gpkit Documentation

Release 0.7.0

MIT Department of Aeronautics and Astronautics

Jun 13, 2018

Contents

GPkit is a Python package for defining and manipulating geometric programming (GP) models.

Our hopes are to bring the mathematics of Geometric Programming into the engineering design process in a disciplined and collaborative way, and to encourage research with and on GPs by providing an easily extensible object-oriented framework.

GPkit abstracts away the backend solver so that users can work directly with engineering equations and optimization concepts. Supported solvers are MOSEK and CVXOPT.

Join our mailing list and/or chatroom for support and examples.

CHAPTER 1

Geometric Programming 101

1.1 What is a GP?

A Geometric Program (GP) is a type of non-linear optimization problem whose objective and constraints have a particular form.

The decision variables must be strictly positive (non-zero, non-negative) quantities. This is a good fit for engineering design equations (which are often constructed to have only positive quantities), but any model with variables of unknown sign (such as forces and velocities without a predefined direction) may be difficult to express in a GP. Such models might be better expressed as *Signomials*.

More precisely, GP objectives and inequalities are formed out of *monomials* and *posynomials*. In the context of GP, a monomial is defined as:

$$f(x) = c x_1^{a_1} x_2^{a_2} ... x_n^{a_n}$$

where c is a positive constant, $x_{1..n}$ are decision variables, and $a_{1..n}$ are real exponents. For example, taking x, y and z to be positive variables, the expressions

$$7x \qquad 4xy^2z \qquad \frac{2x}{y^2z^{0.3}} \qquad \sqrt{2xy}$$

are all monomials. Building on this, a posynomial is defined as a sum of monomials:

$$g(x) = \sum_{k=1}^{K} c_k x_1^{a_1 k} x_2^{a_2 k} ... x_n^{a_n k}$$

For example, the expressions

$$x^2 + 2xy + 1 \qquad 7xy + 0.4(yz)^{-1/3} \qquad 0.56 + \frac{x^{0.7}}{yz}$$

are all posynomials. Alternatively, monomials can be defined as the subset of posynomials having only one term. Using f_i to represent a monomial and g_i to represent a posynomial, a GP in standard form is written as:

$$\begin{array}{ll} \text{minimize} & g_0(x) \\ \text{subject to} & f_i(x) = 1, \quad i = 1, ..., m \\ & g_i(x) \leq 1, \quad i = 1, ..., n \end{array}$$

Boyd et. al. give the following example of a GP in standard form:

$$\begin{array}{ll} \text{minimize} & x^{-1}y^{-1/2}z^{-1} + 2.3xz + 4xyz \\ \text{subject to} & (1/3)x^{-2}y^{-2} + (4/3)y^{1/2}z^{-1} \leq 1 \\ & x + 2y + 3z \leq 1 \\ & (1/2)xy = 1 \end{array}$$

1.2 Why are GPs special?

Geometric programs have several powerful properties:

1. Unlike most non-linear optimization problems, large GPs can be **solved extremely quickly**.
2. If there exists an optimal solution to a GP, it is guaranteed to be **globally optimal**.
3. Modern GP solvers require **no initial guesses** or tuning of solver parameters.

These properties arise because GPs become *convex optimization problems* via a logarithmic transformation. In addition to their mathematical benefits, recent research has shown that many practical problems can be formulated as GPs or closely approximated as GPs.

1.3 What are Signomials / Signomial Programs?

When the coefficients in a posynomial are allowed to be negative (but the variables stay strictly positive), that is called a Signomial.

A Signomial Program has signomial constraints. While they cannot be solved as quickly or to global optima, because they build on the structure of a GP they can often be solved more quickly than a generic nonlinear program. More information can be found under *Signomial Programming*.

1.4 Where can I learn more?

To learn more about GPs, refer to the following resources:

- A tutorial on geometric programming, by S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi.
- Convex optimization, by S. Boyd and L. Vandenberghe.
- Geometric Programming for Aircraft Design Optimization, Hoburg, Abbeel 2014

CHAPTER 2

GPkit Overview

GPkit is a Python package for defining and manipulating geometric programming (GP) models, abstracting away the backend solver.

Our hopes are to bring the mathematics of Geometric Programming into the engineering design process in a disciplined and collaborative way, and to encourage research with and on GPs by providing an easily extensible object-oriented framework.

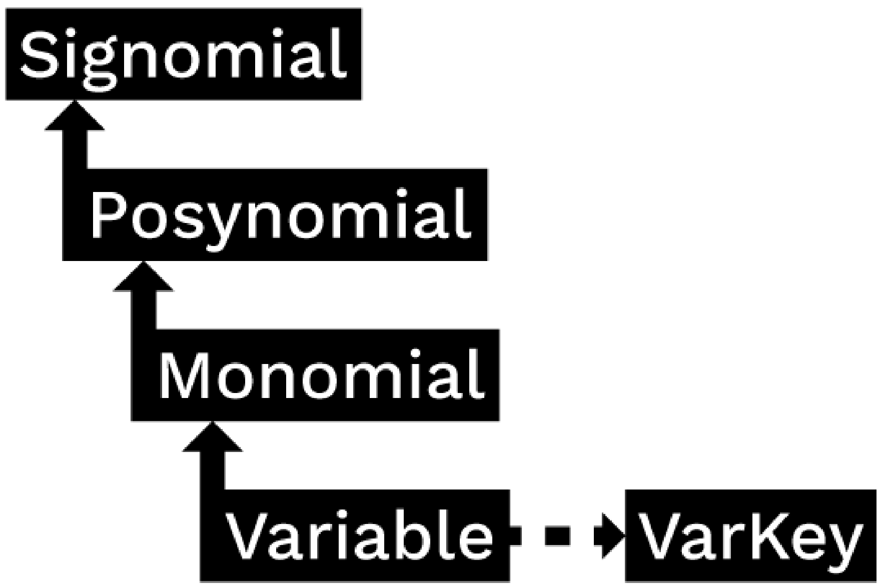
2.1 Symbolic expressions

GPkit is a limited symbolic algebra language, allowing only for the creation of geometric program compatible equations (or signomial program compatible ones, if signomial programming is enabled). As mentioned in *Geometric Programming 101*, one can view monomials as posynomials with a single term, and posynomials as signomials that have only positive coefficients. The inheritance structure of these objects in GPkit follows this mathematical basis.

2.2 Substitution

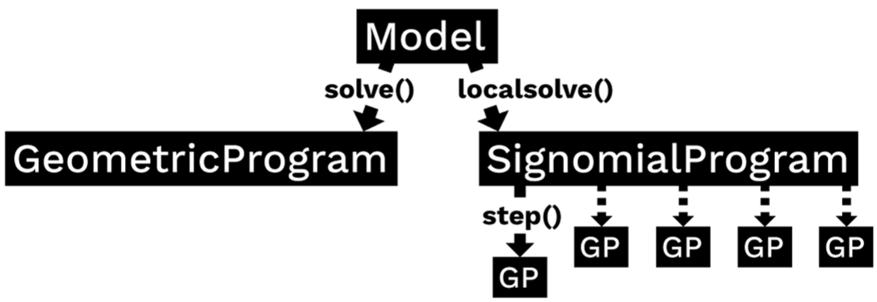
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable's "name". It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric indexing is an important part of the GPkit framework, because it allows a user to keep variable information local and modular.

GPkit keeps its internal representation of objects entirely symbolic until it solves. This means that any expression or Model object can replace any instance of a variable (as represented by a VarKey) with a number, new VarKey, or even an entire Monomial at any time with the `.sub()` method.


Signomial
Posynomial
Monomial
Variable
VarKey

2.3 Model objects

In GPkit, a `Model` object represents a symbolic problem declaration. That problem may be either GP-compatible or SP-compatible. To avoid confusion, calling the `solve()` method on a model will either attempt to solve it for a global optimum (if it's a GP) or return an error immediately (if it's an SP). Similarly, calling `localsolve()` will either start the process of SP-solving (stepping through a sequence of GP-approximations) or return an error for GP-compatible Models. This framework is illustrated below.



CHAPTER 3

Installation

1. If you are on Mac or Windows, we recommend installing Anaconda. Alternatively, install pip and create a virtual environment.
2. (optional) Install the MOSEK solver as directed below
3. Run `pip install gpkit` in the appropriate terminal or command prompt.
4. Open a Python prompt and run `import gpkit` to finish installation and run unit tests.

If you encounter any bugs please email `gpkit@mit.edu` or raise a GitHub issue.

3.1 Installing MOSEK

GPkit interfaces with two off the shelf solvers: cvxopt, and MOSEK. Cvxopt is open source and installed by default; MOSEK requires a commercial licence or (free) academic license.

Mac OS X

- If `which gcc` does not return anything, install the Apple Command Line Tools.
- **Download MOSEK, then:**
  - Move the `mosek` folder to your home directory
  - Follow these steps for Mac.
  - Request an academic license file and put it in `~/mosek/`

Linux

- **Download MOSEK, then:**
  - Move the `mosek` folder to your home directory
  - Follow these steps for Linux.
  - Request an academic license file and put it in `~/mosek/`

Windows

- **Download MOSEK, then:**
  - Follow these steps for Windows.
  - Request an academic license file and put it in `C:\Users\(your_username)\mosek\`
  - **Make sure `gcc` is on your system path.**
    * To do this, type `gcc` into a command prompt.
    * If you get `executable not found`, then install the 64-bit version (x86_64 installer architecture dropdown option) with GCC version 6.4.0 or older of mingw.
    * In an Anaconda command prompt (or equivalent), run `cd C:\Program Files\mingw-64\x86_64-6.4.0-posix-seh-rt_v5-rev0\` (or whatever corresponds to the correct installation directory; note that if mingw is in `Program Files (x86)` instead of `Program Files` you've installed the 32-bit version by mistake)
    * Run `mingw-64` to add it to your executable path. For step 3 of the install process you'll need to run `pip install gpkit` from this prompt.

3.2 Debugging your installation

You may need to rebuild GPkit if any of the following occur:

- You install MOSEK after installing GPkit
- You see `Could not load settings file.` when importing GPkit, or
- `Could not load MOSEK library: ImportError('$HOME/.gpkit/expopt.so not found.')`

To rebuild GPkit run `python -c "from gpkit.build import rebuild; rebuild()"`.

If that doesn't solve your issue then try the following:

- `pip uninstall gpkit`
- `pip install --no-cache-dir --no-deps gpkit`
- `python -c "import gpkit.tests; gpkit.tests.run()"`
- If any tests fail, please email `gpkit@mit.edu` or raise a GitHub issue.

3.3 Bleeding-edge installations

Active developers may wish to install the latest GPkit directly from Github. To do so,

1. `pip uninstall gpkit` to uninstall your existing GPkit.
2. `git clone https://github.com/convexengineering/gpkit.git`
3. `pip install -e gpkit` to install that directory as your environment-wide GPkit.
4. `cd ..; python -c "import gpkit.tests; gpkit.tests.run()"` to test your installation from a non-local directory.

CHAPTER 4

Getting Started

GPkit is a Python package, so we assume basic familiarity with Python: if you're new to Python we recommend you take a look at Learn Python.

Otherwise, *install GPkit* and import away:

```
from gpkit import Variable, VectorVariable, Model
```

4.1 Declaring Variables

Instances of the `Variable` class represent scalar variables. They create a `VarKey` to store the variable's name, units, a description, and value (if the Variable is to be held constant), as well as other metadata.

4.1.1 Free Variables

```
# Declare a variable, x
x = Variable("x")

# Declare a variable, y, with units of meters
y = Variable("y", "m")

# Declare a variable, z, with units of meters, and a description
z = Variable("z", "m", "A variable called z with units of meters")
```

4.1.2 Fixed Variables

To declare a variable with a constant value, use the `Variable` class, as above, but put a number before the units:

```
# Declare \rho equal to 1.225 kg/m^3.
# NOTE: in python string literals, backslashes must be doubled
rho = Variable("\\rho", 1.225, "kg/m^3", "Density of air at sea level")
```

In the example above, the key name `"\rho"` is for LaTeX printing (described later). The unit and description arguments are optional.

```
#Declare pi equal to 3.14
pi = Variable("\\pi", 3.14)
```

4.1.3 Vector Variables

Vector variables are represented by the `VectorVariable` class. The first argument is the length of the vector. All other inputs follow those of the `Variable` class.

```
# Declare a 3-element vector variable "x" with units of "m"
x = VectorVariable(3, "x", "m", "Cube corner coordinates")
x_min = VectorVariable(3, "x", [1, 2, 3], "m", "Cube corner minimum")
```

4.2 Creating Monomials and Posynomials

Monomial and posynomial expressions can be created using mathematical operations on variables.

```
# create a Monomial term xy^2/z
x = Variable("x")
y = Variable("y")
z = Variable("z")
m = x * y**2 / z
type(m)  # gpkit.nomials.Monomial
```

```
# create a Posynomial expression x + xy^2
x = Variable("x")
y = Variable("y")
p = x + x * y**2
type(p)  # gpkit.nomials.Posynomial
```

4.3 Declaring Constraints

`Constraint` objects represent constraints of the form `Monomial >= Posynomial` or `Monomial == Monomial` (which are the forms required for GP-compatibility).

Note that constraints must be formed using <=, >=, or == operators, not < or >.

```
# consider a block with dimensions x, y, z less than 1
# constrain surface area less than 1.0 m^2
x = Variable("x", "m")
y = Variable("y", "m")
z = Variable("z", "m")
S = Variable("S", 1.0, "m^2")
c = (2*x*y + 2*x*z + 2*y*z <= S)
type(c)  # gpkit.nomials.PosynomialInequality
```

4.4 Formulating a Model

The `Model` class represents an optimization problem. To create one, pass an objective and list of Constraints.

By convention, the objective is the function to be *minimized*. If you wish to *maximize* a function, take its reciprocal. For example, the code below creates an objective which, when minimized, will maximize `x*y*z`.

```
objective = 1/(x*y*z)
constraints = [2*x*y + 2*x*z + 2*y*z <= S,
               x >= 2*y]
m = Model(objective, constraints)
```

4.5 Solving the Model

When solving the model you can change the level of information that gets printed to the screen with the `verbosity` setting. A verbosity of 1 (the default) prints warnings and timing; a verbosity of 2 prints solver output, and a verbosity of 0 prints nothing.

```
sol = m.solve(verbosity=0)
```

4.6 Printing Results

The solution object can represent itself as a table:

```
print sol.table()
```

```
Cost
----
 15.59 [1/m**3]

Free Variables
--------------
x : 0.5774   [m]
y : 0.2887   [m]
z : 0.3849   [m]

Constants
---------
S : 1   [m**2]

Sensitivities
-------------
S : -1.5
```

We can also print the optimal value and solved variables individually.

```
print "The optimal value is %s." % sol["cost"]
print "The x dimension is %s." % sol(x)
print "The y dimension is %s." % sol["variables"]["y"]
```

```
The optimal value is 15.5884619886.
The x dimension is 0.5774 meter.
The y dimension is 0.2887 meter.
```

4.7 Sensitivities and dual variables

When a GP is solved, the solver returns not just the optimal value for the problem's variables (known as the "primal solution") but also the effect that relaxing each constraint would have on the overall objective (the "dual solution").

From the dual solution GPkit computes the sensitivities for every fixed variable in the problem. This can be quite useful for seeing which constraints are most crucial, and prioritizing remodeling and assumption-checking.

4.7.1 Using variable sensitivities

Fixed variable sensitivities can be accessed from a SolutionArray's `["sensitivities"]["constants"]` dict, as in this example:

```
import gpkit
x = gpkit.Variable("x")
x_min = gpkit.Variable("x_{min}", 2)
sol = gpkit.Model(x, [x_min <= x]).solve()
assert sol["sensitivities"]["constants"][x_min] == 1
```

These sensitivities are actually log derivatives ($\frac{d\log(y)}{d\log(x)}$); whereas a regular derivative is a tangent line, these are tangent monomials, so the `1` above indicates that `x_min` has a linear relation with the objective. This is confirmed by a further example:

```
import gpkit
x = gpkit.Variable("x")
x_squared_min = gpkit.Variable("x^2_{min}", 2)
sol = gpkit.Model(x, [x_squared_min <= x**2]).solve()
assert sol["sensitivities"]["constants"][x_squared_min] == 2
```

CHAPTER 5

Debugging Models

A number of errors and warnings may be raised when attempting to solve a model. A model may be primal infeasible: there is no possible solution that satisfies all constraints. A model may be dual infeasible: the optimal value of one or more variables is 0 or infinity (negative and positive infinity in logspace).

For a GP model that does not solve, solvers may be able to prove its primal or dual infeasibility, or may return an unknown status.

GPkit contains several tools for diagnosing which constraints and variables might be causing infeasibility. The first thing to do with a model `m` that won't solve is to run `m.debug()`, which will search for changes that would make the model feasible:

```
"Debug examples"

from gpkit import Variable, Model, units

x = Variable("x", "ft")
x_min = Variable("x_min", 2, "ft")
x_max = Variable("x_max", 1, "ft")
y = Variable("y", "volts")

m = Model(x/y, [x <= x_max, x >= x_min])
m.debug()

print "# Now let's try a model unsolvable with relaxed constants\n"

Model(x, [x <= units("inch"), x >= units("yard")]).debug()

print "# And one that's only unbounded\n"

# the value of x_min was used up in the previous model!
x_min = Variable("x_min", 2, "ft")
Model(x/y, [x >= x_min]).debug()
```

```
< DEBUGGING >
> Trying with bounded variables and relaxed constants:

Solves with these variables bounded:
   value near upper bound: y
 sensitive to upper bound: y

and these constants relaxed:
  x_min [ft]: relaxed from 2 to 1

>> Success!

# Now let's try a model unsolvable with relaxed constants

< DEBUGGING >
> Trying with bounded variables and relaxed constants:
>> Failure.
> Trying with relaxed constraints:

Solves with these constraints relaxed:
   1: 3500% relaxed, from 3*x**-1 <= 1
                       to 3*x**-1 <= 36

>> Success!

# And one that's only unbounded

< DEBUGGING >
> Trying with bounded variables and relaxed constants:

Solves with these variables bounded:
   value near upper bound: y
 sensitive to upper bound: y

>> Success!
```

Note that certain modeling errors (such as omitting or forgetting a constraint) may be difficult to diagnose from this output.

