Explanation: SOLVESTATUS is an indication of whether the solver returns a successful solution (1) or is unsuccessful at finding a solution (0). The solver may be unsuccessful for a variety of reasons including reaching a maximum iteration limit, an infeasible constraint, or an unbounded solution.

5.4.54 SOLVETIME

Type: Real, Output

Default Value: 1.0

Description: Solution time (seconds)

Explanation: SOLVETIME is the amount of time in seconds dedicated to solving the problem with the solver. This is less than the overall time required for the entire application because of communication overhead, processing of inputs, and writing the solution files. The overall time can be reduced by setting WEB=0 to avoid writing web-interface files when they are not needed.

5.4.55 SPECS

Type: Integer, Input

Default Value: 1

Description: Specifications from restart file: 1=ON, 0=OFF

Explanation: The default specifications of fixed or calculated are indicated in the restart t0 files that store prior solutions for warm starting the next solution. The SPECS option indicates whether the specifications should be read from the t0 file (ON) or ignored (OFF) when reading the t0 file for initialization.

5.4.56 SPC_CHART

Type: Integer, Input

Default Value: 0

Description: Statistical Process Control chart 0=OFF 1=add +/- 3 set point limits (red lines) 2=add +/- 2 set point limits (yellow lines) 3=add +/- 1 set point limits (green lines)

Explanation: SPC_CHART is a web-interface trend option to create Statistical Process Control (SPC) style charts for Controlled Variables (CVs). When SPC_CHARTS=0, there are no additional limits placed on the trend. As SPC_CHARTS is increased by 1, there are additional ranges added to the CV plots. With a level of 1, new lines at +/- 3 SPHI-SPLO are added to the plot as red upper and lower control limits. With a level of 2, new lines at +/- 2 SPHI-SPLO are added to the plot as yellow upper and lower control limits. With a level of 1, new lines at +/- 1 SPHI-SPLO are added to the plot as green upper and lower control limits. Each level adds an additional two lines.

5.4.57 STREAM_LEVEL

Type: Integer, Input

Default Value: 0

Description: Stream level options 0=Mass Balance 1=Mass, Mole, and Energy Balances

Explanation: STREAM_LEVEL controls the amount of detailed contained in flowsheet models through OBJECT declarations. At the basic level (0), only mass balance equations are solved. Variables for each stream are mass flow

rate and mass fractions of all species declared in the COMPOUNDS section of the model. The last mass fraction is automatically set to make the summation equal to 1.0. At more the more advanced level (1), mole and energy balances are added. The higher stream level adds temperature, pressure, enthalpy, mole fractions, volumetric flow, density, and concentration.

5.4.58 TIME_SHIFT

Type: Integer, Input

Default Value: 1

Description: Time shift for dynamic problems: 1=ON, 0=OFF

Explanation: TIME_SHIFT indicates the number of time steps that a prior solution should be shifted to provide both initial conditions and an initial guess for a dynamic simulation or optimization problem. When TIME_SHIFT = 1 (default), the solution is shifted by one time step. For real-time applications, the cycle time of the solutions should correspond to the first time increment of the application. The first time increment is either set in the CSV file in the time column or else with CTRL_TIME. Failure to synchronize the frequency of solution and the first application step size results in dynamic mismatch. The TIME_SHIFT is set to 0 if the solution should not be shifted over from a prior solution. The TIME_SHIFT can be set to >=2 when multiples of the controller cycle time have elapsed since the prior solution.

5.4.59 WEB

Type: Integer, Input

Default Value: 1

Description: Generate HTML pages: 1=ON, 0=OFF

Explanation: WEB is an option that controls how much web-content is produced. A value of 0 indicates that a browser interface should not be created. This option can improve overall application speed because the web interface files are not created. The default value is 1 to create a single instance of the web interface when the program computes a solution. When DIAGLEVEL>=2, the web interface is also created before the program runs to allow a user to view the initial guess values.

5.4.60 WEB_MENU

Type: Integer, Input

Default Value: 1

Description: Generate HTML navigation menus: 1=ON, 0=OFF

Explanation: WEB_MENU turns OFF (0) or ON (1) the display of a navigation pane at the top of each auto-generated schematic. WEB_MENU should generally be ON (1) but can be turned off to not give options to the end user to access certain configuration options that are normally reserved for a more advanced user.

5.4.61 WEB_REFRESH

Type: Integer, Input

Default Value: 10

Description: Automatic refresh rate on HTML pages (default 10 minutes)

Explanation: WEB_REFRESH is an internal time of the auto-generated web-interface to automatically reload the page. The default value is 10 minutes although it is not typically necessary to update the web-page. New values are automatically loaded to the web-page through AJAX communication that updates the parts of the page that need to be updated.

5.4.62 WEB_PLOT_FREQ

Type: Integer, Input

Default Value: 1

Description: Automatic refresh rate on web interface plots

Explanation: WEB_PLOT_FREQ is an internal time in seconds to refresh the web-interface plots. This option does not automatically reload the page but just the plot within the web-page. The default value is 1 second but this value can be increased to lessen the network load as data is repeatedly sent from the server to the web-interface. Processes with slow dynamics or long cycle times may not need the fast refresh rates. If an error message appears, it may indicate that the plot source files were in the process of being rewritten when the new request for data was initiated. The error message is resolved by reloading the web-page.

5.5 MCP Helpers

The `gekko.gk_mcp` add-on module provides a sidecar helper layer for working with GEKKO models directly from Python code. It does not modify the base GEKKO implementation. Instead, it operates on a live `GEKKO()` object or creates a new one when needed.

Core usage follows this pattern:

```
from gekko import GEKKO
from gekko import gk_mcp

m = GEKKO(remote=False)
helper = gk_mcp.ModelMCP(m)

dof = helper.get_degrees_of_freedom()
options = helper.get_options()
tuning = helper.suggest_tuning_changes(goal="faster solve")
```

The helper can also create a model:

```
from gekko import gk_mcp

m = gk_mcp.create_model(remote=False, name="my_model")
```

5.5.1 Available Functions

- `create_model`: create a new `GEKKO()` object.
- `attach_model` / `ModelMCP`: wrap an existing `GEKKO()` object with direct helper methods.
- `materialize_model`: write *.apm*, *.csv*, *.info*, and *measurements.dbs* files for the current model state.
- `bundle_model`: copy the current model artifacts into `.mcp/runs/<run_id>` for later inspection.
- `get_degrees_of_freedom`: return structural counts and a solver-reported or estimated degrees-of-freedom summary.
- `get_options`: parse `measurements.dbs`, `.info`, and `options.json` into grouped tuning data.
- `summarize_results`: summarize `results.json` when it is available.
- `diagnose_model`: inspect current artifacts plus optional stdout and stderr text to classify failures.
- `suggest_tuning_changes`: return rule-based tuning suggestions from the current model artifacts.

- `check_python_syntax`: catch Python parser errors before running a model script.
- `scaffold_model_script`: generate starter scripts for steady-state, dynamic optimization, MHE, and MPC use cases.
- `run_python_script`: execute a Python model script and bundle its artifacts.

5.5.2 Building a New Model

You can build directly with `GEKKO()` and then inspect the current model:

```
from gekko import GEKKO, gk_mcp

m = GEKKO(remote=False)
x = m.Var(value=1)
y = m.Var(value=2)
m.Equations([x + y == 5, x - y == 1])

helper = gk_mcp.ModelMCP(m)
print(helper.materialize())
print(helper.get_degrees_of_freedom())
```

If you want a script template first:

```
from gekko import gk_mcp
from pathlib import Path

workspace = Path(".")
template = gk_mcp.scaffold_model_script(workspace, problem_type="steady_state", model_
↪name="demo")
syntax = gk_mcp.check_python_syntax(source=template["source"])
```

Syntax errors are reported with the line, column, and offending text so an LLM can repair the script before execution.

5.5.3 Diagnosing Solve Failures

When a solve fails, pass the current model and any captured solver output:

```
from gekko import gk_mcp

helper = gk_mcp.ModelMCP(m)
diagnosis = helper.diagnose(stdout_text=solver_stdout, stderr_text=solver_stderr)
print(diagnosis)
```

The diagnosis looks for:

- Python syntax and runtime exceptions
- APMonitor `@error` blocks
- infeasibility messages
- time limits and convergence issues
- negative or suspicious degrees of freedom

The return value includes supporting file paths so the LLM can inspect the compiled model and tuning files alongside the failure summary.

5.5.4 Tuning to Improve Performance

Use the helper on the current model after materialization or solve:

```
from gekko import gk_mcp

helper = gk_mcp.ModelMCP(m)
options = helper.get_options()
results = helper.summarize_results()
tuning = helper.suggest_tuning_changes(goal="faster solve with smoother MV movement")
```

Typical suggestions include:

- reducing `NODES` for smaller solves
- enabling `SCALING`
- increasing `MV.DCOST` to smooth controller action
- setting a finite `MV.DMAX`
- decreasing `CV.TAU` for tighter tracking
- increasing `FSTATUS` or `WMEAS` for estimation workflows

5.5.5 Optional Run Bundles

If you want a persistent copy of the current model artifacts, create a bundle:

```
bundle = helper.bundle(stdout_text=solver_stdout, stderr_text=solver_stderr)
```

Bundles are stored under:

```
.mcp/runs/<run_id>/
```

This is useful when an LLM needs to inspect a model repeatedly after the original temporary GEKKO run directory changes.

5.6 Tuning Parameters

Local Options: DBS Parameters for Variables

The following is a list of parameters that may be found in the DBS file for variables listed in the INFO file. It is a complete list of the configuration parameters for FV, MV, SV, CV type parameters and variables. Each section includes an indication of the variable type (Integer or Real), the default value, data flow, and description.

5.6.1 AWS

Local Options | Global Options

Type: Floating Point, Output

Default Value: 0

Description: Anti-Windup Status for a Manipulated Variable 0: Not at a limit 1: An upper or lower bound is reached

Explanation: Anti-Windup Status (*AWS*) is terminology borrowed from classical controls such as Proportional Integral Derivative (PID) controllers. Anti-Windup refers to the integral action that is paused when the controller reaches a saturation limit. An example of a saturation limit is a valve that is limited to 0-100% open. If the controller requests -10%, the valve is still limited to 0%. The AWS indication is useful to show when a controller is being driven by an

optimization objective versus constraints that limit movement. A Key Performance Indicator (KPI) of many controllers is the fraction of time that a MV is not at an upper or lower bound limit.

5.6.2 BIAS

Type: Floating Point, Input/Output

Default Value: 0.0

Description: Additive correction factor to align measurement and model values for Controlled Variables (CVs) *BIAS* is additive factor that incorporates the difference between the current measured value and the initial condition of the controller. *FSTATUS* determines how much of the raw measurement is used to update the value of *MEAS*. A feedback status of 0 indicates that the measurement should not be used and the *BIAS* value is kept at the initial value of 0. A feedback status of 1 uses all of the measurement in updating *MEAS*. A feedback status in between 0 and 1 updates *MEAS* with a fractional contribution from *LSTVAL* and the new measurement. The value of *BIAS* is updated from *MEAS* and the unbiased model prediction (*Model_u*).

$$BIAS = MEAS - Model_u$$

The *BIAS* is added to each point in the horizon and the controller objective function drives the biased model (*Model_b*) to the requested set point range.

$$Model_b = Model_u + BIAS$$

The value of BIAS can also be set to an external value by setting the option *BIAS* option directly and setting *FSTATUS* to 0 (OFF).

5.6.3 COST

Type: Real, Input

Default Value: 0.0

Description: Cost weight: (+)=minimize, (-)=maximize

Explanation: Multiplies the parameter by the *COST* value specified. This is used to scale the terms in the objective function. It is important that each term in the objective function is scaled to be of the same order of magnitude to ensure that the optimizer considers each of them (unless different weighting is specifically desired).

5.6.4 CRITICAL

Type: Integer, Input

Default Value: 0

Description: Critical: 0=OFF, 1=ON

Explanation: Turns the application off (*REQCTRLMODE* =1) if the instrument that provides the measurement has a *PSTATUS* =0 (bad measurement). A critical measurement has this flag on to give control back to the operator if the measurement fails to deliver a good value.

5.6.5 DCOST

Type: Real, Input

Default Value: 0.00001

Description: Delta cost penalty for MV movement

Explanation: Adds a term to the objective function that gives a minor penalty for changing the MV. The weight of the penalty is adjusted with the input value. This is useful for systems where excessive movement to the MV causes damage

to equipment or undesirable results. By assigning a weight that is small in comparison to the objective function value, optimal performance is achieved while changing the MV only when necessary.

$$objective \mathrel{+}= \sum_{i=1} \left\| x_{i-1} - x_i \right\|_p$$

Where *p* is equal to *EV_TYPE* (for *IMODE=5or8*) or *CV_TYPE* (for *IMODE=6or9*).

5.6.6 DMAX

Type: Real, Input

Default Value: 1.0e20

Description: Delta MV maximum step per horizon interval

Explanation: Applies a hard constraint that prevents the MV from being changed by more than the specified value in one time step. This can be used to prevent large jumps in the MV value in the case where that is either undesirable or infeasible. The time step is defined as the length of the first time step in the csv file.

5.6.7 DMAXHI

Type: Real, Input

Default Value: 1.0e20

Description: Delta MV positive maximum step per horizon interval

Explanation: Like *DMAX*, but only with positive changes. Applies a hard constraint that prevents the MV from being changed by more than the specified value in one time step, but this constraint only applies to increases in the MV value. This can be used to prevent large jumps in the MV value in the case where that is either undesirable or infeasible.

5.6.8 DMAXLO

Type: Real, Input

Default Value: -1.0e20

Description: Delta MV negative maximum step per horizon interval

Explanation: Like *DMAX*, but only with negative changes. Applies a hard constraint that prevents the MV from being changed by more than the specified value in one time step, but this constraint only applies to decreases in the MV value. This can be used to prevent large jumps in the MV value in the case where that is either undesirable or infeasible.

5.6.9 DPRED

Type: Real, Output

Default Value: 0.0

Description: Delta prediction changes for each step'

Explanation: Changes in the manipulated variables (MVs) are listed for the first 10 steps of the horizon including *DPRED[1]*, *DPRED[2]*, etc. Values of zero indicate that there are no changes. With *REQCTRLMODE=1* (COLD), all *DPRED* values are zero. With *REQCTRLMODE=2* (WARM), only *DPRED[1]* is required to be zero but the other segments may be non-zero. With *REQCTRLMODE=3* (CONTROL), the first *DPRED* value is changing.

5.6.10 FSTATUS

Type: Real, Input

Default Value: 1.0

Description: Feedback status: 1=ON, 0=OFF

Explanation: Determines how much of the measurement (*MEAS*) to use in updating the values in the model. A feedback status of 0 indicates that the measurement should not be used either in estimation or in updating a parameter in the model. A feedback status of 1 uses all of the measurement. A feedback status in between updates the model OR parameter value (x) according to the formula:

$$x = LSTVAL * (1 - FSTATUS) + MEAS * FSTATUS$$

5.6.11 FDELAY

Type: Integer, Input

Default Value: 0

Description: Feedback delay: 0=No Delay, >=1 horizon steps for delay

Explanation: The feedback delay places the measurement at the appropriate point in the horizon for dynamic estimation. Typical examples are laboratory measurement or gas chromatographs that report measurements that were taken in the past, usually with a 10 min - 2 hour delay. When the new value is reported, it should be placed at the appropriate point in the data time horizon. *FDELAY* is the number of horizon steps in the past where the measurement was actually taken.

5.6.12 LOWER

Type: Real, Input

Default Value: -1.0e20

Description: Lower bound

Explanation: *LOWER* is the lower limit of a variable. If the variable guess value is below the lower limit, it is adjusted to the lower limit. The lower limit is also checked with the upper limit to ensure that it is less than or equal to the upper limit. If the lower limit is equal to the upper limit, the variable is fixed at that value.

5.6.13 LSTVAL

Type: Real, Output

Default Value: 1.0

Description: Last value from previous solution

Explanation: The last value (*LSTVAL*) is the value from the prior solution of the optimization problem or simulation.

5.6.14 MEAS

Type: Real, Input

Default Value: 1.0

Description: Measured value

Explanation: The measurement of a variable or parameter. The value of *MEAS* is initialized to the initial model value. The *MEAS* value is used in the application if *FSTATUS* is greater than zero, but not when *FSTATUS=0*.

MEAS must be a scalar and only one measurement is loaded in each *solve()* command. If multiple measurements are needed, they can be loaded through the *VALUE* attribute in their respective locations.

5.6.15 MEAS_GAP

Type: Real, Input

Default Value: 1.0

Description: Deadband for noise rejection of measurements in MHE

Explanation: Used in estimation problems with *EV_TYPE* =1 (l1-norm objective). The measurement gap (*MEAS_GAP*) defines a dead-band region around the measurement. If the model prediction is within that dead-band, there is no objective function penalty. Outside of that region, there is a linear penalty specified with the *WMEAS* parameter. The *WMODEL* parameter is the weighting given to deviation from the prior model prediction but does not have a deadband around the prior model prediction. The gap is only around the measured values.

5.6.16 MODEL

Type: Real, Output

Default Value: 1.0

Description: Model predicted value

Explanation: The MODEL value is a property of SV (State Variable) and CV (Controlled Variable) types. It is the predicted value of the current time. The current time is the first time step for a simulator or controller and the last value in the horizon for estimation.

5.6.17 MV_STEP_HOR

Type: Integer, Input

Default Value: 0 (for global *MV_STEP_HOR*) or 1 (for MV(#).MV_STEP_HOR)

Description: Step length for manipulated variables: 0 uses global *MV_STEP_HOR* as default

Explanation: The manipulated variable step horizon (MV_STEP_HOR) is by default to allow the MV to be adjusted every time set of the collocation. There are cases where the MV should not move every time step but should be constrained to move only a certain multiple of the collocation time step. With MV_STEP_HOR = 2, the manipulated variable is allowed to move on the first step and every other step thereafter. MV_STEP_HOR = 5, the manipulated variable is allowed to move on the first step and every 5th step thereafter. There is also a global parameter *MV_STEP_HOR* that is used as a global configuration for all MVs when the individual MV option is set to 0.

5.6.18 NEWVAL

Type: Real, Output

Default Value: 1.0

Description: New value implemented by the estimator or controller (NEWVAL = MEAS when not in control)

Explanation: The new value of the parameter estimate (FV) or manipulated variable (MV). This value is taken from the first step of the controller or the last step of the estimator. The NEWVAL is set equal to the measured value if the FV or MV status is off and the FSTATUS (feedback status) is ON (1).

