gpkit Documentation

Release 0.5.1

MIT Department of Aeronautics and Astronautics

December 30, 2016

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.

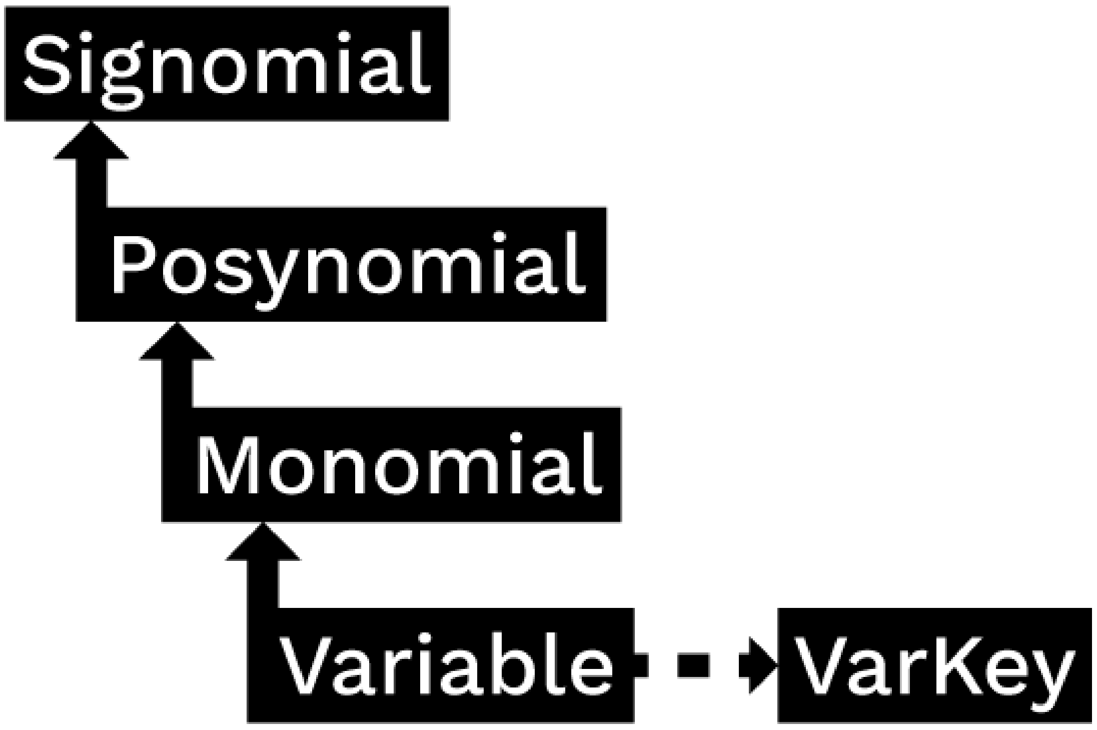
2.2 Substitution

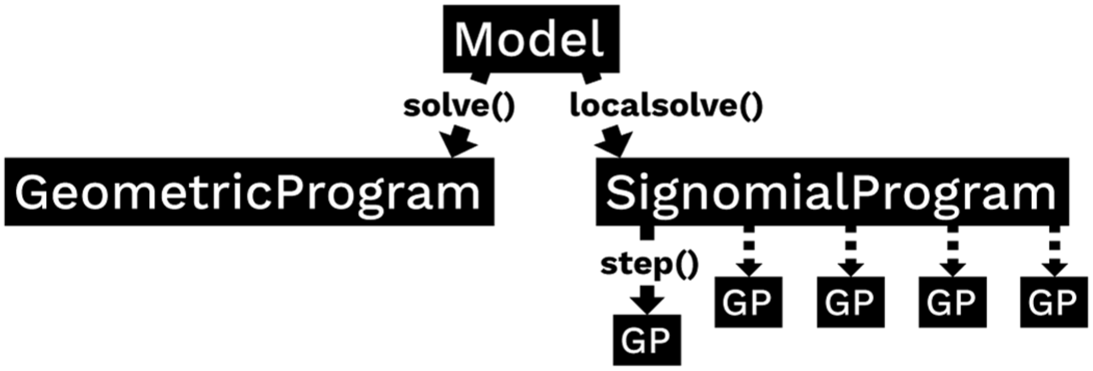
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable's "name". It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric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

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.


Signomial
Posynomial
Monomial
Variable
VarKey


Model
solve()
localsolve()
GeometricProgram
SignomialProgram
step()
GP
GP
GP
GP
GP

Installation Instructions

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

3.1 Installation dependencies

To install GPkit, you'll need to have the following python packages already installed on your system:

- `pip`
- `numpy` version 1.8.1 or newer
- `scipy`
- `pint`

and at least one solver, which we'll choose and install in a later step.

There are many ways to install these dependencies, but here's our suggestion:

3.1.1 Get `pip`

Mac OS X Run `easy_install pip` at a terminal window.

Linux

Use your package manager to install `pip` Ubuntu: `sudo apt-get install python-pip`

Windows Install the Python 2.7 64-bit version of Anaconda.

3.1.2 Get python packages

Mac OS X

Run the following commands:

- `pip install pip --upgrade`
- `pip install numpy`
- `pip install scipy`
- `pip install pint`

Linux

Use your package manager to install `numpy` and `scipy` Ubuntu: `sudo apt-get install python-numpy python-scipy`

Run `pip install pint` (for system python installs, use `sudo pip`)

Windows Do nothing at this step; Anaconda already has the needed packages.

3.2 Install a GP solver

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

At least one solver is required.

3.2.1 Installing cvxopt

Mac OSX Run `pip install cvxopt`

Linux Run `sudo apt-get install libblas-dev liblapack-dev libsuitesparse-dev` or otherwise install those libraries

Run `pip install cvxopt` (for system python installs, use `sudo pip`)

If experiencing issues with wheel in Ubuntu 16.04, try the official installer.

Windows Run `conda install -c omnia cvxopt` in an Anaconda Command Prompt.

3.2.2 Installing mosek

Dependency note: GPkit uses the python package ctypesgen to interface with the MOSEK C bindings.

Licensing note: if you do not have a paid license, you will need an academic or trial license to proceed.

Mac OS X

- If `which gcc` does not return anything, install `XCode` and the Apple Command Line Tools.
- Install ctypesgen with `pip install ctypesgen --pre`.
- **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

- Install ctypesgen with `pip install ctypesgen --pre` (for system python installs, use `sudo pip`)
- **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

- Install ctypesgen by running `pip install ctypesgen --pre` in an Anaconda Command Prompt .
- **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) of mingw.
    - Make sure the `mingw` bin directory is on your system path (you may have to add it manually).

3.3 Install GPkit

- Run `pip install gpkit` at the command line (for system python installs, use `sudo pip`)
- Run `pip install jupyter` to install jupyter notebook (recommended)
- Run `jupyter nbextension enable --py widgetsnbextension` for interactive control of models in jupyter (recommended)
- Run `python -c "import gpkit.tests; gpkit.tests.run()"` to run the tests; if any tests do not pass, please email `gpkit@mit.edu` or raise a GitHub issue.
- Join our mailing list and/or chatroom for support and examples.