5.1 Potential errors and warnings

- **RuntimeWarning: final status of solver 'mosek' was 'DUAL_INFEAS_CER', not 'optima**
  - The solver found a certificate of dual infeasibility: the optimal value of one or more variables is 0 or infinity. See *Dual Infeasibility* below for debugging advice.
- **RuntimeWarning: final status of solver 'mosek' was 'PRIM_INFEAS_CER', not 'optima**
  - The solver found a certificate of primal infeasibility: no possible solution satisfies all constraints. See *Primal Infeasibility* below for debugging advice.
- **RuntimeWarning: final status of solver 'cvxopt' was 'unknown', not 'optimal' or Rur**

- The solver could not solve the model or find a certificate of infeasibility. This may indicate a dual infeasible model, a primal infeasible model, or other numerical issues. Try debugging with the techniques in *Dual* and *Primal Infeasibility* below.

- **`RuntimeWarning: Primal solution violates constraint: 1.0000149786 is greater tha`**
  - this warning indicates that the solver-returned solution violates a constraint of the model, likely because the solver's tolerance for a final solution exceeds GPkit's tolerance during solution checking. This is sometimes seen in dual infeasible models, see *Dual Infeasibility* below. If you run into this, please note on this GitHub issue your solver and operating system.

- **`RuntimeWarning: Dual cost nan does not match primal cost 1.00122315152`**
  - Similarly to the above, this warning may be seen in dual infeasible models, see *Dual Infeasibility* below.

5.2 Dual Infeasibility

In some cases a model will not solve because the optimal value of one or more variables is 0 or infinity (negative or positive infinity in logspace). Such a problem is *dual infeasible* because the GP's dual problem, which determines the optimal values of the sensitivites, does not have any feasible solution. If the solver can prove that the dual is infeasible, it will return a dual infeasibility certificate. Otherwise, it may finish with a solution status of `unknown`.

`gpkit.constraints.bounded.Bounded` is a simple tool that can be used to detect unbounded variables and get dual infeasible models to solve by adding extremely large upper bounds and extremely small lower bounds to all variables in a ConstraintSet.

When a model with a Bounded ConstraintSet is solved, it checks whether any variables slid off to the bounds, notes this in the solution dictionary and prints a warning (if verbosity is greater than 0).

For example, Mosek returns `DUAL_INFEAS_CER` when attempting to solve the following model:

```
"Demonstrate a trivial unbounded variable"
from gpkit import Variable, Model
from gpkit.constraints.bounded import Bounded

x = Variable("x")

constraints = [x >= 1]

m = Model(1/x, constraints)  # MOSEK returns DUAL_INFEAS_CER on .solve()
m = Model(1/x, Bounded(constraints))
# by default, prints bounds warning during solve
sol = m.solve(verbosity=0)
print sol.summary()
print "sol['boundedness'] is:", sol["boundedness"]
```

Upon viewing the printed output,

```

Solves with these variables bounded:
   value near upper bound: x
 sensitive to upper bound: x
```

(continues on next page)

(continued from previous page)

```
Cost
----
 1e-30

Free Variables
--------------
x : 1e+30

sol['boundedness'] is: {'value near upper bound': set([x]), 'sensitive to␣
↪upper bound': set([x])}
```

The problem, unsurprisingly, is that the cost `1/x` has no lower bound because `x` has no upper bound.

For details read the Bounded docstring.

5.3 Primal Infeasibility

A model is primal infeasible when there is no possible solution that satisfies all constraints. A simple example is presented below.

```
"A simple primal infeasible example"
from gpkit import Variable, Model

x = Variable("x")
y = Variable("y")

m = Model(x*y, [
    x >= 1,
    y >= 2,
    x*y >= 0.5,
    x*y <= 1.5
])

# m.solve()  # raises uknown on cvxopt
             # and PRIM_INFEAS_CER on mosek
```

It is not possible for `x*y` to be less than 1.5 while `x` is greater than 1 and `y` is greater than 2.

A common bug in large models that use `substitutions` is to substitute overly constraining values in for variables that make the model primal infeasible. An example of this is given below.

```
"Another simple primal infeasible example"
from gpkit import Variable, Model

#Make the necessary Variables
x = Variable("x")
y = Variable("y", 2)

#make the constraints
constraints = [
    x >= 1,
    0.5 <= x*y,
    x*y <= 1.5
```

(continues on next page)

(continued from previous page)

```
    ]

#declare the objective
objective = x*y

#construct the model
m = Model(objective, constraints)

#solve the model
#raises RuntimeWarning uknown on cvxopt and RuntimeWarning
#PRIM_INFES_CER with mosek
#m.solve()
```

Since y is now set to 2 and x can be no less than 1, it is again impossible for x*y to be less than 1.5 and the model is primal infeasible. If y was instead set to 1, the model would be feasible and the cost would be 1.

5.3.1 Relaxation

If you suspect your model is primal infeasible, you can find the nearest primal feasible version of it by relaxing constraints: either relaxing all constraints by the smallest number possible (that is, dividing the less-than side of every constraint by the same number), relaxing each constraint by its own number and minimizing the product of those numbers, or changing each constant by the smallest total percentage possible.

```
"Relaxation examples"

from gpkit import Variable, Model
x = Variable("x")
x_min = Variable("x_min", 2)
x_max = Variable("x_max", 1)
m = Model(x, [x <= x_max, x >= x_min])
print "Original model"
print "=============="
print m
print
# m.solve()  # raises a RuntimeWarning!

print "With constraints relaxed equally"
print "================================"
from gpkit.constraints.relax import ConstraintsRelaxedEqually
allrelaxed = ConstraintsRelaxedEqually(m)
mr1 = Model(allrelaxed.relaxvar, allrelaxed)
print mr1
print mr1.solve(verbosity=0).table()  # solves with an x of 1.414
print

print "With constraints relaxed individually"
print "====================================="
from gpkit.constraints.relax import ConstraintsRelaxed
constraintsrelaxed = ConstraintsRelaxed(m)
mr2 = Model(constraintsrelaxed.relaxvars.prod() * m.cost**0.01,
            # add a bit of the original cost in
            constraintsrelaxed)
print mr2
```

(continues on next page)

(continued from previous page)

```
print mr2.solve(verbosity=0).table()  # solves with an x of 1.0
print

print "With constants relaxed individually"
print "===================================="
from gpkit.constraints.relax import ConstantsRelaxed
constantsrelaxed = ConstantsRelaxed(m)
mr3 = Model(constantsrelaxed.relaxvars.prod() * m.cost**0.01,
            # add a bit of the original cost in
            constantsrelaxed)
print mr3
print mr3.solve(verbosity=0).table()  # brings x_min down to 1.0
print
```

```
Original model
==============

  # minimize
        x
  # subject to
        x <= x_max
        x >= x_min

With constraints relaxed equally
================================

  # minimize
        C_Relax
  # subject to
        C_Relax >= x*x_max**-1
        C_Relax >= x**-1*x_min
        C_Relax >= 1

Cost
----
 1.414

Free Variables
--------------
x : 1.414

  | Relax
C : 1.414

Constants
---------
x_max : 1
x_min : 2

Sensitivities
-------------
x_min : +0.5
x_max : -0.5


With constraints relaxed individually
```

(continues on next page)

(continued from previous page)

```
=====================================

  # minimize
        C_Relax.1_(0,)*C_Relax.1_(1,)*x**0.01
  # subject to
        C_Relax.1 >= [gpkit.Monomial(x*x_max**-1), gpkit.Monomial(x**-1*x_
↪min)]
        C_Relax.1 >= 1

Cost
----
 2

Free Variables
--------------
x : 1

  | Relax.1
C : [ 1         2         ]

Constants
---------
x_max : 1
x_min : 2

Sensitivities
-------------
x_min : +1
x_max : -0.99


With constants relaxed individually
===================================

  # minimize
        x**0.01*x_max_Relax.2*x_min_Relax.2
  # subject to
        x <= x_max
        x >= x_min
        x_max_Relax.2 >= 1
        x_max >= x_max_Relax.2**-1*x_max_{before}_Relax.2
        x_max <= x_max_Relax.2*x_max_{before}_Relax.2
        x_min_Relax.2 >= 1
        x_min >= x_min_Relax.2**-1*x_min_{before}_Relax.2
        x_min <= x_min_Relax.2*x_min_{before}_Relax.2

Cost
----
 2

Free Variables
--------------
    x : 1
x_max : 1
x_min : 1

      | Relax.2
```

(continues on next page)

(continued from previous page)

```
x_max : 1
x_min : 2

Constants
---------
x_max_{before} : 1
x_min_{before} : 2

Sensitivities
-------------
x_min : +1
x_max : -0.99
```

CHAPTER 6

Visualization and Interaction

6.1 Sankey Diagrams

6.1.1 Requirements

- jupyter notebook
- ipysankeywidget

6.1.2 Example

Code in this section uses the CE solar model

```
from solar import *
Vehicle = Aircraft(Npod=1, sp = False)
M = Mission(Vehicle, latitude=[20])
M.cost = M[M.aircraft.Wtotal]
sol = M.solve()

from gpkit.interactive.sankey import Sankey
Sankey(M).diagram(M.aircraft.Wtotal)
```

```
(objective) adds +1 to the sensitivity of Wtotal_Aircraft
(objective) is Wtotal_Aircraft [lbf]

 adds +0.0075 to the overall sensitivity of Wtotal_Aircraft
 is Wtotal_Aircraft <= 0.5*CL_Mission/Climb/AircraftDrag/WingAero_(0,)*S_
↪Aircraft/Wing/Planform.2*V_Mission/Climb_(0, 0)**2*rho_Mission/Climb_(0, 0)

 adds +0.0117 to the overall sensitivity of Wtotal_Aircraft
 is Wtotal_Aircraft <= 0.5*CL_Mission/Climb/AircraftDrag/WingAero_(1,)*S_
↪Aircraft/Wing/Planform.2*V_Mission/Climb_(0, 1)**2*rho_Mission/Climb_(0, 1)
```

(continues on next page)

(continued from previous page)

```
```

6.1.3 Explanation

Sankey diagrams can be used to visualize sensitivity structure in a model. A blue flow from a constraint to its parent indicates that the sensitivity of the chosen variable (or of making the constraint easier, if no variable is given) is negative; that is, the objective of the overall model would improve if that variable's value were increased *in that constraint alone*. Red indicates a positive sensitivity: the objective and the the constraint 'want' that variable's value decreased. Gray flows indicate a sensitivity whose absolute value is below `1e-7`, i.e. a constraint that is inactive for that variable. Where equal red and blue flows meet, they cancel each other out to gray.

6.1.4 Usage

Variables

In a Sankey diagram of a variable, the variable is on the left with its final sensitivity; to the right of it are all constraints that variable is in.

Free

Free variables have an overall sensitivity of 0, so this visualization shows how the various pressures on that variable in all its constraints cancel each other out; this can get quite complex, as in this diagram of the pressures on wingspan:

```
Sankey(M).diagram(M.aircraft.b)
```

Fixed

Fixed variables can have a nonzero overall sensitivity. Sankey diagrams can how that sensitivity comes together:

Sankey(M).diagram(M['vgust'])

Equivalent Variables

If any variables are equal to the diagram's variable (modulo some constant factor; e.g. `2*x == y` counts for this, as does `2*x <= y` if the constraint is sensitive), they are found and plotted at the same time, and all shown on the left. The constraints responsible for this are shown next to their labels.

```
Sankey(M).sorted_by('constraints', 11)
```

Models

When created without a variable, the diagram shows the sensitivity of every named model to becoming locally easier. Because derivatives are additive, these sensitivities are too: a model's sensitivity is equal to the sum of its constraints' sensitivities. Gray lines in this diagram indicate models without any tight constraints.

```
Sankey(M).diagram(left=60, right=90, width=1050)
```

6.1.5 Syntax

Code	Result
`s = Sankey(M)`	Creates Sankey object of a given model
`s.diagram(vars)`	Creates the diagram in a way Jupyter knows how to present
`d = s.diagram()`	**Don't do this!** Captures output, preventing Jupyter from seeing it.
`s.diagram(width=...)`	Sets width in pixels. Same for height.
`s.diagram(left=...)`	Sets top margin in pixels. Same for right, top. bottom. Use if the left-hand text is being cut off.
`s.diagram(flowright=True)`	Shows the variable / top constraint on the right instead of the left.
`s.sorted_by("maxflow", 0)`	Creates diagram of the variable with the largest single constraint sensitivity. (change the `0` index to go down the list)
`s.sorted_by("constraints", 0)`	Creates diagram of the variable that's in the most constraints. (change the `0` index to go down the list)

6.2 Plotting a 1D Sweep

Methods exist to facilitate creating, solving, and plotting the results of a single-variable sweep (see *Sweeps* for details). Example usage is as follows:

```python
"Demonstrates manual and auto sweeping and plotting"
import matplotlib as mpl
mpl.use('Agg')
# comment out the lines above to show figures in a window
import numpy as np
from gpkit import Model, Variable, units

x = Variable("x", "m", "Swept Variable")
y = Variable("y", "m^2", "Cost")
m = Model(y, [y >= (x/2)**-0.5 * units.m**2.5 + 1*units.m**2, y >= (x/2)**2])

# arguments are: model, swept: values, posnomial for y-axis
sol = m.sweep({x: np.linspace(1, 3, 20)}, verbosity=0)
f, ax = sol.plot(y)
ax.set_title("Manually swept (20 points)")
f.show()
```

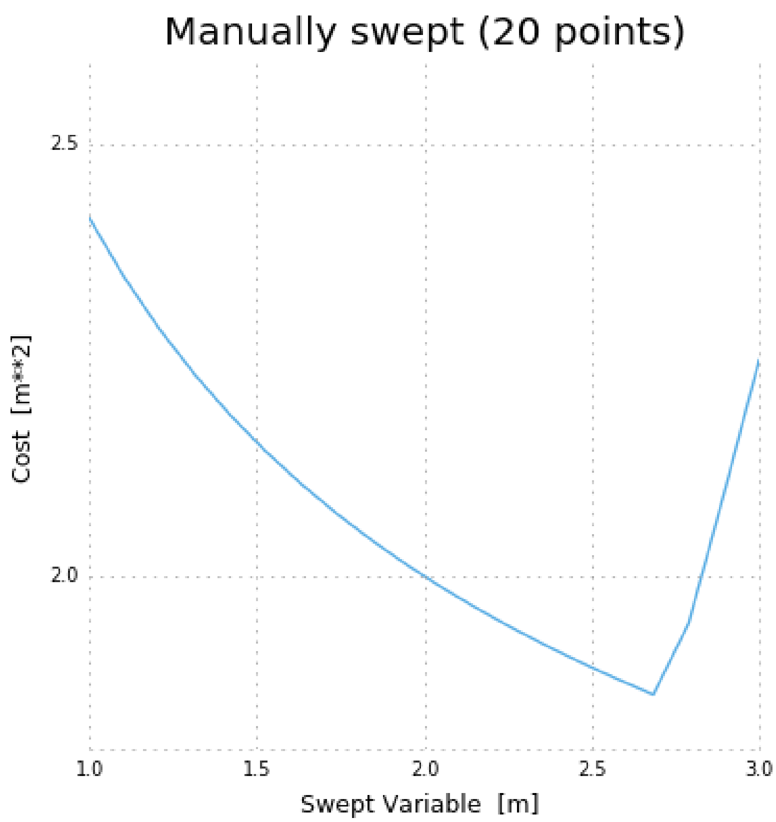
(continues on next page)

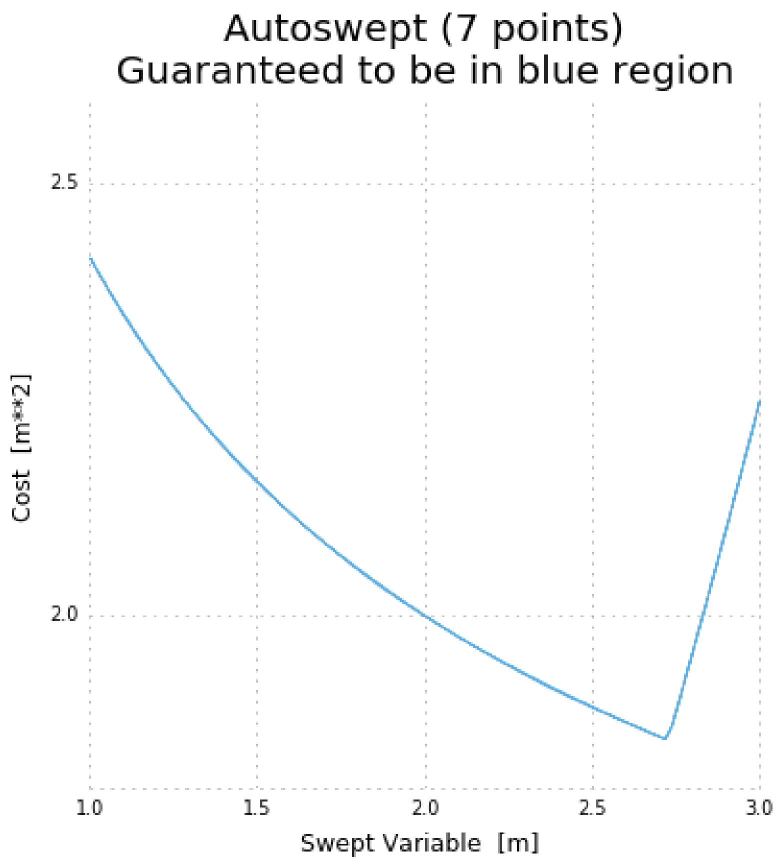
(continued from previous page)

```
f.savefig("plot_sweep1d.png")

# arguments are: model, swept: (min, max, optional logtol), posnomial for y-
↪axis
sol = m.autosweep({x: (1, 3)}, tol=0.001, verbosity=0)
f, ax = sol.plot(y)
ax.set_title("Autoswept (7 points)\nGuaranteed to be in blue region")
f.show()
f.savefig("plot_autosweep1d.png")
```

Which results in:




Autoswept (7 points)
Guaranteed to be in blue region
2.5
2.0
Cost [m**2]
1.0
1.5
2.0
2.5
3.0
Swept Variable [m]

CHAPTER 7

Building Complex Models

7.1 Checking for result changes

Tracking the effects of changes to complex models can get out of hand; we recommend saving solutions with `sol.save()`, then checking that new solutions are almost equivalent with `sol1.almost_equal(sol2)` and/or `print sol1.diff(sol2)`, as shown below.

```
import cPickle as pickle
...  # build the model
sol = m.solve()
# uncomment the line below to verify a new model
# sol.save("last_verified.sol")
last_verified_sol = pickle.load(open("last_verified.sol"))
if not sol.almost_equal(last_verified_sol, reltol=1e-3):
    print last_verified_sol.diff(sol)

# Note you can replace the last three lines above with
print sol.diff("last_verified.sol")
# if you don't mind doing the diff in that direction.
```

You can also check differences between swept solutions, or between a point solution and a sweep.