5.6.19 NXTVAL

Type: Real, Output

Default Value: 1.0

Description: Next value that the estimator or controller would implement if CTRLMODE=3.

Explanation: The next value (NXTVAL) to be implemented by the controller. This is especially useful for a controller in WARM mode (CTRLMODE=2) where no values are changed on the first step (still in manual mode) but the control

actions are computed beyond the first step. This is a useful mode to inspect the controller performance before it is turned on.

5.6.20 OBJPCT

Type: Real, Output

Default Value: 0.0

Description: Percent of objective function contribution considering all SV and CV variables

Explanation: The objective function percent is useful to see which controlled variables (CVs) are contributing the most to the controller overall objective function. If one of the CVs has a high OBJPCT, it may be dominating the controller action and the tuning factors WSP (CV_TYPE=2) or WSPHI/WSPLO (CV_TYPE=1) should be adjusted accordingly.

5.6.21 OSTATUS

Type: Integer, Output

Default Value: 0

Description: Bit encoding for status messages

Explanation: Status messages encoded in binary form for transfer and decoding. This is deprecated and will be removed in a future release.

5.6.22 OSTATUSCHG

Type: Integer, Output

Default Value: 0

Description: Change in bit encoding for status messages

Explanation: Change of status messages, encoded in binary form for transfer and decoding. This is deprecated and will be removed in a future release.

5.6.23 PRED

Type: Real, Output

Default Value: 1.0

Description: Prediction horizon

Explanation: The first predictions of a state (SV) or controlled (CV) variable. The number of PRED values is limited to the first 10 but can be less with a shorter horizon. PRED[0] is the initial condition while PRED[1] is the first predicted step. The other values PRED[2], PRED[3], … , PRED[10] are the predicted model values up to a horizon of 10 time points.

5.6.24 PSTATUS

Type: Integer, Input

Default Value: 1

Description: Instrument status: 1=GOOD, 0=BAD

Explanation: An indication of the instrument health. If PSTATUS is 0 then the instrument is determined to be bad and the measurement should not be used. By default, all instruments are assumed to be reporting good values.

5.6.25 REQONCTRL

Type: Integer, Input

Default Value: 0

Description: Required for control mode to be ON (3): 1=ON, 0=OFF

Explanation: Certain Manipulated Variables (MVs) and Controlled Variables (CVs) are critical to the operation of the entire application. When any of the MVs or CVs with REQONCTRL are turned off, the entire application is turned off (CTRLMODE=1). The requested control mode (REQCTRLMODE) is the requested controller mode but this option downgrades the controller to a simulation mode if a critical MV or CV is OFF.

5.6.26 SP

Type: Real, Input

Default Value: 0.0

Description: Set point for squared error model

Explanation: The target value for a controller that is using a squared error objective (single trajectory track). The setpoint (SP) is the final target value.

5.6.27 SPHI

Type: Real, Input

Default Value: 1.0e20

Description: Set point high for linear error model

Explanation: The upper range of the target region (dead-band) for a controller that is using an l1-norm error objective. The setpoint high (SPHI) is the upper final target value.

5.6.28 SPLO

Type: Real, Input

Default Value: -1.0e20

Description: Set point low for linear error model

Explanation: The lower range of the target region (dead-band) for a controller that is using an l1-norm error objective. The setpoint low (SPLO) is the lower final target value.

5.6.29 STATUS

Type: Integer, Input

Default Value: 0

Description: Status: 1=ON, 0=OFF

Explanation: The STATUS specifies when a parameter (FV or MV) that is normally fixed (OFF) can become calculated (ON). Similarly, STATUS set to ON, allows a controlled variable (CV) to be included as a model predictive controller set point target or steady state optimizer target. The STATUS value indicates whether that variable should be included in the optimization (ON) or is merely a fixed input or prediction (OFF). It is acceptable to have only a subset of parameters (FVs or MVs) or variables (CVs) with STATUS set to ON. The STATUS can be turned ON or OFF for each cycle of the controller as needed without disrupting the overall application. An estimator uses STATUS for FVs and MVs but uses FSTATUS (not STATUS) to determine when measurements are used.

5.6.30 TAU

Type: Real, Input

Default Value: 60.0

Description: Time constant for controlled variable response

Explanation: The time constant is a tuning parameter for the speed of response of a reference trajectory. When the set point is stepped to a new value, the time constant (TAU) adjusts the speed of response with SP_tr = (1-exp(-t/TAU)) (SPnew - SPold) + SPold where SPold is the prior set point, SPnew is the new set point, t is the time, TAU is the time constant, and SP_tr is the resultant trajectory.

5.6.31 TIER

Type: Integer, Input

Default Value: 1

Description: Tiered order of Manipulated Variables (MVs) and Controlled Variables (CVs)

Explanation: TIER is an ascending order of precedence for optimization. Tuning an application with multiple objectives can be challenging to coordinate the weights of competing objectives although there is a clear rank priority order. TIER gives the option to split the degrees of freedom into multiple sub-optimization problems. The highest priority values are optimized first while the subsequent priority values are optimized as a next step. Valid TIER values for MVs and CVs are between 1 and 100. There are up to 100 optimization levels and individual MVs / CVs must be at the same TIER value to be included in the sub-optimization problem. The STATUS must also be ON (1) for an MV to be a degree of freedom. The STATUS must also be ON (1) for an CV to be included in the objective function. If there is a sub-optimization problem that has no MV degrees of freedom, a warning message is displayed.

5.6.32 TR_OPEN

Type: Real, Input

Default Value: 1.0

Description: Initial trajectory opening ratio (0=ref traj, 1=tunnel, 2=funnel)

Explanation: TR_OPEN controls the trajectory opening for *CV_TYPE* = 1. It is the ratio of opening gap to the final gap of *SPHI* - *SPLO*. A value of *TR_OPEN = 2* means that the initial trajectory is twice the width of the final gap of *SPHI* - *SPLO*. With *TR_OPEN = 0*, the initial trajectory starts at the same point and widens with a first order response as specified by *TAU* to final destinations of SPHI and SPLO. Each CV can have a different TR_OPEN.

5.6.33 TR_INIT

Type: Integer, Input

Default Value: 0

Description: Setpoint trajectory initialization (0=dead-band, 1=re-center with coldstart/out-of-service, 2=re-center always)

Explanation: *TR_INIT* is an option to specify how the initial conditions of the controlled variable's (CV) setpoint reference trajectory should change with each cycle. The trajectory is set by *TAU*. An option of 0 specifies that the initial conditions should start at SPHI and SPLO, creating an unchanging target over the horizon. An option of 1 makes the initial conditions equal to the current process variable value only on coldstart (*COLDSTART* >=1) or with *CTRLMODE* <=2 when the controller is out of service. Otherwise, the reference trajectory is updated from the first step of the prior solution. When *TR_INIT=2*, the reference trajectory realigns to the variable's initial condition with each cycle. Each CV can have a different *TR_INIT*. The trajectory is also influenced by *TR_OPEN*.

5.6.34 UPPER

Type: Real, Input

Default Value: 1.0e20

Description: Upper bound

Explanation: *UPPER* is the the upper limit of a variable. If the variable guess value is above the upper limit, it is adjusted to the upper limit before it is given to the solver. The upper limit is also checked with the lower limit (*LOWER*) to ensure that it is greater than or equal to the lower limit. If the upper limit is equal to the lower limit, the variable is fixed at that value.

5.6.35 VALUE

Type: Real, Input/Output

Default Value: 1

Description: Value of the variable

Explanation: *VALUE* is a floating point number of a variable.

5.6.36 VDVL

Type: Real, Input

Default Value: 1.0e20

Description: Delta validity limit

Explanation: *VDVL* is the maximum change of a measured value before it is considered an unrealistic change. The change in measured values is recorded at every cycle of the application and compared to the *VDVL* limit. Validity limits are placed to catch instrument errors that may otherwise create bad inputs to the application. If a delta validity limit is exceeded, the action is to either freeze the measured value at the last good value (*VLACTION* =0) or change the measured value by a maximum of *VDVL* in the direction of the bad measurement (*VLACTION* =1). Another way to minimize the impact of unrealistic changes in measurements is to set *FSTATUS* between 0 and 1 with values closer to 0 becoming insensitive to measurement changes.

5.6.37 VLACTION

Type: Integer, Input

Default Value: 0

Description: Validity Limit Action

Explanation: *VLACTION* is the validity limit action when *VDVL* is exceeded. The change in measured values is recorded at every cycle of the application and compared to the *VDVL* limit. Validity limits are placed to catch instrument errors that may otherwise create bad inputs to the application. If a delta validity limit is exceeded, the action is to either freeze the measured value at the last good value (*VLACTION=0*) or change the measured value by a maximum of *VDVL* in the direction of the bad measurement (*VLACTION=1*).

5.6.38 VLHI

Type: Real, Input

Default Value: 1.0e20

Description: Upper validity limit

Explanation: *VLHI* is the upper validity limit for a measured value. Validity limits are one way to protect an application against bad measurements. This gross error detection relies on a combination of change values and absolute limits to

determine when a measurement should be rejected. If *VLHI* is exceeded, the measured value is placed at *VLHI* or the maximum move allowed by *VDVL* when *VLACTION* =1. If *VLACTION* =0, there is no change in the measured value when a limit (*VDVL*, *VLHI*, *VLLO*) is exceeded.

5.6.39 VLLO

Type: Real, Input

Default Value: -1.0e20

Description: Lower validity limit

Explanation: VLLO is the lower validity limit for a measured value. Validity limits are one way to protect an application against bad measurements. This gross error detection relies on a combination of change values and absolute limits to determine when a measurement should be rejected. If the VLLO limit is crossed, the measured value is placed at VLLO or the maximum move allowed by VDVL when VLACTION=1. If VLACTION=0, there is no change in the measured value when a limit (VDVL, VLHI, VLLO) is exceeded.

5.6.40 WMEAS

Type: Real, Input

Default Value: 20.0

Description: Objective function weight on measured value

Explanation: A weighting factor to penalize deviation of current model predictions from measured values. This is used in estimation applications (*Modes* =2, 5, or 8) where the penalty. The infinite estimation horizon approximation is especially useful for systems that have weakly observable or unobservable states. A higher *WMODEL* can also help to reduce the aggressiveness of the estimator in aligning with the measurements by balancing with a penalty against shifting too far from the prior predictions. The *WMODEL* value should never be equal to or larger than the *WMEAS* value for *EV_TYPE* =1 (l1-norm). A *WMODEL* value higher than *WMEAS* will ignore measured values in favor of matching prior model predictions.

5.6.41 WMODEL

Type: Real, Input

Default Value: 2.0d0

Description: Objective function weight on model value

Explanation: A weighting factor to penalize deviation of current model predictions from prior model predictions. This is used in estimation applications (IMODE=2, 5, or 8) where the penalty from a prior model prediction is a "forgetting factor" that approximates an infinite estimation horizon or favors prior predictions. The infinite estimation horizon approximation is especially useful for systems that have weakly observable or unobservable states. A higher WMODEL can also help to reduce the aggressiveness of the estimator in aligning with the measurements by balancing with a penalty against shifting too far from the prior predictions. The WMODEL value should never be equal to or larger than the WMEAS value for EV_TYPE=1 (l1-norm). A WMODEL value higher than WMEAS will ignore measured values in favor of matching prior model predictions.

5.6.42 WSP

Type: Real, Input

Default Value: 20.0

Description: Objective function weight on set point for squared error model

Explanation: A weighting factor to penalize a squared error from the setpoint trajectory with final target SP. The weighting factor is positive to drive the response to the SP trajectory and negative to drive it away from the SP. A negative WSP is highly unlikely and the value should generally be positive.

5.6.43 WSPHI

Type: Real, Input

Default Value: 20.0

Description: Objective function weight on upper set point for linear error model

Explanation: A weighting factor to penalize deviation above the upper setpoint trajectory with final target SPHI. If there is no penalty to cross the upper setpoint, WSPHI can be set to zero.

5.6.44 WSPLO

Type: Real, Input

Default Value: 20.0

Description: Objective function weight on lower set point for linear error model

Explanation: A weighting factor to penalize deviation below the lower setpoint trajectory with final target SPLO. If there is no penalty to cross the lower setpoint trajectory, WSPLO can be set to zero.

5.7 MV Options

The following is a list of parameters associated with FVs and MVs.

5.7.1 FV Options

This is a list of all available options for FVs and their default values:

CRITICAL = 0

DMAX = 1.0e20

DMAXHI = 1.0e20

DMAXLO = -1.0e20

FSTATUS = 1.0

LOWER = -1.0e20

LSTVAL = 1.0

MEAS = 1.0

NEWVAL = 1.0

PSTATUS = 1

STATUS = 0

UPPER = 1.0e20

VDVL = 1.0e20

VLACTION = 0

VLHI = 1.0e20

VLLO = -1.0e20

5.7.2 MV Options

This is a list of all available options for MVs and their default values:

AWS = 0

COST = 0.0

CRITICAL = 0

DCOST = 0.00001

DMAX = 1.0e20

DMAXHI = 1.0e20

DMAXLO = -1.0e20

DPRED = 0.0

FSTATUS = 1.0

LOWER = -1.0e20

LSTVAL = 1.0

MEAS = 1.0

MV_STEP_HOR = 0

NEWVAL = 1.0

NXTVAL = 1.0

PRED = 1.0

PSTATUS = 1

REQONCTRL = 0

STATUS = 0

TIER = 1

UPPER = 1.0e20

VDVL = 1.0e20

VLACTION = 0

VLHI = 1.0e20

VLLO = -1.0e20

5.8 CV Options

5.8.1 SV Options

This is a list of all available options for SVs and their default values:

FSTATUS = 0

LOWER = -1.0e20

MEAS = 1.0

MODEL = 1.0

PRED = 1.0

UPPER = 1.0e20

5.8.2 CV Options

This is a list of all available options for CVs and their default values:

BIAS = 0.0

COST = 0.0

CRITICAL = 0

FDELAY = 0

FSTATUS = 0

LOWER = -1.0e20

LSTVAL = 1.0

MEAS = 1.0

MEAS_GAP = 1.0e-3

MODEL = 1.0

PRED = 1.0

PSTATUS = 1

SP = 0.0

SPHI = 1.0e20

SPLO = -1.0e20

STATUS = 0

TAU = 60.0

TIER = 1

TR_INIT = 0

TR_OPEN = 1.0

UPPER = 1.0e20

VDVL = 1.0e20

VLACTION = 0

VLHI = 1.0e20

VLLO = -1.0e20

WMEAS = 20.0

WMODEL = 2.0

WSP = 20.0

WSPHI = 20.0

WSPLO = 20.0

5.9 Model Building

5.9.1 Model Functions

m = GEKKO(remote=True, [server], [name]):

Creates a GEKKO model *m*.

If *remote* is *True*, the problem is sent to *self.server* to be solved. If *False*, GEKKO looks for local binaries of APMonitor.

Certain solver options are not available for local solve because of distribution restrictions and the requirement for solver licenses.

c = m.Const(value, [name])

A constant value in the optimization problem. This is a static value and is not changed by the optimizer. Constants are fixed values that represent model inputs, fixed constants, or any other value that does not change. Constants are not modified by the solver as it searches for a solution. As such, constants do not contribute to the number of degrees of freedom (DOF):

```
c = m.Const(3)
```

The constants may be defined in one section or in multiple declarations throughout the model. Constant initialization is performed sequentially, from top to bottom. If a constant does not have an initial value given, a default value of 0.0 is assigned. Constants may also be a function of other constants. These initial conditions are processed once as the first step after the model parsing. All constants have global scope in the model.

p = m.Param([value], [name])

Parameters are values that are nominally fixed at initial values but can be changed with input data, by the user, or can become calculated by the optimizer to minimize an objective function if they are indicated as decision variables. Parameters are values that represent model inputs, fixed constants, or measurements that may change over time. Parameters are not modified by the solver as it searches for a solution but they can be upgraded to an FV or MV as a decision variable for the optimizer. As a parameter, it does not contribute to the number of degrees of freedom (DOF):

```
p = m.Param(value=[0,0.1,0.2])
```

The parameters may be defined in one section or in multiple declarations throughout the model. Parameter initialization is performed sequentially, from top to bottom. If a parameter does not have an initial value given, a default value of 0.0 is assigned. Parameters may also be a function of other parameters or variable initial conditions. These initial conditions are processed once as the first step after the model parsing. All parameters have global scope in the model.

v = m.Var([value], [lb], [ub], [integer=False], [fixed_initial=True], [name])

Variables are always calculated values as determined by the set of equations. Some variables are either measured and/or controlled to a desired target value. Variables are modified by the solver as it searches for a solution. Each additional variable adds a decision (degree of freedom) to the problem. The following is an example of declaring an integer variable (0,1,2,...) that is constrained to be between 0 and 10 with a default value of 2:

```
v = m.Var(2,lb=0,ub=10,integer=True)
```

The variables may be defined in one section or in multiple declarations throughout the model. Variable initialization is performed sequentially, from top to bottom. If a variable does not have an initial value given, a default value of 0.0 is assigned. Variables may also be initialized from parameters or variable initial conditions. These initial conditions are processed once as the first step after the model parsing. All variables have global scope in the model.

fv = m.FV([value], [lb], [ub], [integer=False], [fixed_initial=True], [name])

Fixed Values or Feedforward Variables (FVs) are model coefficients that change to fit process data or minimize an objective function. These parameters can change the behavior and structure of the model. An FV has a single value over all time points for dynamic problems. It also has a single value when fitting a model to many data points, such as with steady state regression (IMODE=2). An FV is defined with a starting value of 3 and constrained between 0 and 10. The STATUS option set to 1 tells the optimizer that it can be adjusted to minimize the objective:

```
fv = m.FV(3,lb=0,ub=10)
fv.STATUS = 1
```

mv = m.MV([value], [lb], [ub], [integer=False], [fixed_initial=True], [name])

Manipulated variables (MVs) are decision variables for an estimator or controller. These decision variables are adjusted by the optimizer to minimize an objective function at every time point or with every data set. Unlike FVs, MVs may have different values at the discretized time points. An MV is defined with a starting value of 4 and constrained between 5 and 10. The STATUS option set to 1 tells the optimizer that it can be adjusted to minimize the objective:

```
mv = m.MV(4,lb=5,ub=10)
mv.STATUS = 1
```

sv = m.SV([value], [lb], [ub], [integer=False], [fixed_initial=True], [name])

State variables (SVs) are an upgraded version of a regular variable (m.Var) with additional logic to implement simple feedback and adjust the initial condition in dynamic simulations, estimators, or controllers. State variables may have upper and lower constraints but these should be used with caution to avoid an infeasible solution. A state variable is uninitialized (default=0) but is updated with a measurement of 6:

```
sv = m.SV()
sv.FSTATUS = 1
sv.MEAS = 6
```

cv = m.CV([value], [lb], [ub], [integer=False], [fixed_initial=True], [name])

Controlled variables are model variables that are included in the objective of a controller or optimizer. These variables are controlled to a range, maximized, or minimized. Controlled variables may also be measured values that are included for data reconciliation. State variables may have upper and lower constraints but these should be used with caution to avoid an infeasible solution. A controlled variable in a model predictive control application is given a default value of 7 with a setpoint range of 30 to 40:

```
cv = m.CV(7)
cv.STATUS = 1
cv.SPHI = 40
cv.SPLO = 30
```

i = m.Intermediate(equation, [name])

Intermediates are explicit equations where the variable is set equal to an expression that may include constants, parameters, variables, or other intermediate values that are defined previously. Intermediates are not implicit equations but are explicitly calculated with each model function evaluation. An intermediate variable is declared as the product of parameter p and variable v:

```
i = m.Intermediate(p*v)
```

Intermediate variables are useful to decrease the complexity of the model. These variables store temporary calculations with results that are not reported in the final solution reports. In many models, the temporary variables

outnumber the regular variables by many factors. This model reduction often aides the solver in finding a solution by reducing the problem size.