3.4 Debugging installation

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

- You install a new solver (mosek or cvxopt) after installing GPkit
- You delete the `.gpkit` folder from your home directory
- 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, first try running `python -c "from gpkit.build import rebuild; rebuild()"`. If that do

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

3.5 Bleeding-edge / developer installations

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

1. Run `pip uninstall gpkit` to uninstall your existing GPkit.
2. Run `git clone https://github.com/hoburg/gpkit.git` to clone the GPkit repository.
3. Run `pip install -e gpkit` to install that directory as your environment-wide GPkit.
4. Run `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
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(verbosity=0)
```

```
Debugging...
_____________________

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
_____________________

Model does not solve with relaxed constraints.
_____________________
```

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 **`Run`**
  - 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:

```python
"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.table()
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]


Cost
----
 1e-30

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

sol['boundedness'] is: {'value near upper bound': array([x], dtype=object), 'sensitive to upper
```

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
    ]

#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
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
=====================================

  # minimize
        C_Relax_(0,)*C_Relax_(1,)*x**0.01
  # subject to
        C >= [gpkit.Monomial(x*x_max**-1), gpkit.Monomial(x**-1*x_min)]
        C >= 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*x_min_Relax
  # subject to
        x <= x_max
        x >= x_min
        x_min >= 2*x_min_Relax**-1
        x_min <= 2*x_min_Relax
        x_min_Relax >= 1
        x_max >= x_max_Relax**-1
        x_max <= x_max_Relax
        x_max_Relax >= 1

Cost
----
 2

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

```
         x : 1
     x_max : 1
     x_min : 1

           | Relax.2
     x_max : 1
     x_min : 2
```

CHAPTER 6

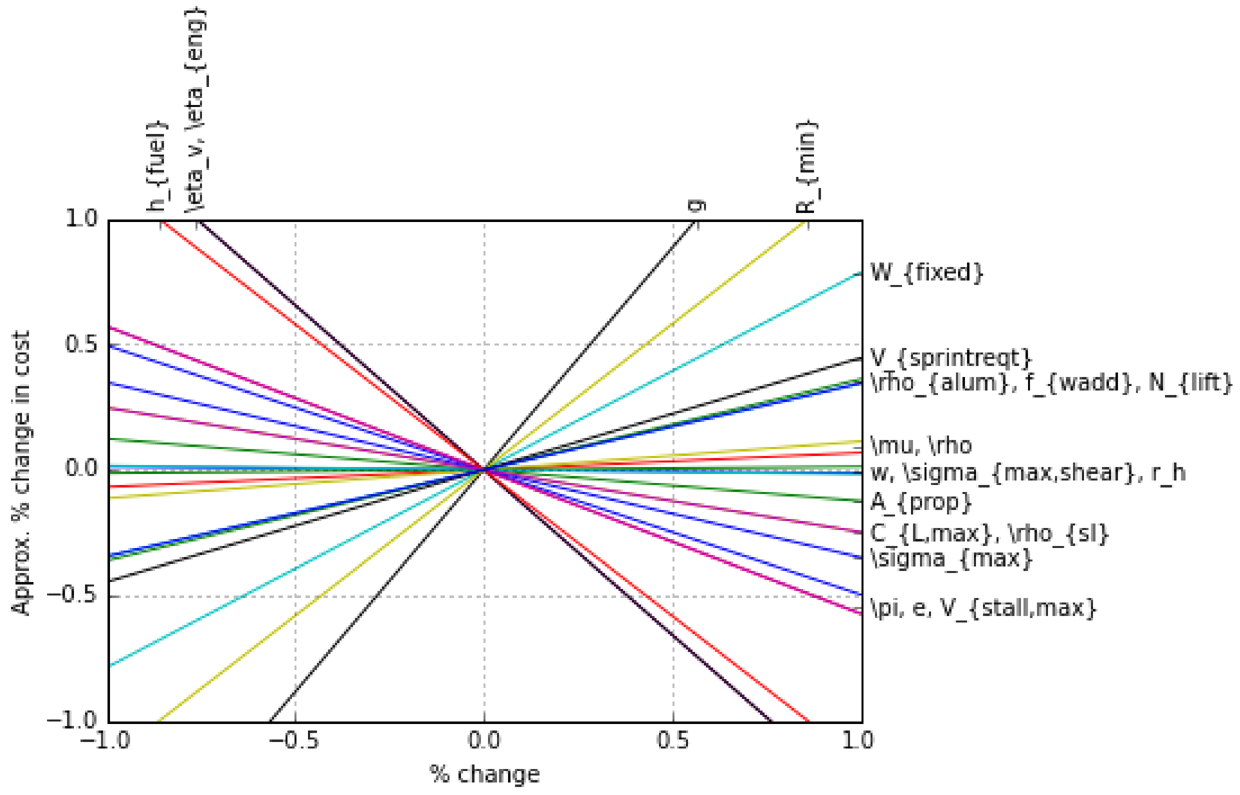
Visualization and Interaction

6.1 Plotting variable sensitivities

Sensitivities are a useful way to evaluate the tradeoffs in your model, as well as what aspects of the model are driving the solution and should be examined. To help with this, `gpkit.interactive` has an automatic sensitivity plotting function that can be accessed as follows:

```
from gpkit.interactive.plotting import sensitivity_plot
sensitivity_plot(m)
```

Which produces the following plot:



In this plot, steep lines that go up to the right are variables whose increase sharply increases (makes worse) the objective. Steep lines going down to the right are variables whose increase sharply decreases (improves) the objective. Only local sensitivities are displayed, so the lines are optimistic: the real effect of changing parameters may lead to higher costs than shown.

CHAPTER 7

Building Complex Models

7.1 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.

7.2 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")
            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:

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

class Test(Model):
    "A simple scalar model"
    def setup(self):
        x = Variable("x")
        return [x >= 1]

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

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).table()
```

```
SCALAR

Cost
----
 1

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

__________

VECTORIZED

Cost
----
 2

Free Variables
--------------
x : [ 1          2          1         ]
```