7.2 Inheriting from `Model`

GPkit encourages an object-oriented modeling approach, where the modeler creates objects that inherit from Model to break large systems down into subsystems and analysis domains. The benefits of this approach include modularity, reusability, and the ability to more closely follow mental models of system hierarchy. For example: two different models for a simple beam, designed by different modelers, should be able to be used interchangeably inside another subsystem (such as an aircraft wing) without either modeler having to write specifically with that use in mind.

When you create a class that inherits from Model, write a `.setup()` method to create the model's variables and return its constraints. `GPkit.Model.__init__` will call that method and automatically add your model's name and unique ID to any created variables.

Variables created in a `setup` method are added to the model even if they are not present in any constraints. This allows for simplistic 'template' models, which assume constant values for parameters and can grow incrementally in complexity as those variables are freed.

At the end of this page a detailed example shows this technique in practice.

7.3 Accessing Variables in Models

GPkit provides several ways to access a Variable in a `Model` (or `ConstraintSet`):

- using `Model.variables_byname(key)`. This returns all Variables in the Model, as well as in any submodels, that match the key.
- using `Model.topvar(key)`. This returns the top-level Variable that matches the key. The Variable must appear at the top level, not in a submodel.
- using `Model.__getitem__`. `Model[key]` returns the only variable matching the key, even if the match occurs in a submodel. If multiple variables match the key, an error is raised.

These methods are illustrated in the following example.

```
"Demo of accessing variables in models"
from gpkit import Model, Variable


class Battery(Model):
    """A simple battery

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
    E

    """
    def setup(self):
        h = Variable("h", 200, "Wh/kg", "specific energy")
        E = self.E = Variable("E", "MJ", "stored energy")
        m = self.m = Variable("m", "lb", "battery mass")
        return [E <= m*h]


class Motor(Model):
    """Electric motor

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
```

(continues on next page)

(continued from previous page)

```
    Pmax

    """
    def setup(self):
        m = self.m = Variable("m", "lb", "motor mass")
        f = Variable("f", 20, "lb/hp", "mass per unit power")
        Pmax = self.Pmax = Variable("P_{max}", "hp", "max output power")
        return [m >= f*Pmax]


class PowerSystem(Model):
    """A battery powering a motor

    Upper Unbounded
    ---------------
    m

    Lower Unbounded
    ---------------
    E, Pmax

    """
    def setup(self):
        battery, motor = Battery(), Motor()
        components = [battery, motor]
        m = self.m = Variable("m", "lb", "mass")
        self.E = battery.E
        self.Pmax = motor.Pmax

        return [components,
                m >= sum(comp.m for comp in components)]

PS = PowerSystem()
print "Getting the only var 'E': ", PS["E"]
print "The top-level var 'm': ", PS.m
print "All the variables 'm': ", PS.variables_byname("m")
```

```
Getting the only var 'E':  E_PowerSystem/Battery [MJ]
The top-level var 'm':  m_PowerSystem [lb]
All the variables 'm':  [gpkit.Variable(m_PowerSystem [lb]), gpkit.
↪Variable(m_PowerSystem/Battery [lb]), gpkit.Variable(m_PowerSystem/Motor␣
↪[lb])]
```

7.4 Vectorization

`gpkit.Vectorize` creates an environment in which Variables are created with an additional dimension:

```
"from gpkit/tests/t_vars.py"

def test_shapes(self):
    with gpkit.Vectorize(3):
        with gpkit.Vectorize(5):
            y = gpkit.Variable("y")
```

(continues on next page)

(continued from previous page)

```
            x = gpkit.VectorVariable(2, "x")
        z = gpkit.VectorVariable(7, "z")

    self.assertEqual(y.shape, (5, 3))
    self.assertEqual(x.shape, (2, 5, 3))
    self.assertEqual(z.shape, (7, 3))
```

This allows models written with scalar constraints to be created with vector constraints:

```
"Vectorization demonstration"
from gpkit import Model, Variable, Vectorize

class Test(Model):
    """A simple scalar model

    Upper Unbounded
    ---------------
    x
    """
    def setup(self):
        x = self.x = Variable("x")
        return [x >= 1]

print "SCALAR"
m = Test()
m.cost = m["x"]
print m.solve(verbosity=0).summary()

print "__________\n"
print "VECTORIZED"
with Vectorize(3):
    m = Test()
m.cost = m["x"].prod()
m.append(m["x"][1] >= 2)
print m.solve(verbosity=0).summary()
```

```
SCALAR

Cost
----
 1

Free Variables
--------------
x : 1

__________

VECTORIZED

Cost
----
 2

Free Variables
--------------
```

(continues on next page)

(continued from previous page)

```
x : [ 1          2          1         ]
```

7.5 Multipoint analysis modeling

In many engineering models, there is a physical object that is operated in multiple conditions. Some variables correspond to the design of the object (size, weight, construction) while others are vectorized over the different conditions (speed, temperature, altitude). By combining named models and vectorization we can create intuitive representations of these systems while maintaining modularity and interoperability.

In the example below, the models `Aircraft` and `Wing` have a `.dynamic()` method which creates instances of `AircraftPerformance` and `WingAero`, respectively. The `Aircraft` and `Wing` models create variables, such as size and weight without fuel, that represent a physical object. The `dynamic` models create properties that change based on the flight conditions, such as drag and fuel weight.

This means that when an aircraft is being optimized for a mission, you can create the aircraft (`AC` in this example) and then pass it to a `Mission` model which can create vectorized aircraft performance models for each flight segment and/or flight condition.

```
"""Modular aircraft concept"""
import cPickle as pickle
import numpy as np
from gpkit import Model, Vectorize, parse_variables


class AircraftP(Model):
    """Aircraft flight physics: weight <= lift, fuel burn

    Variables
    ---------
    Wfuel  [lbf]  fuel weight
    Wburn  [lbf]  segment fuel burn

    Upper Unbounded
    ---------------
    Wburn, aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
    ---------------
    Wfuel, aircraft.W, state.mu

    """
    def setup(self, aircraft, state):
        self.aircraft = aircraft
        self.state = state
        exec parse_variables(AircraftP.__doc__)

        self.wing_aero = aircraft.wing.dynamic(aircraft.wing, state)
        self.perf_models = [self.wing_aero]

        W = aircraft.W
        S = aircraft.wing.S

        V = state.V
```

(continues on next page)

(continued from previous page)

```
        rho = state.rho

        D = self.wing_aero.D
        CL = self.wing_aero.CL

        return [W + Wfuel <= 0.5*rho*CL*S*V**2,
                Wburn >= 0.1*D], self.perf_models


class Aircraft(Model):
    """The vehicle model

    Variables
    ---------
    W   [lbf]   weight

    Upper Unbounded
    ---------------
    W

    Lower Unbounded
    ---------------
    wing.c, wing.S
    """
    def setup(self):
        exec parse_variables(Aircraft.__doc__)
        self.fuse = Fuselage()
        self.wing = Wing()
        self.components = [self.fuse, self.wing]

        return self.components, W >= sum(c.W for c in self.components)

    dynamic = AircraftP


class FlightState(Model):
    """Context for evaluating flight physics

    Variables
    ---------
    V     40         [knots]    true airspeed
    mu    1.628e-5  [N*s/m^2]  dynamic viscosity
    rho   0.74      [kg/m^3]   air density

    """
    def setup(self):
        exec parse_variables(FlightState.__doc__)


class FlightSegment(Model):
    """Combines a context (flight state) and a component (the aircraft)

    Upper Unbounded
    ---------------
    Wburn, aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
```

(continues on next page)

(continued from previous page)

```
    ---------------
    Wfuel, aircraft.W

    """
    def setup(self, aircraft):
        self.aircraft = aircraft

        self.flightstate = FlightState()
        self.aircraftp = aircraft.dynamic(aircraft, self.flightstate)

        self.Wburn = self.aircraftp.Wburn
        self.Wfuel = self.aircraftp.Wfuel

        return self.flightstate, self.aircraftp


class Mission(Model):
    """A sequence of flight segments

    Upper Unbounded
    ---------------
    aircraft.wing.c, aircraft.wing.A

    Lower Unbounded
    ---------------
    aircraft.W
    """
    def setup(self, aircraft):
        self.aircraft = aircraft

        with Vectorize(4):  # four flight segments
            self.fs = FlightSegment(aircraft)

        Wburn = self.fs.aircraftp.Wburn
        Wfuel = self.fs.aircraftp.Wfuel
        self.takeoff_fuel = Wfuel[0]

        return self.fs, [Wfuel[:-1] >= Wfuel[1:] + Wburn[:-1],
                         Wfuel[-1] >= Wburn[-1]]


class WingAero(Model):
    """Wing aerodynamics

    Variables
    ---------
    CD        [-]    drag coefficient
    CL        [-]    lift coefficient
    e    0.9  [-]    Oswald efficiency
    Re        [-]    Reynold's number
    D         [lbf]  drag force

    Upper Unbounded
    ---------------
    D, Re, wing.A, state.mu

    Lower Unbounded
```

(continues on next page)

(continued from previous page)

```
    ---------------
    CL, wing.S, state.mu, state.rho, state.V
    """
    def setup(self, wing, state):
        self.wing = wing
        self.state = state
        exec parse_variables(WingAero.__doc__)

        c = wing.c
        A = wing.A
        S = wing.S
        rho = state.rho
        V = state.V
        mu = state.mu

        return [
            CD >= 0.074/Re**0.2 + CL**2/np.pi/A/e,
            Re == rho*V*c/mu,
            D >= 0.5*rho*V**2*CD*S]


class Wing(Model):
    """Aircraft wing model

    Variables
    ---------
    W         [lbf]       weight
    S         [ft^2]      surface area
    rho     1 [lbf/ft^2]  areal density
    A      27 [-]         aspect ratio
    c         [ft]        mean chord

    Upper Unbounded
    ---------------
    W

    Lower Unbounded
    ---------------
    c, S
    """
    def setup(self):
        exec parse_variables(Wing.__doc__)
        return [W >= S*rho, c == (S/A)**0.5]

    dynamic = WingAero


class Fuselage(Model):
    """The thing that carries the fuel, engine, and payload

    A full model is left as an exercise for the reader.

    Variables
    ---------
    W  100 [lbf]  weight

    """
```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        exec parse_variables(Fuselage.__doc__)


AC = Aircraft()
MISSION = Mission(AC)
M = Model(MISSION.takeoff_fuel, [MISSION, AC])
sol = M.solve(verbosity=0)
# save solution to a file and retrieve it
sol.save("solution.p")
sol_loaded = pickle.load(open("solution.p"))

vars_of_interest = set(AC.varkeys)
vars_of_interest.update(MISSION.fs.aircraftp.unique_varkeys)
vars_of_interest.add("D")
print sol.summary(vars_of_interest)
```

Note that the output table can be filtered with a list of variables to show.

```

Cost
----
 1.091 [lbf]

Free Variables
--------------
        | Aircraft
    W : 144.1                                       [lbf]    weight

        | Aircraft/Wing
    S : 44.14                                       [ft**2] surface area
    W : 44.14                                       [lbf]    weight
    c : 1.279                                       [ft]     mean chord

        | Mission/FlightSegment/AircraftP
Wburn : [ 0.274      0.273      0.272      0.272     ]  [lbf]    segment fuel burn
Wfuel : [ 1.09       0.817      0.544      0.272     ]  [lbf]    fuel weight

        | Mission/FlightSegment/AircraftP/WingAero
    D : [ 2.74       2.73       2.72       2.72      ]  [lbf]    drag force

Sensitivities
-------------
      | Aircraft/Fuselage
  W : +0.97   weight

      | Aircraft/Wing
  A : -0.67   aspect ratio
rho : +0.43   areal density

Next Largest Sensitivities
--------------------------
      | Mission/FlightSegment/AircraftP/WingAero
  e : [ -0.18      -0.18      -0.18      -0.18     ] Oswald efficiency

      | Mission/FlightSegment/FlightState
  V : [ -0.22      -0.21      -0.21      -0.21     ] true airspeed
```

(continues on next page)

(continued from previous page)

```
rho : [ -0.12      -0.11      -0.11      -0.11     ] air density
```

CHAPTER 8

Advanced Commands

8.1 Derived Variables

8.1.1 Evaluated Fixed Variables

Some fixed variables may be derived from the values of other fixed variables. For example, air density, viscosity, and temperature are functions of altitude. These can be represented by a substitution or value that is a one-argument function accepting `model.substitutions` (for details, see *Substitutions* below).

```
# code from t_GPSubs.test_calcconst in tests/t_sub.py
x = Variable("x", "hours")
t_day = Variable("t_{day}", 12, "hours")
t_night = Variable("t_{night}", lambda c: 24 - c[t_day], "hours")
# note that t_night has a function as its value
m = Model(x, [x >= t_day, x >= t_night])
sol = m.solve(verbosity=0)
self.assertAlmostEqual(sol(t_night)/gpkit.ureg.hours, 12)
m.substitutions.update({t_day: ("sweep", [8, 12, 16])})
sol = m.solve(verbosity=0)
self.assertEqual(len(sol["cost"]), 3)
npt.assert_allclose(sol(t_day) + sol(t_night), 24)
```

These functions are automatically differentiated with the ad package to provide more accurate sensitivities. In some cases may require using functions from the `ad.admath` instead of their python or numpy equivalents; the ad documentation contains details on how to do this.

8.1.2 Evaluated Free Variables

Some free variables may be evaluated from the values of other (non-evaluated) free variables after the optimization is performed. For example, if the efficiency ν of a motor is not a GP-compatible variable, but $(1-\nu)$ is a valid GP variable, then ν can be calculated after solving. These evaluated free variables can be represented by a `Variable` with `evalfn` metadata. Note that this variable should not be used in constructing your model!

```
# code from t_constraints.test_evalfn in tests/t_sub.py
x = Variable("x")
x2 = Variable("x^2", evalfn=lambda v: v[x]**2)
m = Model(x, [x >= 2])
m.unique_varkeys = set([x2.key])
sol = m.solve(verbosity=0)
self.assertAlmostEqual(sol(x2), sol(x)**2)
```

For evaluated variables that can be used during a solution, see `externalfn` under *Sequential Geometric Programs*.

8.2 Sweeps

Sweeps are useful for analyzing tradeoff surfaces. A sweep "value" is an Iterable of numbers, e.g. `[1, 2, 3]`. The simplest way to sweep a model is to call `model.sweep({sweepvar: sweepvalues})`, which will return a solution array but not change the model's substitutions dictionary. If multiple `sweepvars` are given, the method will run them all as independent one-dimensional sweeps and return a list of one solution per sweep. The method `model.autosweep({sweepvar: (start, end)}, tol=0.01)` behaves very similarly, except that only the bounds of the sweep need be specified and the region in betwen will be swept to a maximum possible error of tol in the log of the cost. For details see *1D Autosweeps* below.

8.2.1 Sweep Substitutions

Alternatively, or to sweep a higher-dimensional grid, Variables can swept with a substitution value takes the form `('sweep', Iterable)`, such as `('sweep', np.linspace(1e6, 1e7, 100))`. During variable declaration, giving an Iterable value for a Variable is assumed to be giving it a sweep value: for example, `x = Variable("x", [1, 2, 3])` will sweep `x` over three values.

Vector variables may also be substituted for: `{y: ("sweep" ,[[1, 2], [1, 2], [1, 2]])}` will sweep $y \; \forall \; y_i \in \{1, 2\}$. These sweeps cannot be specified during Variable creation.

A Model with sweep substitutions will solve for all possible combinations: e.g., if there's a variable `x` with value `('sweep', [1, 3])` and a variable `y` with value `('sweep', [14, 17])` then the gp will be solved four times, for $(x, y) \in \{(1, 14), \; (1, 17), \; (3, 14), \; (3, 17)\}$. The returned solutions will be a one-dimensional array (or 2-D for vector variables), accessed in the usual way.

8.2.2 1D Autosweeps

If you're only sweeping over a single variable, autosweeping lets you specify a tolerance for cost error instead of a number of exact positions to solve at. GPkit will then search the sweep segment for a locally optimal number of sweeps that can guarantee a max absolute error on the log of the cost.

Accessing variable and cost values from an autosweep is slightly different, as can be seen in this example:

```
"Show autosweep_1d functionality"
import cPickle as pickle
import numpy as np
import gpkit
from gpkit import units, Variable, Model
```

(continues on next page)

(continued from previous page)

```
from gpkit.tools.autosweep import autosweep_1d
from gpkit.small_scripts import mag

A = Variable("A", "m**2")
l = Variable("l", "m")

m1 = Model(A**2, [A >= l**2 + units.m**2])
tol1 = 1e-3
bst1 = autosweep_1d(m1, tol1, l, [1, 10], verbosity=0)
print "Solved after %2i passes, cost logtol +/-%.3g" % (bst1.nsols, bst1.tol)
# autosweep solution accessing
l_vals = np.linspace(1, 10, 10)
sol1 = bst1.sample_at(l_vals)
print "values of l:", l_vals
print "values of A:", sol1("A")
cost_estimate = sol1["cost"]
cost_lb, cost_ub = sol1.cost_lb(), sol1.cost_ub()
print "cost lower bound:", cost_lb
print "cost estimate:   ", cost_estimate
print "cost upper bound:", cost_ub
# you can evaluate arbitrary posynomials
np.testing.assert_allclose(mag(2*sol1(A)), mag(sol1(2*A)))
assert (sol1["cost"] == sol1(A**2)).all()
# the cost estimate is the logspace mean of its upper and lower bounds
np.testing.assert_allclose((np.log(mag(cost_lb)) + np.log(mag(cost_ub)))/2,
                           np.log(mag(cost_estimate)))
# save autosweep to a file and retrieve it
bst1.save("autosweep.p")
bst1_loaded = pickle.load(open("autosweep.p"))

# this problem is two intersecting lines in logspace
m2 = Model(A**2, [A >= (l/3)**2, A >= (l/3)**0.5 * units.m**1.5])
tol2 = {"mosek": 1e-12, "cvxopt": 1e-7,
        "mosek_cli": 1e-6}[gpkit.settings["default_solver"]]
bst2 = autosweep_1d(m2, tol2, l, [1, 10], verbosity=0)
print "Solved after %2i passes, cost logtol +/-%.3g" % (bst2.nsols, bst2.tol)
print "Table of solutions used in the autosweep:"
print bst2.solarray.table()
```

If you need access to the raw solutions arrays, the smallest simplex tree containing any given point can be gotten with `min_bst = bst.min_bst(val)`, the extents of that tree with `bst.bounds` and solutions of that tree with `bst.sols`. More information is in `help(bst)`.

8.3 Tight ConstraintSets

Tight ConstraintSets will warn if any inequalities they contain are not tight (that is, the right side does not equal the left side) after solving. This is useful when you know that a constraint *should* be tight for a given model, but representing it as an equality would be non-convex.