The intermediate variables may be defined in one section or in multiple declarations throughout the model. Intermediate variables are parsed sequentially, from top to bottom. To avoid inadvertent overwrites, intermediate variable can be defined once. In the case of intermediate variables, the order of declaration is critical. If an intermediate is used before the definition, an error reports that there is an uninitialized value.

The intermediate variables are processed before the implicit equation residuals, every time the solver requests model information. As opposed to implicitly calculated variables, the intermediates are calculated repeatedly and substituted into other intermediate or implicit equations.

eq = m.Equation(equation)

Add a constraint *equation* built from GEKKO Parameters, Variables and Intermediates, and python scalars. Valid operators include python math and comparisons (+,-,*,/,**,==,<,>). Available functions are listed below in *Equation Functions*.

[eqs] = m.Equations(equations)

Accepts a list or array of equations.

classmethod m.**Obj**(*obj*)

The problem objective to minimize. If multiple objective are provided, they are summed.

classmethod m.**Minimize**(*obj*)

The problem objective to minimize. The *m.Minimize(obj)* function is the same as *m.Obj(obj)* but it improves the optimization problem understanding by specifying that the objective function is minimized.

classmethod m.**Maximize**(*obj*)

The problem objective to maximize. This is the same as function *m.Obj(-obj)* or *m.Minimize(-obj)*. Multiplying *-1* converts an objective function from a maximization to a minimization problem. All optimization problems are converted to minimize form for solution by the solvers. As such, *m.options.OBJFCNVAL* reports the minimized result with a successful solution. Multiple objectives can be part of a single optimization problem. Multiple objectives are added together to create an overall objective:

```
m.Maximize(revenue)          # maximize revenue
m.Minimize(operating_cost)   # minimize operating costs
m.Minimize(10*emissions)     # increase importance of minimizing emissions by 10x
```

m.time

Sets the time array indicating the discrete elements of time discretization for dynamic modes (*IMODE > 3*). Accepts a python list or a numpy array.

classmethod dt()

Differential equations are specified by differentiating a variable with the *dt()* method or *dt* property. For example, velocity *v* is the derivative of position *x*:

```
m.Equation( v == x.dt() )  # derivative (dx/dt) with parenthesis
m.Equation( v == x.dt )    # derivative (dx/dt) without parenthesis also okay
```

Discretization is determined by the model *time* attribute. For example, *m.time = [0,1,2,3]* will discretize all equations and variable at the 4 points specified. Time or space discretization is available with Gekko, but not both. If the model contains a partial differential equation, the discretization in the other dimensions is performed with Gekko array operations as shown in the hyperbolic and parabolic PDE Gekko examples.

a = m.Array(type,dimension,args)**

Create an n-dimensional array (as defined in tuple input *dimension*) of GEKKO variables of type *type* . The

optional keyword arguments (***args*) are applied to each element of the array. The following example demonstrates the use of a 3x2 Array, a Parameter, Intermediates, and an Objective. The array values are initialized to 2.0 and bounds are set to -10.0 to 10.0:

```
from gekko import GEKKO
m = GEKKO()
# variable array dimension
n = 3 # rows
p = 2 # columns
# create array
x = m.Array(m.Var,(n,p))
for i in range(n):
  for j in range(p):
            x[i,j].value = 2.0
            x[i,j].lower = -10.0
            x[i,j].upper = 10.0
# create parameter
y = m.Param(value = 1.0)
# sum columns
z = [None]*p
for j in range(p):
   z[j] = m.Intermediate(sum([x[i,j] for i in range(n)]))
# objective
m.Obj(sum([ (z[j]-1)**2 + y for j in range(p)]))
# minimize objective
m.solve()
print('x:', x)
print('z:', z)
```

classmethod m.**solve**(*disp=True*, *debug=False*)

Solve the optimization problem.

This function has these substeps:

- Validates the model and write .apm file
- Validate and write .csv file
- Write options to .dbs file
- Solve the problem using apm.exe
- Load results into python variables.

If *disp* is *True*, APM and solve output are printed.

If *debug* is *True*, variable names are checked for problems, tuning parameters are checked for common errors, and user-defined input options are compared against options used by APM. This is useful in debugging strange results.

If *GUI* is *True*, the results of this solve are sent to the GUI. If the GUI is not open yet, the GUI object is created, the server is spawned and the browser client is launched.

classmethod m.**Connection**(*var1*, *var2*, *pos1=None*, *pos2=None*, *node1='end'*, *node2='end'*)

var1 must be a GEKKO variable, but *var2* can be a static value. If *pos1* or *pos2* is not *None*, the associated var must be a GEKKO variable and the position is the (0-indexed) time-discretized index of the variable.

Connections are processed after the parameters and variables are parsed, but before the initialization of the values. Connections are the merging of two variables or connecting specific nodes of a discretized variable. Once the

variable is connected to another, the variable is only listed as an alias. Any other references to the connected value are referred to the principal variable (*var1*). The alias variable (*var2*) can be referenced in other parts of the model, but will not appear in the solution files.

The position is 0-index and the last position is *len(m.time)-1*. Alternatively, the *end* string can be used for either the position or node. Additional documentation is available in the APMonitor documentation on Nodes.

classmethod `m.fix`(*var*, *val=None*, *pos=None*)

Fix a variable at a specific value so that the solver cannot adjust the value:

```
fix(var,val=None,pos=None)
```

Inputs:

- var = variable to fix
- val = specified value or None to use default
- pos = position within the horizon or None for all

The `var` variable must be a Gekko Parameter or Variable. When `val==None`, the current default value is retained. When `pos==None`, the value is fixed over all horizon nodes.

The `fix` function calls the `Connection` function with `var2` as a static value (`val`) and adds the `fixed` specification.

classmethod `m.fix_initial`(*var*, *value=None*)

Fix a variable at the initial condition so that the solver cannot adjust the value:

```
fix_initial(var,value=None)
```

Inputs:

- var = variable to fix at initial condition
- val = specified value or None to use default

classmethod `m.fix_final`(*var*, *value=None*)

Fix a variable at the final time point in the horizon so that the solver cannot adjust the value:

```
fix_final(var,value=None)
```

Inputs:

- var = variable to fix at the final time point
- val = specified value or None to use default

classmethod `m.free`(*var*, *pos=None*)

Free a variable at a specific position so that the solver can adjust the value:

```
free(var,pos=None)
```

Inputs:

- var = variable to free
- pos = position within the horizon or `None` for all

The `var` variable must be a Gekko Parameter or Variable. When `pos==None`, the value is calculated over all horizon nodes.

The `free` function calls the `Connection` function with `var2` with the string `calculated`.

classmethod m.**free_initial**(*var*)

Free a variable at the initial condition so that the solver can adjust the value:

```
free_initial(var)
```

Inputs:

- var = variable to free at initial condition

classmethod m.**free_final**(*var*)

Free a variable at the final time point in the horizon so that the solver can adjust the value:

```
free_final(var)
```

Inputs:

- var = variable to free at the final time point

m.solver_options

A list of strings to pass to the solver; one string for each option name and value. For example:

```
m = GEKKO()
m.solver_options = ['max_iter 100','max_cpu_time 100']
```

5.9.2 Equation Functions

Special function besides algebraic operators are available through GEKKO functions. These must be used (not numpy or other equivalent functions):

classmethod m.**sin**(*other*)

classmethod m.**cos**(*other*)

classmethod m.**tan**(*other*)

classmethod m.**asin**(*other*)

classmethod m.**acos**(*other*)

classmethod m.**atan**(*other*)

classmethod m.**sinh**(*other*)

classmethod m.**cosh**(*other*)

classmethod m.**tanh**(*other*)

classmethod m.**exp**(*other*)

classmethod m.**log**(*other*)

classmethod m.**log10**(*other*)

classmethod m.**sqrt**(*other*)

classmethod m.**erf**(*other*)

classmethod m.**erfc**(*other*)

classmethod m.**sigmoid**(*other*)

5.9.3 Logical Functions

Traditional logical expressions such as if statements cannot be used in gradient based optimization because they create discontinuities in the problem derivatives. The logical expressions built into Gekko provide a workaround by either using MPCC formulations (Type 2), or by introducing integer variables (Type 3). Additionally, all Type 3 functions require a mixed integer solver such as APOPT (SOLVER=1) to solve, and **Gekko changes the solver to APOPT automatically if these functions are found in a model.** See additional information on logical conditions in optimization and if statements in Gekko for additional examples and explanations of the methods.

y = abs2(x)

Generates the absolute value with continuous first and second derivatives.

The traditional method for absolute value (abs) has a point that is not continuously differentiable at an argument value of zero and can cause a gradient-based optimizer to fail to converge. The abs2 function creates a continuous differentiable function with a complementarity constraint:

```
Usage: y = m.abs2(x)
```

Input:

GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = abs3(x)

Generates the absolute value with a binary switch.

The traditional method for absolute value (abs) has a point that is not continuously differentiable at an argument value of zero and can cause a gradient-based optimizer to fail to converge. The abs3 introduces a new binary variable and equation that requires a mixed integer solver, such as APOPT:

```
Usage: y = m.abs3(x)
```

Input:

GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = if2(condition,x1,x2)

IF conditional with complementarity constraint switch variable.

The traditional method for IF statements is not continuously differentiable and can cause a gradient-based optimizer to fail to converge. The if2 method uses a complementarity condition to determine whether y=x1 (when condition<0) or y=x2 (when condition>=0):

```
Usage: y = m.if2(condition,x1,x2)
```

Inputs:

condition: GEKKO variable, parameter, or expression

x1 and x2: GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = x1 when condition<0

y = x2 when condition>0

When condition~0, the solution may be *x1*, *x2*, or a linear combination of the two with *if2*. Use *if3* to avoid the linear combination of the two values *x1* and *x2*.

Example usage:

```
import numpy as np
from gekko import gekko
m = gekko()
x1 = m.Const(5)
x2 = m.Const(6)
t = m.Var(0)
m.Equation(t.dt()==1)
m.time = np.linspace(0,10)
y = m.if2(t-5,x1,x2)
m.options.IMODE = 6
m.solve()
import matplotlib.pyplot as plt
plt.plot(m.time,y)
plt.show()
```

y = if3(condition,x1,x2)

IF conditional with a binary switch variable.

The traditional method for IF statements is not continuously differentiable and can cause a gradient-based optimizer to fail to converge. The if3 method uses a binary switching variable to determine whether y=x1 (when condition<0) or y=x2 (when condition>=0):

```
Usage: y = m.if3(condition,x1,x2)
```

Inputs:

condition: GEKKO variable, parameter, or expression

x1 and x2: GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = x1 when condition<0

y = x2 when condition>=0

When condition~0, the solution may be x1 or x2. Gekko computes a numerical solution with a solver tolerance that is 1e-6 by default.

Example usage:

```
import numpy as np
import matplotlib.pyplot as plt
from gekko import GEKKO
m = GEKKO(remote=False)
p = m.Param()
y = m.if3(p-4,p**2,p+1)

# solve with condition<0
p.value = 3
m.solve(disp=False)
print(y.value)
```

(continues on next page)

(continued from previous page)

```
# solve with condition>=0
p.value = 5
m.solve(disp=False)
print(y.value)
```

y = max2(x1,x2)

Generates the maximum value with continuous first and second derivatives.

The traditional method for max value (max) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge. The max2 function creates the complementarity constraints with continuous derivatives:

```
Usage: y = m.max2(x1,x2)
```

Input:

GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = max3(x1,x2)

Generates the maximum value with a binary switch variable.

The traditional method for max value (max) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge:

```
Usage: y = m.max3(x1,x2)
```

Input:

GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = min2(x1,x2)

Generates the minimum value with continuous first and second derivatives.

The traditional method for min value (min) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge:

```
Usage: y = m.min2(x1,x2)
```

Input:

GEKKO variable, parameter, or expression

Output:

GEKKO variable

y = min3(x1,x2)

Generates the maximum value with a binary switch variable.

The traditional method for max value (max) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge. The max3 function creates a new binary variable that must be solved with a mixed integer solver, such as APOPT:

```
Usage: y = m.max3(x1,x2)
```

Input:
GEKKO variable, parameter, or expression

Output:
GEKKO variable

classmethod pwl(*x*, *y*, *x_data*, *y_data*, *bound_x=False*)

Generate a 1d piecewise linear function with continuous derivatives from vectors of x and y data that link to GEKKO variables x and y with a constraint that y=f(x) with piecewise linear units.

Inputs:

- x: GEKKO parameter or variable
- y: GEKKO variable
- x_data: array of x data
- y_data: array of y data that matches x_data size
- bound_x: boolean to state if x should be bounded at the upper and lower bounds of x_data to avoid extrapolation error of the piecewise linear region.

Output: none

y = sos1(values)

Special Ordered Set (SOS), Type-1.
Chose one from a set of possible numeric values that are mutually exclusive options. The SOS is a combination of binary variables with only one that is allowed to be non-zero.

values = [y0,y1,…,yn]

b0 + b1 + … + bn = 1, 0<=bi<=1

y = y0*b0 + y1*b1 + … + yn*bn

The binary variable (bi) signals which option is selected:

```
Usage: y = m.sos1(values)
```

Input:
values (possible y numeric values as a list)

Output:
y (GEKKO variable)

Example usage:

```
from gekko import GEKKO
m = GEKKO()
y = m.sos1([19.05, 25.0, 29.3, 30.2])
m.Obj(y) # select the minimum value
m.solve()
print(y.value)
```

y = sign2(x)

Generates the sign of an argument with MPCC.
The traditional method for signum (sign) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge:

```
Usage: y = m.sign2(x)
```

Input:
GEKKO variable, parameter, or expression

Output:
GEKKO variable

y = sign3(x)

Generates the sign of an argument with binary switching variable.
The traditional method for signum (sign) is not continuously differentiable and can cause a gradient-based optimizer to fail to converge:

```
Usage: y = m.sign3(x)
```

Input:
GEKKO variable, parameter, or expression

Output:
GEKKO variable

5.9.4 Pre-built Objects

Pre-built objects are common model constructs that facilitate data analysis, regression, and model building. Additional object documentation gives insight on the supported objects as well as examples for building other custom objects or libraries. Other object libraries are the *Chemical Library* and *Deep Learning* Library. Additional object libraries are under development.

y,u = arx(p,y=None,u=None)

Build a GEKKO model from ARX representation.

Inputs:

- parameter dictionary p[‘a’], p[‘b’], p[‘c’]
- a (coefficients for a polynomial, na x ny)
- b (coefficients for b polynomial, ny x (nb x nu))
- c (coefficients for output bias, ny)
- y (Optioanl: Controlled Variable Array)
- u (Optional: Manipulated Variable Array)

x = axb(A,b,x=None,etype='=',sparse=False)

Create Ax=b, Ax<b, Ax>b, Ax<=b, or Ax>=b models:

```
Usage: x = m.axb(A,b,x=None,etype='=,<,>,<=,>=',sparse=[True,False])
```

Inputs:

- A = numpy 2D array or list in dense or sparse form
- b = numpy 1D array or list in dense or sparse form
- x = 1D array of gekko variables (optional). If None on entry then the array is created and returned.
- etype = ['=',```’<’,’>’,’>=’,’<=’```] for equality or inequality form

- sparse = True if data is in sparse form, otherwise dense

sparse matrices are stored in COO form with [row,col,value] with starting index 1 for optional matrix A and in [row,value] for * vector b

Output:
GEKKO variables x

classmethod bspline(*x*, *y*, *z*, *x_data*, *y_data*, *z_data*, *data=True*, *kx=3*, *ky=3*, *sf=None*)

Generate a 2D Bspline with continuous first and seconds derivatives from 1-D arrays of x_data and y_data coordinates (in strictly ascending order) and 2-D z data of size (x.size,y.size). GEKKO variables x, y and z are linked with function z=f(x,y) where the function f is a bspline.

Inputs:

x,y = independent Gekko parameters or variables as predictors for z

z = dependent Gekko variable with z = f(x,y)

If data is True (default) then the bspline is built from data

x_data = 1D list or array of x values, size (nx)

y_data = 1D list or array of y values, size (ny)

z_data = 2D list or matrix of z values, size (nx,ny)

If data is False then the bspline knots and coefficients are loaded

x_data = 1D list or array of x knots, size (nx)

y_data = 1D list or array of y knots, size (ny)

z_data = 2D list or matrix of c coefficients, size (nx-kx-1)*(ny-ky-1)

kx = degree of spline in x-direction, default=3

ky = degree of spline in y-direction, default=3

sf = smooth factor (sf), only for data=True

sf controls the tradeoff between smoothness and closeness of fit. If *sf* is small, the approximation may follow too much signal noise. If *sf* is large, the approximation does not follow the general trend. A proper *sf* depends on the data and level of noise when *sf* is None a default value of *nx*ny*(0.1)**2* is used where 0.1 is the approximate statistical error of each point the *sf* is only used when constructing the *bspline* (*data=True*)

Outputs:

None

Generate a 2d Bspline with continuous first and seconds derivatives from 1-D arrays of x_data and y_data coordinates (in strictly ascending order) and 2-D z data of size (x.size,y.size). GEKKO variables x, y and z are linked with function z=f(x,y) where the function f is a bspline.

c = cov(x,y=None,ddof=1,name=None)

Covariance (scalar or matrix) built with GEKKO equations.

