

---

# **gpkit Documentation**

*Release 0.5.3*

**MIT Department of Aeronautics and Astronautics**

**Jul 13, 2017**



<b>1</b>	<b>Geometric Programming 101</b>	<b>3</b>
1.1	What is a GP? . . . . .	3
1.2	Why are GPs special? . . . . .	4
1.3	What are Signomials / Signomial Programs? . . . . .	4
1.4	Where can I learn more? . . . . .	4
<b>2</b>	<b>GPkit Overview</b>	<b>5</b>
2.1	Symbolic expressions . . . . .	5
2.2	Substitution . . . . .	5
2.3	Model objects . . . . .	7
<b>3</b>	<b>Installation Instructions</b>	<b>9</b>
3.1	Installation dependencies . . . . .	9
3.2	Install a GP solver . . . . .	10
3.3	Install GPkit . . . . .	11
3.4	Debugging installation . . . . .	11
3.5	Bleeding-edge / developer installations . . . . .	12
<b>4</b>	<b>Getting Started</b>	<b>13</b>
4.1	Declaring Variables . . . . .	13
4.2	Creating Monomials and Posynomials . . . . .	14
4.3	Declaring Constraints . . . . .	14
4.4	Formulating a Model . . . . .	15
4.5	Solving the Model . . . . .	15
4.6	Printing Results . . . . .	15
4.7	Sensitivities and dual variables . . . . .	16
<b>5</b>	<b>Debugging Models</b>	<b>17</b>
5.1	Potential errors and warnings . . . . .	18
5.2	Dual Infeasibility . . . . .	18
5.3	Primal Infeasibility . . . . .	19
<b>6</b>	<b>Visualization and Interaction</b>	<b>25</b>
6.1	Interactive Control Panel . . . . .	25
6.2	Plotting a 1D Sweep . . . . .	25
<b>7</b>	<b>Building Complex Models</b>	<b>29</b>

7.1	Inheriting from <code>Model</code> . . . . .	29
7.2	Accessing Variables in Models . . . . .	29
7.3	Vectorization . . . . .	30
7.4	Multipoint analysis modeling . . . . .	31
<b>8</b>	<b>Advanced Commands</b>	<b>37</b>
8.1	Derived Variables . . . . .	37
8.2	Sweeps . . . . .	38
8.3	Tight ConstraintSets . . . . .	39
8.4	Substitutions . . . . .	40
8.5	Composite Objectives . . . . .	42
<b>9</b>	<b>Signomial Programming</b>	<b>43</b>
9.1	Example Usage . . . . .	43
9.2	Sequential Geometric Programs . . . . .	44
<b>10</b>	<b>Examples</b>	<b>49</b>
10.1	iPython Notebook Examples . . . . .	49
10.2	A Trivial GP . . . . .	49
10.3	Maximizing the Volume of a Box . . . . .	50
10.4	Water Tank . . . . .	51
10.5	Simple Wing . . . . .	52
10.6	Simple Beam . . . . .	54
<b>11</b>	<b>Glossary</b>	<b>57</b>
11.1	Subpackages . . . . .	57
11.2	Submodules . . . . .	90
11.3	<code>gpkit.build</code> module . . . . .	90
11.4	<code>gpkit.exceptions</code> module . . . . .	91
11.5	<code>gpkit.keydict</code> module . . . . .	91
11.6	<code>gpkit.modified_ctypesgen</code> module . . . . .	93
11.7	<code>gpkit.repr_conventions</code> module . . . . .	93
11.8	<code>gpkit.small_classes</code> module . . . . .	93
11.9	<code>gpkit.small_scripts</code> module . . . . .	94
11.10	<code>gpkit.solution_array</code> module . . . . .	94
11.11	<code>gpkit.varkey</code> module . . . . .	96
11.12	Module contents . . . . .	97
<b>12</b>	<b>Citing GPKit</b>	<b>99</b>
<b>13</b>	<b>Acknowledgements</b>	<b>101</b>
<b>14</b>	<b>Release Notes</b>	<b>103</b>
14.1	Version 0.5.3 . . . . .	103
14.2	Version 0.5.2 . . . . .	103
14.3	Version 0.5.1 . . . . .	104
14.4	Version 0.5.0 . . . . .	104
14.5	Version 0.4.2 . . . . .	105
14.6	Version 0.4.0 . . . . .	106
14.7	Version 0.3.4 . . . . .	106
14.8	Version 0.3.2 . . . . .	106
14.9	Version 0.3 . . . . .	106
14.10	Version 0.2 . . . . .	107
	<b>Python Module Index</b>	<b>109</b>



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.