```
from gpkit import Variable, Model
from gpkit.constraints.tight import Tight

Tight.reltol = 1e-2  # set the global tolerance of Tight
x = Variable('x')
```

(continues on next page)

(continued from previous page)

```
x_min = Variable('x_{min}', 2)
m = Model(x, [Tight([x >= 1], reltol=1e-3),  # set the specific tolerance
              x >= x_min])
m.solve(verbosity=0)  # prints warning
```

8.4 Loose ConstraintSets

Loose ConstraintSets will warn if any GP-compatible constraints they contain are not loose (that is, their sensitivity is above some threshold after solving). This is useful when you want a constraint to be inactive for a given model because it represents an important model assumption (such as a fit only valid over a particular interval).

```
from gpkit import Variable, Model
from gpkit.constraints.tight import Loose

Tight.reltol = 1e-4  # set the global tolerance of Tight
x = Variable('x')
x_min = Variable('x_{min}', 1)
m = Model(x, [Loose([x >= 2], senstol=1e-4),  # set the specific tolerance
              x >= x_min])
m.solve(verbosity=0)  # prints warning
```

8.5 Substitutions

Substitutions are a general-purpose way to change every instance of one variable into either a number or another variable.

8.5.1 Substituting into Posynomials, NomialArrays, and GPs

The examples below all use Posynomials and NomialArrays, but the syntax is identical for GPs (except when it comes to sweep variables).

```
# adapted from t_sub.py / t_NomialSubs / test_Basic
from gpkit import Variable
x = Variable("x")
p = x**2
assert p.sub(x, 3) == 9
assert p.sub(x.varkeys["x"], 3) == 9
assert p.sub("x", 3) == 9
```

Here the variable `x` is being replaced with `3` in three ways: first by substituting for `x` directly, then by substituting for the `VarKey("x")`, then by substituting the string "x". In all cases the substitution is understood as being with the VarKey: when a variable is passed in the VarKey is pulled out of it, and when a string is passed in it is used as an argument to the Posynomial's `varkeys` dictionary.

8.5.2 Substituting multiple values

```
# adapted from t_sub.py / t_NomialSubs / test_Vector
from gpkit import Variable, VectorVariable
x = Variable("x")
y = Variable("y")
z = VectorVariable(2, "z")
p = x*y*z
assert all(p.sub({x: 1, "y": 2}) == 2*z)
assert all(p.sub({x: 1, y: 2, "z": [1, 2]}) == z.sub(z, [2, 4]))
```

To substitute in multiple variables, pass them in as a dictionary where the keys are what will be replaced and values are what it will be replaced with. Note that you can also substitute for VectorVariables by their name or by their NomialArray.

8.5.3 Substituting with nonnumeric values

You can also substitute in sweep variables (see *Sweeps*), strings, and monomials:

```
# adapted from t_sub.py / t_NomialSubs
from gpkit import Variable
from gpkit.small_scripts import mag

x = Variable("x", "m")
xvk = x.varkeys.values()[0]
descr_before = x.exp.keys()[0].descr
y = Variable("y", "km")
yvk = y.varkeys.values()[0]
for x_ in ["x", xvk, x]:
    for y_ in ["y", yvk, y]:
        if not isinstance(y_, str) and type(xvk.units) != str:
            expected = 0.001
        else:
            expected = 1.0
        assert abs(expected - mag(x.sub(x_, y_).c)) < 1e-6
if type(xvk.units) != str:
    # this means units are enabled
    z = Variable("z", "s")
    # y.sub(y, z) will raise ValueError due to unit mismatch
```

Note that units are preserved, and that the value can be either a string (in which case it just renames the variable), a varkey (in which case it changes its description, including the name) or a Monomial (in which case it substitutes for the variable with a new monomial).

8.5.4 Updating ConstraintSet substitutions

ConstraintSets have a `.substitutions` KeyDict attribute which will be substituted before solving. This KeyDict accepts variable names, VarKeys, and Variable objects as keys, and can be updated (or deleted from) like a regular Python dictionary to change the substitutions that will be used at solve-time. If a ConstraintSet itself contains ConstraintSets, it and all its elements share pointers to the same substitutions dictionary object, so that updating any one of them will update all of them.

8.5.5 Substituting with replacement

Any of the substitutions above can be run with `p.subinplace(*args)` to substitute directly into the object in question.

8.5.6 Fixed Variables

When a Model is created, any fixed Variables are used to form a dictionary: `{var: var.descr["value"] for var in self.varlocs if "value" in var.descr}`. This dictionary in then substituted into the Model's cost and constraints before the `substitutions` argument is (and hence values are supplanted by any later substitutions).

`solution.subinto(p)` will substitute the solution(s) for variables into the posynomial `p`, returning a NomialArray. For a non-swept solution, this is equivalent to `p.sub(solution["variables"])`.

You can also substitute by just calling the solution, i.e. `solution(p)`. This returns a numpy array of just the coefficients (`c`) of the posynomial after substitution, and will raise a' `ValueError`` if some of the variables in `p` were not found in `solution`.

8.5.7 Freeing Fixed Variables

After creating a Model, it may be useful to "free" a fixed variable and resolve. This can be done using the command `del m.substitutions["x"]`, where `m` is a Model. An example of how to do this is shown below.

```
from gpkit import Variable, Model
x = Variable("x")
y = Variable("y", 3)  # fix value to 3
m = Model(x, [x >= 1 + y, y >= 1])
_ = m.solve()  # optimal cost is 4; y appears in sol["constants"]

del m.substitutions["y"]
_ = m.solve()  # optimal cost is 2; y appears in Free Variables
```

Note that `del m.substitutions["y"]` affects `m` but not `y.key`. `y.value` will still be 3, and if `y` is used in a new model, it will still carry the value of 3.

CHAPTER 9

Signomial Programming

Signomial programming finds a local solution to a problem of the form:

$$\begin{array}{lll} \text{minimize} & g_0(x) & \\ \text{subject to} & f_i(x) = 1, & i = 1, ..., m \\ & g_i(x) - h_i(x) \leq 1, & i = 1, ..., n \end{array}$$

where each f is monomial while each g and h is a posynomial.

This requires multiple solutions of geometric programs, and so will take longer to solve than an equivalent geometric programming formulation.

In general, when given the choice of which variables to include in the positive-posynomial / g side of the constraint, the modeler should:

1. maximize the number of variables in g,
2. prioritize variables that are in the objective,
3. then prioritize variables that are present in other constraints.

The `.localsolve` syntax was chosen to emphasize that signomial programming returns a local optimum. For the same reason, calling `.solve` on an SP will raise an error.

By default, signomial programs are first solved conservatively (by assuming each h is equal only to its constant portion) and then become less conservative on each iteration.

9.1 Example Usage

```
"""Adapted from t_SP in tests/t_geometric_program.py"""
import gpkit

# Decision variables
x = gpkit.Variable('x')
y = gpkit.Variable('y')
```

(continues on next page)

(continued from previous page)

```
# must enable signomials for subtraction
with gpkit.SignomialsEnabled():
    constraints = [x >= 1-y, y <= 0.1]

# create and solve the SP
m = gpkit.Model(x, constraints)
print m.localsolve(verbosity=0).summary()
assert abs(m.solution(x) - 0.9) < 1e-6
```

When using the `localsolve` method, the `reltol` argument specifies the relative tolerance of the solver: that is, by what percent does the solution have to improve between iterations? If any iteration improves less than that amount, the solver stops and returns its value.

If you wish to start the local optimization at a particular point x_k, however, you may do so by putting that position (a dictionary formatted as you would a substitution) as the `xk` argument.

9.2 Sequential Geometric Programs

The method of solving local GP approximations of a non-GP compatible model can be generalized, at the cost of the general smoothness and lack of a need for trust regions that SPs guarantee.

For some applications, it is useful to call external codes which may not be GP compatible. Imagine we wished to solve the following optimization problem:

$$\begin{array}{ll} \text{minimize} & y \\ \text{subject to} & y \geq \sin(x) \\ & \frac{\pi}{4} \leq x \leq \frac{\pi}{2} \end{array}$$

This problem is not GP compatible due to the sin(x) constraint. One approach might be to take the first term of the Taylor expansion of sin(x) and attempt to solve:

```
"Can be found in gpkit/docs/source/examples/sin_approx_example.py"
import numpy as np
from gpkit import Variable, Model


x = Variable("x")
y = Variable("y")

objective = y

constraints = [y >= x,
               x <= np.pi/2.,
               x >= np.pi/4.,
              ]

m = Model(objective, constraints)
print m.solve(verbosity=0).summary()
```

```

Cost
----
 0.7854
```

(continues on next page)

(continued from previous page)

```
Free Variables
--------------
x : 0.7854
y : 0.7854
```

We can do better, however, by utilizing some built in functionality of GPkit. For simple cases with a single Variable, GPkit looks for `externalfn` metadata:

```
"Can be found in gpkit/docs/source/examples/external_sp2.py"
import numpy as np
from gpkit import Variable, Model

x = Variable("x")


def y_ext(self, x0):
    "Returns constraints on y derived from x0"
    if x not in x0:
        return self >= x
    return self >= x/x0[x] * np.sin(x0[x])

y = Variable("y", externalfn=y_ext)

m = Model(y, [np.pi/4 <= x, x <= np.pi/2])
print m.localsolve(verbosity=0).summary()
```

```
Cost
----
 0.7071

Free Variables
--------------
x : 0.7854
y : 0.7071
```

However, for external functions not intrinsically tied to a single variable it's best to use the full ConstraintSet API, as follows:

Assume we have some external code which is capable of evaluating our incompatible function:

```
"""External function for GPkit to call.  Can be found
in gpkit/docs/source/examples/external_function.py"""
import numpy as np

def external_code(x):
    "Returns sin(x)"
    return np.sin(x)
```

Now, we can create a ConstraintSet that allows GPkit to treat the incompatible constraint as though it were a signomial programming constraint:

```python
"Can be found in gpkit/docs/source/examples/external_constraint.py"
from gpkit.exceptions import InvalidGPConstraint
from external_function import external_code


class ExternalConstraint(object):
    "Class for external calling"
    varkeys = {}

    def __init__(self, x, y):
        # We need a GPkit variable defined to return in our constraint.  The
        # easiest way to do this is to read in the parameters of interest in
        # the initiation of the class and store them here.
        self.x = x
        self.y = y

    def as_posyslt1(self, _):
        "Ensures this is treated as an SGP constraint"
        raise InvalidGPConstraint("ExternalConstraint cannot solve as a GP.")

    def as_gpconstr(self, x0):
        "Returns locally-approximating GP constraint"

        # Unpacking the GPkit variables
        x = self.x
        y = self.y

        # Creating a default constraint for the first solve
        if not x0:
            return (y >= x)

        # Returns constraint updated with new call to the external code
        else:
            # Unpack Design Variables at the current point
            x_star = x0["x"]

            # Call external code
            res = external_code(x_star)

        # Return linearized constraint
        return (y >= res*x/x_star)
```

and replace the incompatible constraint in our GP:

```python
"Can be found in gpkit/docs/source/examples/external_sp.py"

import numpy as np
from gpkit import Variable, Model
from external_constraint import ExternalConstraint

x = Variable("x")
y = Variable("y")

objective = y

constraints = [ExternalConstraint(x, y),
               x <= np.pi/2.,
```

(continues on next page)

(continued from previous page)

```
                x >= np.pi/4.,
               ]

m = Model(objective, constraints)
print m.localsolve(verbosity=0).summary()
```

```

Cost
----
 0.7071

Free Variables
--------------
x : 0.7854
y : 0.7071

```

which is the expected result. This method has been generalized to larger problems, such as calling XFOIL and AVL.

If you wish to start the local optimization at a particular point x_0, however, you may do so by putting that position (a dictionary formatted as you would a substitution) as the `x0` argument

CHAPTER 10

Examples

10.1 iPython Notebook Examples

More examples, including some with in-depth explanations and interactive visualizations, can be seen on nbviewer.

10.2 A Trivial GP

The most trivial GP we can think of: minimize x subject to the constraint $x \geq 1$.

```
"Very simple problem: minimize x while keeping x greater than 1."
from gpkit import Variable, Model

# Decision variable
x = Variable("x")

# Constraint
constraints = [x >= 1]

# Objective (to minimize)
objective = x

# Formulate the Model
m = Model(objective, constraints)

# Solve the Model
sol = m.solve(verbosity=0)

# print selected results
print("Optimal cost:  %.4g" % sol["cost"])
print("Optimal x val: %s" % sol(x))
```

Of course, the optimal value is 1. Output:

```
Optimal cost:  1
Optimal x val: 1.0 dimensionless
```

10.3 Maximizing the Volume of a Box

This example comes from Section 2.4 of the GP tutorial, by S. Boyd et. al.

```python
"Maximizes box volume given area and aspect ratio constraints."
from gpkit import Variable, Model

# Parameters
alpha = Variable("alpha", 2, "-", "lower limit, wall aspect ratio")
beta = Variable("beta", 10, "-", "upper limit, wall aspect ratio")
gamma = Variable("gamma", 2, "-", "lower limit, floor aspect ratio")
delta = Variable("delta", 10, "-", "upper limit, floor aspect ratio")
A_wall = Variable("A_{wall}", 200, "m^2", "upper limit, wall area")
A_floor = Variable("A_{floor}", 50, "m^2", "upper limit, floor area")

# Decision variables
h = Variable("h", "m", "height")
w = Variable("w", "m", "width")
d = Variable("d", "m", "depth")

# Constraints
constraints = [A_wall >= 2*h*w + 2*h*d,
               A_floor >= w*d,
               h/w >= alpha,
               h/w <= beta,
               d/w >= gamma,
               d/w <= delta]

# Objective function
V = h*w*d
objective = 1/V  # To maximize V, we minimize its reciprocal

# Formulate the Model
m = Model(objective, constraints)

# Solve the Model and print the results table
print m.solve(verbosity=0).table()
```

The output is

```

Cost
----
 0.003674 [1/m**3]

Free Variables
--------------
d : 8.17   [m] depth
h : 8.163  [m] height
w : 4.081  [m] width

Constants
```

(continues on next page)

(continued from previous page)

```
---------
A_{floor} : 50   [m**2] upper limit, floor area
 A_{wall} : 200  [m**2] upper limit, wall area
    alpha : 2           lower limit, wall aspect ratio
     beta : 10          upper limit, wall aspect ratio
    delta : 10          upper limit, floor aspect ratio
    gamma : 2           lower limit, floor aspect ratio

Sensitivities
-------------
A_{wall} : -1.5  upper limit, wall area
   alpha : +0.5  lower limit, wall aspect ratio
```

10.4 Water Tank

Say we had a fixed mass of water we wanted to contain within a tank, but also wanted to minimize the cost of the material we had to purchase (i.e. the surface area of the tank):

```
"Minimizes cylindrical tank surface area for a particular volume."
from gpkit import Variable, VectorVariable, Model

M = Variable("M", 100, "kg", "Mass of Water in the Tank")
rho = Variable("\\rho", 1000, "kg/m^3", "Density of Water in the Tank")
A = Variable("A", "m^2", "Surface Area of the Tank")
V = Variable("V", "m^3", "Volume of the Tank")
d = VectorVariable(3, "d", "m", "Dimension Vector")

constraints = (A >= 2*(d[0]*d[1] + d[0]*d[2] + d[1]*d[2]),
               V == d[0]*d[1]*d[2],
               M == V*rho)

m = Model(A, constraints)
sol = m.solve(verbosity=0)
print sol.summary()
```

The output is

```

Cost
----
 1.293 [m**2]

Free Variables
--------------
A : 1.293                               [m**2] Surface Area of the Tank
V : 0.1                                 [m**3] Volume of the Tank
d : [ 0.464     0.464     0.464     ]   [m]    Dimension Vector

Sensitivities
-------------
   M : +0.67  Mass of Water in the Tank
\rho : -0.67  Density of Water in the Tank
```

10.5 Simple Wing

This example comes from Section 3 of Geometric Programming for Aircraft Design Optimization, by W. Hoburg and P. Abbeel.