The covariance between two vectors is:

```
cov(x,y) = sum_{i=1..N} (x_i - mean(x)) (y_i - mean(y))/(N - ddof)
```

Usage:

```
c = m.cov(x, y, ddof=1)  # sample covariance (default)
c = m.cov(x, y, ddof=0)  # population covariance

C = m.cov(X, ddof=1)     # covariance matrix from list of vectors
                                # X = [x1, x2, ..., xp]
```

Inputs:

x:
1D vector (list/tuple/np.ndarray) if y is provided, or a 2D collection of vectors (list of vectors or 2D ndarray) if y is None.

y:
Optional 1D vector, same length as x when x is 1D.

ddof:
Delta degrees of freedom. Use ddof=0 for population covariance or ddof=1 for sample covariance. Must satisfy 0 <= ddof < N.

name:
Optional base name for output variable(s). For covariance matrices, entries are named like name_i_j.

Output:

If y is provided:
GEKKO variable c (scalar covariance)

If y is None and x is a list of vectors:
Covariance matrix C as a list-of-lists of GEKKO variables

Example usage (scalar covariance):

```
import numpy as np
from gekko import GEKKO

m = GEKKO(remote=False)

xi = [2.1,2.5,3.6,4.0]
yi = [8,10,12,14]

x = [m.Param(value=v) for v in xi]
y = [m.Param(value=v) for v in yi]

c_pop  = m.cov(x, y, ddof=0, name='cov_pop')
c_samp = m.cov(x, y, ddof=1, name='cov_samp')

m.solve(disp=False)

print('Gekko Solution')
print('population cov:', c_pop.value[0])
print('sample cov:', c_samp.value[0])

print('Numpy Solution')
c_pop_np  = np.cov(xi, yi, ddof=0)[0,1]
c_samp_np = np.cov(xi, yi, ddof=1)[0,1]
```

(continues on next page)

(continued from previous page)

```
print('population cov:', c_pop_np)
print('sample cov:', c_samp_np)
```

Example usage (covariance matrix):

```
import numpy as np
from gekko import GEKKO

m = GEKKO(remote=False)

Xv = [[0.01, 0.02, -0.01, 0.03],
      [0.00, 0.01,  0.02, 0.01],
      [0.02, 0.00,  0.01, 0.00]]

X = m.Array(m.Param,(3,4))
for i in range(3):
    for j in range(4):
        X[i][j].value = Xv[i][j]

C = m.cov(X, ddof=1, name='C')

m.solve(disp=False)

# C is a 3x3 list-of-lists of GEKKO Vars
print('Gekko Solution')
for i in range(3):
    print(C[i][0:])

print('Numpy Solution')
print(np.cov(Xv))
```

classmethod cspline(*x*, *y*, *x_data*, *y_data*, *bound_x=False*)

Generate a 1d cubic spline with continuous first and seconds derivatives from arrays of x and y data which link to GEKKO variables x and y with a constraint that y=f(x).

Inputs:

x: GEKKO variable

y: GEKKO variable

x_data: array of x data

y_data: array of y data that matches x_data

bound_x: boolean to state that x should be bounded at the upper and lower bounds of x_data to avoid extrapolation error of the cspline.

classmethod delay(*u*, *y*, *steps=1*)

Build a delay with number of time steps between input (u) and output (y) with a discrete time series model.

Inputs:

u: delay input as a GEKKO variable

y: delay output as a GEKKO variable

steps: integer number of steps (default=1)

classmethod integral(*x*)

Integral of a constant, parameter, intermediate, variable, or expression.

Inputs:

x: input variable, parameter, or expression

y: GEKKO variable that is the integral up to that time in the horizon

Example:

```
from gekko import GEKKO
import numpy as np
m = GEKKO()
m.time = np.linspace(0,2,5)
m.options.IMODE=4; m.options.NODES=6
x = m.Var(5)
m.Equation(x.dt()==-x)
y = m.Var(0)
m.Equation(y==m.integral(x))
m.options.SOLVER=1
m.solve()
```

classmethod periodic(*v*)

Makes the variable argument periodic by adding an equation to constrains v[end] = v[0]. This does not affect the default behavior of fixing initial conditions (v[0]).

x = qobj(b,A=[],x=None,otype='min',sparse=False)

Create quadratic objective = 0.5 x^T A x + c^T x:

```
Usage: x = m.qobj(c,Q=[2d array],otype=['min','max'],sparse=[True,False])
```

Input:

- b = numpy 1D array or list in dense or sparse form
- A = numpy 2D array or list in dense or sparse form
- x = array of gekko variables (optional). If None on entry then the array is created and returned.
- sparse = True if data is in sparse form, otherwise dense

sparse matrices are stored in COO form with [row,col,value] with starting index 1 for optional matrix A and in [row,value] for vector b sparse matrices must have 3 columns

Output:

GEKKO variables x

x,y,u = state_space(A,B,C,D=None,E=None,discrete=False,dense=False)

For State Space models, input SS matricies A,B,C, and optionally D and E. Returns a GEKKO array of states (SV) *x*, array of outputs (CV) *y* and array of inputs (MV) *u*. A,B,C,D, and E must be 2-dimensional matricies of the appropriate size.

The *discrete* Boolean parameter indicates a discrete-time model, which requires constant time steps and 2 *NODES*. The *dense* Boolean parameter indicates if A,B,C,D, and E should be written as dense or sparse matrices. Sparse matricies will be faster unless it is known that the matricies are very dense. See examples of discrete time simulation and model predictive control with state space models:

```
import numpy as np
from gekko import GEKKO
A = np.array([[-.003, 0.039, 0, -0.322],
              [-0.065, -0.319, 7.74, 0],
              [0.020, -0.101, -0.429, 0],
              [0, 0, 1, 0]])
B = np.array([[0.01, 1, 2],
              [-0.18, -0.04, 2],
              [-1.16, 0.598, 2],
              [0, 0, 2]]
            )
C = np.array([[1, 0, 0, 0],
              [0, -1, 0, 7.74]])
m = GEKKO()
x,y,u = m.state_space(A,B,C,D=None)
```

y = sum(x)

Summation using APM object.:

```
Usage: y = m.sum(x)
```

Input:

Numpy array or List of GEKKO variables, parameters, constants, intermediates, or expressions

Output:

GEKKO variable

y,p,K = sysid(t,u,y,na=1,nb=1,nk=0,shift='calc',scale=True,diaglevel=0,pred='model', objf=100)

Identification of linear time-invariant models:

```
y,p,K = sysid(t,u,y,na,nb,shift=0,pred='model',objf=1)
```

Input:

- t = time data
- u = input data for the regression
- y = output data for the regression
- na = number of output coefficients (default=1)
- nb = number of input coefficients (default=1)
- nk = input delay steps (default=0)
- shift (optional) with `'none'` (no shift), `'init'` (initial pt), `'mean'` (mean center), or `'calc'` (calculate c)
- scale (optional) scale data to between zero to one unless data range is already less than one
- pred (option) `'model'` for output error regression form, implicit solution. Favors an unbiased model prediction but can require more time to compute, especially for large data sets. `'meas'` for ARX regression form, explicit solution. Computes the coefficients of the time series model with an explicit solution.

- objf = Objective scaling factor, when `pred='model'`: minimize objf*(model-meas)**2 + 1e-3 * (a^2 + b^2 + c^2) and when `pred='meas'`: minimize (model-meas)**2
- diaglevel sets display solver output and diagnostics (0-6)

Output:

- ypred (predicted outputs)
- p as coefficient dictionary with keys `'a','b','c'`
- K gain matrix

An example of system identification with 2 MVs and 2 CVs with data from the Temperature Control Lab is shown below:

```
from gekko import GEKKO
import pandas as pd
import matplotlib.pyplot as plt
url = 'http://apmonitor.com/do/uploads/Main/tclab_dyn_data2.txt'
data = pd.read_csv(url)
t = data['Time']
u = data[['H1','H2']]
y = data[['T1','T2']]
m = GEKKO()
na = 2 # output coefficients
nb = 2 # input coefficients
yp,p,K = m.sysid(t,u,y,na,nb,diaglevel=1)
plt.figure()
plt.subplot(2,1,1)
plt.plot(t,u)
plt.subplot(2,1,2)
plt.plot(t,y)
plt.plot(t,yp)
plt.xlabel('Time')
plt.show()
```

y = vsum(x)

Summation of variable in the data or time direction. This is similar to an integral but only does the summation of all points, not the integral area that considers time intervals. Below is an example of `vsum` with `IMODE=2` for a regression problem:

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt
xm = np.array([0,1,2,3,4,5])
ym = np.array([0.1,0.2,0.3,0.5,0.8,2.0])
m = GEKKO()
x = m.Param(value=xm)
a = m.FV()
a.STATUS=1
y = m.CV(value=ym)
y.FSTATUS=1
z = m.Var()
m.Equation(y==0.1*m.exp(a*x))
m.Equation(z==m.vsum(x))
```

(continues on next page)

(continued from previous page)

```
m.options.IMODE = 2
m.solve()
```

5.9.5 Internal Methods

These are the major methods used internal by GEKKO. They are not intended for external use, but may prove useful for highly customized applications.

static build_model(*self*)

Write the .apm model file for the executable to read. The .apm file contains all constants, parameters, variables, intermediates, equations and objectives. Single values and/or initializations, along with variable bounds, are passed throught the .apm model file.

static write_csv()

Any array values are passed through the csv, including variable initializations. If `imode > 3` then `time` must be discretized in the csv.

static generate_overrides_dbs_file()

All global and local variable options are listed in the overrides database file.

static load_json()

Reads back global and variable options from the options.json file. Stores output and input/output options to the associated variable.

static load_results()

The executable returns variable value results in a json. This function reads the json and loads the results back into local python variables.

static verify_input_options()

Called when optional *solve* argument *verify_input=True*. Compares input options of the model and variables from GEKKO to those reported by APM to find discrepencies.

5.9.6 Internal Attributes

These are GEKKO model attributes used internally. They are not intended for external use, but may prove useful in advanced applications.

server

String representation of the server url where the model is solved. The default is 'http://byu.apmonitor.com'. This is set by the optional argument *server* when intializing a model.

remote

Boolean that determines if solutions are offloaded to the server or executed locally.

id

_constants

A python list of pointers to GEKKO Constants attributed to the model.

_parameters

A python list of pointers to GEKKO Parameters, FVs and MVs attributed to the model.

_variables

A python list of pointers to GEKKO Variables, SVs and CVs attributed to the model.

_intermediates

A python list of pointers to GEKKO Intermediate variables attributed to the model.

_inter_equations

A python list of the explicit intermediate equations. The order of this list must match the order of intermediates in the *intermediates* attribute.

_equations

A python list of equations

_objectives

A python list of objective.

_connections

A python list of connections

csv_status

Set to 'generated' if any time, parameter or variable data is communicated through the csv file. Otherwise set to 'none'.

model_name

The name of the model as a string. Used in local temporary file name and application name for remote solves. This is set by the optional argument *name* when intializing a model. Default names include the model *id* attribute to maintain unique names.

_path

The absolute path of the temporary file used to store all input/output files for the APMonitor executable.

5.9.7 Clean Up an Application

Gekko creates a temporary folder *m.path* for each model instance that is created with *m.GEKKO()*. With a local solution *m.GEKKO(remote=False)*, all processing is done in the local folder with input and output files. The folder is viewable with *m.open_folder()*. Linux, MacOS, and Windows manage the temporary folder differently. Windows does not automatically delete temporary folder files while Linux and MacOS remove the files on reboot or after a certain period of time. Gekko deletes the temporary folder and any files associated with the application with the *cleanup* command.

classmethod `m.cleanup()`

Remove temporary folder and application files:

```
Usage: m.cleanup()
```

Users that are processing many thousands of large jobs may need to use the *m.cleanup()* function to free up disk space. The function is not automatically called because of several applications (e.g. Model Predictive Control) that repeatedly solve with updated inputs or objectives.

5.10 Deep Learning

GEKKO specializes in a optimization and control. The **brain** module extends GEKKO with machine learning objects to facilitate the building, fitting, and validation of deep learning methods.

5.10.1 Deep Learning Properties

GEKKO specializes in a unique subset of machine learning. However, it can be used for various types of machine learning. This is a module to facilitate Artificial Neural Networks in GEKKO. Most ANN packages use gradient decent optimization. The solvers used in GEKKO use more advanced techniques than gradient decent. However, training neural networks may require extremely large datasets. For these

large problems, gradient decent does prove more useful because of the ability to massively parallelize. Nevertheless, training in GEKKO is available for cases where the data set is of moderate size, for combined physics-based and empirical modeling, or for other predictive modeling and optimization applications that warrant a different solution strategy.

b = brain.Brain(m=[],remote=True,bfgs=True,explicit=True):

Creates a new *brain* object as a GEKKO model *m*. Option *remote* specifies if the problem is solved locally or remotely, *bfgs* uses only first derivative information with a BFGS update when *True* and otherwise uses first and second derivatives from automatic differentiation, *explicit* calculates the layers with *Intermediate* equations instead of implicit *Equations*:

```
from gekko import brain
b = brain.Brain()
```

b.input_layer(size):

Add an input layer to the artificial neural network. The input layer *size* is equal to the number of features or predictors that are inputs to the network.:

```
from gekko import brain
b = brain.Brain()
b.input_layer(1)
```

b.layer(linear=0,relu=0,tanh=0,gaussian=0,bent=0,leaky=0,ltype='dense'):

Layer types: dense, convolution, pool (mean)

Activation options: none, softmax, relu, tanh, sigmoid, linear:

```
from gekko import brain
b = brain.Brain()
b.input_layer(1)
b.layer(linear=2)
b.layer(tanh=2)
b.layer(linear=2)
```

Each layer of the neural network may include one or multiple types of activation nodes. A typical network structure is to use a linear layer for the first internal and last internal layers with other activation functions in between.

b.output_layer(size,ltype='dense',activation='linear'):

Layer types: dense, convolution, pool (mean)

Activation options: none, softmax, relu, tanh, sigmoid, linear:

```
from gekko import brain
b = brain.Brain()
b.input_layer(1)
b.layer(linear=2)
b.layer(tanh=2)
b.layer(linear=2)
b.output_layer(1)
```

b.learn(inputs,outputs,obj=2,gap=0,disp=True):

Make the *brain* **learn** by adjusting the network weights to minimize the loss (objective) function. Give inputs as (n)xm, where n = input layer dimensions, m = number of datasets:

```
from gekko import brain
import numpy as np
b = brain.Brain()
b.input_layer(1)
b.layer(linear=2)
b.layer(tanh=2)
b.layer(linear=2)
b.output_layer(1)
x = np.linspace(0,2*np.pi)
y = np.sin(x)
b.learn(x,y)
```

Give outputs as (n)xm, where n = output layer dimensions, m = number of datasets. Objective can be 1 (l1 norm) or 2 (l2 norm). If obj=1, gap provides a deadband around output matching.

b.shake(percent):

Neural networks are non-convex. Some stochastic shaking can sometimes help bump the problem to a new region. This function perturbs all weights by +/-percent their values.

b.think(inputs):

Predict output based on *inputs*. The *think* method is used after the network is trained. The trained parameter weights are used to calculates the new network outputs based on the input values.:

```
from gekko import brain
import numpy as np
import matplotlib.pyplot as plt
b = brain.Brain()
b.input_layer(1)
b.layer(linear=2)
b.layer(tanh=2)
b.layer(linear=2)
b.output_layer(1)
x = np.linspace(0,2*np.pi)
y = np.sin(x)
b.learn(x,y)
xp = np.linspace(-2*np.pi,4*np.pi,100)
yp = b.think(xp)
plt.figure()
plt.plot(x,y,'bo')
plt.plot(xp,yp[0],'r-')
plt.show()
```

5.11 Machine Learning

Gekko specializes in optimization, dynamic simulation, and control. The ML module in GEKKO interfaces compatible machine learning algorithms into the optimization suite to be used for data-based optimization. Trained models from *scikit-learn*, *gpflow*, *nonconformist*, and *tensorflow* are imported into Gekko for design optimization, model predictive control, and physics-informed hybrid modeling.

5.11.1 Machine Learning Interface models

These functions allows interfaces of various models into Gekko. They can be found in the ML subpackage of gekko, imported like so:

```
from gekko import ML
```

Model = ML.Gekko_GPR(model,Gekko_Model,modelType='sklearn',fixedKernel=True)

Convert a gaussian process model from *sklearn* into the Gekko package.

model: The first argument is the trained gaussian model, either from *sklearn* GaussianProcessRegressor or a model from *gpflow*. Custom kernels are not implemented, but all kernels and combinations of kernel in *sklearn* are allowed.

Gekko_Model: Gekko model (created by *GEKKO()*) that is appended with the new GPR model.

modeltype: sklearn indicates that the model is from scikit-learn, otherwise from gpflow. If it is not sklearn, it convert from gpflow to sklearn.

fixedKernel: conversion from gpflow to sklearn. If it is set to *True*, the kernel hyperparameters are set to fixed; otherwise, *False* allows the hyperparameters to be changed during training.

Model = ML.Gekko_SVR(model,Gekko_Model)

Imports an SVR model into Gekko.

model: Model must be a variant of sklearn *svm.SVR()* or *svm.NuSVR()*.

Gekko_Model: Gekko model (created by *GEKKO()*) that is appended with the new SVR model.

Model = ML.Gekko_NN_SKlearn(model,minMaxArray,Gekko_Model)

Import an sklearn Neural Network into Gekko.

model: model trained with the MLPRegressor function from sklearn.

minMaxArray: min-max array for scaling created by the custom min max scaler. This is necessary as neural networks often use a scaled dataset.

Gekko_Model: Gekko model (created by *GEKKO()*) that is appended with the new NN model.

Model = ML.Gekko_LinearRegression(model,Gekko_Model)

Import a trained linear regression model from sklearn.

model: trained model from sklearn as a Ridge Regression or Linear Regression model.

Gekko_Model: Gekko model (created by *GEKKO()*) that is appended with the new Linear Regression model.

Model = ML.Gekko_NN_TF(model,Gekko_Model)

Import a Tensorflow and Keras Neural Network into Gekko. A customized loss function must be used during training to calculate uncertainty.

model: trained model from the TensorFlow.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

Model = ML.Gekko_DecisionTree(model,gekkoModel,ifo=2,eps=1e-3)

Import a sklearn decision tree model (sklearn.tree.DecisionTreeRegressor) into Gekko. Tree-based Gekko models use if2 and if3 functions, defaulting the solver to APOPT. These functions may be inaccurate, so it may be good to test different 'ifo'/'eps' parameters to achieve desired accuracy. When predicting with tree models, pass in the argument 'return_proba' to return the probability of the selected class, or 'return_conds' to validate which tree leaf is active for the Gekko prediction.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

ifo: The 'if' Gekko function to use. ifo=2 uses GEKKO.if2(), ifo3 uses GEKKO.if3().

eps: An error term to add to the conditional logic to encourage the solver failing on tree conditional statements.

`Model = ML.Gekko_RandomForest(model,gekkoModel,ifo=2,eps=1e-3)`

Import a sklearn Random Forest model (sklearn.ensemble.RandomForestRegressor) into Gekko. For tree-based ensemble methods, too many base_estimators or excessively deep trees may cause the solver to stall or fail to converge. Tree-based Gekko models use if2 and if3 functions, defaulting the solver to APOPT. These functions may be inaccurate, so it may be good to test different 'ifo'/'eps' parameters to achieve desired accuracy. When predicting with tree models, pass in the argument 'return_proba' to return the probability of the selected class, or 'return_conds' to validate which tree leaf is active for the Gekko prediction.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

ifo: The 'if' Gekko function to use. ifo=2 uses GEKKO.if2(), ifo3 uses GEKKO.if3().

eps: An error term to add to the conditional logic to encourage the solver failing on tree conditional statements.

`Model = ML.Gekko_GradientBooster(model,gekkoModel,ifo=2,eps=1e-3)`

Import a sklearn Gradient Boosting Regressor model (sklearn.ensemble.GradientBoostingRegressor) into Gekko. For tree-based ensemble methods, too many base_estimators or excessively deep trees may cause the solver to stall or fail to converge. Tree-based Gekko models use if2 and if3 functions, defaulting the solver to APOPT. These functions may be inaccurate, so it may be good to test different 'ifo'/'eps' parameters to achieve desired accuracy. When predicting with tree models, pass in the argument 'return_proba' to return the probability of the selected class, or 'return_conds' to validate which tree leaf is active for the Gekko prediction.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

ifo: The 'if' Gekko function to use. ifo=2 uses GEKKO.if2(), ifo3 uses GEKKO.if3().

eps: An error term to add to the conditional logic to encourage the solver failing on tree conditional statements.

`Model = ML.Gekko_LinearTree(model,gekkoModel,ifo=2,eps=1e-3)`

Import a linear tree regressor model from the linear-tree package into Gekko. Tree-based Gekko models use if2 and if3 functions, defaulting the solver to APOPT. These functions may be inaccurate, so it may be good to test different 'ifo'/'eps' parameters to achieve desired accuracy. When predicting with tree models, pass in the argument 'return_proba' to return the probability of the selected class, or 'return_conds' to validate which tree leaf is active for the Gekko prediction.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

ifo: The 'if' Gekko function to use. ifo=2 uses GEKKO.if2(), ifo3 uses GEKKO.if3().

eps: An error term to add to the conditional logic to encourage the solver failing on tree conditional statements.

`Model = ML.Boootstrap(models,Gekko_Model)`

Perform an ensemble/bootstrap calculation method.

models: an array of models including GPR, SVR, and/or sklearn-NN models.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

Model = ML.Conformist(model,Gekko_Model,u)

Conformal prediction wrapper for the previous listed machine learning models.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

u: The uncertainty interval provided by split conformal prediction. For conformalized ensembles, a custom version of the Bootstrap() interface is needed.

Model = ML.Delta(model,Gekko_Model,X,s)

Delta uncertainty wrapper for interfaced models. For machine learning models, a least squares model is used as surrogate for uncertainty quantification.

model: trained model from sklearn.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

X: The design matrix for a least squares model / surrogate model.

s The approximation of the model error, or root mean square error.

prediction = Model.predict(xi,return_std=True):

For any model class built by the above functions, the function predict is called to generate a prediction. Some models may have multi-output (like neural networks) or additional return statements.

xi: input array. It must be the same shape as the features used to train the model. It can be scalar/array quantities, or it can be gekko variables as the input can be gekko variables. It allows optimization and control to be accessible to these models.

return_std: return standard deviation of prediction. For most models, this return 0 as this is not natively calculated. If the model is a gaussian model or is wrapped in one of the wrappers, then it provides an uncertainty. Some methods may increase runtime of the process, especially if the training set is large for the model.

model = Gekko_Scaled_Model(gmodel,scaler_x=None,scaler_y=None)

This interface wraps any gekko interface-model with scalers from sklearn. A scaler can be provided for the input and/or output. Sklearn's MinMaxScaler and StandardScaler are currently supported.

gmodel: a ML model (listed above) already interfaced into Gekko.

Gekko_Model: Gekko model solver object (created by *GEKKO()*).

scaler_x: sklearn scaler object for the input features.

scaler_y: sklearn scaler object for output features.

5.11.2 Example problem

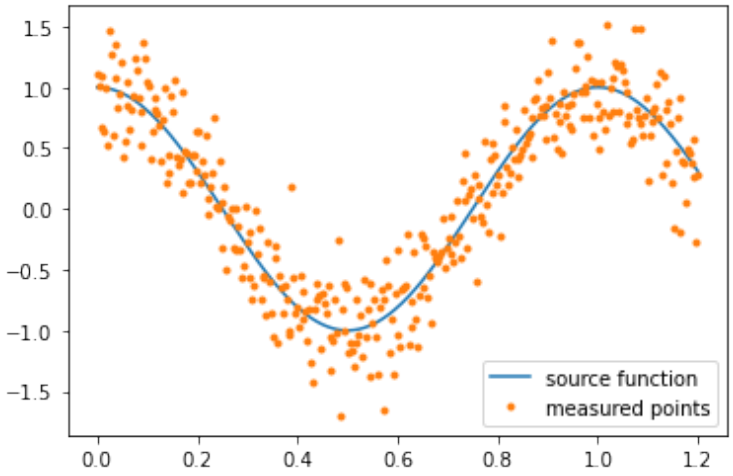
The example problem is a simple case study for the integration of Machine Learning models into Gekko. Noise is added to the data to represent measurement uncertainty and create a necessity for fitting a regression model to the data.

```
import numpy as np
import matplotlib.pyplot as plt

#Source function to generate data
def f(x):
    return np.cos(2*np.pi*x)

#represent noise from a data sample
```

(continues on next page)

(continued from previous page)

```
N = 350
xl = np.linspace(0,1.2,N)
noise = np.random.normal(0,.3,N)
y_measured = f(xl) + noise

plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
plt.legend()
plt.show()
```



Gekko's optimization functionality is used to find a minimum of this function.

```
from gekko import GEKKO
m = GEKKO()
x = m.Var(0,lb=0,ub=1)
y = m.Intermediate(m.cos(x*2*np.pi)) #function is used here
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solve Time:',m.options.SOLVETIME,'s')
```

```
solution: -1.0
x: 0.5
Gekko Solve Time: 0.0078999999996 s
```

If the original source function is unknown, but the data is available, data can be used to train machine learning models and then these trained models can be used to optimize the required function. In this case, the models are being used as the objective function, but they can be used as constraint functions as well. Currently, Gaussian Process Regression, Support Vector Machines, and Artificial Neural Networks from sklearn can be interfaced and integrated into Gekko. Below is a basic training script for each of the three models.

```
#Import the ML interface functions
from gekko.ML import Gekko_GPR,Gekko_SVR,Gekko_NN_SKlearn
from gekko.ML import Gekko_NN_TF,Gekko_LinearRegression
from gekko.ML import Bootstrap,Conformist,CustomMinMaxGekkoScaler
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
from sklearn.model_selection import train_test_split

#Training the Data and split it
data = pd.DataFrame(np.array([xl,y_measured]).T,columns=['x','y'])
features = ['x']
label = ['y']

train,test = train_test_split(data,test_size=0.2,shuffle=True)

#Training the models
import sklearn.gaussian_process as gp
from sklearn.neural_network import MLPRegressor
from sklearn import svm
from sklearn.metrics import r2_score
import tensorflow as tf

#GPR
k = gp.kernels.RBF() * gp.kernels.ConstantKernel() + gp.kernels.WhiteKernel()
gpr = gp.GaussianProcessRegressor(kernel=k,\
                            n_restarts_optimizer=10,\
                            alpha=0.1,\
                            normalize_y=True)
gpr.fit(train[features],train[label])
r2 = gpr.score(test[features],test[label])
print('gpr r2:',r2)

#SVR
svr = svm.SVR()
svr.fit(train[features],np.ravel(train[label]))
r2 = svr.score(test[features],np.ravel(test[label]))
print('svr r2:',r2)

#NNSK
s = CustomMinMaxGekkoScaler(data,features,label)
ds = s.scaledData()
mma = s.minMaxValues()

trains,tests = train_test_split(ds,test_size=0.2,shuffle=True)
hl= [25,15]
mlp = MLPRegressor(hidden_layer_sizes= hl, activation='relu',
                   solver='adam', batch_size = 32,
                   learning_rate = 'adaptive',learning_rate_init = .0005,
                   tol=1e-6 ,n_iter_no_change = 200,
                   max_iter=12000)
mlp.fit(trains[features],np.ravel(trains[label]))
r2 = mlp.score(tests[features],np.ravel(tests[label]))
print('nnSK r2:',r2)

#NNTF
s = CustomMinMaxGekkoScaler(data,features,label)
ds = s.scaledData()
```

(continues on next page)

(continued from previous page)

```
mma = s.minMaxValues()

trains,tests = train_test_split(ds,test_size=0.2,shuffle=True)
def loss(y_true, y_pred):
    mu = y_pred[:, :1] # first output neuron
    log_sig = y_pred[:, 1:] # second output neuron
    sig = tf.exp(log_sig) # undo the log

    return tf.reduce_mean(2*log_sig + ((y_true-mu)/sig)**2)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(25, activation='relu'),
    tf.keras.layers.Dense(20, activation='relu'),
    #tf.keras.layers.Dense(20, activation='relu'),
    #tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(1) # Output = (, ln()) if using loss fxn
])

model.compile(loss='mse')
model.fit(trains[features],np.ravel(trains[label]),
          batch_size = 10,epochs = 450,verbose = 0)
pred = model(np.ravel(tests[features]))[:,0]
r2 = r2_score(pred,np.ravel(tests[label]))
print('nnTF r2:',r2)
```

```
gpr r2: 0.838216347698638
svr r2: 0.8410099238165618
nnSK r2: 0.8642445702764527
nnTF r2: 0.8136166599386209
```

Models are plotted against the source function and data.

```
plt.figure(figsize=(8,8))
plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
gpr_pred,gpr_u = gpr.predict(data[features],return_std = True)
gpr_u *= 1.645
plt.errorbar(xl,gpr_pred,fmt='--',yerr=gpr_u,label='gpr')
plt.plot(xl,svr.predict(data[features]),'--',label='svr')
plt.plot(xl,s.unscale_y(mlp.predict(ds[features])),'--',label='nn_sk')
plt.plot(xl,s.unscale_y(model.predict(np.ravel(ds[features]))[:,0]),'--',label='nn_tf')
plt.legend()
```

```
from gekko import GEKKO
m = GEKKO()
x = m.Var(0,lb=0,ub=1)
y = Gekko_GPR(gpr,m).predict(x) #function is used here
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -1.0091446783
x: 0.50302287841
Gekko Solvetime: 0.0609 s
```

Gekko_GPR interfaces gpr from *sklearn* or *gpflow* into Gekko. Gaussian Processes allows for the calculation of prediction intervals in the model. While this isn't shown here, for more complicated problems this uncertainty can be used with optimization and decision making when these models are used. All kernels implemented in *sklearn*, anisotropic and isotropic, are working in Gekko, however, some may converge to an infeasibility during solving, so careful kernel consideration is key. These kernels can be combined together, and a custom kernel can be used if a corresponding function is implemented in both *sklearn* and *Gekko*.

```
from gekko import GEKKO
m = GEKKO()
x = m.Var(0,lb=0,ub=1)
y = Gekko_SVR(svr,m).predict(x)
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -0.98631267957
x: 0.49993325357
Gekko Solvetime: 0.015799999999 s
```

Gekko_SVR interfaces *svr* from *sklearn* into Gekko. Support vector machines are more simple than GPR, but do not produce the same uncertainty calculations. All 4 kernels from sklearn are implemented and compatible with Gekko.

```
from gekko import GEKKO
m = GEKKO()
x = m.Var(0,lb=0,ub=1)
y = Gekko_NN_SKlearn(mlp,mma,m).predict([x])
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -1.1718634886
x: 0.47205029204
Gekko Solvetime: 0.1068 s
```

Gekko_NN_SKlearn implements the ANN from *sklearn*, specifically the one created by MLPRegressor. Since scaling is necessary for neural networks, a custom min max scaler was replicated so that the interface could automatically scale and unscale data for prediction. Any layer combination or activation function from sklearn is applicable in Gekko.

```
from gekko import GEKKO
x = m.Var(.0,lb = 0,ub=1)
y = Gekko_NN_TF(model,mma,m,n_output = 1).predict([x])
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
```

(continues on next page)

(continued from previous page)

```
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -1.1491270614
x: 0.49209592754
Gekko Solvetime: 0.2622 s
```

```
from gekko import GEKKO
x = m.Var(.0,lb = 0,ub=1)
y = Gekko_NN_TF(model,mma,m,n_output = 1).predict([x])
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -1.1491270614
x: 0.49209592754
Gekko Solvetime: 0.2622 s
```

5.11.3 Bootstrap Uncertainty Quantification

For some cases where uncertainty intervals are necessary, resampling can be used to train multiple models, where the mean and standard deviation is then used as prediction and uncertainty. Models can even be combined in this resampling method.

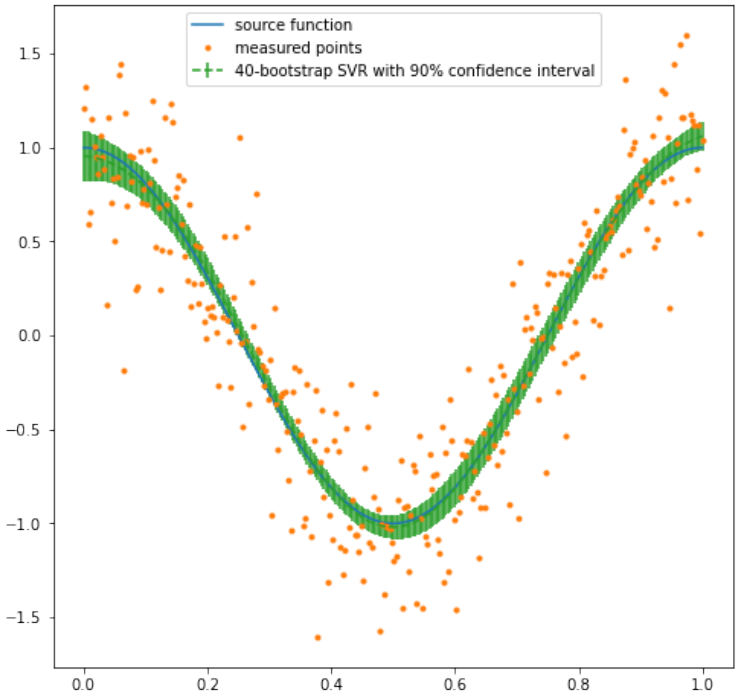
```
from sklearn.metrics import r2_score
Train,test = train_test_split(data,test_size=0.2,shuffle=True)
models = []
for i in range(40):
    train,extra = train_test_split(Train,test_size=0.5,shuffle=True)
    svr = svm.SVR()
    svr.fit(train[features],np.ravel(train[label]))
    models.append(svr)

pred = []
std = []
for i in range(len(data)):
    predicted = []
    for j in range(len(models)):
        predicted.append(models[j].predict(data[features].values[i].reshape(-1,1)))
    pred.append(np.mean(predicted))
    std.append(1.645*np.std(predicted))

r2 = r2_score(pred,data[label])
print('ensemble r2:',r2)

plt.figure(figsize=(8,8))
plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
plt.errorbar(xl,pred,fmt='--',label='40-bootstrap SVR with 90% prediction interval',
```

(continues on next page)

(continued from previous page)

```
↪yerr=std)
plt.legend()
```

```
ensemble r2: 0.8063013376330483
```




```
from gekko import GEKKO
m = GEKKO()
x = m.Var(0,lb=0,ub=1)
y = Bootstrap(models,m).predict(x) #function is used here
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -1.0201166145
x: 0.49964423862
Gekko Solvetime: 0.186 s
```

5.11.4 Linear Regression

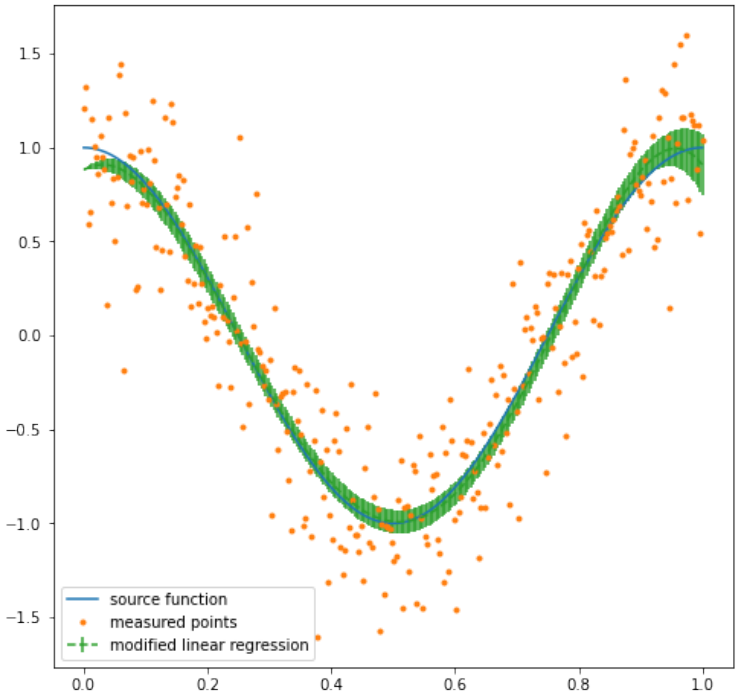
Linear regression is also interfaced in Gekko. This works for Sklearn's RidgeRegression and LinearRegression functions. For nonlinear functions, linear regression can be extended to polynomial/multivariate regression with feature engineering. It is possible to calculate the uncertainty of prediction for linear regression, so the training set and RMSE is also provided to the Interface function.