7.3 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.

```python
"""Modular aircraft concept"""
import numpy as np
from gpkit import Model, Variable, Vectorize

class Aircraft(Model):
    "The vehicle model"
    def setup(self):
        self.fuse = Fuselage()
        self.wing = Wing()
        self.components = [self.fuse, self.wing]

        W = Variable("W", "lbf", "weight")
        self.weight = W

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

    def dynamic(self, state):
        "This component's performance model for a given state."
        return AircraftP(self, state)


class AircraftP(Model):
    "Aircraft flight physics: weight <= lift, fuel burn"
    def setup(self, aircraft, state):
        self.aircraft = aircraft
        self.wing_aero = aircraft.wing.dynamic(state)
        self.perf_models = [self.wing_aero]
        Wfuel = Variable("W_{fuel}", "lbf", "fuel weight")
        Wburn = Variable("W_{burn}", "lbf", "segment fuel burn")

        return self.perf_models, [
            aircraft.weight + Wfuel <= (0.5*state["\\rho"]*state["V"]**2
                                        * self.wing_aero["C_L"]
                                        * aircraft.wing["S"]),
            Wburn >= 0.1*self.wing_aero["D"]
            ]


class FlightState(Model):
    "Context for evaluating flight physics"
    def setup(self):
        Variable("V", 40, "knots", "true airspeed")
        Variable("\\mu", 1.628e-5, "N*s/m^2", "dynamic viscosity")
        Variable("\\rho", 0.74, "kg/m^3", "air density")


class FlightSegment(Model):
    "Combines a context (flight state) and a component (the aircraft)"
    def setup(self, aircraft):
        self.flightstate = FlightState()
        self.aircraftp = aircraft.dynamic(self.flightstate)
        return self.flightstate, self.aircraftp
```

```
class Mission(Model):
    "A sequence of flight segments"
    def setup(self, aircraft):
        with Vectorize(4):  # four flight segments
            fs = FlightSegment(aircraft)

        Wburn = fs.aircraftp["W_{burn}"]
        Wfuel = fs.aircraftp["W_{fuel}"]
        self.takeoff_fuel = Wfuel[0]

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


class Wing(Model):
    "Aircraft wing model"
    def dynamic(self, state):
        "Returns this component's performance model for a given state."
        return WingAero(self, state)

    def setup(self):
        W = Variable("W", "lbf", "weight")
        S = Variable("S", 190, "ft^2", "surface area")
        rho = Variable("\\rho", 1, "lbf/ft^2", "areal density")
        A = Variable("A", 27, "-", "aspect ratio")
        c = Variable("c", "ft", "mean chord")

        return [W >= S*rho,
                c == (S/A)**0.5]


class WingAero(Model):
    "Wing aerodynamics"
    def setup(self, wing, state):
        CD = Variable("C_D", "-", "drag coefficient")
        CL = Variable("C_L", "-", "lift coefficient")
        e = Variable("e", 0.9, "-", "Oswald efficiency")
        Re = Variable("Re", "-", "Reynold's number")
        D = Variable("D", "lbf", "drag force")

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


class Fuselage(Model):
    "The thing that carries the fuel, engine, and payload"
    def setup(self):
        # fuselage needs an external dynamic drag model,
        # left as an exercise for the reader
        # V = Variable("V", 16, "gal", "volume")
        # d = Variable("d", 12, "in", "diameter")
        # S = Variable("S", "ft^2", "wetted area")
        # cd = Variable("c_d", .0047, "-", "drag coefficient")
        # CDA = Variable("CDA", "ft^2", "drag area")
        Variable("W", 100, "lbf", "weight")
```

```
AC = Aircraft()
MISSION = Mission(AC)
M = Model(MISSION.takeoff_fuel, [MISSION, AC])
SOL = M.solve(verbosity=0)
print SOL.table()
```

```
Cost
----
 1.943 [lbf]

Free Variables
--------------
         | Aircraft
       W : 290                                     [lbf] weight

         | Aircraft/Wing
       W : 190                                     [lbf] weight
       c : 2.653                                   [ft]  mean chord

         | Mission/FlightSegment/AircraftP
W_{burn} : [ 0.487     0.486     0.485     0.485    ]  [lbf] segment fuel burn
W_{fuel} : [ 1.94      1.46      0.97      0.485    ]  [lbf] fuel weight

         | Mission/FlightSegment/AircraftP/WingAero
     C_D : [ 0.00783   0.00782   0.00781   0.0078   ]        drag coefficient
     C_L : [ 0.47      0.469     0.468     0.467    ]        lift coefficient
       D : [ 4.87      4.86      4.85      4.85     ]  [lbf] drag force
      Re : [ 7.56e+05  7.56e+05  7.56e+05  7.56e+05 ]        Reynold's number

Constants
---------
     | Aircraft/Fuselage
   W : 100                                     [lbf]       weight

     | Aircraft/Wing
   A : 27                                                  aspect ratio
   S : 190                                     [ft**2]     surface area
\rho : 1                                       [lbf/ft**2] areal density

     | Mission/FlightSegment/AircraftP/WingAero
   e : [ 0.9       0.9       0.9       0.9      ]            Oswald efficiency

     | Mission/FlightSegment/FlightState
   V : [ 40        40        40        40       ]  [kt]        true airspeed
 \mu : [ 1.63e-05  1.63e-05  1.63e-05  1.63e-05 ]  [N*s/m**2]  dynamic viscosity
\rho : [ 0.74      0.74      0.74      0.74     ]  [kg/m**3]   air density

Sensitivities
-------------
     | Mission/FlightSegment/FlightState
   V : [ 0.0997    0.1       0.101     0.102    ] true airspeed
\rho : [ 0.034     0.0344    0.0347    0.0351   ] air density
 \mu : [ 0.0316    0.0317    0.0317    0.0318   ] dynamic viscosity

     | Mission/FlightSegment/AircraftP/WingAero
   e : [ -0.0926   -0.0924   -0.0922   -0.092   ] Oswald efficiency
```

```
        | Aircraft/Wing
    S : 0.6831                                   surface area
 \rho : 0.4816                                   areal density
    A : -0.3056                                  aspect ratio

        | Aircraft/Fuselage
    W : 0.2535                                   weight
```

CHAPTER 8

Advanced Commands

8.1 Substitutions

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

8.1.1 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.

8.1.2 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.

8.1.3 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).

8.1.4 Substituting with replacement

Any of the substitutions above can be run with `p.sub(*args, replace=True)` to clobber any previously-substituted values.

8.1.5 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`.

8.1.6 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 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.

8.2 Tight ConstraintSets

Tight ConstraintSets will warn if any inequalities they contain are not tight (that is, the right side equals the left side) after solving. This is useful when you know that a constraint _should_ be tight for a given model, but reprenting 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')
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
```

8.3 Sweeps

8.3.1 Declaring Sweeps

Sweeps are useful for analyzing tradeoff surfaces. A sweep "value" is an Iterable of numbers, e.g. `[1, 2, 3]`. Variables are swept when their substitution value takes the form `('sweep', Iterable), (e.g. '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]`. Sweeps can also be declared during later substitution (`gp.sub("x", ('sweep', [1, 2, 3]))`, or if the variable was already substituted for a constant, `gp.sub("x", ('sweep', [1, 2, 3]), replace=True))`.

8.3.2 Solving Sweeps

A Model with sweeps 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. Sweeping Vector Variables

Vector variables may also be substituted for: `y = VectorVariable(3, "y", value=('sweep' ,[[1, 2], [1, 2], [1, 2]])` will sweep $y \,\forall\, y_i \in \{1, 2\}$.

8.3.3 Parallel Sweeps

During a normal sweep, each result is independent, so they can be run in parallel. To use this feature, run `$ ipcluster start` at a terminal: it will automatically start a number of iPython parallel computing engines equal to the number of cores on your machine, and when you next import gpkit you should see a note like `Using parallel execution of sweeps on 4 clients`. If you do, then all sweeps performed with that import of gpkit will be parallelized.

This parallelization sets the stage for gpkit solves to be outsourced to a server, which may be valuable for faster results; alternately, it could allow the use of gpkit without installing a solver.

8.3.4 Linked Sweeps

Some constants may be "linked" to another sweep variable. This can be represented by a Variable whose value is `('sweep', fn)`, where the argument of the function `fn` is the dictionary of non-linked constants. If you declare a variables value to be a function, then it will assume you meant that as a sweep value: for example, `a_ = gpkit.Variable("a_", lambda c: 1-c[a.key], "-")` will create a constant whose value is always 1 minus the value of a (valid for values of a less than 1). Note that this declaration requires the variable `a` to already have been declared.

8.3.5 Example Usage