```
"Minimizes airplane drag for a simple drag and structure model."
import cPickle as pickle
import numpy as np
from gpkit import Variable, Model
pi = np.pi


# Constants
k = Variable("k", 1.2, "-", "form factor")
e = Variable("e", 0.95, "-", "Oswald efficiency factor")
mu = Variable("\\mu", 1.78e-5, "kg/m/s", "viscosity of air")
rho = Variable("\\rho", 1.23, "kg/m^3", "density of air")
tau = Variable("\\tau", 0.12, "-", "airfoil thickness to chord ratio")
N_ult = Variable("N_{ult}", 3.8, "-", "ultimate load factor")
V_min = Variable("V_{min}", 22, "m/s", "takeoff speed")
C_Lmax = Variable("C_{L,max}", 1.5, "-", "max CL with flaps down")
S_wetratio = Variable("(\\frac{S}{S_{wet}})", 2.05, "-", "wetted area ratio")
W_W_coeff1 = Variable("W_{W_{coeff1}}", 8.71e-5, "1/m",
                      "Wing Weight Coefficent 1")
W_W_coeff2 = Variable("W_{W_{coeff2}}", 45.24, "Pa",
                      "Wing Weight Coefficent 2")
CDA0 = Variable("(CDA0)", 0.031, "m^2", "fuselage drag area")
W_0 = Variable("W_0", 4940.0, "N", "aircraft weight excluding wing")

# Free Variables
D = Variable("D", "N", "total drag force")
A = Variable("A", "-", "aspect ratio")
S = Variable("S", "m^2", "total wing area")
V = Variable("V", "m/s", "cruising speed")
W = Variable("W", "N", "total aircraft weight")
Re = Variable("Re", "-", "Reynold's number")
C_D = Variable("C_D", "-", "Drag coefficient of wing")
C_L = Variable("C_L", "-", "Lift coefficent of wing")
C_f = Variable("C_f", "-", "skin friction coefficient")
W_w = Variable("W_w", "N", "wing weight")

constraints = []

# Drag model
C_D_fuse = CDA0/S
C_D_wpar = k*C_f*S_wetratio
C_D_ind = C_L**2/(pi*A*e)
constraints += [C_D >= C_D_fuse + C_D_wpar + C_D_ind]

# Wing weight model
W_w_strc = W_W_coeff1*(N_ult*A**1.5*(W_0*W*S)**0.5)/tau
W_w_surf = W_W_coeff2 * S
constraints += [W_w >= W_w_surf + W_w_strc]

# and the rest of the models
constraints += [D >= 0.5*rho*S*C_D*V**2,
                Re <= (rho/mu)*V*(S/A)**0.5,
```

(continues on next page)

(continued from previous page)

```
                C_f >= 0.074/Re**0.2,
                W <= 0.5*rho*S*C_L*V**2,
                W <= 0.5*rho*S*C_Lmax*V_min**2,
                W >= W_0 + W_w]

print("SINGLE\n======")
m = Model(D, constraints)
sol = m.solve(verbosity=0)
print(sol.summary())
# save solution to a file and retrieve it
sol.save("solution.p")
print(sol.diff("solution.p"))

print("SWEEP\n=====")
N = 2
sweeps = {V_min: ("sweep", np.linspace(20, 25, N)),
          V: ("sweep", np.linspace(45, 55, N)), }
m.substitutions.update(sweeps)
sweepsol = m.solve(verbosity=0)
print(sweepsol.summary())
sol_loaded = pickle.load(open("solution.p"))
print(sweepsol.diff(sol_loaded))
```

The output is

```
SINGLE
======

Cost
----
 303.1 [N]

Free Variables
--------------
  A : 8.46                aspect ratio
C_D : 0.02059             Drag coefficient of wing
C_L : 0.4988              Lift coefficent of wing
C_f : 0.003599            skin friction coefficient
  D : 303.1       [N]     total drag force
 Re : 3.675e+06           Reynold's number
  S : 16.44       [m**2]  total wing area
  V : 38.15       [m/s]   cruising speed
  W : 7341        [N]     total aircraft weight
W_w : 2401        [N]     wing weight

Most Sensitive
--------------
                W_0 : +1     aircraft weight excluding wing
                  e : -0.48  Oswald efficiency factor
                  k : +0.43  form factor
(\frac{S}{S_{wet}}) : +0.43  wetted area ratio
            V_{min} : -0.37  takeoff speed

Solution difference
-------------------
The largest difference is 0%
```

(continues on next page)

(continued from previous page)

```
Solution sensitivity delta
--------------------------
The largest sensitivity delta is +0

SWEEP
=====

Cost
----
 [ 338       396       294       326       ] [N]

Sweep Variables
---------------
      V : [ 45        55        45        55        ]  [m/s] cruising speed
V_{min} : [ 20        20        25        25        ]  [m/s] takeoff speed

Free Variables
--------------
  A : [ 6.2       4.77      8.84      7.16      ]         aspect ratio
C_D : [ 0.0146    0.0123    0.0196    0.0157    ]         Drag coefficient of wing
C_L : [ 0.296     0.198     0.463     0.31      ]         Lift coefficent of wing
C_f : [ 0.00333   0.00314   0.00361   0.00342   ]         skin friction coefficient
  D : [ 338       396       294       326       ]  [N]    total drag force
 Re : [ 5.38e+06  7.24e+06  3.63e+06  4.75e+06 ]         Reynold's number
  S : [ 18.6      17.3      12.1      11.2      ]  [m**2] total wing area
  W : [ 6.85e+03  6.4e+03   6.97e+03  6.44e+03 ]  [N]    total aircraft weight
W_w : [ 1.91e+03  1.46e+03  2.03e+03  1.5e+03   ]  [N]    wing weight

Most Sensitive
--------------
                W_0 : [ +0.92     +0.85     +0.95     +0.85     ] aircraft weight excluding wing
            V_{min} : [ -0.82     -1        -0.41     -0.71     ] takeoff speed
                  V : [ +0.59     +0.97     +0.25     +0.75     ] cruising speed
                  k : [ +0.56     +0.63     +0.45     +0.54     ] form factor
(\frac{S}{S_{wet}}) : [ +0.56     +0.63     +0.45     +0.54     ] wetted area ratio

Solution difference
(positive means the argument is bigger)
-------------------
    C_L : [  +68.3%   +151.5%     +7.7%    +60.9%  ] Lift coefficent of wing
    W_w : [  +26.0%    +64.7%    +18.5%    +59.8%  ] wing weight
    C_D : [  +40.8%    +67.7%     +5.3%    +31.3%  ] Drag coefficient of wing
      A : [  +36.5%    +77.2%     -4.3%    +18.1%  ] aspect ratio
     Re : [  -31.7%    -49.3%     +1.1%    -22.6%  ] Reynold's number
      S : [  -11.4%     -5.2%    +36.1%    +47.1%  ] total wing area
      V : [  -15.2%    -30.6%    -15.2%    -30.6%  ] cruising speed
V_{min} : [  +10.0%    +10.0%    -12.0%    -12.0%  ] takeoff speed
      D : [  -10.3%    -23.5%     +3.0%     -7.0%  ] total drag force
      W : [   +7.2%    +14.7%     +5.4%    +13.9%  ] total aircraft weight
```

(continues on next page)

(continued from previous page)

```
    C_f : [   +7.9%    +14.5%      -.2%     +5.3%  ] skin friction␣
↪coefficient

Solution sensitivity delta
(positive means the argument has a higher sensitivity)
--------------------------
                  V : [  -.59    -.97    -.25    -.75   ] cruising speed
            V_{min} : [  +.45    +.67    +.05    +.34   ] takeoff speed
          C_{L,max} : [  +.23    +.34    +.02    +.17   ] max CL with flaps␣
↪down
                  e : [  -.15    -.25    -.06    -.19   ] Oswald efficiency␣
↪factor
                W_0 : [  +.09    +.17    +.06    +.16   ] aircraft weight␣
↪excluding wing
               \rho : [  -.05    -.13    -.10    -.19   ] density of air
      W_{W_{coeff1}} : [  +.11    +.18    +.04    +.14   ] Wing Weight␣
↪Coefficent 1
            N_{ult} : [  +.11    +.18    +.04    +.14   ] ultimate load factor
               \tau : [  -.11    -.18    -.04    -.14   ] airfoil thickness␣
↪to chord ratio
                  k : [  -.13    -.20    -.02    -.11   ] form factor
(\frac{S}{S_{wet}}) : [  -.13    -.20    -.02    -.11   ] wetted area ratio
             (CDA0) : [  -.02    -.05    -.04    -.09   ] fuselage drag area
      W_{W_{coeff2}} : [  -.01    0.00    +.04    +.05   ] Wing Weight␣
↪Coefficent 2
```

10.6 Simple Beam

In this example we consider a beam subjected to a uniformly distributed transverse force along its length. The beam has fixed geometry so we are not optimizing its shape, rather we are simply solving a discretization of the Euler-Bernoulli beam bending equations using GP.

```
"""
A simple beam example with fixed geometry. Solves the discretized
Euler-Bernoulli beam equations for a constant distributed load
"""
import numpy as np
from gpkit import Variable, VectorVariable, Model, ureg
from gpkit.small_scripts import mag

eps = 2e-4   # has to be quite small for consistent cvxopt printouts;
             # normally you'd set this to something more like 1e-20


class Beam(Model):
    """Discretization of the Euler beam equations for a distributed load.

    Arguments
    ---------
    N : int
        Number of finite elements that compose the beam.
    L : float
        [m] Length of beam.
```

(continues on next page)

(continued from previous page)

```
    EI : float
        [N m^2] Elastic modulus times cross-section's area moment of inertia.
    q : float or N-vector of floats
        [N/m] Loading density: can be specified as constants or as an array.

    Upper Unbounded
    ---------------
    w_tip
    """
    def setup(self, N=4):
        EI = Variable("EI", 1e4, "N*m^2")
        dx = Variable("dx", "m", "Length of an element")
        L = Variable("L", 5, "m", "Overall beam length")
        q = VectorVariable(N, "q", 100*np.ones(N), "N/m",
                           "Distributed load at each point")
        V = VectorVariable(N, "V", "N", "Internal shear")
        V_tip = Variable("V_{tip}", eps, "N", "Tip loading")
        M = VectorVariable(N, "M", "N*m", "Internal moment")
        M_tip = Variable("M_{tip}", eps, "N*m", "Tip moment")
        th = VectorVariable(N, "\\theta", "-", "Slope")
        th_base = Variable("\\theta_{base}", eps, "-", "Base angle")
        w = VectorVariable(N, "w", "m", "Displacement")
        w_base = Variable("w_{base}", eps, "m", "Base deflection")
        # below: trapezoidal integration to form a piecewise-linear
        #        approximation of loading, shear, and so on
        # shear and moment increase from tip to base (left > right)
        shear_eq = (V >= V.right + 0.5*dx*(q + q.right))
        shear_eq[-1] = (V[-1] >= V_tip)  # tip boundary condition
        moment_eq = (M >= M.right + 0.5*dx*(V + V.right))
        moment_eq[-1] = (M[-1] >= M_tip)
        # slope and displacement increase from base to tip (right > left)
        theta_eq = (th >= th.left + 0.5*dx*(M + M.left)/EI)
        theta_eq[0] = (th[0] >= th_base)  # base boundary condition
        displ_eq = (w >= w.left + 0.5*dx*(th + th.left))
        displ_eq[0] = (w[0] >= w_base)
        # minimize tip displacement (the last w)
        self.cost = self.w_tip = w[-1]
        return [shear_eq, moment_eq, theta_eq, displ_eq,
                L == (N-1)*dx]


b = Beam(N=6, substitutions={"L": 6, "EI": 1.1e4, "q": 110*np.ones(6)})
sol = b.solve(verbosity=0)
print sol.summary(maxcolumns=6)
w_gp = sol("w")  # deflection along beam

L, EI, q = sol("L"), sol("EI"), sol("q")
x = np.linspace(0, mag(L), len(q))*ureg.m  # position along beam
q = q[0]  # assume uniform loading for the check below
w_exact = q/(24.*EI) * x**2 * (x**2 - 4*L*x + 6*L**2)  # analytic soln

assert max(abs(w_gp - w_exact)) <= 1.1*ureg.cm

PLOT = False
if PLOT:
    import matplotlib.pyplot as plt
    x_exact = np.linspace(0, L, 1000)
```

(continues on next page)

(continued from previous page)

```
    w_exact = q/(24.*EI) * x_exact**2 * (x_exact**2 - 4*L*x_exact + 6*L**2)
    plt.plot(x, w_gp, color='red', linestyle='solid', marker='^',
             markersize=8)
    plt.plot(x_exact, w_exact, color='blue', linestyle='dashed')
    plt.xlabel('x [m]')
    plt.ylabel('Deflection [m]')
    plt.axis('equal')
    plt.legend(['GP solution', 'Analytical solution'])
    plt.show()
```

The output is

```

Cost
----
 1.621 [m]

Free Variables
--------------
    dx : 1.2                                                            [m]␣
↪   Length of an element
     M : [ 1.98e+03  1.27e+03  713       317       79.2      0.0002    ] ␣
↪[N*m] Internal moment
     V : [ 660       528       396       264       132       0.0002    ]  [N]␣
↪   Internal shear
\theta : [ 0.0002    0.177     0.285     0.341     0.363     0.367     ]      ␣
↪   Slope
     w : [ 0.0002    0.107     0.384     0.76      1.18      1.62      ]  [m]␣
↪   Displacement

Most Sensitive
--------------
 L : +4                                                            Overall␣
↪beam length
EI : -1
 q : [ +0.0072   +0.042    +0.12     +0.23     +0.37     +0.22     ]␣
↪Distributed load at each point

```

By plotting the deflection, we can see that the agreement between the analytical solution and the GP solution is good.

CHAPTER 11

Glossary

For an alphabetical listing of all commands, check out the genindex

11.1 gpkit package

11.1.1 Subpackages

gpkit.constraints package

Submodules

gpkit.constraints.array module

Implements ArrayConstraint

class gpkit.constraints.array.**ArrayConstraint**(*constraints*, *left*, *oper*, *right*)

Bases: *gpkit.constraints.single_equation.SingleEquationConstraint*, *gpkit.constraints.set.ConstraintSet*

A ConstraintSet for prettier array-constraint printing.

ArrayConstraint gets its *sub* method from ConstrainSet, and so *left* and *right* are only used for printing.

When created by NomialArray *left* and *right* are likely to be be either NomialArrays or Varkeys of VectorVariables.

subinplace(*subs*)

Substitutes in place, updating self.substitutions accordingly.

Keys substituted with *subinplace* are no longer present, so if such a key is also in self.substitutions that substitution is now orphaned. If *subs[key]* describes some key in the

ConstraintSet (i.e. one key has been substituted for another), then a substitution is added, mapping the orphaned value to this new key; otherwise, an error is raised.

gpkit.constraints.bounded module

Implements Bounded

class `gpkit.constraints.bounded.`**`Bounded`**(*constraints*, *verbosity=1*, *eps=1e-30*, *lower=None*, *upper=None*)

Bases: `gpkit.constraints.set.ConstraintSet`

Bounds contained variables so as to ensure dual feasibility.

constraints [iterable] constraints whose varkeys will be bounded

substitutions [dict] as in ConstraintSet.__init__

verbosity [int]

how detailed of a warning to print 0: nothing 1: print warnings

eps [float] default lower bound is eps, upper bound is 1/eps

lower [float] lower bound for all varkeys, replaces eps

upper [float] upper bound for all varkeys, replaces 1/eps

`check_boundaries`(*result*)
Creates (and potentially prints) a dictionary of unbounded variables.

`process_result`(*result*)
Add boundedness to the model's solution

`sens_from_dual`(*las*, *nus*, *result*)
Return sensitivities while capturing the relevant lambdas

`gpkit.constraints.bounded.`**`varkey_bounds`**(*varkeys*, *lower*, *upper*)
Returns constraints list bounding all varkeys.

varkeys [iterable] list of varkeys to create bounds for

lower [float] lower bound for all varkeys

upper [float] upper bound for all varkeys

gpkit.constraints.costed module

Implement CostedConstraintSet

class `gpkit.constraints.costed.`**`CostedConstraintSet`**(*cost*, *constraints*, *substitutions=None*)

Bases: `gpkit.constraints.set.ConstraintSet`

A ConstraintSet with a cost

cost : gpkit.Posynomial constraints : Iterable substitutions : dict

`constrained_varkeys`()
Return all varkeys in the cost and non-ConstraintSet constraints

controlpanel(**args*, ***kwargs*)
Easy model control in IPython / Jupyter

Like interact(), but with the ability to control sliders and their ranges live. args and kwargs are passed on to interact()

interact(*ranges=None*, *fn_of_sol=None*, ***solvekwargs*)
Easy model interaction in IPython / Jupyter

By default, this creates a model with sliders for every constant which prints a new solution table whenever the sliders are changed.

fn_of_sol [function] The function called with the solution after each solve that displays the result. By default prints a table.

ranges [dictionary {str: Slider object or tuple}] Determines which sliders get created. Tuple values may contain two or three floats: two correspond to (min, max), while three correspond to (min, step, max)

****solvekwargs** kwargs which get passed to the solve()/localsolve() method.

reset_varkeys()
Resets varkeys to what is in the cost and constraints

rootconstr_latex(*excluded=None*)
Latex showing cost, to be used when this is the top constraint

rootconstr_str(*excluded=None*)
String showing cost, to be used when this is the top constraint

subinplace(*subs*)
Substitutes in place.

gpkit.constraints.gp module

Implement the GeometricProgram class

class gpkit.constraints.gp.**GeometricProgram**(*cost*, *constraints*, *substitutions*, *allow_missingbounds=False*)
Bases: *gpkit.constraints.costed.CostedConstraintSet*, *gpkit.nomials.data.NomialData*

Standard mathematical representation of a GP.

cost [Constraint] Posynomial to minimize when solving

constraints [list of Posynomials] Constraints to maintain when solving (implicitly Posynomials <= 1) GeometricProgram does not accept equality constraints (e.g. x == 1);

instead use two inequality constraints (e.g. x <= 1, 1/x <= 1)

verbosity [int (optional)] If verbosity is greater than zero, warns about missing bounds on creation.

solver_out and *solver_log* are set during a solve *result* is set at the end of a solve if solution status is optimal

```
>>> gp = gpkit.geometric_program.GeometricProgram(
                    # minimize
                    x,
                    [   # subject to
                        1/x  # <= 1, implicitly
```

(continues on next page)

(continued from previous page)