```
from sklearn.metrics import mean_squared_error
```

(continues on next page)

(continued from previous page)

```
newdata = data
newdata['x^2'] = data['x']**2
newdata['x^3'] = data['x']**3
newdata['sinx'] = np.sin(data['x'])
newfeatures = ['x','x^2','x^3','sinx']

from sklearn.linear_model import Ridge

train,test = train_test_split(newdata,test_size=0.2,shuffle=True)

lr = Ridge(alpha=0)
lr.fit(train[newfeatures],train[label])
lr.score(test[newfeatures],test[label])
pred = lr.predict(data[newfeatures])
RMSE = np.sqrt(mean_squared_error(pred,data[label]))
Xtrain = train[newfeatures].values

#predict with the Linear model and uncertainties
import scipy.stats as ss
def predict(model,xi,Xtrain,RMSE,conf=0.9):
    pred = model.predict(xi.reshape(1,-1))[0][0]
    G = np.linalg.inv(np.dot(Xtrain.T,Xtrain))
    n = len(Xtrain)
    p = len(G)
    t = ss.t.isf(q=(1 - conf) / 2, df=n-p)
    uncertainty = RMSE*t*np.sqrt(np.dot(np.dot(xi.T,G),xi))
    return pred,uncertainty

pred = []
std = []
for i in range(len(newdata)):
    p,s = predict(lr,newdata[newfeatures].values[i],Xtrain,RMSE)
    pred.append(p)
    std.append(s)

plt.figure(figsize=(8,8))
plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
plt.errorbar(xl,pred,fmt='--',yerr=std,label='modified linear regression')
plt.legend()
```




```
from gekko import GEKKO
m = GEKKO()
x = m.Var(.1,lb=0,ub=1)
#needs to be modified due to feature engineering
x1 = x
x2 = x**2
x3 = x**3
x4 = m.sin(x)
y = Gekko_LinearRegression(lr,m).predict([x1,x2,x3,x4]) #function is used here
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -0.99222938638
x: 0.50971895084
Gekko Solvetime: 0.013500000001 s
```

5.11.5 Conformal Prediction Uncertainty Quantification

Prediction intervals can also be calculated with a distance-metric based method called non-conformist prediction. This requires an additional datasplit of the training set into a training set and calibration set. This calibration set is then used to calibrate the model and give a prediction interval that represents the desired quantile. This method works with every model and produces a constant uncertainty rather than a variable one. The model is first trained with the nonconformist library before it is interfaced with Gekko. It typically results in a higher variance but can be more consistent than other methods.

```
from nonconformist.base import RegressorAdapter
from nonconformist.icp import IcpRegressor
```

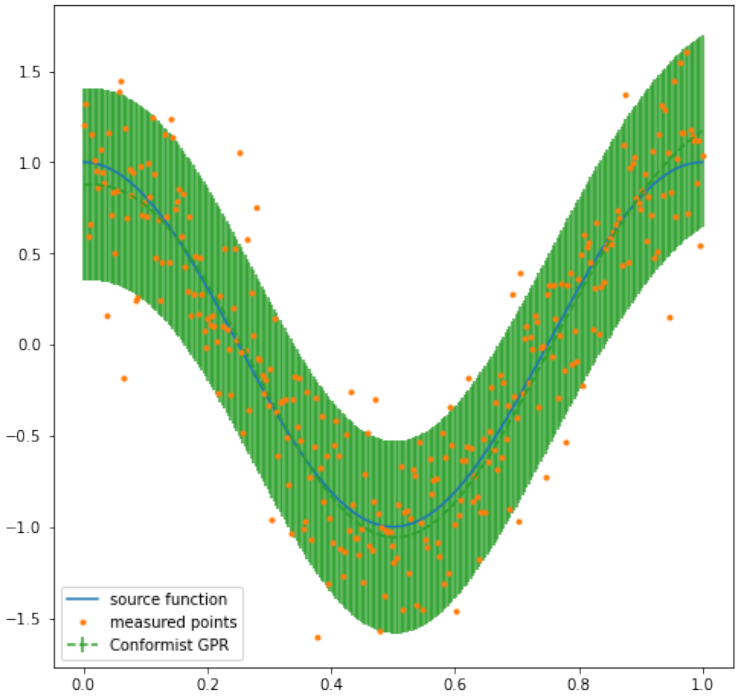
(continues on next page)

(continued from previous page)

```
from nonconformist.nc import RegressorNc, RegressorNormalizer
from nonconformist.nc import InverseProbabilityErrFunc
from nonconformist.nc import MarginErrFunc,AbsErrorErrFunc,SignErrorErrFunc
from sklearn.neural_network import MLPRegressor
import sklearn.gaussian_process as gp
from sklearn import svm

train,test = train_test_split(data,test_size=0.2,shuffle=True)
train,calibrate = train_test_split(train,test_size=0.5,shuffle=True)


k = gp.kernels.RBF() * gp.kernels.ConstantKernel() + gp.kernels.WhiteKernel()
gpr = gp.GaussianProcessRegressor(kernel=k,\
                              n_restarts_optimizer=10,\
                              alpha=0.1,\
                              normalize_y=True)


mod = RegressorAdapter(gpr)

nc = RegressorNc(mod,AbsErrorErrFunc()) #assign an error function
icp = IcpRegressor(nc) #create the icp regressor
icp.fit(train[features],train[label].values.reshape(len(train)))
icp.calibrate(calibrate[features],calibrate[label].values.reshape(len(calibrate)))

all_prediction = icp.predict(data[features].values,significance=0.1)
pred = (all_prediction[:,0] + all_prediction[:,1])/2
margin = (abs(all_prediction[:,0] - all_prediction[:,1])/2)[0]
r2a = r2_score(pred,data[label])
print('r2:',r2a)
print('90% prediction margin:',margin)
```

```
r2: 0.8134537732195808
90% prediction margin: 0.527352056347252
```

```
plt.figure(figsize=(8,8))
plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
plt.errorbar(xl,pred,fmt='--',yerr=margin,label='Conformist GPR')
plt.legend()
```




```
from gekko import GEKKO
m = GEKKO()
x = m.Var(.1,lb=0,ub=1)
#needs to be modified due to feature engineering
y = Conformist([icp.get_params()['nc_function__model'].get_params()['model'],margin],m).
↪predict([x])
m.Obj(y)
m.solve(disp=False)
print('solution:',y.value[0])
print('x:',x.value[0])
print('Gekko Solvetime:',m.options.SOLVETIME,'s')
```

```
solution: -0.95151964117
x: 0.51036497097
Gekko Solvetime: 0.023100000006 s
```

There are several methods of calculating prediction uncertainty for the interfaced models. Uncertainty is calculated within Gaussian Processes. Several models can be trained and used with resampling for a prediction mean and variation. Linear regression can create uncertainty through least squared equations. This same uncertainty method can be applied to other nonlinear regression methods with the Delta function. The nonconformist library can generate an uncertainty margin as well. Different variants of neural networks, specifically in Tensorflow, can produce uncertainty based on the loss function.

5.11.6 Control Applications

These models can be used for a wide variety of applications in Gekko, not just optimization. Here, a simple control problem is presented where the xt function is replaced by a model. It generates nearly the same result

```
def control1(model=[],returns=False,plot=True):
```

(continues on next page)

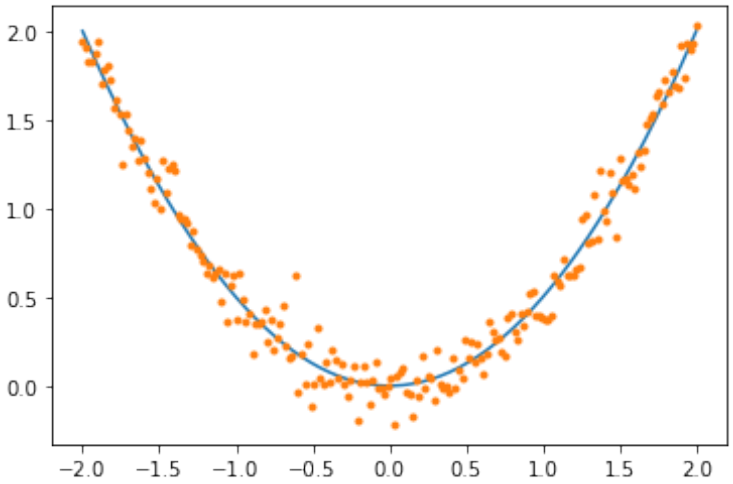
(continued from previous page)

```
    global m

    m = GEKKO() # initialize gekko
    nt = 101
    m.time = np.linspace(0,2,nt)
    # Variables
    x1 = m.Var(value=1)
    x2 = m.Var(value=0)
    u = m.Var(value=0,lb=-1,ub=1)
    p = np.ones(nt) # mark final time point
    p[-1] = 1.0
    final = m.Param(value=p)
    # Equations
    m.Equation(x1.dt()==u)
    if(model == []):
        f = 0.5*x1**2
    else:
        if(model[0] == 'GPR'):
            f = Gekko_GPR(model[1],m).predict([x1])
        elif(model[0] == 'SVR'):
            f = Gekko_SVR(model[1],m).predict([x1])
        elif(model[0] == 'NNSK'):
            f = Gekko_NN_SKlearn(model[1],model[2],m).predict([x1])
        elif(model[0] == 'NNTF'):
            f = Gekko_NN_TF(model[1],model[2],m,1).predict([x1])

    m.Equation(x2.dt()==f)
    m.Obj(x2*final) # Objective function
    m.options.IMODE = 6 # optimal control mode
    m.solve(disp=False) # solve

    if plot:
        plt.figure(1) # plot results
        plt.plot(m.time,x1.value,'k-',label=r'$x_1$')
        plt.plot(m.time,x2.value,'b-',label=r'$x_2$')
        plt.plot(m.time,u.value,'r--',label=r'$u$')
        plt.legend(loc='best')
        plt.xlabel('Time')
        plt.ylabel('Value')
        plt.show()
    if returns:
        return m.time,x1.value,x2.value,u.value

#Generate Training set
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return 0.5*x**2

N = 200
xl = np.linspace(-2,2,N)
```

(continues on next page)

(continued from previous page)

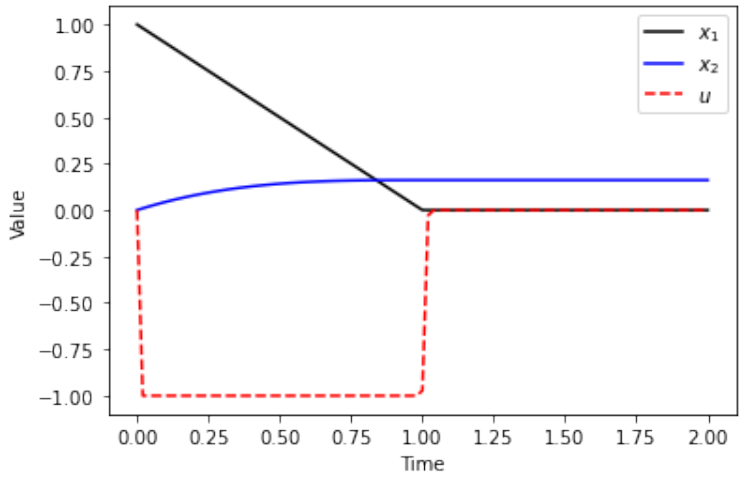
```
noise = np.random.normal(0,.1,N)
y_measured = f(xl) + noise

plt.plot(xl,f(xl),label='source function')
plt.plot(xl,y_measured,'.',label='measured points')
```



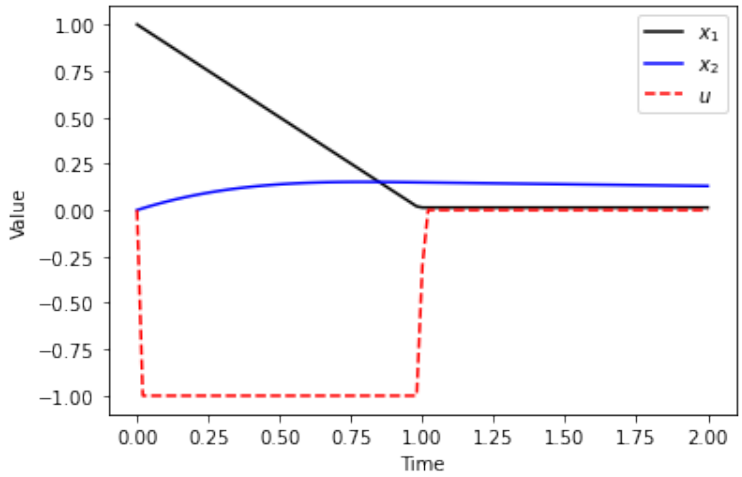

```
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.DataFrame(np.array([xl,y_measured]).T,columns=['x','y'])
features = ['x']
label = ['y']

train,test = train_test_split(data,test_size=0.2,shuffle=True)

import sklearn.gaussian_process as gp
from sklearn.neural_network import MLPRegressor
from sklearn import svm

#gpr
k = gp.kernels.RBF() * gp.kernels.ConstantKernel() + gp.kernels.WhiteKernel()
gpr = gp.GaussianProcessRegressor(kernel=k,\
                            n_restarts_optimizer=10,\
                            alpha=0.1,\
                            normalize_y=True)
gpr.fit(train[features],train[label])
r2 = gpr.score(test[features],test[label])
print('gpr r2:',r2)


#svr
svr = svm.SVR()
svr.fit(train[features],np.ravel(train[label]))
r2 = svr.score(test[features],np.ravel(test[label]))
print('svr r2:',r2)
```

```
gpr r2: 0.9786408113248649
svr r2: 0.9731400906454939
```

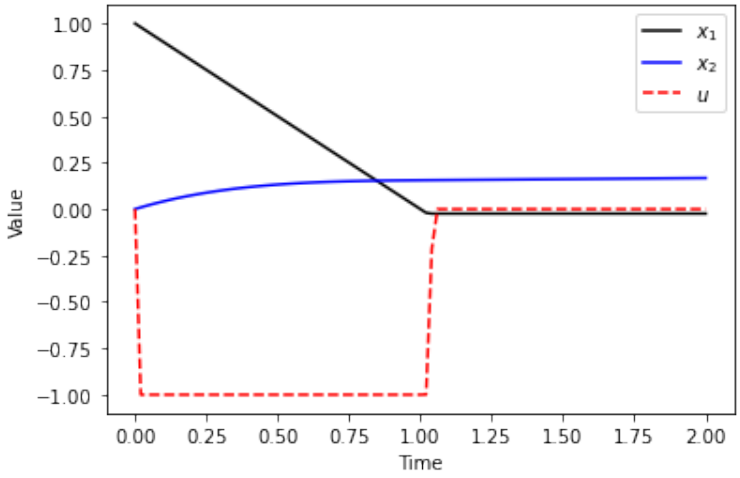
```
control1()
```




```
control1(["GPR",gpr])
```




```
control1(["SVR",svr])
```



5.11.7 Remarks

All of these functions import trained models and tools provided by other packages. The packages used for this extension to Gekko are:

Scikit-learn: https://scikit-learn.org/stable/

Tensorflow: https://www.tensorflow.org/

GPflow: https://github.com/GPflow/GPflow

Nonconformist: https://github.com/donlnz/nonconformist

as well as other standard packages like Numpy, Pandas, and others.

Authors of the Machine Learning package for Gekko are graduate research assistants LaGrande Gunnell and Kyle Manwaring. Thanks to John Vienna and Xiaonan Lu of Pacific Northwest National Laboratory for providing technical direction and sponsorship of the work through a Department of Energy (DOE) grant.

5.12 Chemical Library

GEKKO specializes in a optimization and control. The *chemical* module extends GEKKO with chemical compounds, thermodynamic properties, and flowsheet objects.

5.12.1 Thermodynamic Properties

Thermodynamic properties form the basis for the flowsheet objects. The thermodynamic properties are also accessible as either temperature independent or temperature dependent quantities.

c = chemical.Properties(m):

Creates a chemical property object with a GEKKO model *m*.

Chemical properties are defined to specify the chemicals involved in thermodynamic and flowsheet objects.:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
```

classmethod c.compound(*name*)

Add chemical compound to model with one of the following:

1. IUPAC Name (1,2-ethanediol)
2. Common Name (ethylene glycol)
3. CAS Number (107-21-1)
4. Formula (C2H6O2)

Repeated compounds are permitted. All compounds should be declared before thermo objects are created. An error message will occur if the compound is not in the database and a file 'compounds.txt' will be created to communicate the available compounds.:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('water')
c.compound('hexane')
```

prop = c.thermo(name)

Thermodynamic Properties:

```
# usage: thermo('mw') for constants
# thermo('lvp',T) for temperature dependent
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
# add compounds
c.compound('water')
c.compound('hexane')
c.compound('heptane')
# molecular weight
mw = c.thermo('mw')
# liquid vapor pressure
T = m.Param(value=310)
vp = c.thermo('lvp',T)
m.solve(disp=False)
print(mw)
print(vp)
```

Temperature Independent

- mw = Molecular Weight (kg/kmol)
- tc = Critical Temperature (K)
- pc = Critical Pressure (Pa)
- vc = Critical Volume (m^3/kmol)
- ccf = Crit Compress Factor (unitless)
- mp = Melting Point (K)
- tpt = Triple Pt Temperature (K)
- tpp = Triple Pt Pressure (Pa)
- nbp = Normal Boiling Point (K)
- lmv = Liq Molar Volume (m^3/kmol)
- ighf = IG Heat of Formation (J/kmol)

- iggf = IG Gibbs of Formation (J/kmol)
- igae = IG Absolute Entropy (J/kmol*K)
- shf = Std Heat of Formation (J/kmol)
- sgf = Std Gibbs of Formation (J/kmol)
- sae = Std Absolute Entropy (J/kmol*K)
- hfmp = Heat Fusion at Melt Pt (J/kmol)
- snhc = Std Net Heat of Comb (J/kmol)
- af = Acentric Factor (unitless)
- rg = Radius of Gyration (m)
- sp = Solubility Parameter ((J/m^3)^0.5)
- dm = Dipole Moment (c*m)
- r = van der Waals Volume (m^3/kmol)
- q = van der Waals Area (m^2)
- ri = Refractive Index (unitless)
- fp = Flash Point (K)
- lfl = Lower Flammability Limit (K)
- ufl = Upper Flammability Limit (K)
- lflt = Lower Flamm Limit Temp (K)
- uflt = Upper Flamm Limit Temp (K)
- ait = Auto Ignition Temp (K)

Temperature Dependent

- sd = Solid Density (kmol/m^3)
- ld = Liquid Density (kmol/m^3)
- svp = Solid Vapor Pressure (Pa)
- lvp = Liquid Vapor Pressure (Pa)
- hvap = Heat of Vaporization (J/kmol)
- scp = Solid Heat Capacity (J/kmol*K)
- lcp = Liquid Heat Capacity (J/kmol*K)
- igcp = Ideal Gas Heat Capacity (J/kmol*K)
- svc = Second Virial Coefficient (m^3/kmol)
- lv = Liquid Viscosity (Pa*s)
- vv = Vapor Viscosity (Pa*s)
- sk = Solid Thermal Conductivity (W/m*K)
- lk = Liq Thermal Conductivity (W/m*K)
- vk = Vap Thermal Conductivity (W/m*K)
- st = Surface Tension (N/m)

- sh = Solid Enthalpy (J/kmol)
- lh = Liq Enthalpy (J/kmol)
- vh = Vap Enthalpy (J/kmol)

5.12.2 Flowsheet Objects

Flowsheet objects are created with the chemical library with basic unit operations that mix, separate, react, and model the dynamics of chemical mixtures in processing equipment. The basis for flowsheet objects is:

- Pressure (Pa)
- Temperature (K)
- Mole Fractions
- Molar Flow (kmol/sec)
- Moles (kmol)

These fundamental quantities are used to track other derived quantities such as concentration (kmol/m^3), mass (kg), mass flow (kg/sec), enthalpy (J/kmol), heat capacity (J/kmol-K), mass fractions, and many others for mixtures and streams.

`f = chemical.Flowsheet(m,[stream_level=1]):`

Creates a chemical flowsheet object with a GEKKO model *m* and a *stream_level*.

The *stream_level* either includes only chemical compositions (*stream_level=0*) or also pressure and temperature (*stream_level=1*). Most methods in the Flowsheet object require *stream_level=1* but there are a few cases such as blending applications that don't the additional equations (e.g. energy balance equations to simulate temperature changes.

A code example shows the use of a *Flowsheet* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
```

`f.connect(s1,s2):`

Connect two objects The first name dictates the properties of the combined object.