```
# code from t_GPSubs.test_VectorSweep in tests/t_sub.py
from gpkit import Variable, VectorVariable, Model

x = Variable("x")
y = VectorVariable(2, "y")
m = Model(x, [x >= y.prod()])
m.substitutions.update({y: ('sweep', [[2, 3], [5, 7, 11]])})
a = m.solve(printing=False)["cost"]
b = [10, 14, 22, 15, 21, 33]
assert all(abs(a-b)/(a+b) < 1e-7)
```

8.4 Composite Objectives

Given n posynomial objectives g_i, you can sweep out the problem's Pareto frontier with the composite objective:

$g_0 w_0 \prod_{i \neq 0} v_i + g_1 w_1 \prod_{i \neq 1} v_i + ... + g_n \prod_i v_i$

where $i \in 0...n-1$ and $v_i = 1 - w_i$ and $w_i \in [0, 1]$

GPkit has the helper function `composite_objective` for constructing these.

8.4.1 Example Usage

```
import numpy as np
import gpkit

L, W = gpkit.Variable("L"), gpkit.Variable("W")

eqns = [L >= 1, W >= 1, L*W == 10]
```

```
    co_sweep = [0] + np.logspace(-6, 0, 10).tolist()

    obj = gpkit.tools.composite_objective(L+W, W**-1 * L**-3,
                                          normsub={L:10, W: 10},
                                          sweep=co_sweep)

    m = gpkit.Model(obj, eqns)
    m.solve()
```

The `normsub` argument specifies an expected value for your solution to normalize the different g_i (you can also do this by hand). The feasibility of the problem should not depend on the normalization, but the spacing of the sweep will.

The `sweep` argument specifies what points between 0 and 1 you wish to sample the weights at. If you want different resolutions or spacings for different weights, the `sweeps` argument accepts a list of sweep arrays.

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.

The specification of a signomial problem can affect its solve time in a nuanced way: `gpkit.Model(x, [x >= 0.1, x+y >= 1, y <= 0.1]).localsolve()` takes about twice as long to solve with cvxopt as `gpkit.Model(x, [x >= 1-y, y <= 0.1]).localsolve()`, despite the two formulations being arithmetically equivalent and taking the same number of iterations.

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')

# 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)
sol = m.localsolve(verbosity=0)
print sol.table()
assert abs(sol(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)
sol = m.solve(verbosity=0)
print sol.table()
```

```
Cost
----
 0.7854

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

We can do better, however, by utilizing some built in functionality of GPkit. Assume we have some external code which is capable of evaluating our incompatible function:

```python
"""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 import ConstraintSet
from gpkit.exceptions import InvalidGPConstraint
from external_function import external_code


class ExternalConstraint(ConstraintSet):
    "Class for external calling"
    # Overloading the __init__ function here permits the constraint class to be
    # called more cleanly at the top level GP.
    def __init__(self, x, y, **kwargs):

        # Calls the ConstriantSet __init__ function
        super(ExternalConstraint, self).__init__([], **kwargs)

        # 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

    # Prevents the ExternalConstraint class from solving in a GP, thus forcing
    # iteration
    def as_posyslt1(self, substitutions=None):
        raise InvalidGPConstraint("ExternalConstraint cannot solve as a GP.")

    # Returns the ExternalConstraint class as a GP compatible constraint when
    # requested by the GPkit solver
    def as_gpconstr(self, x0):

        # 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:

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

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

```
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:  %s" % sol['cost'])
print("Optimal x val: %s" % sol(x))
```

Of course, the optimal value is 1. Output:

```
Optimal cost:  1.0
Optimal x val: 1.0
```

10.3 Maximizing the Volume of a Box

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

```
"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
sol = m.solve(verbosity=0)
print sol.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
---------
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
  -------------
     alpha : 0.5  lower limit, wall aspect ratio
  A_{wall} : -1.5 upper limit, wall area
```

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.table()
```

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

Constants
---------
   M : 100   [kg]      Mass of Water in the Tank
\rho : 1000  [kg/m**3] Density of Water in the Tank

Sensitivities
-------------
   M : 0.6667  Mass of Water in the Tank
\rho : -0.6667 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 numpy as np
from gpkit import Variable, Model


# 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")
pi = Variable("\\pi", np.pi, "-", "half of the circle constant")
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,
                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.table())

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.table())
```

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

Constants
---------
             (CDA0) : 0.031     [m**2]    fuselage drag area
(\frac{S}{S_{wet}}) : 2.05                wetted area ratio
          C_{L,max} : 1.5                 max CL with flaps down
            N_{ult} : 3.8                 ultimate load factor
            V_{min} : 22        [m/s]     takeoff speed
                W_0 : 4940      [N]       aircraft weight excluding wing
     W_{W_{coeff1}} : 8.71e-05  [1/m]     Wing Weight Coefficent 1
     W_{W_{coeff2}} : 45.24     [Pa]      Wing Weight Coefficent 2
                \mu : 1.78e-05  [kg/m/s]  viscosity of air
                \pi : 3.142               half of the circle constant
               \rho : 1.23      [kg/m**3] density of air
               \tau : 0.12                airfoil thickness to chord ratio
                  e : 0.95                Oswald efficiency factor
                  k : 1.2                 form factor

Sensitivities
-------------
                W_0 : 1.011   aircraft weight excluding wing
                  k : 0.4299  form factor
```

```
(\frac{S}{S_{wet}}) : 0.4299  wetted area ratio
     W_{W_{coeff1}} : 0.2903  Wing Weight Coefficent 1
            N_{ult} : 0.2903  ultimate load factor
     W_{W_{coeff2}} : 0.1303  Wing Weight Coefficent 2
             (CDA0) : 0.09156 fuselage drag area
                \mu : 0.08599 viscosity of air
          C_{L,max} : -0.1839 max CL with flaps down
               \rho : -0.2269 density of air
               \tau : -0.2903 airfoil thickness to chord ratio
            V_{min} : -0.3678 takeoff speed
                  e : -0.4785 Oswald efficiency factor
                \pi : -0.4785 half of the circle constant

SWEEP
=====

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

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

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

Constants
---------
             (CDA0) : 0.031     [m**2]    fuselage drag area
(\frac{S}{S_{wet}}) : 2.05                wetted area ratio
          C_{L,max} : 1.5                 max CL with flaps down
            N_{ult} : 3.8                 ultimate load factor
                W_0 : 4940      [N]       aircraft weight excluding wing
     W_{W_{coeff1}} : 8.71e-05  [1/m]     Wing Weight Coefficent 1
     W_{W_{coeff2}} : 45.24     [Pa]      Wing Weight Coefficent 2
                \mu : 1.78e-05  [kg/m/s]  viscosity of air
                \pi : 3.142               half of the circle constant
               \rho : 1.23      [kg/m**3] density of air
               \tau : 0.12                airfoil thickness to chord ratio
                  e : 0.95                Oswald efficiency factor
                  k : 1.2                 form factor

Sensitivities
-------------
                W_0 : [ 0.919     0.947     0.845     0.847    ] aircraft weight excluding wing
                  V : [ 0.589     0.249     0.975     0.746    ] cruising speed
                  k : [ 0.561     0.454     0.63      0.536    ] form factor
```

```
(\frac{S}{S_{wet}}) : [ 0.561     0.454     0.63      0.536    ] wetted area ratio
     W_{W_{coeff1}} : [ 0.179     0.247     0.108     0.155    ] Wing Weight Coefficent 1
            N_{ult} : [ 0.179     0.247     0.108     0.155    ] ultimate load factor
             (CDA0) : [ 0.114     0.131     0.146     0.177    ] fuselage drag area
     W_{W_{coeff2}} : [ 0.141     0.0911    0.126     0.0787   ] Wing Weight Coefficent 2
                \mu : [ 0.112     0.0907    0.126     0.107    ] viscosity of air
               \rho : [ -0.172    -0.129    -0.097    -0.0331  ] density of air
               \tau : [ -0.179    -0.247    -0.108    -0.155   ] airfoil thickness to chord rati
                  e : [ -0.325    -0.415    -0.225    -0.287   ] Oswald efficiency factor
                \pi : [ -0.325    -0.415    -0.225    -0.287   ] half of the circle constant
          C_{L,max} : [ -0.411    -0.207    -0.521    -0.353   ] max CL with flaps down
            V_{min} : [ -0.822    -0.415    -1.04     -0.705   ] takeoff speed
```

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.