```
                        ])
>>> gp.solve()
```

check_solution (*cost*, *primal*, *nu*, *la*, *tol=0.001*, *abstol=1e-20*)
Run a series of checks to mathematically confirm sol solves this GP

cost: float cost returned by solver

primal: list primal solution returned by solver

nu: numpy.ndarray monomial lagrange multiplier

la: numpy.ndarray posynomial lagrange multiplier

RuntimeWarning, if any problems are found

gen ()
Generates nomial and solve data (A, p_idxs) from posynomials

solve (*solver=None*, *verbosity=1*, *warn_on_check=False*, *process_result=True*, ***kwargs*)
Solves a GeometricProgram and returns the solution.

solver [str or function (optional)] By default uses one of the solvers found during installation. If set to "mosek", "mosek_cli", or "cvxopt", uses that solver. If set to a function, passes that function cs, A, p_idxs, and k.

verbosity [int (optional)] If greater than 0, prints solver name and solve time.

****kwargs :** Passed to solver constructor and solver function.

result [dict] A dictionary containing the translated solver result; keys below.

- **cost** [float] The value of the objective at the solution.
- **variables** [dict] The value of each variable at the solution.
- **sensitivities** [dict]
  - **monomials** [array of floats] Each monomial's dual variable value at the solution.
  - **posynomials** [array of floats] Each posynomials's dual variable value at the solution.

varkeys
The GP's varkeys, created when necessary.

gpkit.constraints.gp.**check_mono_eq_bounds** (*missingbounds*, *meq_bounds*)
Bounds variables with monomial equalities

gpkit.constraints.gp.**genA** (*exps*, *varlocs*, *meq_idxs*)
Generates A matrix from exps and varidxs

exps [list of Hashvectors] Exponents for each monomial in a GP

varidxs [dict] Locations of each variable in exps

A [sparse Cootmatrix] Exponents of the various free variables for each monomial: rows of A are monomials, columns of A are variables.

missingbounds [dict] Keys: variables that lack bounds. Values: which bounds are missed.

gpkit.constraints.model module

Implements Model

class gpkit.constraints.model.**Model**(*cost=None*, *constraints=None*, **args*, ***kwargs*)

Bases: *gpkit.constraints.costed.CostedConstraintSet*

Symbolic representation of an optimization problem.

The Model class is used both directly to create models with constants and sweeps, and indirectly inherited to create custom model classes.

cost [Posynomial (optional)] Defaults to *Monomial(1)*.

constraints [ConstraintSet or list of constraints (optional)] Defaults to an empty list.

substitutions [dict (optional)] This dictionary will be substituted into the problem before solving, and also allows the declaration of sweeps and linked sweeps.

name [str (optional)] Allows "naming" a model in a way similar to inherited instances, and overrides the inherited name if there is one.

program is set during a solve *solution* is set at the end of a solve

as_gpconstr(*x0*)
Returns approximating constraint, keeping name and num

autosweep(*sweeps*, *tol=0.01*, *samplepoints=100*, ***solveargs*)
Autosweeps {var: (start, end)} pairs in sweeps to tol.

Returns swept and sampled solutions. The original simplex tree can be accessed at sol.bst

debug(*solver=None*, *verbosity=1*, ***solveargs*)
Attempts to diagnose infeasible models.

If a model debugs but errors in a process_result call, debug again with *process_results=False*

gp(*constants=None*, ***kwargs*)
Return program version of self

program: NomialData Class to return, e.g. GeometricProgram or SequentialGeometricProgram

return_attr: string attribute to return in addition to the program

localsolve(*solver=None*, *verbosity=1*, *skipsweepfailures=False*, ***kwargs*)
Forms a mathematical program and attempts to solve it.

solver [string or function (optional)] If None, uses the default solver found in installation.

verbosity [int (optional)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

skipsweepfailures [bool (optional)] If True, when a solve errors during a sweep, skip it.

**kwargs : Passed to solver

sol [SolutionArray] See the SolutionArray documentation for details.

ValueError if the program is invalid. RuntimeWarning if an error occurs in solving or parsing the solution.

name = None

naming = None

num = None

program = None

solution = None

solve (*solver=None*, *verbosity=1*, *skipsweepfailures=False*, ***kwargs*)
Forms a mathematical program and attempts to solve it.

solver [string or function (optional)] If None, uses the default solver found in installation.

verbosity [int (optional)] If greater than 0 prints runtime messages. Is decremented by one and then passed to programs.

skipsweepfailures [bool (optional)] If True, when a solve errors during a sweep, skip it.

**kwargs : Passed to solver

sol [SolutionArray] See the SolutionArray documentation for details.

ValueError if the program is invalid. RuntimeWarning if an error occurs in solving or parsing the solution.

sp (*constants=None*, ***kwargs*)
Return program version of self

program: NomialData Class to return, e.g. GeometricProgram or SequentialGeometricProgram

return_attr: string attribute to return in addition to the program

subconstr_latex (*excluded=None*)
The collapsed appearance of a ConstraintBase

subconstr_str (*excluded=None*)
The collapsed appearance of a ConstraintBase

sweep (*sweeps*, ***solveargs*)
Sweeps {var: values} pairs in sweeps. Returns swept solutions.

verify_docstring ()
Verifies docstring bounds are sufficient but not excessive.

gpkit.constraints.model.**get_relaxed** (*relaxvals*, *mapped_list*, *min_return=1*)
Determines which relaxvars are considered 'relaxed'

gpkit.constraints.prog_factories module

Scripts for generating, solving and sweeping programs

gpkit.constraints.prog_factories.**evaluate_linked** (*constants*, *linked*)
Evaluates the values and gradients of linked variables.

gpkit.constraints.prog_factories.**run_sweep** (*genfunction*, *self*, *solution*, *skipsweepfailures*, *constants*, *sweep*, *linked*, *solver*, *verbosity*, ***kwargs*)
Runs through a sweep.

gpkit.constraints.relax module

Models for assessing primal feasibility

class gpkit.constraints.relax.**ConstantsRelaxed**(*constraints*, *include_only=None*, *exclude=None*)

Bases: *gpkit.constraints.set.ConstraintSet*

Relax constants in a constraintset.

constraints [iterable] Constraints which will be relaxed (made easier).

include_only [set] if declared, variable names must be on this list to be relaxed

exclude [set] if declared, variable names on this list will never be relaxed

relaxvars [Variable] The variables controlling the relaxation. A solved value of 1 means no relaxation was necessary or optimal for a particular constant. Higher values indicate the amount by which that constant has been made easier: e.g., a value of 1.5 means it was made 50 percent easier in the final solution than in the original problem. Of course, this can also be determined by looking at the constant's new value directly.

process_result(*result*)

Does arbitrary computation / manipulation of a program's result

There's no guarantee what order different constraints will process results in, so any changes made to the program's result should be careful not to step on other constraint's toes.

- check that an inequality was tight
- add values computed from solved variables

class gpkit.constraints.relax.**ConstraintsRelaxed**(*constraints*)

Bases: *gpkit.constraints.set.ConstraintSet*

Relax constraints, as in Eqn. 11 of [Boyd2007].

constraints [iterable] Constraints which will be relaxed (made easier).

relaxvars [Variable] The variables controlling the relaxation. A solved value of 1 means no relaxation was necessary or optimal for a particular constraint. Higher values indicate the amount by which that constraint has been made easier: e.g., a value of 1.5 means it was made 50 percent easier in the final solution than in the original problem.

[Boyd2007] : "A tutorial on geometric programming", Optim Eng 8:67-122

class gpkit.constraints.relax.**ConstraintsRelaxedEqually**(*constraints*)

Bases: *gpkit.constraints.set.ConstraintSet*

Relax constraints the same amount, as in Eqn. 10 of [Boyd2007].

constraints [iterable] Constraints which will be relaxed (made easier).

relaxvar [Variable] The variable controlling the relaxation. A solved value of 1 means no relaxation. Higher values indicate the amount by which all constraints have been made easier: e.g., a value of 1.5 means all constraints were 50 percent easier in the final solution than in the original problem.

[Boyd2007] : "A tutorial on geometric programming", Optim Eng 8:67-122

gpkit.constraints.set module

Implements ConstraintSet

class gpkit.constraints.set.**ConstraintSet**(*constraints*, *substitutions=None*)
Bases: list

Recursive container for ConstraintSets and Inequalities

append(*value*)
L.append(object) – append object to end

as_gpconstr(*x0*)
Returns GPConstraint approximating this constraint at x0

When x0 is none, may return a default guess.

as_posyslt1(*substitutions=None*)
Returns list of posynomials which must be kept <= 1

constrained_varkeys()
Return all varkeys in non-ConstraintSet constraints

flat(*constraintsets=True*)
Yields contained constraints, optionally including constraintsets.

latex(*excluded=None*)
LaTeX representation of a ConstraintSet.

process_result(*result*)
Does arbitrary computation / manipulation of a program's result

There's no guarantee what order different constraints will process results in, so any changes made to the program's result should be careful not to step on other constraint's toes.

- check that an inequality was tight
- add values computed from solved variables

reset_varkeys()
Goes through constraints and collects their varkeys.

rootconstr_latex(*excluded=None*)
The appearance of a ConstraintSet in addition to its contents

rootconstr_str(*excluded=None*)
The appearance of a ConstraintSet in addition to its contents

sens_from_dual(*las*, *nus*, *result*)
Computes constraint and variable sensitivities from dual solution

las [list] Sensitivity of each posynomial returned by *self.as_posyslt1*

nus: list of lists Each posynomial's monomial sensitivities

constraint_sens [dict] The interesting and computable sensitivities of this constraint

var_senss [dict] The variable sensitivities of this constraint

str_without(*excluded=None*)
String representation of a ConstraintSet.

subconstr_latex(*excluded=None*)
The collapsed appearance of a ConstraintSet

subconstr_str (*excluded=None*)
The collapsed appearance of a ConstraintSet

subinplace (*subs*)
Substitutes in place, updating self.substitutions accordingly.

Keys substituted with *subinplace* are no longer present, so if such a key is also in self.substitutions that substitution is now orphaned. If *subs[key]* describes some key in the ConstraintSet (i.e. one key has been substituted for another), then a substitution is added, mapping the orphaned value to this new key; otherwise, an error is raised.

unique_varkeys = frozenset([])

variables_byname (*key*)
Get all variables with a given name

varkeys = None

gpkit.constraints.set.**add_meq_bounds** (*bounded, meq_bounded*)
Iterates through meq_bounds until convergence

gpkit.constraints.set.**raise_badelement** (*cns, i, constraint*)
Identify the bad element and raise a ValueError

gpkit.constraints.set.**raise_elementhasnumpybools** (*constraint*)
Identify the bad subconstraint array and raise a ValueError

gpkit.constraints.sgp module

Implement the SequentialGeometricProgram class

class gpkit.constraints.sgp.**SequentialGeometricProgram** (*cost, constraints, substitutions*)

Bases: *gpkit.constraints.costed.CostedConstraintSet*

Prepares a collection of signomials for a SP solve.

cost [Posynomial] Objective to minimize when solving

constraints [list of Constraint or SignomialConstraint objects] Constraints to maintain when solving (implicitly Signomials <= 1)

verbosity [int (optional)] Currently has no effect: SequentialGeometricPrograms don't know anything new after being created, unlike GeometricPrograms.

gps is set during a solve *result* is set at the end of a solve

```
>>> gp = gpkit.geometric_program.SequentialGeometricProgram(
                    # minimize
                    x,
                    [   # subject to
                        1/x - y/x,  # <= 1, implicitly
                        y/10  # <= 1
                    ])
>>> gp.solve()
```

gp (*x0=None, mutategp=False*)
The GP approximation of this SP at x0.

init_gp (*substitutions, x0=None*)
Generates a simplified GP representation for later modification

localsolve (*solver=None*, *verbosity=1*, *x0=None*, *reltol=0.0001*, *iteration_limit=50*, *mutategp=True*, ***kwargs*)
Locally solves a SequentialGeometricProgram and returns the solution.

solver [str or function (optional)] By default uses one of the solvers found during installation. If set to "mosek", "mosek_cli", or "cvxopt", uses that solver. If set to a function, passes that function cs, A, p_idxs, and k.

verbosity [int (optional)] If greater than 0, prints solve time and number of iterations. Each GP is created and solved with verbosity one less than this, so if greater than 1, prints solver name and time for each GP.

x0 [dict (optional)] Initial location to approximate signomials about.

reltol [float] Iteration ends when this is greater than the distance between two consecutive solve's objective values.

iteration_limit [int] Maximum GP iterations allowed.

***args, **kwargs :** Passed to solver function.

result [dict] A dictionary containing the translated solver result.

gpkit.constraints.sigeq module

Implements SignomialEquality

class gpkit.constraints.sigeq.**SignomialEquality** (*left*, *right*)
Bases: *gpkit.constraints.set.ConstraintSet*

A constraint of the general form posynomial == posynomial

gpkit.constraints.single_equation module

Implements SingleEquationConstraint

class gpkit.constraints.single_equation.**SingleEquationConstraint** (*left*, *oper*, *right*)
Bases: object

Constraint expressible in a single equation.

func_opers = {'<=': <built-in function le>, '=': <built-in function eq>, '>=':

latex (*excluded=None*)
Latex representation without attributes in excluded list

latex_opers = {'<=': '\\leq', '=': '=', '>=': '\\geq'}

process_result (*result*)
Process solver results

str_without (*excluded=None*)
String representation without attributes in excluded list

subconstr_latex (*excluded*)
The collapsed latex of a constraint

subconstr_str (*excluded*)
The collapsed string of a constraint

`gpkit.constraints.single_equation.trycall(obj, attr, arg, default)`
Try to call method of an object, returning *default* if it does not exist

gpkit.constraints.tight module

Implements Tight

class `gpkit.constraints.tight.Tight(constraints, reltol=None, raiseerror=False)`
Bases: *gpkit.constraints.set.ConstraintSet*

ConstraintSet whose inequalities must result in an equality.

process_result(*result*)
Checks that all constraints are satisfied with equality

reltol = 1e-06

Module contents

Contains ConstraintSet and related classes and objects

gpkit.interactive package

Submodules

gpkit.interactive.chartjs module

gpkit.interactive.plot_sweep module

Implements plot_sweep1d function

`gpkit.interactive.plot_sweep.assign_axes(var, posys, axes)`
Assigns axes to posys, creating and formatting if necessary

`gpkit.interactive.plot_sweep.format_and_label_axes(var, posys, axes, ylabel=True)`
Formats and labels axes

`gpkit.interactive.plot_sweep.plot_1dsweepgrid(model, sweeps, posys, origsol=None, tol=0.01, **solveargs)`
Creates and plots a sweep from an existing model

Example usage: f, _ = plot_sweep_1d(m, {'x': np.linspace(1, 2, 5)}, 'y') f.savefig('mysweep.png')

gpkit.interactive.plotting module

Plotting methods

`gpkit.interactive.plotting.compare(models, sweeps, posys, tol=0.001)`
Compares the values of posys over a sweep of several models.

If posys is of the same length as models, this will plot different variables from different models.

Currently only supports a single sweepvar.

Example Usage: compare([aec, fbc], {“R”: (160, 300)},

["cost", ("W_{rm batt}", "W_{rm fuel}")], tol=0.001)

`gpkit.interactive.plotting.`**`plot_convergence`**(*model*)

Plots the convergence of a signomial programming model

model: Model Signomial programming model that has already been solved

matplotlib.pyplot Figure Plot of cost as functions of SP iteration #

gpkit.interactive.ractor module

Implements Ractor-based interactive CADtoons

`gpkit.interactive.ractor.`**`ractorjs`**(*title*, *model*, *update_py*, *ranges*, *constraint_js=”*)

Creates Javascript/HTML for CADtoon interaction without installing GPkit.

`gpkit.interactive.ractor.`**`ractorpy`**(*model*, *update_py*, *ranges*, *constraint_js=”*, *showtables=('cost', 'sensitivities')*)

Creates interactive iPython widget for controlling a CADtoon

`gpkit.interactive.ractor.`**`showcadtoon`**(*title*, *css=”*)

Displays cadtoon as iPython HTML

gpkit.interactive.sankey module

gpkit.interactive.widgets module

Module contents

Module for the interactive and plotting functions of GPkit

gpkit.nomials package

Submodules

gpkit.nomials.array module

Module for creating NomialArray instances.

Example

```
>>> x = gpkit.Monomial('x')
>>> px = gpkit.NomialArray([1, x, x**2])
```

`class` `gpkit.nomials.array.`**`NomialArray`**

Bases: `numpy.ndarray`

A Numpy array with elementwise inequalities and substitutions.

input_array : array-like