Inputs:

- s1 = object or name of object 1 (string)
- s2 = object or name of object 2 (string)

A code example shows the use of the *connect* function:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
mix = f.mixer()
```

(continues on next page)

(continued from previous page)

```
spl = f.splitter()
f.connect(mix.outlet,spl.inlet)
m.solve()
```

f.set_phase(y,phase='liquid'):

Set the phase (vapor, liquid, solid) of a stream or accumulation.

y = object or name of object (string)

phase = phase of the object (vapor, liquid, solid). A code example demonstrates the *set_phase* method:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
fl = f.flash()
f.set_phase(fl.inlet,'vapor')
m.solve()
```

f.reserve(fixed=False):

Create an accumulation that is a quantity (moles) of chemical holdup.

Output: Reserve Object

- P = Pressure (Pa)
- T = Temperature (K)
- n = Molar holdup (kmol)
- x = Array of mole fractions
- phase = Phase (solid, liquid, vapor)
- fixed = Gekko parameter (True) or variable (False) if None or []

A code example demonstrates the creation of a *reserve* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
r = f.reserve()
m.solve()
```

f.stream(fixed=True):

Create a stream that is a flow (moles/sec) of chemical compounds.

Output: Stream Object

- P = Pressure (Pa)
- T = Temperature (K)
- ndot = Molar flow rate (kmol/sec)

- x = Array of mole fractions
- phase = Phase (solid, liquid, vapor)
- fixed = Gekko parameter (True) or variable (False) if None or []

A code example demonstrates the creation of a *stream* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
r = f.stream()
m.solve()
```

f.flash():

Create a flash object that separates a stream into a liquid and vapor outlet. The flash object does not have a liquid holdup. See flash_column for a flash object with holdup.

Output: Flash object

- P = Pressure (Pa)
- T = Temperature (K)
- Q = Heat input (J/sec)
- gamma = Activity coefficients for each compound
- inlet = inlet stream name
- vapor = vapor outlet stream name
- liquid = liquid outlet stream name

A code example demonstrates the creation and solution of a *flash* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
fl = f.flash()
m.solve()
```

f.flash_column():

Create a flash column object that separates a stream into a liquid and vapor outlet. The flash object does not have a liquid holdup. See flash for a flash object without holdup.

Output: Flash column object

- P = Pressure (Pa)
- T = Temperature (K)
- Q = Heat input (J/sec)
- n = Holdup (kmol)
- gamma = Activity coefficients for each compound

- inlet = inlet stream name
- vapor = vapor outlet stream name
- liquid = liquid outlet stream name

A code example demonstrates the creation and solution of a *flash_column* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
fl = f.flash()
m.solve()
```

f.mass(y=None,rn=''):

Create a mass object that calculates the mass (kg) in a mixture holdup.

Inputs:

y = Mass Object (mo)

- m = mass (kg)
- mx = mass of components (kg)
- reserve = ‘’

  rn = Reserve name if already created

Output: Mass object

A code example demonstrates how a *mass* object is created and linked to a reserve object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
r = f.reserve()
ms = f.mass(rn=r.name)
m.options.solver=1
m.solve()
```

f.massflow(y=None,sn=''):

Create a mass flow object that calculates the mass flow (kg/sec) in a mixture stream.

Inputs:

y = Mass Flow Object (mo)

- mdot = mass flow (kg/sec)
- mx = mass of components (kg)
- stream = ‘’

  sn = Stream name if already created

Output: Mass flow object

A code example demonstrates how a *massflow* object is created and linked to a stream object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
s = f.stream()
mf = f.massflow(sn=s.name)
m.options.solver=1
m.solve()
```

f.massflows(y=None,sn=''):

Create a mass flow object that calculates the mass flow (kg/sec) in a mixture stream.

Inputs:

y = Mass Flow Object (mo)

- mdot = mass flow (kg/sec)
- mdoti = mass flow of components (kg)
- stream = ‘’

  sn = Stream name if already created

Output: Mass flows object

A code example demonstrates how a *massflow* object is created and linked to a stream object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
s = f.stream()
mf = f.massflows(sn=s.name)
m.options.solver=1
m.solve()
```

f.mixer(ni=2):

Create a mixer object that combines two or more streams. The mixer object does not have a liquid holdup. See vessel for a mixer object with holdup.

Input:

- ni = Number of inlets (default=2)

Output: Mixer object

- inlet = inlet stream names (inlet[1], inlet[2], . . . , inlet[ni])
- outlet = outlet stream name

A code example demonstrates the creation and solution of a *mixer* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
mx = f.mixer()
m.options.SOLVER=1
m.solve()
```

f.molarflows(y=None,sn=''):

Create a molar flows object that calculates the molar flow (kmol/sec) of a mixture stream as well as the molar flow of the individual components.

Inputs:

y = Molar Flows Object (mo)

- ndot = molar flow (kmol/sec)
- ndoti = molar flow of components (kmol)
- stream = ''

sn = Stream name if already created

Output: Mass flows object

A code example demonstrates how a *molarflows* object is created and linked to a stream object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
s = f.stream()
mf = f.molarflows(sn=s.name)
m.options.solver=1
m.solve()
```

f.pid():

Create a PID (Proportional Integral Derivative) control object that relates the controller output (CO), process variable (PV), and set point (SP) of a control loop. This is a continuous form of a PID controller that approximates discrete PID controllers typically used in industrial practice.

Output: PID object

- co = Controller Output (u)
- pv = Process Variable (y)
- sp = Process Variable Set Point (ysp)
- Kc = PID Proportional constant
- tauI = PID Integral constant
- tauD = PID Derivative constant

- i = Integral term

Description: PID: Proportional Integral Derivative Controller In the frequency domain the PID controller is described by

U(s) = Kc*Y(s) + Y(s)*Kc/s*tauI + Kc*taud*s*Y(s)

In the time domain the PID controller is described by

u(t) = Kc*(ysp-y(t)) + (Kc/taui)*Integral(t=0...t)(ysp-y(t))dt + Kc*taud*dy(t)/dt

This implementation splits the single equation into two equations The second equation is necessary to avoid the numerical integration. The equations are posed in an open equation form. The integral time

constant is multiplied through to avoid potential divide by zero.

This form may have an advantage over placing the term taui in with the integral equation for cases where taui becomes very small.

0 = -u*taui + Kc*((ysp-y)*taui + Integral + taud*(dy/dt)*taui)

0 = d(Integral)/dt - (ysp-y)

A code example demonstrates the creation and solution of a *pid* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
f = chemical.Flowsheet(m)
p = f.pid()
m.solve()
```

f.pump():

Create a pump object that changes the pressure of a stream.

Output: Pump object

- dp = change in pressure (Pa)
- inlet = inlet stream name
- outlet = outlet stream name

A code example demonstrates the creation and solution of a *pump* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
p = f.pump()
m.solve()
```

f.reactor(ni=1):

Create a reactor object that combines two or more streams and includes a generation term for each chemical species. The reactor object is similar to the vessel object but includes reactions. The reaction rates are defined as (+) generation and (-) consumption. In addition to the reaction rates, there is a term for heat generation from exothermic reactions (+) or heat removal from endothermic reactions (-).

Input:

- ni = Number of inlets (default=2)

Output: Reactor object

- V = Volume (m^3)
- Q = Heat input (J/sec)
- Qr = Heat generation by reaction (J/sec)
- r = Mole generation (kmol/sec)
- rx = Mole generation by species (kmol/sec)
- inlet = inlet stream names (inlet[1], inlet[2], ..., inlet[ni])
- reserve = Molar holdup name
- outlet = Outlet stream name

A code example demonstrates the creation and solution of a *reactor* object with two inlet streams:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
r = f.reactor(ni=2)
m.options.SOLVER = 1
m.solve()
```

f.recovery():

Create a recovery object that splits out components of a stream. This object is commonly used in applications such as separation systems (membranes, filters, fluidized bed production, etc). The last split fraction is calculated as the remainder split amount that sums to a total quantity of one.

Output: Recovery object

- split = Split fraction to outlet 1 (0-1)
- inlet = inlet stream name
- outlet = outlet stream name

A code example demonstrates the creation and solution of a *recovery* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
p = f.recovery()
m.options.SOLVER = 1
m.solve()
```

f.splitter(no=2):

Create a splitter object that divides a stream into two outlets. This object is used in flowsheeting applications where the stream may be diverted to a different downstream process or a recycle split. The last split fraction is calculated as the remainder split amount that sums to a total quantity of one.

Input:

- no = Number of outlets

Output: Splitter object

- split = Split fraction to outlet 1 (0-1)
- inlet = inlet stream name
- outlet = outlet stream name

A code example demonstrates the creation and solution of a *splitter* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
p = f.splitter()
m.options.SOLVER = 1
m.solve()
```

f.stage(opt=2):

Create an equilibrium stage distillation object that has a vapor and liquid inlet, a vapor and liquid outlet, pressure drop, and heat addition or loss rate. The stage object is available either in Index-1 or Index-2 DAE (Differential and Algebraic Equation) form determined by the ***opt*** parameter. The stage model is one stage (tray, packing height) of a distillation column.

Input:

- opt = Index-1 (1) or Index-2 (2=default) form

Output: Stage object

- l_in = Inlet liquid stream
- l_out = Outlet liquid stream
- v_in = Inlet vapor stream
- v_out = Outlet vapor stream
- q = Heat addition (+) or loss (-) rate
- dp_in_liq = Pressure drop below stage
- dp_in_vap = Pressure drop above stage

A code example demonstrates the creation and solution of a *stage* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
s = f.stage(opt=1)
m.options.SOLVER = 1
m.solve()
```

f.stream_lag():

Create a stream_lag object that approximates first-order blending of a stream that passes through a vessel. The time constant (tau) is approximately the volume divided by the volumetric flow. Molar fractions in the outlet stream are blended inputs.

Output: Stream lag object

- tau = time constant (sec)
- inlet = inlet stream name
- outlet = outlet stream name

A code example demonstrates the creation and solution of a *stream_lag* object:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
s = f.stream_lag()
m.solve()
```

f.vessel(ni=1,mass=False):

Create a vessel object that simulates a container with volume *V*. The vessel object is similar to the reactor object but does not include reactions. There is a term for heat addition (+) or heat removal (-). The options *mass* parameter is *True* if the inlet and outlet are expressed as mass flows instead of molar flows.

Input:

- ni = Number of inlets (default=1)

Output: Vessel object

- V = Volume (m^3)
- Q = Heat input (J/sec)
- inlet = inlet stream names (inlet[1], inlet[2], . . . , inlet[ni])
- reserve = Molar holdup name
- outlet = Outlet stream name

A code example demonstrates the creation and solution of a *vessel* object with three inlet streams:

```
from gekko import GEKKO, chemical
m = GEKKO()
c = chemical.Properties(m)
c.compound('propane')
c.compound('water')
f = chemical.Flowsheet(m)
v = f.vessel(ni=3)
m.options.SOLVER = 1
m.solve()
```

5.13 Solver Extension

GEKKO includes a limited interface to access more solvers. The solver extension module allows for converting the GEKKO model to other mathematical optimization libraries, opening up access to more solvers. It currently contains two converters, allowing the GEKKO model to be solved through AMPLPY or Pyomo.

5.13.1 Setup

To use the solver extension module:

- Set the SOLVER_EXTENSION option to the converter you want to use (either `AMPLPY` or `PYOMO`)
  - `m.options.SOLVER_EXTENSION = <converter>`
- Set the SOLVER option to the solver you want to use (eg. `ipopt`). Note that the string specifying the solver may be case sensitive.
  - `m.options.SOLVER = <solver>`
- The GEKKO model can be declared like normal. The results from the solve are placed back into the GEKKO model variables.

Solver Options

Solver options are specified within m.solver_options as normal:

```
# Use options relevant to the solver you are using.
m.solver_options = ['max_iter 10', \
                    'tol 0.01', \
                    'outlev 1' \
                    # etc...
                    ]
```

5.13.2 AMPLPY

The solver extension module supports converting to AMPL syntax, allowing access to various supported solvers by making use of the AMPLPY library. Alternatively, a .mod file (AMPL model file) can be output and solved by uploading to NEOS.

The base version of AMPL limits a model to 500 variables and 500 constraints (300 for nonlinear problems, and fewer for certain solvers). AMPL offers a free Community Edition license with no limitations on variables or constraints. Licensing and more details can be obtained from the AMPL website.

The converter between the GEKKO model and AMPLPY is limited and does not support the full range of model building functions and options available in GEKKO. However, basic model building functions such as (but not limited to) variables, parameters, constants, intermediates, constraints, and objectives are supported by the converter. Functions relating to dynamic optimization (time, derivatives, etc) are not supported by AMPLPY and cannot be used in the converter.

Setup

The solver extension module requires AMPLPY to solve within GEKKO:

```
$ pip install amplpy
```

Solvers are installed through amplpy.modules. See https://dev.ampl.com/ampl/python/modules.html:

```
$ python -m amplpy.modules install <solver>
```

AMPLPY Example

Example use of the solver extension module with the AMPLPY converter is shown below:

```
from gekko import GEKKO
m = GEKKO(remote=False)  # remote=True not supported
x = m.Var()
y = m.Var()
m.Equations([3*x+2*y==1, x+2*y==0])
# enable solver extension and use AMPLPY converter
m.options.SOLVER_EXTENSION = "AMPLPY"
m.options.SOLVER = "bonmin"  # use BONMIN solver
m.solve()    # solve
print(x.value,y.value)
```

Additional AMPLPY converter methods

The AMPLPY converter provides some utility methods to the GEKKO model object:

classmethod m.**create_amplpy_object**()

Returns an amplpy model object:

```
ampl = m.create_amplpy_object()
# do some stuff with the amplpy object
#...
```

For more information view the amplpy documentation.

classmethod m.**generate_ampl_file**(*filename='model.mod'*)

Generates an ampl model (.mod) file in the current directory or as specified by `filename`:

```
m.generate_ampl_file()
```

5.13.3 Pyomo

The solver extension module supports converting to Pyomo, a python library for mathematical optimization. Pyomo supports a variety of solvers, including an interface with ASL (AMPL Solver Library). One of the primary advantages of Pyomo is that it is entirely open source (BSD license), and therefore has no constraints on model size and is free for commercial use.

The converter to Pyomo supports the basic GEKKO model building functions such as variables, parameters, constants, intermediates, constraints, and objectives. In its current state, it does not support dynamic optimization (time, derivatives, etc). However, this has the potential to be implemented in the future through use of Pyomo DAE (Differential Algebraic Equations) module.

Setup

The solver extension module requires Pyomo to solve within GEKKO:

```
$ pip install Pyomo
```

Solvers should be installed by either compiling them from source or obtaining the relevant binaries. The location of the executable should then be added to PATH in order for Pyomo to recognise the solver.

You can check if the solver has been installed correctly by running a version check, ie. `<solver> -v`.

Pyomo Example

Example use of the solver extension module with the Pyomo converter is shown below:

```
from gekko import GEKKO
m = GEKKO(remote=False)  # remote=True not supported
x = m.Var()
y = m.Var()
m.Equations([3*x+2*y==1, x+2*y==0])
# enable solver extension and use Pyomo converter
m.options.SOLVER_EXTENSION = "PYOMO"
m.options.SOLVER = "cbc"  # use CBC solver
m.solve()    # solve
print(x.value,y.value)
```

Additional Pyomo converter methods

The Pyomo converter provides some utility methods to the GEKKO model object:

classmethod `m.create_pyomo_object()`

Returns an pyomo ConcreteModel object:

```
pyomo_model = m.create_pyomo_object()
# do some stuff with the pyomo ConcreteModel
#...
```

For more information view the Pyomo documentation.