```python
"""
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


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.
    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.
    """
    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}", 0, "N", "Tip loading")
        M = VectorVariable(N, "M", "N*m", "Internal moment")
        M_tip = Variable("M_{tip}", 0, "N*m", "Tip moment")
        th = VectorVariable(N, "\\theta", "-", "Slope")
        th_base = Variable("\\theta_{base}", 0, "-", "Base angle")
        w = VectorVariable(N, "w", "m", "Displacement")
        w_base = Variable("w_{base}", 0, "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 = 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)})
b.zero_lower_unbounded_variables()
sol = b.solve(verbosity=0)
print sol.table()
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)
    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
----
 1.62 [m]

Free Variables
--------------
    dx : 1.2                                         [m]   Length of an element
     M : [ 1.98e+03  1.27e+03  713       317      ... ]  [N*m] Internal moment
     V : [ 660       528       396       264      ... ]  [N]   Internal shear
\theta : [  -        0.177     0.285     0.341    ... ]        Slope
```

```
     w : [  -        0.106     0.384     0.759    ... ]  [m]   Displacement

Constants
---------
           EI : 1.1e+04                                       [N*m**2]
            L : 6                                             [m]      Overall beam length
      M_{tip} : 0                                             [N*m]    Tip moment
      V_{tip} : 0                                             [N]      Tip loading
\theta_{base} : 0                                                      Base angle
     w_{base} : 0                                             [m]      Base deflection
            M : [  -         -         -         -       ... ]  [N*m]    Internal moment
            V : [  -         -         -         -       ... ]  [N]      Internal shear
       \theta : [ 0         -         -         -       ... ]           Slope
            q : [ 110       110       110       110      ... ]  [N/m]    Distributed load at eac
            w : [ 0         -         -         -       ... ]  [m]      Displacement

Sensitivities
-------------
 L : 4                                              Overall beam length
 q : [ 0.0072    0.0416    0.118     0.234    ... ] Distributed load at each point
EI : -1
```

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 Subpackages

11.1.1 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.

gpkit.constraints.bounded module

Implements Bounded

class `gpkit.constraints.bounded.`**`Bounded`**(*constraints*, *substitutions=None*, *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

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

`sens_from_dual`(*las*, *nus*)
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

`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`(*init_dict=None*)
Resets varkeys to what is in the cost and constraints

`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

subinplace(*subs*)
Substitutes in place.

gpkit.constraints.linked module

Implements LinkedConstraintSet

class gpkit.constraints.linked.**LinkedConstraintSet**(*constraints*, *include_only=None*, *exclude=None*)

Bases: *gpkit.constraints.set.ConstraintSet*

A ConstraintSet that links duplicate variables in its constraints

VarKeys with the same *.str_without([”models”])* are linked.

The new linking varkey will have the same attributes as the first linked varkey of that name, without any value, models, or modelnums.

If any of the constraints have a substitution for a linked varkey, the linking varkey will have that substitution as well; if more than one linked varkey has a substitution a ValueError will be raised.

constraints: iterable valid argument to ConstraintSet

include_only: set whitelist of variable names to include

exclude: set blacklist of variable names, supercedes include_only

process_result(*result*)

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

debug(*verbosity=1*, ***solveargs*)
Attempts to diagnose infeasible models.

gp(*verbosity=1*, *substitutions=None*, ***kwargs*)
Return program version of self

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

return_attr: string attribute to return in addition to the program

`link`(*other*, *include_only=None*, *exclude=None*)
Connects this model with a set of constraints

`localsolve`(*solver=None*, *verbosity=1*, *skipsweepfailures=False*, **args*, ***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.

*args, **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*, **args*, ***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.

*args, **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`(*verbosity=1*, *substitutions=None*, ***kwargs*)
Return program version of self

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

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

`zero_lower_unbounded_variables`()
Recursively substitutes 0 for variables that lack a lower bound

gpkit.constraints.prog_factories module

Scripts for generating, solving and sweeping programs

`gpkit.constraints.prog_factories.`**`run_sweep`**(*genfunction*, *self*, *solution*, *skipsweepfailures*, *constants*, *sweep*, *linkedsweep*, *solver*, *verbosity*, **args*, ***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.

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

`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

`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`(*init_dict=None*)

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*)

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.

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

varkeys = None

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.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.signomial_program module

Implement the SignomialProgram class

class gpkit.constraints.signomial_program.**SignomialProgram** (*cost*, *constraints*, *substitutions=None*, *verbosity=1*)
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: SignomialPrograms 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.SignomialProgram(
                    # minimize
                    x,
                    [   # subject to
                        1/x - y/x,  # <= 1, implicitly
                        y/10  # <= 1
                    ])
>>> gp.solve()
```

gp (*x0=None*, *verbosity=1*)
The GP approximation of this SP at x0.

localsolve(*solver=None*, *verbosity=1*, *x0=None*, *rel_tol=0.0001*, *iteration_limit=50*, ***kwargs*)

Locally solves a SignomialProgram 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.

rel_tol [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.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>, '>=': <built-in function ge>}

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

sub(*subs*)

Returns a substituted version of this constraint.

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*, *substitutions=None*, *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**

class `gpkit.constraints.tight.`**`TightConstraintSet`**(**args*, ***kwargs*)