```
>>> px = gpkit.NomialArray([1, x, x**2])
```

latex (*matwrap=True*)
Returns 1D latex list of contents.

left
Returns (0, self[0], self[1] … self[N-1])

outer (*other*)
Returns the array and argument's outer product.

padleft (*padding*)
Returns ({padding}, self[0], self[1] … self[N])

padright (*padding*)
Returns (self[0], self[1] … self[N], {padding})

prod (**args*, ***kwargs*)
Returns a product. O(N) if no arguments and only contains monomials.

right
Returns (self[1], self[2] … self[N], 0)

str_without (*excluded=None*)
Returns string without certain fields (such as 'models').

sub (*subs*, *require_positive=True*)
Substitutes into the array

sum (**args*, ***kwargs*)
Returns a sum. O(N) if no arguments are given.

units
units must have same dimensions across the entire nomial array

vectorize (*function*, **args*, ***kwargs*)
Apply a function to each terminal constraint, returning the array

gpkit.nomials.array.**array_constraint** (*symbol*, *func*)
Return function which creates constraints of the given operator.

gpkit.nomials.core module

The shared non-mathematical backbone of all Nomials

class gpkit.nomials.core.**Nomial** (*hmap*)
Bases: *gpkit.nomials.data.NomialData*

Shared non-mathematical properties of all nomials

latex (*excluded=None*)
Latex representation, parsing *excluded* just as .str_without does

prod ()
Return self for compatibility with NomialArray

str_without (*excluded=None*)
String representation, excluding fields ('units', varkey attributes)

sub = None

sum()
Return self for compatibility with NomialArray

to(*units*)
Create new Signomial converted to new units

unitstr(*into='%s', options=None, dimless=''*)
Returns the string corresponding to an object's units.

value
Self, with values substituted for variables that have values

float, if no symbolic variables remain after substitution (Monomial, Posynomial, or Nomial), otherwise.

gpkit.nomials.data module

Machinery for exps, cs, varlocs data – common to nomials and programs

class gpkit.nomials.data.**NomialData**(*hmap*)
Bases: object

Object for holding cs, exps, and other basic 'nomial' properties.

cs: array (coefficient of each monomial term) exps: tuple of {VarKey: float} (exponents of each monomial term) varlocs: {VarKey: list} (terms each variable appears in) units: pint.UnitsContainer

cs
Create cs or return cached cs

diff(*var*)
Derivative of this with respect to a Variable

var (Variable): Variable to take derivative with respect to

NomialData

exps
Create exps or return cached exps

values
The NomialData's values, created when necessary.

varkeys
The NomialData's varkeys, created when necessary for a substitution.

varlocs
Create varlocs or return cached varlocs

gpkit.nomials.map module

Implements the NomialMap class

class gpkit.nomials.map.**NomialMap**
Bases: *gpkit.small_classes.HashVector*

Class for efficent algebraic represention of a nomial

A NomialMap is a mapping between hashvectors representing exponents and their coefficients in a posynomial.

For example, {{x : 1}: 2.0, {y: 1}: 3.0} represents 2*x + 3*y, where x and y are VarKey objects.

csmap = None

diff(*varkey*)
Differentiates a NomialMap with respect to a varkey

expmap = None

mmap(*orig*)
Maps substituted monomials back to the original nomial

self.expmap is the map from pre- to post-substitution exponents, and takes the form {original_exp: new_exp}

self.csmap is the map from pre-substitution exponents to coefficients.

m_from_ms is of the form {new_exp: [old_exps,]}

pmap is of the form [{orig_idx1: fraction1, orig_idx2: fraction2, },] where at the index corresponding to each new_exp is a dictionary mapping the indices corresponding to the old exps to their fraction of the post-substitution coefficient

remove_zeros()
Removes zeroed exponents and monomials.

If *only_check_cs* is True, checks only whether any values are zero. If False also checks whether any exponents in the keys are zero.

sub(*substitutions*, *varkeys*, *parsedsubs=False*)
Applies substitutions to a NomialMap

substitutions [(dict-like)] list of substitutions to perform

varkeys [(set-like)] varkeys that are present in self (required argument so as to require efficient code)

parsedsubs [bool] flag if the substitutions have already been parsed to contain only keys in varkeys

to(*units*)
Returns a new NomialMap of the given units

units = None

units_of_product(*thing*, *thing2=None*)
Sets units to those of *thing*thing2*

gpkit.nomials.map.**subinplace**(*cp*, *exp*, *o_exp*, *vk*, *cval*, *expval*, *exps_covered*)
Modifies cp by substituing cval/expval for vk in exp

gpkit.nomials.math module

Signomial, Posynomial, Monomial, Constraint, & MonoEQCOnstraint classes

class gpkit.nomials.math.**Monomial**(*hmap=None*, *cs=1*, *require_positive=True*, ***descr*)
Bases: *gpkit.nomials.math.Posynomial*

A Posynomial with only one term

Same as Signomial. Note: Monomial historically supported several different init formats

These will be deprecated in the future, replaced with a single __init__ syntax, same as Signomial.

c
Creates c or returns a cached c

exp
Creates exp or returns a cached exp

mono_approximation(*x0*)
Monomial approximation about a point x0

x0 (dict): point to monomialize about

Monomial (unless self(x0) < 0, in which case a Signomial is returned)

class gpkit.nomials.math.**MonomialEquality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.math.PosynomialInequality*

A Constraint of the form Monomial == Monomial.

as_posyslt1(*substitutions=None*)
Tags posynomials for dual feasibility checking

sens_from_dual(*la*, *nu*, *result*)
Returns the variable/constraint sensitivities from lambda/nu

class gpkit.nomials.math.**Posynomial**(*hmap=None*, *cs=1*, *require_positive=True*, ***descr*)
Bases: *gpkit.nomials.math.Signomial*

A Signomial with strictly positive cs

Same as Signomial. Note: Posynomial historically supported several different init formats

These will be deprecated in the future, replaced with a single __init__ syntax, same as Signomial.

mono_lower_bound(*x0*)
Monomial lower bound at a point x0

x0 (dict): point to make lower bound exact

Monomial

class gpkit.nomials.math.**PosynomialInequality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.math.ScalarSingleEquationConstraint*

A constraint of the general form monomial >= posynomial Stored in the posylt1_rep attribute as a single Posynomial (self <= 1) Usually initialized via operator overloading, e.g. cc = (y**2 >= 1 + x)

as_gpconstr(*x0*)
The GP version of a Posynomial constraint is itself

as_posyslt1(*substitutions=None*)
Returns the posys <= 1 representation of this constraint.

sens_from_dual(*la*, *nu*, *result*)
Returns the variable/constraint sensitivities from lambda/nu

class gpkit.nomials.math.**ScalarSingleEquationConstraint**(*left*, *oper*, *right*)
Bases: *gpkit.constraints.single_equation.SingleEquationConstraint*

A SingleEquationConstraint with scalar left and right sides.

nomials = []

subinplace (*substitutions*)
Modifies the constraint in place with substitutions.

class gpkit.nomials.math.**Signomial** (*hmap=None, cs=1, require_positive=True, **descr*)
Bases: *gpkit.nomials.core.Nomial*

A representation of a Signomial.

exps: tuple of dicts Exponent dicts for each monomial term

cs: tuple Coefficient values for each monomial term

require_positive: bool If True and Signomials not enabled, c <= 0 will raise ValueError

Signomial Posynomial (if the input has only positive cs) Monomial (if the input has one term and only positive cs)

diff (*var*)
Derivative of this with respect to a Variable

var (Variable): Variable to take derivative with respect to

Signomial (or Posynomial or Monomial)

mono_approximation (*x0*)
Monomial approximation about a point x0

x0 (dict): point to monomialize about

Monomial (unless self(x0) < 0, in which case a Signomial is returned)

posy_negy ()
Get the positive and negative parts, both as Posynomials

Posynomial, Posynomial: p_pos and p_neg in (self = p_pos - p_neg) decomposition,

sub (*substitutions, require_positive=True*)
Returns a nomial with substitued values.

3 == (x**2 + y).sub({'x': 1, y: 2}) 3 == (x).gp.sub(x, 3)

substitutions [dict or key] Either a dictionary whose keys are strings, Variables, or VarKeys, and whose values are numbers, or a string, Variable or Varkey.

val [number (optional)] If the substitutions entry is a single key, val holds the value

require_positive [boolean (optional, default is True)] Controls whether the returned value can be a Signomial.

Returns substituted nomial.

subinplace (*substitutions*)
Substitutes in place.

class gpkit.nomials.math.**SignomialInequality** (*left, oper, right*)
Bases: *gpkit.nomials.math.ScalarSingleEquationConstraint*

A constraint of the general form posynomial >= posynomial Stored internally (exps, cs) as a single Signomial (0 >= self) Usually initialized via operator overloading, e.g. cc = (y**2 >= 1 + x - y) Additionally retains input format (lhs vs rhs) in self.left and self.right Form is self.left >= self.right.

as_approxsgt (*x0*)
Returns monomial-greater-than sides, to be called after as_approxlt1

as_approxslt()
Returns posynomial-less-than sides of a signomial constraint

as_gpconstr(*x0*)
Returns GP approximation of an SP constraint at x0

as_posyslt1(*substitutions=None*)
Returns the posys <= 1 representation of this constraint.

sens_from_dual(*la*, *nu*, *result*)

We want to do the following chain: dlog(Obj)/dlog(monomial[i]) = nu[i] * dlog(monomial)/d(monomial) = 1/(monomial value) * d(monomial)/d(var) = see below * d(var)/dlog(var) = var = dlog(Obj)/dlog(var)

each final monomial is really (coeff signomial)/(negy signomial)

and by the chain rule d(monomial)/d(var) = d(coeff)/d(var)*1/negy + d(1/negy)/d(var)*coeff = d(coeff)/d(var)*1/negy - d(negy)/d(var)*coeff*1/negy**2

class gpkit.nomials.math.**SingleSignomialEquality**(*left*, *right*)
Bases: *gpkit.nomials.math.SignomialInequality*

A constraint of the general form posynomial == posynomial

as_approxsgt(*x0*)
Returns monomial-greater-than sides, to be called after as_approxlt1

as_approxslt()
Returns posynomial-less-than sides of a signomial constraint

as_gpconstr(*x0*)
Returns GP approximation of an SP constraint at x0

as_posyslt1(*substitutions=None*)
Returns the posys <= 1 representation of this constraint.

gpkit.nomials.substitution module

Scripts to parse and collate substitutions

gpkit.nomials.substitution.**append_sub**(*sub*, *keys*, *constants*, *sweep*, *linkedsweep*)
Appends sub to constants, sweep, or linkedsweep.

gpkit.nomials.substitution.**parse_subs**(*varkeys*, *substitutions*, *clean=False*)
Seperates subs into constants, sweeps linkedsweeps actually present.

gpkit.nomials.variables module

Implement Variable and ArrayVariable classes

class gpkit.nomials.variables.**ArrayVariable**
Bases: *gpkit.nomials.array.NomialArray*

A described vector of singlet Monomials.

shape [int or tuple] length or shape of resulting array

***args :**

may contain "name" (Strings)

"value" (Iterable) "units" (Strings)

and/or "label" (Strings)

****descr :** VarKey description

NomialArray of Monomials, each containing a VarKey with name '$name_{i}', where $name is the vector's name and i is the VarKey's index.

class gpkit.nomials.variables.**Variable**(**args, **descr*)
Bases: *gpkit.nomials.math.Monomial*

A described singlet Monomial.

***args** [list]

may contain "name" (Strings)

"value" (Numbers + Quantity) or (Iterable) for a sweep "units" (Strings)

and/or "label" (Strings)

****descr** [dict] VarKey description

Monomials containing a VarKey with the name '$name', where $name is the vector's name and i is the VarKey's index.

sub(**args, **kwargs*)
Same as nomial substitution, but also allows single-argument calls

x = Variable('x') assert x.sub(3) == Variable('x', value=3)

to(*units*)
Create new Signomial converted to new units

class gpkit.nomials.variables.**VectorizableVariable**(**args, **descr*)
Bases: *gpkit.nomials.variables.Variable*, *gpkit.nomials.variables.ArrayVariable*

A Variable outside a vectorized environment, an ArrayVariable within.

gpkit.nomials.variables.**addmodelstodescr**(*descr, addtonamedvars=None*)
Add models to descr, optionally adding the second argument to NAMEDVARS

gpkit.nomials.variables.**veclinkedfn**(*linkedfn, i*)
Generate an indexed linking function.

Module contents

Contains nomials, inequalities, and arrays

gpkit.tools package

Submodules

gpkit.tools.autosweep module

Tools for optimal fits to GP sweeps

class gpkit.tools.autosweep.**BinarySweepTree**(*bounds*, *sols*, *sweptvar*, *costposy*)

Bases: object

Spans a line segment. May contain two subtrees that divide the segment.

bounds [two-element list] The left and right boundaries of the segment

sols [two-element list] The left and right solutions of the segment

costs [array] The left and right logcosts of the segment

splits [None or two-element list] If not None, contains the left and right subtrees

splitval [None or float] The worst-error point, where the split will be if tolerance is too low

splitlb [None or float] The cost lower bound at splitval

splitub [None or float] The cost upper bound at splitval

add_split(*splitval*, *splitsol*)
Creates subtrees from bounds[0] to splitval and splitval to bounds[1]

add_splitcost(*splitval*, *splitlb*, *splitub*)
Adds a splitval, lower bound, and upper bound

cost_at(_, *value*, *bound=None*)
Logspace interpolates between split and costs. Guaranteed bounded.

min_bst(*value*)
Returns smallest bst around value.

posy_at(*posy*, *value*)
Logspace interpolates between sols to get posynomial values.

No guarantees, just like a regular sweep.

sample_at(*values*)
Creates a SolutionOracle at a given range of values

save(*filename='autosweep.p'*)
Pickles the autosweep and saves it to a file.

The saved autosweep is identical except for two things:

- the cost is made unitless
- each solution's 'program' attribute is removed

Solution can then be loaded with e.g.: >>> import cPickle as pickle >>> pickle.load(open("autosweep.p"))

solarray
Returns a solution array of all the solutions in an autosweep

sollist
Returns a list of all the solutions in an autosweep

class gpkit.tools.autosweep.**SolutionOracle**(*bst*, *sampled_at*)

Bases: object

Acts like a SolutionArray for autosweeps

cost_lb()
Gets cost lower bounds from the BST and units them

cost_ub()
Gets cost upper bounds from the BST and units them

plot(*posys=None*, *axes=None*)
Plots the sweep for each posy

solarray
Returns a solution array of all the solutions in an autosweep

gpkit.tools.autosweep.**autosweep_1d**(*model*, *logtol*, *sweepvar*, *bounds*, ***solvekwargs*)
Autosweep a model over one sweepvar

gpkit.tools.autosweep.**get_tol**(*costs*, *bounds*, *sols*, *variable*)
Gets the intersection point and corresponding bounds from two solutions.

gpkit.tools.autosweep.**recurse_splits**(*model*, *bst*, *variable*, *logtol*, *solvekwargs*, *sols*)
Recursively splits a BST until logtol is reached

gpkit.tools.docstring module

Docstring-parsing methods

gpkit.tools.docstring.**check_and_parse_flag**(*string*, *flag*, *errorcatch*, *declaration_func=None*)
Checks for instances of flag in string and parses them.

gpkit.tools.docstring.**constant_declare**(*string*, *flag*, *idx2*, *countstr*)
Turns Variable declarations into Constant ones

gpkit.tools.docstring.**expected_unbounded**(*instance*, *doc*)
Gets expected-unbounded variables from a string

gpkit.tools.docstring.**parse_variables**(*string*, *errorcatch=True*)
Parses a string to determine what variables to create from it

gpkit.tools.docstring.**variable_declaration**(*nameval*, *units*, *label*, *line*, *errorcatch=True*)
Turns parsed output into a Variable declaration

gpkit.tools.docstring.**vv_declare**(*string*, *flag*, *idx2*, *countstr*)
Turns Variable declarations into VectorVariable ones

gpkit.tools.fmincon module

A module to facilitate testing GPkit against fmincon

gpkit.tools.fmincon.**generate_mfiles**(*model*, *logspace=False*, *algorithm='interior-point'*, *guess='ones'*, *gradobj='on'*, *gradconstr='on'*, *writefiles=True*)
A method for preparing fmincon input files to run a GPkit program

INPUTS: model [GPkit model] The model to replicate in fmincon

logspace [Boolean] Whether to re-produce the model in logspace

algorithm: [string] Algorithm used by fmincon 'interior-point': uses the interior point solver 'SQP': uses the sequential quadratic programming solver

guess: [string] The type of initial guess used 'ones': One for each variable 'order-of-magnitude-floor': The "log-floor" order of

magnitude of the GP/SP optimal solution (i.e. O(99)=10)

'order-of-magnitude-round': The "log-nearest" order of magnitude of the GP/SP optimal solution (i.e. O(42)=100)

'almost-exact-solution': The GP/SP optimal solution rounded to 1 significant figure

OR [list] The actual values of initial guess to use

gradconstr: [string] Include analytical constraint gradients? 'on': Yes 'off': No

gradobj: [string] Include analytical objective gradients? 'on': Yes 'off': No

writefiles: [Boolean] whether or not to actually write the m files

`gpkit.tools.fmincon.`**`make_initial_guess`**(*model*, *newlist*, *guess='ones'*, *logspace=False*)

Returns initial guess

gpkit.tools.spdata module

Implements SPData class

class `gpkit.tools.spdata.`**`SPData`**(*model*)

Bases: *gpkit.nomials.data.NomialData*

Generates matrices describing an SP.

```
>>> spdata = SPData(m)
>>> spdata.save('example_sp.h5')
```

`save`(*filename*)

Save spdata to an h5 file.

gpkit.tools.tools module

Non-application-specific convenience methods for GPkit

`gpkit.tools.tools.`**`mdmake`**(*filename*, *make_tex=True*)

Make a python file and (optional) a pandoc-ready .tex.md file

`gpkit.tools.tools.`**`mdparse`**(*filename*, *return_tex=False*)

Parse markdown file, returning as strings python and (optionally) .tex.md

`gpkit.tools.tools.`**`te_exp_minus1`**(*posy*, *nterm*)

Taylor expansion of e^{posy} - 1

posy [gpkit.Posynomial] Variable or expression to exponentiate

nterm [int] Number of non-constant terms in resulting Taylor expansion

gpkit.Posynomial Taylor expansion of e^{posy} - 1, carried to nterm terms

`gpkit.tools.tools.`**`te_secant`**(*var*, *nterm*)

Taylor expansion of secant(var).

var [gpkit.monomial] Variable or expression argument

nterm [int] Number of non-constant terms in resulting Taylor expansion

gpkit.Posynomial Taylor expansion of secant(x), carried to nterm terms

`gpkit.tools.tools.te_tangent(var, nterm)`
Taylor expansion of tangent(var).

var [gpkit.monomial] Variable or expression argument

nterm [int] Number of non-constant terms in resulting Taylor expansion

gpkit.Posynomial Taylor expansion of tangent(x), carried to nterm terms

Module contents

Contains miscellaneous tools including fmincon comparison tool

11.1.2 Submodules

11.1.3 gpkit.build module

Finds solvers, sets gpkit settings, and builds gpkit

class `gpkit.build.CVXopt`
Bases: *gpkit.build.SolverBackend*

CVXopt finder.

look()
Attempts to import cvxopt.

name = 'cvxopt'

class `gpkit.build.Mosek`
Bases: *gpkit.build.SolverBackend*

MOSEK finder and builder.

bin_dir = None

build()
Builds a dynamic library to GPKITBUILD or $HOME/.gpkit

expopt_files = None

flags = None

lib_name = None

lib_path = None

look()
Looks in default install locations for latest mosek version.

name = 'mosek'

patches = {'dgopt.c': {'printf("Number of Hessian non-zeros: %d\\n",nlh[0]->numh

version = None

class gpkit.build.**MosekCLI**
    Bases: *gpkit.build.SolverBackend*

    MOSEK command line interface finder.

    look()
        Attempts to run mskexpopt.

    name = 'mosek_cli'

class gpkit.build.**SolverBackend**
    Bases: `object`

    Inheritable class for finding solvers. Logs.

    build = None

    installed = False

    look = None

    name = None

gpkit.build.**build**()
    Builds GPkit

gpkit.build.**call**(*cmd*)
    Calls subprocess. Logs.

gpkit.build.**diff**(*filename*, *diff_dict*)
    Applies a simple diff to a file. Logs.

gpkit.build.**isfile**(*path*)
    Returns true if there's a file at $path. Logs.

gpkit.build.**log**(**args*)
    Print a line and append it to the log string.

gpkit.build.**pathjoin**(**args*)
    Join paths, collating multiple arguments.

gpkit.build.**replacedir**(*path*)
    Replaces directory at $path. Logs.

11.1.4 gpkit.exceptions module

GPkit-specific Exception classes

exception gpkit.exceptions.**InvalidGPConstraint**
    Bases: `exceptions.Exception`

    Raised when a non-GP-compatible constraint is used in a GP

11.1.5 gpkit.globals module

global mutable variables

class gpkit.globals.**NamedVariables**(*model*)
    Bases: `object`

    Creates an environment in which all variables have a model name and num appended to their varkeys.

class gpkit.globals.**SignomialsEnabled**
Bases: object

Class to put up and tear down signomial support in an instance of GPkit.