5.14 Examples

5.14.1 Solve Linear Equations

$$3x + 2y = 1$$
$$x + 2y = 0$$

```
from gekko import GEKKO
m = GEKKO()             # create GEKKO model
x = m.Var()             # define new variable, default=0
y = m.Var()             # define new variable, default=0
m.Equations([3*x+2*y==1, x+2*y==0])  # equations
m.solve(disp=False)     # solve
print(x.value,y.value) # print solution
```

[0.5] [-0.25]

5.14.2 Solve Nonlinear Equations

$$x + 2y = 0$$
$$x^2 + y^2 = 1$$

```
from gekko import GEKKO
m = GEKKO()              # create GEKKO model
x = m.Var(value=0)       # define new variable, initial value=0
y = m.Var(value=1)       # define new variable, initial value=1
m.Equations([x + 2*y==0, x**2+y**2==1]) # equations
m.solve(disp=False)      # solve
print([x.value[0],y.value[0]]) # print solution
```

[-0.8944272, 0.4472136]

5.14.3 Variable and Equation Arrays

$$x_1 = x_0 + p$$
$$x_2 - 1 = x_1 + x_0$$
$$x_2 = x_1^2$$

This example demonstrates how to define a parameter with a value of 1.2, a variable array, an equation, and an equation array using GEKKO. After the solution with m.solve(), the x values are printed:

```
from gekko import GEKKO
m=GEKKO()
p=m.Param(1.2)
x=m.Array(m.Var,3)
eq0 = x[1]==x[0]+p
eq1 = x[2]-1==x[1]+x[0]
m.Equation(x[2]==x[1]**2)
m.Equations([eq0,eq1])
m.solve()
for i in range(3):
   print('x['+str(i)+']='+str(x[i].value))
```

x[0]=[-1.094427] x[1]=[0.1055728] x[2]=[0.01114562]

5.14.4 HS 71 Benchmark

$$\min$$
$$x_1 x_4 (x_1 + x_2 + x_3) + x_3$$
$$s.t.$$
$$x_1 x_2 x_3 x_4 >= 25$$

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40$$

$$1 <= x_1, x_2, x_3, x_4 <= 5$$

$$x_0 = (1, 5, 5, 1)$$

This example demonstrates how to solve the HS71 benchmark problem using GEKKO:

```
from gekko import GEKKO

# Initialize Model
```

(continues on next page)

(continued from previous page)

```
m = GEKKO(remote=True)

#help(m)

#define parameter
eq = m.Param(value=40)

#initialize variables
x1,x2,x3,x4 = [m.Var() for i in range(4)]

#initial values
x1.value = 1
x2.value = 5
x3.value = 5
x4.value = 1

#lower bounds
x1.lower = 1
x2.lower = 1
x3.lower = 1
x4.lower = 1

#upper bounds
x1.upper = 5
x2.upper = 5
x3.upper = 5
x4.upper = 5

#Equations
m.Equation(x1*x2*x3*x4>=25)
m.Equation(x1**2+x2**2+x3**2+x4**2==eq)

#Objective
m.Obj(x1*x4*(x1+x2+x3)+x3)

#Set global options
m.options.IMODE = 3 #steady state optimization

#Solve simulation
m.solve() # solve on public server

#Results
print('')
print('Results')
print('x1: ' + str(x1.value))
print('x2: ' + str(x2.value))
print('x3: ' + str(x3.value))
print('x4: ' + str(x4.value))
```

5.14.5 Solver Selection

Solve y2=1 with APOPT solver. See APMonitor documentation or GEKKO documentation for additional solver options:

```
from gekko import GEKKO
m = GEKKO()           # create GEKKO model
y = m.Var(value=2)    # define new variable, initial value=2
m.Equation(y**2==1)   # define new equation
m.options.SOLVER=1    # change solver (1=APOPT,3=IPOPT)
m.solve(disp=False)   # solve locally (remote=False)
print('y: ' + str(y.value)) # print variable value
```

y: [1.0]

5.14.6 Interpolation with Cubic Spline

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

xm = np.array([0,1,2,3,4,5])
ym = np.array([0.1,0.2,0.3,0.5,1.0,0.9])

m = GEKKO()              # create GEKKO model
m.options.IMODE = 2      # solution mode
x = m.Param(value=np.linspace(-1,6)) # prediction points
y = m.Var()              # prediction results
m.cspline(x, y, xm, ym) # cubic spline
m.solve(disp=False)      # solve

# create plot
plt.plot(xm,ym,'bo')
plt.plot(x.value,y.value,'r--',label='cubic spline')
plt.legend(loc='best')
```

5.14.7 Linear and Polynomial Regression

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

xm = np.array([0,1,2,3,4,5])
ym = np.array([0.1,0.2,0.3,0.5,0.8,2.0])

#### Solution
m = GEKKO()
m.options.IMODE=2
# coefficients
c = [m.FV(value=0) for i in range(4)]
x = m.Param(value=xm)
y = m.CV(value=ym)
y.FSTATUS = 1
```

(continues on next page)

(continued from previous page)

```
# polynomial model
m.Equation(y==c[0]+c[1]*x+c[2]*x**2+c[3]*x**3)

# linear regression
c[0].STATUS=1
c[1].STATUS=1
m.solve(disp=False)
p1 = [c[1].value[0],c[0].value[0]]

# quadratic
c[2].STATUS=1
m.solve(disp=False)
p2 = [c[2].value[0],c[1].value[0],c[0].value[0]]

# cubic
c[3].STATUS=1
m.solve(disp=False)
p3 = [c[3].value[0],c[2].value[0],c[1].value[0],c[0].value[0]]

# plot fit
plt.plot(xm,ym,'ko',markersize=10)
xp = np.linspace(0,5,100)
plt.plot(xp,np.polyval(p1,xp),'b--',linewidth=2)
plt.plot(xp,np.polyval(p2,xp),'r--',linewidth=3)
plt.plot(xp,np.polyval(p3,xp),'g:',linewidth=2)
plt.legend(['Data','Linear','Quadratic','Cubic'],loc='best')
plt.xlabel('x')
plt.ylabel('y')
```

5.14.8 Nonlinear Regression

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

# measurements
xm = np.array([0,1,2,3,4,5])
ym = np.array([0.1,0.2,0.3,0.5,0.8,2.0])

# GEKKO model
m = GEKKO()

# parameters
x = m.Param(value=xm)
a = m.FV()
a.STATUS=1

# variables
y = m.CV(value=ym)
y.FSTATUS=1
```

(continues on next page)

(continued from previous page)

```
# regression equation
m.Equation(y==0.1*m.exp(a*x))

# regression mode
m.options.IMODE = 2

# optimize
m.solve(disp=False)

# print parameters
print('Optimized, a = ' + str(a.value[0]))

plt.plot(xm,ym,'bo')
plt.plot(xm,y.value,'r-')
```

5.14.9 Solve Differential Equation(s)

Solve the following differential equation with initial condition $y(0) = 5$:

$$k\frac{dy}{dt} = ty$$

where $k = 10$. The solution of $y(t)$ should be reported from an initial time 0 to final time 20. Create a plot of the result for $y(t)$ versus t.:

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

m = GEKKO()
m.time = np.linspace(0,20,100)
k = 10
y = m.Var(value=5.0)
t = m.Param(value=m.time)
m.Equation(k*y.dt()==-t*y)
m.options.IMODE = 4
m.solve(disp=False)

plt.plot(m.time,y.value)
plt.xlabel('time')
plt.ylabel('y')
```

5.14.10 Mixed Integer Nonlinear Programming

```
from gekko import GEKKO
m = GEKKO() # Initialize gekko
m.options.SOLVER=1  # APOPT is an MINLP solver

# optional solver settings with APOPT
m.solver_options = ['minlp_maximum_iterations 500', \
                    # minlp iterations with integer solution
                    'minlp_max_iter_with_int_sol 10', \
```

(continues on next page)

(continued from previous page)

```
                    # treat minlp as nlp
                    'minlp_as_nlp 0', \
                    # nlp sub-problem max iterations
                    'nlp_maximum_iterations 50', \
                    # 1 = depth first, 2 = breadth first
                    'minlp_branch_method 1', \
                    # maximum deviation from whole number
                    'minlp_integer_tol 0.05', \
                    # covergence tolerance
                    'minlp_gap_tol 0.01']

# Initialize variables
x1 = m.Var(value=1,lb=1,ub=5)
x2 = m.Var(value=5,lb=1,ub=5)
# Integer constraints for x3 and x4
x3 = m.Var(value=5,lb=1,ub=5,integer=True)
x4 = m.Var(value=1,lb=1,ub=5,integer=True)
# Equations
m.Equation(x1*x2*x3*x4>=25)
m.Equation(x1**2+x2**2+x3**2+x4**2==40)
m.Obj(x1*x4*(x1+x2+x3)+x3) # Objective
m.solve(disp=False) # Solve
print('Results')
print('x1: ' + str(x1.value))
print('x2: ' + str(x2.value))
print('x3: ' + str(x3.value))
print('x4: ' + str(x4.value))
print('Objective: ' + str(m.options.objfcnval))
```

Results x1: [1.358909] x2: [4.599279] x3: [4.0] x4: [1.0] Objective: 17.5322673

5.14.11 Moving Horizon Estimation

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

# Estimator Model
m = GEKKO()
m.time = p.time
# Parameters
m.u = m.MV(value=u_meas) #input
m.K = m.FV(value=1, lb=1, ub=3)    # gain
m.tau = m.FV(value=5, lb=1, ub=10) # time constant
# Variables
m.x = m.SV() #state variable
m.y = m.CV(value=y_meas) #measurement
# Equations
m.Equations([m.tau * m.x.dt() == -m.x + m.u,
             m.y == m.K * m.x])
# Options
m.options.IMODE = 5 #MHE
```

(continues on next page)

(continued from previous page)

```
m.options.EV_TYPE = 1
# STATUS = 0, optimizer doesn't adjust value
# STATUS = 1, optimizer can adjust
m.u.STATUS = 0
m.K.STATUS = 1
m.tau.STATUS = 1
m.y.STATUS = 1
# FSTATUS = 0, no measurement
# FSTATUS = 1, measurement used to update model
m.u.FSTATUS = 1
m.K.FSTATUS = 0
m.tau.FSTATUS = 0
m.y.FSTATUS = 1
# DMAX = maximum movement each cycle
m.K.DMAX = 2.0
m.tau.DMAX = 4.0
# MEAS_GAP = dead-band for measurement / model mismatch
m.y.MEAS_GAP = 0.25

# solve
m.solve(disp=False)

# Plot results
plt.subplot(2,1,1)
plt.plot(m.time,u_meas,'b:',label='Input (u) meas')
plt.legend()
plt.subplot(2,1,2)
plt.plot(m.time,y_meas,'gx',label='Output (y) meas')
plt.plot(p.time,p.y.value,'k-',label='Output (y) actual')
plt.plot(m.time,m.y.value,'r--',label='Output (y) estimated')
plt.legend()
plt.show()
```

5.14.12 Model Predictive Control

```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

m = GEKKO()
m.time = np.linspace(0,20,41)

# Parameters
mass = 500
b = m.Param(value=50)
K = m.Param(value=0.8)

# Manipulated variable
p = m.MV(value=0, lb=0, ub=100)
p.STATUS = 1  # allow optimizer to change
p.DCOST = 0.1 # smooth out gas pedal movement
```

(continues on next page)

(continued from previous page)

```
p.DMAX = 20   # slow down change of gas pedal

# Controlled Variable
v = m.CV(value=0)
v.STATUS = 1  # add the SP to the objective
m.options.CV_TYPE = 2 # squared error
v.SP = 40     # set point
v.TR_INIT = 1 # set point trajectory
v.TAU = 5     # time constant of trajectory

# Process model
m.Equation(mass*v.dt() == -v*b + K*b*p)

m.options.IMODE = 6 # control
m.solve(disp=False)

# get additional solution information
import json
with open(m.path+'//results.json') as f:
    results = json.load(f)

plt.figure()
plt.subplot(2,1,1)
plt.plot(m.time,p.value,'b-',label='MV Optimized')
plt.legend()
plt.ylabel('Input')
plt.subplot(2,1,2)
plt.plot(m.time,results['v1.tr'],'k-',label='Reference Trajectory')
plt.plot(m.time,v.value,'r--',label='CV Response')
plt.ylabel('Output')
plt.xlabel('Time')
plt.legend(loc='best')
plt.show()
```

5.14.13 Optimization of Multiple Linked Phases

```
import numpy as np
from gekko import GEKKO
import matplotlib.pyplot as plt

# Initialize gekko model
m = GEKKO()
# Number of collocation nodes
nodes = 3

# Number of phases
n = 5

# Time horizon (for all phases)
m.time = np.linspace(0,1,100)
```

(continues on next page)

(continued from previous page)

```
# Input (constant in IMODE 4)
u = [m.Var(1,lb=-2,ub=2,fixed_initial=False) for i in range(n)]

# Example of same parameter for each phase
tau = 5

# Example of different parameters for each phase
K = [2,3,5,1,4]

# Scale time of each phase
tf = [1,2,4,8,16]

# Variables (one version of x for each phase)
x = [m.Var(0) for i in range(5)]

# Equations (different for each phase)
for i in range(n):
    m.Equation(tau*x[i].dt()/tf[i]==-x[i]+K[i]*u[i])

# Connect phases together at endpoints
for i in range(n-1):
    m.Connection(x[i+1],x[i],1,len(m.time)-1,1,nodes)
    m.Connection(x[i+1],'CALCULATED',pos1=1,node1=1)

# Objective
# Maximize final x while keeping third phase = -1
m.Obj(-x[n-1]+(x[2]+1)**2*100)

# Solver options
m.options.IMODE = 6
m.options.NODES = nodes

# Solve
m.solve()

# Calculate the start time of each phase
ts = [0]
for i in range(n-1):
    ts.append(ts[i] + tf[i])

# Plot
plt.figure()
tm = np.empty(len(m.time))
for i in range(n):
    tm = m.time * tf[i] + ts[i]
    plt.plot(tm,x[i])
```

5.14.14 Additional Examples

- 18 Applications with Python GEKKO
- Dynamic Optimization Course (see Homework Solutions)

- GEKKO Search on APMonitor Documentation
- GEKKO (optimization software) on Wikipedia
- GEKKO Journal Article
- GEKKO Webinar to the AIChE CAST Division

5.15 Support

5.15.1 Bugs

GEKKO is in its early stages and still under active development. If you identify a bug in the package, please submit an issue on GitHub.

If you would like to contribute to GEKKO development, please make pull requests on the GitHub repo.

5.15.2 Questions

Further clarification regarding GEKKO, please visit Stack Overflow with questions tagged *gekko* for human-generated questions and answers. Generative AI is built into Gekko with the *support* module. Queries are sent to a web service and a text response is returned. The web service searches a list of hundreds of related Gekko questions and answers to add question context and receive a more relevant answer.

```
from gekko import support
a = support.agent()
a.ask("Can you optimize the Rosenbrock function?")
```

Create a support agent with *support.agent()* and then ask any questions. Prior questions and answers are stored as context for follow-up questions. User questions are not stored for LLM training. The Gekko AI Assistant web service may be unavailable or slow down during periods of high use.

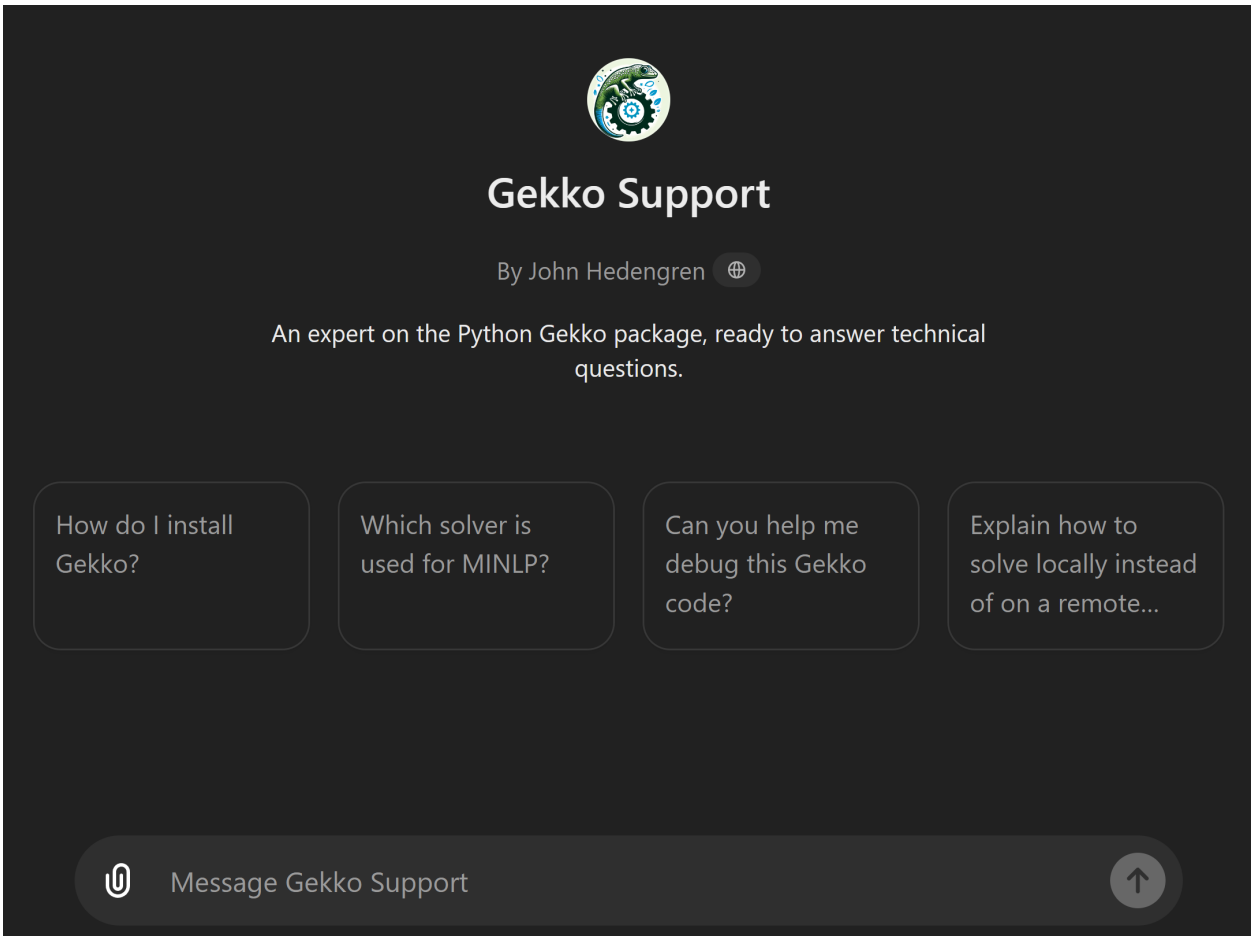
A local Gekko AI Assistant can be run with Retrieval Augmented Generation (RAG) and choice of LLM on a local *ollama* server. Advanced Process Solutions, LLC (APS) is not responsible in any way for the Gekko AI Assistant accuracy, completeness, performance, timeliness, reliability, content, upgrades, or availability of any information and/or software received as a result of the use of the service. See APS terms and conditions.

5.15.3 Advanced Debugging

The files read/written by GEKKO are in the folder indicated in *m._path*. Information on debugging these files is available on APMonitor.

5.15.4 Generative AI

Generative AI can also help with basic and advanced questions. Most Large Language Models (LLMs) have knowledge of the Gekko syntax and are able to help with generating prototype applications, suggesting performance improvements, answering questions that would be found in the documentation, designing applications, and working through errors. A context aware OpenAI GPT model is available for Gekko GenAI Support.



A PDF of the complete Gekko (latest release) documentation is available from this link.

CHAPTER

SIX

OVERVIEW OF GEKKO

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

V

W