Bases: *`gpkit.constraints.tight.Tight`*

kept for backwards compatibility, discard in 0.6

Module contents

Contains ConstraintSet and related classes and objects

11.1.2 gpkit.interactive package

Submodules

gpkit.interactive.chartjs module

gpkit.interactive.linking_diagram module

gpkit.interactive.plotting module

gpkit.interactive.ractor module

gpkit.interactive.sensitivity_map module

gpkit.interactive.widgets module

Module contents

11.1.3 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])
```

c
: The coefficient vector in the GP input data sense

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.data module

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

class `gpkit.nomials.data.`**`NomialData`**(*exps=None*, *cs=None*, *simplify=True*)
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

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

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

NomialData

`init_from_nomials`(*nomials*)
Way to initialize from nomials. Calls __init__. Used by subclass __init__ methods.

`values`
The NomialData's values, created when necessary.

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

`gpkit.nomials.data.`**`simplify_exps_and_cs`**(*exps*, *cs*, *return_map=False*)
Reduces the number of monomials, and casts them to a sorted form.

exps [list of Hashvectors] The exponents of each monomial

cs [array of floats or Quantities] The coefficients of each monomial

return_map [bool (optional)] Whether to return the map of which monomials combined to form a simpler monomial, and their fractions of that monomial's final c.

exps [list of Hashvectors] Exponents of simplified monomials.

cs [array of floats or Quantities] Coefficients of simplified monomials.

mmap [list of HashVectors] List for each new monomial of {originating indexes: fractions}

gpkit.nomials.nomial_core module

The shared non-mathematical backbone of all Nomials

class `gpkit.nomials.nomial_core.`**`Nomial`**(*exps=None*, *cs=None*, *simplify=True*)
Bases: *`gpkit.nomials.data.NomialData`*

Shared non-mathematical properties of all nomials

`c` **= None**

`convert_to`(*arg*)
Convert this signomial to new units

`latex`(*excluded=None*)
For pretty printing with Sympy

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

`sub` **= None**

to(*arg*)
Create new Signomial converted to new 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.nomial_math module

Signomial, Posynomial, Monomial, Constraint, & MonoEQCOnstraint classes

class `gpkit.nomials.nomial_math.`**Monomial**(*exps=None*, *cs=1*, *require_positive=True*, *simplify=True*, ***descr*)
Bases: *gpkit.nomials.nomial_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.

mono_approximation(*x0*)

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

A Constraint of the form Monomial == Monomial.

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

class `gpkit.nomials.nomial_math.`**Posynomial**(*exps=None*, *cs=1*, *require_positive=True*, *simplify=True*, ***descr*)
Bases: *gpkit.nomials.nomial_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.nomial_math.`**PosynomialInequality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.nomial_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*)
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*)
Returns the variable/constraint sensitivities from lambda/nu

class gpkit.nomials.nomial_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.nomial_math.**Signomial**(*exps=None*, *cs=1*, *require_positive=True*, *simplify=True*, ***descr*)
Bases: *gpkit.nomials.nomial_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(*wrt*)
Derivative of this with respect to a Variable

wrt (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.nomial_math.**SignomialInequality**(*left*, *oper*, *right*)
Bases: *gpkit.nomials.nomial_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_gpconstr(*x0*)
Returns GP approximation of an SP constraint at x0

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

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

A constraint of the general form posynomial == posynomial

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

Module containing the substitution function

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

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

gpkit.nomials.substitution.**substitution**(*nomial*, *substitutions*)
Efficient substituton into a list of monomials.

varlocs [dict] Dictionary mapping variables to lists of monomial indices.

exps [Iterable of dicts] Dictionary mapping variables to exponents, for each monomial.

cs [list] Coefficient for each monomial.

substitutions [dict] Substitutions to apply to the above.

val [number (optional)] Used to substitute singlet variables.

varlocs_ [dict] Dictionary of monomial indexes for each variable.

exps_ [dict] Dictionary of variable exponents for each monomial.

cs_ [list] Coefficients each monomial.

subs_ [dict] Substitutions to apply to the above.

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 + Quantity)

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.nomial_math.Monomial`*

A described singlet Monomial.

***args** [list]

may contain "name" (Strings)

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

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.

`descr`

a Variable's descr is derived from its VarKey.

`key`

Get the VarKey associated with this Variable

`sub``(`**args*, ***kwargs*`)`

Same as nomial substitution, but also allows single-argument calls

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

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.

Module contents

Contains nomials, inequalities, and arrays

11.1.4 gpkit.tests package

Submodules

gpkit.tests.helpers module

Convenience classes and functions for unit testing

class `gpkit.tests.helpers.`**`NewDefaultSolver`**(*solver*)

Bases: `object`

Creates an environment with a different default solver

class `gpkit.tests.helpers.`**`NullFile`**

Bases: `object`

A fake file interface that does nothing

`close`()

Having not written, cease.

`write`(*string*)

Do not write, do not pass go.

class `gpkit.tests.helpers.`**`StdoutCaptured`**(*logfilepath=None*)

Bases: `object`

Puts everything that would have printed to stdout in a log file instead

`gpkit.tests.helpers.`**`generate_example_tests`**(*path*, *testclasses*, *solvers=None*, *newtest_fn=None*)

Mutate TestCase class so it behaves as described in TestExamples docstring

path [str] directory containing example modules to test

testclass [class] class that inherits from *unittest.TestCase*

newtest_fn [function] function that returns new tests. defaults to import_test_and_log_output

solvers [iterable] solvers to run for; or only for default if solvers is None

`gpkit.tests.helpers.`**`logged_example_testcase`**(*name*, *imported*, *path*)

Returns a method for attaching to a unittest.TestCase that imports or reloads module 'name' and stores in imported[name]. Runs top-level code, which is typically a docs example, in the process.

Returns a method.

`gpkit.tests.helpers.`**`new_test`**(*name*, *solver*, *import_dict*, *path*, *testfn=None*)

logged_example_testcase with a NewDefaultSolver

`gpkit.tests.helpers.`**`run_tests`**(*tests*, *xmloutput=None*, *verbosity=2*)

Default way to run tests, to be used in __main__.

tests: iterable of unittest.TestCase xmloutput: string or None

if not None, generate xml output for continuous integration, with name given by the input string

verbosity: int verbosity level for unittest.TextTestRunner

gpkit.tests.run_tests module

Script for running all gpkit unit tests

`gpkit.tests.run_tests.`**`import_tests`**`()`

Get a list of all GPkit unit test TestCases

`gpkit.tests.run_tests.`**`run`**`(`*xmloutput=False*, *tests=None*, *unitless=True*`)`

Run all gpkit unit tests.

xmloutput: bool If true, generate xml output files for continuous integration

gpkit.tests.t_constraints module

Unit tests for Constraint, MonomialEquality and SignomialInequality

class `gpkit.tests.t_constraints.`**`TestBounded`**`(`*methodName='runTest'*`)`

Bases: `unittest.case.TestCase`

Test bounded constraint set

`test_substitution_issue905``()`

class `gpkit.tests.t_constraints.`**`TestConstraint`**`(`*methodName='runTest'*`)`

Bases: `unittest.case.TestCase`

Tests for Constraint class

`test_additive_scalar``()`

Make sure additive scalars simplify properly

`test_additive_scalar_gt1``()`

1 can’t be greater than (1 + something positive)

`test_bad_elements``()`

`test_constraintget``()`

`test_equality_relaxation``()`

`test_exclude_vector``()`

`test_init``()`

Test Constraint __init__

`test_link_conflict``()`

Check that substitution conflicts are flagged during linking.

`test_oper_overload``()`

Test Constraint initialization by operator overloading

class `gpkit.tests.t_constraints.`**`TestMonomialEquality`**`(`*methodName='runTest'*`)`

Bases: `unittest.case.TestCase`

Test monomial equality constraint class

`test_inheritance``()`

Make sure MonomialEquality inherits from the right things

`test_init``()`

Test initialization via both operator overloading and __init__

`test_non_monomial``()`

Try to initialize a MonomialEquality with non-monomial args

test_str()
Test that MonomialEquality.__str__ returns a string

class gpkit.tests.t_constraints.**TestSignomialInequality**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test Signomial constraints

test_init()
Test initialization and types

test_posyslt1()

class gpkit.tests.t_constraints.**TestTight**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test tight constraint set

test_posyconstr_in_gp()
Tests tight constraint set with solve()

test_posyconstr_in_sp()

test_sigconstr_in_sp()
Tests tight constraint set with localsolve()

gpkit.tests.t_examples module

Unit testing of tests in docs/source/examples

class gpkit.tests.t_examples.**TestExamples**(*methodName='runTest'*)
Bases: unittest.case.TestCase

To test a new example, add a function called *test_$EXAMPLENAME*, where $EXAMPLENAME is the name of your example in docs/source/examples without the file extension.

This function should accept two arguments (e.g. 'self' and 'example'). The imported example script will be passed to the second: anything that was a global variable (e.g, "sol") in the original script is available as an attribute (e.g., "example.sol")

If you don't want to perform any checks on the example besides making sure it runs, just put "pass" as the function's body, e.g.:

def test_dummy_example(self, example): pass

But it's good practice to ensure the example's solution as well, e.g.:

def test_dummy_example(self, example): self.assertAlmostEqual(example.sol["cost"], 3.121)

test_beam(*example*)

test_debug(*example*)

test_external_sp(*example*)

test_performance_modeling(*example*)

test_primal_infeasible_ex1(*example*)

test_primal_infeasible_ex2(*example*)

test_relaxation(*example*)

test_simple_box(*example*)

test_simple_sp(*example*)

test_simpleflight(*example*)

test_sin_approx_example(*example*)

test_unbounded(*example*)

test_vectorize(*example*)

test_water_tank(*example*)

test_x_greaterthan_1(*example*)

gpkit.tests.t_keydict module

Test KeyDict class

class gpkit.tests.t_keydict.**TestKeyDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the KeyDict class

test_dictlike()