```
>>> import gpkit
>>> x = gpkit.Variable("x")
>>> y = gpkit.Variable("y", 0.1)
>>> with SignomialsEnabled():
>>>     constraints = [x >= 1-y]
>>> gpkit.Model(x, constraints).localsolve()
```

class gpkit.globals.**Vectorize**(*dimension_length*)
Bases: object

Creates an environment in which all variables are exended in an additional dimension.

gpkit.globals.**begin_variable_naming**(*model*)
Appends a model name and num to the environment.

gpkit.globals.**end_variable_naming**()
Pops a model name and num from the environment.

gpkit.globals.**load_settings**(*path=None*, *firstattempt=True*)
Load the settings file at SETTINGS_PATH; return settings dict

gpkit.globals.**reset_modelnumbers**()
Zeroes all model number counters

11.1.6 gpkit.keydict module

Implements KeyDict and KeySet classes

class gpkit.keydict.**KeyDict**(**args*, ***kwargs*)
Bases: dict

KeyDicts do two things over a dict: map keys and collapse arrays.

>>>> kd = gpkit.keydict.KeyDict()

If `.keymapping` is True, a KeyDict keeps an internal list of VarKeys as canonical keys, and their values can be accessed with any object whose *key* attribute matches one of those VarKeys, or with strings matching any of the multiple possible string interpretations of each key:

For example, after creating the KeyDict kd and setting kd[x] = v (where x is a Variable or VarKey), v can be accessed with by the following keys:

- x
- x.key
- x.name (a string)
- "x_modelname" (x's name including modelname)

Note that if a item is set using a key that does not have a *.key* attribute, that key can be set and accessed normally.

If `.collapse_arrays` is True then VarKeys which have a *shape* parameter (indicating they are part of an array) are stored as numpy arrays, and automatically de-indexed when a matching VarKey with a particular *idx* parameter is used as a key.

See also: gpkit/tests/t_keydict.py.

collapse_arrays = True

get (*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

keymapping = True

parse_and_index (*key*)
Returns key if key had one, and veckey/idx for indexed veckeys.

update (**args*, ***kwargs*)
Iterates through the dictionary created by args and kwargs

update_keymap ()
Updates the keymap with the keys in _unmapped_keys

class gpkit.keydict.**KeySet** (**args*, ***kwargs*)
Bases: gpkit.keydict.KeyDict

KeyDicts that don't collapse arrays or store values.

add (*item*)
Adds an item to the keyset

collapse_arrays = False

update (**args*, ***kwargs*)
Iterates through the dictionary created by args and kwargs

gpkit.keydict.**clean_value** (*key*, *value*)
Gets the value of variable-less monomials, so that *x.sub({x: gpkit.units.m})* and *x.sub({x: gpkit.ureg.m})* are equivalent.

Also converts any quantities to the key's units, because quantities can't/shouldn't be stored as elements of numpy arrays.

11.1.7 gpkit.repr_conventions module

Repository for representation standards

gpkit.repr_conventions.**unitstr** (*units*, *into='%s'*, *options=None*, *dimless=''*)
Returns the string corresponding to an object's units.

11.1.8 gpkit.small_classes module

Miscellaneous small classes

class gpkit.small_classes.**CootMatrix** (*row*, *col*, *data*)
Bases: object

A very simple sparse matrix representation.

append (*row*, *col*, *data*)
Appends entry to matrix.

dot (*arg*)
Returns dot product with arg.

tocoo ()
Converts to another type of matrix.

tocsc ()
Converts to another type of matrix.

tocsr()
Converts to a Scipy sparse csr_matrix

todense()
Converts to another type of matrix.

todia()
Converts to another type of matrix.

todok()
Converts to another type of matrix.

class gpkit.small_classes.**Count**
Bases: object

Like python 2's itertools.count, for Python 3 compatibility.

next()
Increment self.count and return it

class gpkit.small_classes.**DictOfLists**
Bases: dict

A hierarchy of dicionaries, with lists at the bottom.

append(*sol*)
Appends a dict (of dicts) of lists to all held lists.

atindex(*i*)
Indexes into each list independently.

to_arrays()
Converts all lists into array.

class gpkit.small_classes.**HashVector**
Bases: dict

A simple, sparse, string-indexed vector. Inherits from dict.

The HashVector class supports element-wise arithmetic: any undeclared variables are assumed to have a value of zero.

arg : iterable

```
>>> x = gpkit.nomials.Monomial('x')
>>> exp = gpkit.small_classes.HashVector({x: 2})
```

copy()
Return a copy of this

class gpkit.small_classes.**SolverLog**(*verbosity=0*, *output=None*, ***kwargs*)
Bases: list

Adds a *write* method to list so it's file-like and can replace stdout.

write(*writ*)
Append and potentially write the new line.

gpkit.small_classes.**matrix_converter**(*name*)
Generates conversion function.

11.1.9 gpkit.small_scripts module

Assorted helper methods

gpkit.small_scripts.**is_sweepvar**(*sub*)
Determines if a given substitution indicates a sweep.

gpkit.small_scripts.**latex_num**(*c*)
Returns latex string of numbers, potentially using exponential notation.

gpkit.small_scripts.**mag**(*c*)
Return magnitude of a Number or Quantity

gpkit.small_scripts.**maybe_flatten**(*value*)
Extract values from 0-d numpy arrays, if necessary

gpkit.small_scripts.**nomial_latex_helper**(*c*, *pos_vars*, *neg_vars*)
Combines (varlatex, exponent) tuples, separated by positive vs negative exponent, into a single latex string

gpkit.small_scripts.**try_str_without**(*item*, *excluded*)
Try to call item.str_without(excluded); fall back to str(item)

11.1.10 gpkit.solution_array module

Defines SolutionArray class

class gpkit.solution_array.**SolutionArray**
Bases: *gpkit.small_classes.DictOfLists*

A dictionary (of dictionaries) of lists, with convenience methods.

cost : array variables: dict of arrays sensitivities: dict containing:

monomials : array posynomials : array variables: dict of arrays

localmodels [NomialArray] Local power-law fits (small sensitivities are cut off)

```
>>> import gpkit
>>> import numpy as np
>>> x = gpkit.Variable("x")
>>> x_min = gpkit.Variable("x_{min}", 2)
>>> sol = gpkit.Model(x, [x >= x_min]).solve(verbosity=0)
>>>
>>> # VALUES
>>> values = [sol(x), sol.subinto(x), sol["variables"]["x"]]
>>> assert all(np.array(values) == 2)
>>>
>>> # SENSITIVITIES
>>> senss = [sol.sens(x_min), sol.sens(x_min)]
>>> senss.append(sol["sensitivities"]["variables"]["x_{min}"])
>>> assert all(np.array(senss) == 1)
```

almost_equal(*sol*, *reltol=0.001*, *sens_abstol=0.01*)
Checks for almost-equality between two solutions

diff(*sol*, *min_percent=1.0*, *show_sensitivities=True*, *min_senss_delta=0.1*)
Outputs differences between this solution and another

sol [solution or string] Strings are treated as paths to valid pickled solutions

min_percent [float] The smallest percentage difference in the result to consider

show_sensitivities [boolean] if True, also computer sensitivity deltas

min_senss_delta [float] The smallest absolute difference in sensitivities to consider

str

plot (*posys=None*, *axes=None*)
Plots a sweep for each posy

program = None

save (*filename='solution.p'*)
Pickles the solution and saves it to a file.

The saved solution is identical except for two things:

- the cost is made unitless
- the solution's 'program' attribute is removed

Solution can then be loaded with e.g.: >>> import cPickle as pickle >>> pickle.load(open("solution.p"))

savecsv (*filename='solution.csv'*, *include=None*, *min_data=False*)
Saves primal solution as csv

savemat (*filename='solution.mat'*, *include=None*, *min_data=True*)
Saves primal solution as matlab file

subinto (*posy*)
Returns NomialArray of each solution substituted into posy.

summary (*showvars=()*, *ntopsenss=5*, ***kwargs*)
Print summary table, showing top sensitivities and no constants

table (*showvars=()*, *tables=('cost', 'sweepvariables', 'freevariables', 'constants', 'sensitivities')*, ***kwargs*)
A table representation of this SolutionArray

tables: Iterable

Which to print of ("cost", "sweepvariables", "freevariables", "constants", "sensitivities")

fixedcols: If true, print vectors in fixed-width format latex: int

If > 0, return latex format (options 1-3); otherwise plain text

included_models: Iterable of strings If specified, the models (by name) to include

excluded_models: Iterable of strings If specified, model names to exclude

str

table_titles = {'constants': 'Constants', 'freevariables': 'Free Variables', 'sw

todataframe (*include=None*, *min_data=False*)
Returns primal solution as pandas dataframe

gpkit.solution_array.**insenss_table** (*data*, *_*, *maxval=0.1*, ***kwargs*)
Returns insensitivity table lines

`gpkit.solution_array.`**`results_table`**(*data*, *title*, *printunits=True*, *fixedcols=True*, *varfmt='%s : '*, *valfmt='%-.4g '*, *vecfmt='%-8.3g'*, *included_models=None*, *excluded_models=None*, *latex=False*, *minval=0*, *sortbyvals=False*, *hidebelowminval=False*, *columns=None*, *maxcolumns=5*, ***_*)

Pretty string representation of a dict of VarKeys Iterable values are handled specially (partial printing)

data: dict whose keys are VarKey's data to represent in table

title: string minval: float

skip values with all(abs(value)) < minval

printunits: bool fixedcols: bool

if True, print rhs (val, units, label) in fixed-width cols

varfmt: string format for variable names

valfmt: string format for scalar values

vecfmt: string format for vector values

latex: int If > 0, return latex format (options 1-3); otherwise plain text

included_models: Iterable of strings If specified, the models (by name) to include

excluded_models: Iterable of strings If specified, model names to exclude

sortbyvals [boolean] If true, rows are sorted by their average value instead of by name.

`gpkit.solution_array.`**`senss_table`**(*data*, *showvars=()*, *title='Sensitivities'*, ***kwargs*)

Returns sensitivity table lines

`gpkit.solution_array.`**`topsenss_filter`**(*data*, *showvars*, *nvars=5*)

Filters sensitivities down to top N vars

`gpkit.solution_array.`**`topsenss_table`**(*data*, *showvars*, *nvars=5*, ***kwargs*)

Returns top sensitivity table lines

11.1.11 gpkit.varkey module

Defines the VarKey class

class `gpkit.varkey.`**`VarKey`**(*name=None*, ***kwargs*)

Bases: `object`

An object to correspond to each 'variable name'.

name [str, VarKey, or Monomial] Name of this Variable, or object to derive this Variable from.

****kwargs :** Any additional attributes, which become the descr attribute (a dict).

VarKey with the given name and descr.

`latex`(*excluded=None*)

Returns latex representation.

latex_unitstr()
Returns latex unitstr

naming
Returns this varkey's naming tuple

str_without(*excluded=None*)
Returns string without certain fields (such as 'models').

subscripts = ('models', 'idx')

classmethod unique_id()
Increment self.count and return it

unitstr(*into='%s'*, *options=None*, *dimless=''*)
Returns the string corresponding to an object's units.

11.1.12 Module contents

GP and SP modeling package

CHAPTER 12

Citing GPkit

If you use GPkit, please cite it with the following bibtex:

```
@Misc{gpkit,
      author={Edward Burnell and Warren Hoburg},
      title={GPkit software for geometric programming},
      howpublished={\url{https://github.com/convexengineering/gpkit}},
      year={2018},
      note={Version 0.7.0}
     }
```

CHAPTER 13

Acknowledgements

We thank the following contributors for helping to improve GPkit:

- Marshall Galbraith for setting up continuous integration.
- Stephen Boyd for inspiration and suggestions.
- Kirsten Bray for designing the GPkit logo.

CHAPTER 14

Release Notes

This page lists the changes made in each point version of gpkit.

14.1 Version 0.7.0

- Variable's values are now used only in their first ConstraintSet; in other ConstraintSets they're free variables
- Variable values can be preserved by setting `constant=True` during variable declaration
- MOSEK home directory can be set by a `MSKHOME` environment variable at build time
- `sol(var)` now always returns Pint Quantities, even if the variable is dimensionless
- `sol[...][var]`, on the other hand, now always returns floats / numpy arrays of floats
- Optional boundedness checking in docstring (see usage in docs)
- Automatic boundedness checking for GPs
- Sankey diagrams
- Many other fixes

14.2 Version 0.6.0

- new much faster NomialMap data structure (#682)
- Many bug fixes and small improvements.
- 6900 lines of code, 2200 lines of tests, 2100 lines of docstring.

14.3 Version 0.5.3

- faster SP solves (#1109)
- LinkedConstraintSet deprecated (#1110)
- Fixes to autosweep, ConstraintSet, interactive
- Solution time is now stored with soltutions (including sweeps/SPs)
- Model strings are divided with slashes (e.g. Airplane/Wing)

14.4 Version 0.5.2

- **Added new `sweep` and `autosweep` methods to Model**
  - Added `plot` routines to the results of those routines to make it easy to plot a 1D sweep.
- **Added new `summary` method to solution_array.**
  - It and table accept iterables of vars, will only print vars in that iterable (or, by default, all vars)
- **Cleaned up and documented the `interactive` submodule**
  - removed contour and sensitivity plots
  - added a 1D-sweep plotting function
  - added that plotting function as an option within the control panel interface
- **Overhauled and documented three types of variables whose value is determined by functions:**
  - calculated constants
  - post-solve calculated variables
  - between-GP-solves calculated variables (for Sequential Geometric Programs)
- Fix `Bounded` and implement `debug()` for SPs
- Apply `subinplace` to substitutions dictionary as well
- Require GP substitutions to be Numbers only
- Extend Bounded to one-sided bounds
- Print model's numbers by default, unless `"modelnums" in exclude`
- Implement lazy keymapping, allowing GP/SP results to be KeyDicts
- Handle Signomial Inequalities that become Posynomial Inequalities after substitution
- Various documentation updates
- Various bug fixes

14.5 Version 0.5.1

- O(N) sums and monomial products

- Warn about invalid ConstraintSet elements
- allow setting Tight tolerance as a class attribute
- full backwards compatibility for __init__ methods
- scripts to test remote repositories
- minor fixes, tests, and refactors
- 3550 lines of code, 1800 lines of tests, 1700 lines of docstring. (not counting *interactive*)

14.6 Version 0.5.0

- No longer recommend the use of linked variables and subinplace (see below)
- Switched default solver to MOSEK
- Added Linked Variable diagram (PR #915)
- Changed how overloaded operators interact with pint (PR #938)
- Added and documented debugging tools (PR #933)
- Added and documented vectorization tools
- Documented modular model construction
- 3200 lines of code, 1800 lines of tests, 1700 lines of docstring. (not counting *interactive*)

14.6.1 Changes to named models / Model inheritance

We are deprecating the creation of named submodels with custom `__init__` methods. Previously, variables created during `__init__` in any class inheriting from Model were replaced by a copy with `__class__.__name__` added as varkey metadata. This was slow, a bit irregular, and hacky.

We're moving to an explicitly-irregular `setup` method, which (if declared for a class inheriting from Model) is automatically called during `Model.__init__` inside a `NamedVariables(self.__class__.__name__)` environment. This 1) handles the naming of variables more explicitly and efficiently, and 2) allows us to capture variables created within `setup`, so that constants that are not a part of any constraint can be used directly (several examples of such template models are in the new *Building Complex Models* documentation).

`Model.__init__` calls `setup` with the arguments given to the constructor, with the exception of the reserved keyword `substitutions`. This allows for the easy creation of a named model with custom parameter values (as in the documentation's Beam example). `setup` methods should return an iterable (list, tuple, ConstraintSet, ...) of constraints or nothing if the model contains no constraints. To declare a submodel cost, set `self.cost` during `setup`. However, we often find declaring a model's cost explicitly just before solving to be a more legible practice.

In addition to permitting us to name variables at creation, and include unconstrained variables in a model, we hope that `setup` methods will clarify the side effects of named model creation.

14.7 Version 0.4.2

- prototype handling of SignomialEquality constraints

- fix an issue where solution tables printed incorrect units (despite the units being correct in the `SolutionArray` data structure)
- fix `controlpanel` slider display for newer versions of ipywidgets
- fix an issue where identical unit-ed variables could have different hashes
- Make the text of several error messages more informative
- Allow monomial approximation of monomials
- bug fixes and improvements to TightConstraintSet
- Don't print results table automatically (it was unwieldy for large models). To print it, `print sol.table()`.
- Use cvxopt's ldl kkt solver by default for more robustness to rank issues
- Improved `ConstraintSet.__getitem__`, only returns top-level Variable
- Move toward the varkeys of a ConstraintSet being an immutable set
- CPI update
- numerous pylint fixes
- BoundedConstraint sets added for dual feasibility debugging
- SP sweep compatibility

14.8 Version 0.4.0

- New model for considering constraints: all constraints are considered as sets of constraints which may contain other constraints, and are asked for their substitutions / posynomial less than 1 representation as late as possible.
- Support for calling external code during an SP solve.
- New class KeyDict to allow referring to variables by name or with objects.
- Many many other bug fixes, speed ups, and refactors under the hood.

14.9 Version 0.3.4

- Modular / model composition fixes and improvements
- Working controlpanel() for Model
- ipynb and numpy dependency fixes
- printing fixes
- El Capitan fix
- slider widgets now have units

14.10 Version 0.3.2

- Assorted bug fixes
- Assorted internal improvements and simplifications
- Refactor signomial constraints, resulting in smarter SP heuristic
- Simplify and strengthen equality testing for nomials
- Not counting submodules, went from 2400 to 2500 lines of code and from 1050 to 1170 lines of docstrings and comments.

14.11 Version 0.3

- Integrated GP and SP creation under the Model class
- Improved and simplified under-the-hood internals of GPs and SPs
- New experimental SP heuristic
- Improved test coverage
- Handles vectors which are partially constants, partially free
- Simplified interaction with Model objects and made it more pythonic
- Added SP "step" method to allow single-stepping through an SP
- Isolated and corrected some solver-specific behavior
- Fully allowed substitutions of variables for 0 (commit 4631255)
- Use "with" to create a signomials environment (commit cd8d581)
- Continuous integration improvements, thanks @galbramc !
- Not counting subpackages, went from 2200 to 2400 lines of code (additions were mostly longer error messages) and from 650 to 1050 lines of docstrings and comments.
- Add automatic feasibility-analysis methods to Model and GP
- Simplified solver logging and printing, making it easier to access solver output.

14.12 Version 0.2

- Various bug fixes
- Python 3 compatibility
- Added signomial programming support (alpha quality, may be wrong)
- Added composite objectives
- Parallelized sweeping
- Better table printing
- Linked sweep variables
- Better error messages

- Closest feasible point capability
- Improved install process (no longer requires ctypesgen; auto-detects MOSEK version)
- Added examples: wind turbine, modular GP, examples from 1967 book, maintenance (part replacement)
- Documentation grew by ~70%
- Added Advanced Commands section to documentation
- Many additional unit tests (more than doubled testing lines of code)

Python Module Index

g

Index

A

B

C

H

I

K

L

M

N

O

P

R

S

T

U

V

W