test_failed_getattr()

test_getattr()

test_setattr()

test_vector()

gpkit.tests.t_model module

Tests for GP and SP classes

class gpkit.tests.t_model.**Box**(*cost=None*, *constraints=None*, **args*, ***kwargs*)
Bases: *gpkit.constraints.model.Model*

simple box for model testing

setup()

class gpkit.tests.t_model.**BoxAreaBounds**(*cost=None*, *constraints=None*, **args*, ***kwargs*)
Bases: *gpkit.constraints.model.Model*

for testing functionality of separate analysis models

setup(*box*)

class gpkit.tests.t_model.**TestGP**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test GeometricPrograms. This TestCase gets run once for each installed solver.

name = 'TestGP_'

ndig = None

solver = None

test_601()

test_additive_constants()

test_constants_in_objective_1()
Issue 296

test_constants_in_objective_2()
Issue 296

test_cost_freeing()
Test freeing a variable that's in the cost.

test_exps_is_tuple()
issue 407

test_mdd_example()

test_posy_simplification()
issue 525

test_sigeq()

test_simple_united_gp()

test_singular()
Create and solve GP with a singular A matrix

test_terminating_constant_()

test_trivial_gp()

Create and solve a trivial GP: minimize x + 2y subject to xy >= 1

The global optimum is (x, y) = (sqrt(2), 1/sqrt(2)).

test_trivial_vector_gp()
Create and solve a trivial GP with VectorVariables

test_zero_lower_unbounded()

test_zeroing()

class gpkit.tests.t_model.**TestModelNoSolve**(*methodName='runTest'*)
Bases: unittest.case.TestCase

model tests that don't require a solver

test_modelname_added()

test_no_naming_on_var_access()

class gpkit.tests.t_model.**TestModelSolverSpecific**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test cases run only for specific solvers

test_cvxopt_kwargs()

class gpkit.tests.t_model.**TestSP**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test case for SP class – gets run for each installed solver

name = 'TestSP_'

ndig = None

solver = None

test_initially_infeasible()

test_issue180()

test_partial_sub_signomial()
Test SP partial x0 initialization

test_relaxation()

test_sigs_not_allowed_in_cost()

test_small_named_signomial()

test_sp_initial_guess_sub()

test_sp_substitutions()

test_trivial_sp()

test_trivial_sp2()

test_unbounded_debugging()
Test nearly-dual-feasible problems

test_values_vs_subs()

class gpkit.tests.t_model.**Thing**(*cost=None*, *constraints=None*, **args*, ***kwargs*)
Bases: gpkit.constraints.model.Model

a thing, for model testing

setup(*length*)

gpkit.tests.t_model.**test**
alias of TestSP_cvxopt

gpkit.tests.t_model.**testcase**
alias of TestSP

gpkit.tests.t_nomial_array module

Tests for NomialArray class

class gpkit.tests.t_nomial_array.**TestNomialArray**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the NomialArray class. Also tests VectorVariable, since VectorVariable returns a NomialArray

test_array_mult()

test_constraint_gen()

test_elementwise_mult()

test_empty()

test_getitem()

test_left_right()

test_ndim()

test_outer()

test_prod()

test_shape()

test_substition()

test_sum()

test_units()

gpkit.tests.t_nomials module

Tests for Monomial, Posynomial, and Signomial classes

class gpkit.tests.t_nomials.**TestMonomial**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the Monomial class

test_div()
Test Monomial division

test_eq_ne()
Test equality and inequality comparators

test_init()
Test multiple ways to create a Monomial

test_latex()
Test latex string creation

test_mul()
Test monomial multiplication

test_numerical_precision()
not sure what to test here, placeholder for now

test_pow()
Test Monomial exponentiation

test_repr()
Simple tests for __repr__, which prints more than str

test_str_with_units()
Make sure __str__() works when units are involved

test_units()
make sure multiplication with units works (issue 492)

class gpkit.tests.t_nomials.**TestPosynomial**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the Posynomial class

test_constraint_gen()
Test creation of Constraints via operator overloading

test_diff()
Test differentiation (!!)

test_eq()
Test Posynomial __eq__

test_eq_units()

test_init()
Test Posynomial construction

test_integer_division()
Make sure division by integer doesn't use Python integer division

test_mono_lower_bound()
Test monomial approximation

test_posyposy_mult()
Test multiplication of Posynomial with Posynomial

test_simplification()
Make sure like monomial terms get automatically combined

class `gpkit.tests.t_nomials.`**TestSignomial**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

TestCase for the Signomial class

test_eq_ne()
Test Signomial equality and inequality operators

test_init()
Test Signomial construction

gpkit.tests.t_small module

Tests for small_classes.py and small_scripts.py

class `gpkit.tests.t_small.`**TestHashVector**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

TestCase for the HashVector class

test_init()
Make sure HashVector acts like a dict

test_mul_add()
Test multiplication and addition

test_neg()
Test negation

test_pow()
Test exponentiation

class `gpkit.tests.t_small.`**TestSmallScripts**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

TestCase for gpkit.small_scripts

test_pint_366()

test_unitstr()

gpkit.tests.t_solution_array module

Tests for SolutionArray class

class `gpkit.tests.t_solution_array.`**TestResultsTable**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

TestCase for results_table()

test_nan_printing()
Test that solution prints when it contains nans

test_result_access()

class gpkit.tests.t_solution_array.**TestSolutionArray**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for the SolutionArray class

test_call()

test_call_time()

test_call_units()

test_call_vector()

test_subinto()

test_table()

test_units_sub()

gpkit.tests.t_sub module

Test substitution capability across gpkit

class gpkit.tests.t_sub.**TestGPSubs**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test substitution for Model and GP objects

test_getkey()

test_linked_sweep()

test_model_composition_units()

test_model_recursion()

test_persistence()

test_phantoms()

test_skipfailures()

test_united_sub_sweep()

test_vector_init()

test_vector_sweep()
Test sweep involving VectorVariables

class gpkit.tests.t_sub.**TestNomialSubs**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Test substitution for nomial-family objects

test_basic()
Basic substitution, symbolic

test_dimensionless_units()

test_numeric()
Basic substitution of numeric value

test_quantity_sub()

test_scalar_units()

test_signomial()
Test Signomial substitution

test_string_mutation()

test_unitless_monomial_sub()
Tests that dimensionless and undimensioned subs can interact.

test_variable()
Test special single-argument substitution for Variable

test_vector()

gpkit.tests.t_tools module

Tests for tools module

class gpkit.tests.t_tools.**TestMathModels**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for math models

test_composite_objective()

test_fmincon_generator()
Test fmincon comparison tool

test_te_exp_minus1()
Test Taylor expansion of e^x - 1

test_te_secant()
Test Taylor expansion of secant(var)

test_te_tangent()
Test Taylor expansion of tangent(var)

gpkit.tests.t_vars module

Test VarKey, Variable, VectorVariable, and ArrayVariable classes

class gpkit.tests.t_vars.**TestArrayVariable**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the ArrayVariable class

test_is_vector_variable()
Make sure ArrayVariable is a shortcut to VectorVariable (we want to know if this changes).

test_str()
Make sure string looks something like a numpy array

class gpkit.tests.t_vars.**TestVarKey**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the VarKey class

test_dict_key()
make sure variables are well-behaved dict keys

test_eq_neq()
Test boolean equality operators

test_init()
Test VarKey initialization

test_repr()
Test __repr__ method

test_units_attr()
Make sure VarKey objects have a units attribute

class gpkit.tests.t_vars.**TestVariable**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the Variable class

test_hash()
Hashes should collide independent of units

test_init()
Test Variable initialization

test_unit_parsing()

test_value()
Detailed tests for value kwarg of __init__

class gpkit.tests.t_vars.**TestVectorVariable**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for the VectorVariable class. Note: more relevant tests in t_posy_array.

test_constraint_creation_units()

test_init()
Test VectorVariable initialization

class gpkit.tests.t_vars.**TestVectorize**(*methodName='runTest'*)
Bases: unittest.case.TestCase

TestCase for gpkit.vectorize

test_shapes()

Module contents

GPkit testing module

11.1.5 gpkit.tools package

Submodules

gpkit.tools.fmincon module

A module to facilitate testing GPkit against fmincon

gpkit.tools.fmincon.**generate_mfiles**(*model*, *algorithm='interior-point'*, *guesstype='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

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

guesstype: [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

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*, *guesstype='ones'*)
Returns initial guess

gpkit.tools.tools module

Non-application-specific convenience methods for GPkit

`gpkit.tools.tools.`**`composite_objective`**(**objectives*, ***kwargs*)
Creates a cost function that sweeps between multiple objectives.

`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

11.2 Submodules

11.3 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.

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

`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]->numhesnz);': 'MSK_echotask(task**

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_gpkit`**`()`
: 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.`**`rebuild`**`()`
: Changes to the installed gpkit directory and runs build_gpkit()

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

11.4 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

11.5 gpkit.geometric_program module

11.6 gpkit.keydict module

Implements KeyDict and KeySet classes

class `gpkit.keydict.`**`FastKeyDict`**`(`**args*, ***kwargs*`)`

Bases: *`gpkit.keydict.KeyDict`*

KeyDicts that don't map keys, only collapse arrays

`keymapping` **= False**

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**

`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

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

11.7 gpkit.modified_ctypesgen module

11.8 gpkit.repr_conventions module

Repository for representation standards

11.9 gpkit.small_classes module

Miscellaneous small classes

class `gpkit.small_classes.`**`CootMatrix`**
Bases: *`gpkit.small_classes.CootMatrix`*

A very simple sparse matrix representation.

append (*row*, *col*, *data*)
Appends entry to matrix.

dot (*arg*)
Returns dot product with arg.

shape = None

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.

gpkit.small_classes.**CootMatrixTuple**
alias of *CootMatrix*

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.

classify (*cls*)
Converts dictionaries whose first key isn't a string to given class.

to_united_array (*unitless_keys=()*, *united=False*)
Converts all lists into array, potentially grabbing units from keys.

class gpkit.small_classes.**HashVector** (**args*, ***kwargs*)
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})
```

class gpkit.small_classes.**SolverLog** (*verbosity=0*, *output=None*, **args*, ***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.

11.10 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.**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)

gpkit.small_scripts.**unitstr** (*units, into='%s', options='~', dimless='-'*)
Returns the unitstr of a given object.

gpkit.small_scripts.**veckeyed** (*key*)
Return a veckey version of a VarKey

11.11 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)
```

program = None

subinto(*posy*)
Returns NomialArray of each solution substituted into posy.

table(*tables=('cost', 'sweepvariables', 'freevariables', 'constants', 'sensitivities')*, *latex=False*, ***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 = {'freevariables': 'Free Variables', 'variables': 'Variables', 'cost': 'Cost', 'sweepvariables': 'Swe

gpkit.solution_array.**results_table**(*data*, *title*, *minval=0*, *printunits=True*, *fixedcols=True*, *varfmt='%s : '*, *valfmt='%-.4g '*, *vecfmt='%-8.3g'*, *included_models=None*, *excluded_models=None*, *latex=False*, *sortbyvals=False*)
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.

11.12 gpkit.varkey module

Defines the VarKey class

class `gpkit.varkey.`**`Count`**

Bases: `object`

Like python 2's itertools.count, for Python 3 compatibility.

`next``()`

Increment self.count and return it

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.

`eq_ignores` **= frozenset(['units', 'value'])**

`latex``(`*excluded=None*`)`

Returns latex representation.

`latex_unitstr``()`

Returns latex unitstr

`new_unnamed_id``()`

Increment self.count and return it

`str_without``(`*excluded=None*`)`

Returns string without certain fields (such as 'models').

`subscripts` **= ('models', 'idx')**

11.13 Module contents

GP and SP Modeling Package

For examples please see the examples folder.

11.13.1 Requirements

numpy MOSEK or CVXOPT scipy(optional): for complete sparse matrix support sympy(optional): for latex printing in iPython Notebook

11.13.2 Attributes

settings [dict] Contains settings loaded from `./env/settings`

class `gpkit.`**`GPkitUnits`**

Bases: `object`

Return monomials instead of Quantitites

class `gpkit.`**`NamedVariables`**(*model*)

Bases: `object`

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

class `gpkit.`**`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.`**`Vectorize`**(*dimension_length*)

Bases: `object`

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

`gpkit.`**`begin_variable_naming`**(*model*)

Appends a model name and num to the environment.

`gpkit.`**`disable_units`**()

Disables units support in a particular instance of GPkit.

Posynomials created after calling this are incompatible with those created before.

If gpkit is imported multiple times, this needs to be run each time.

The correct way to call this is: import gpkit gpkit.disable_units()

The following will *not* have the intended effect: from gpkit import disable_units disable_units()

`gpkit.`**`enable_units`**(*path=None*)

Enables units support in a particular instance of GPkit.

Posynomials created after calling this are incompatible with those created before.

If gpkit is imported multiple times, this needs to be run each time.

`gpkit.`**`end_variable_naming`**()

Pops a model name and num from the environment.

`gpkit.`**`load_settings`**(*path='/home/docs/checkouts/readthedocs.org/user_builds/gpkit/envs/v0.5.1/local/lib/python2.7/site packages/gpkit/env/settings'*)

Load the settings file at SETTINGS_PATH; return settings dict

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/hoburg/gpkit}},
      year={2016},
      note={Version 0.5.1}
     }
```

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 0.5.1

- 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*)

14.2 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*)

14.2.1 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.

14.3 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

14.4 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.

14.5 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

14.6 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.

14.7 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.

14.8 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