## 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:

$$f(x) = cx_1^{a_1}x_2^{a_2}\dots x_n^{a_n}$$

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

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

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

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

For example, the expressions

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

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:

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

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

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

## 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.

## 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*.

## 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



---

## 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.

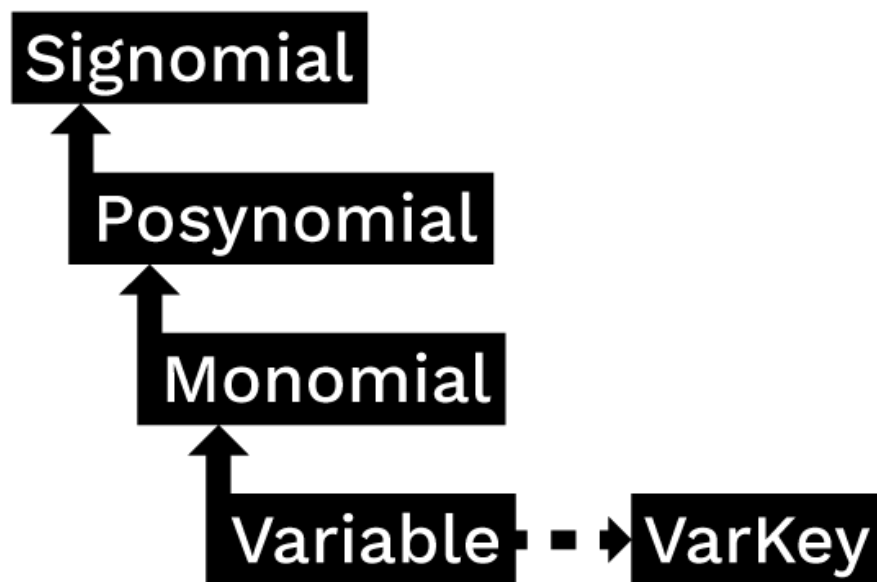
### Symbolic expressions

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.

### Substitution

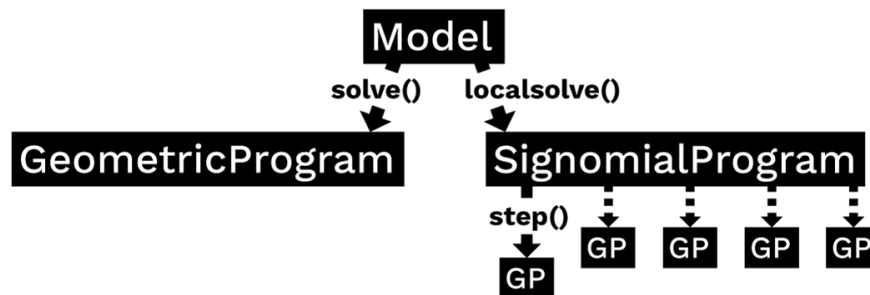
The `Varkey` object in the graph above is not a algebraic expression, but what GPkit uses as a variable's "name". It carries the LaTeX representation of a variable and its units, as well as any other information the user wishes to associate with a variable. The use of `VarKeys` as opposed to numeric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



## 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.





---

## Installation Instructions

---

If you encounter bugs during installation, please email [gpkit@mit.edu](mailto:gpkit@mit.edu) or raise a [GitHub issue](#).

### 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:

#### 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](#).

#### 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.

## 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.

### 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.

### 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).

## 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](mailto:gpkit@mit.edu) or raise a [GitHub issue](#).
- Join our [mailing list](#) and/or [chatroom](#) for support and examples.

## 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](mailto:gpkit@mit.edu) or raise a [GitHub issue](#).

## 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
```

## 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.

### 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")
```

### 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)
```

## 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")
```

## 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
```

## 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
```

## 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)
```

## 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)
```

## 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.
```

## 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.

### 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
```

---

## 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)
```

```
> Trying to solve with bounded variables and relaxed constants

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

and these constants relaxed:
  x_min [ft]: relaxed from 2 to 1
> ...success!

> Trying to solve with relaxed constraints
> ...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.

## Potential errors and warnings

- **RuntimeWarning: final status of solver 'mosek' was 'DUAL\_INFEAS\_CER', not 'optimal'**
  - 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 'optimal'**
  - 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 'unbounded'**
  - 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 than 1.0000000000**
  - 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.

## 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 sensitivities, does not have any feasible solution. If the solver can prove that the dual is infeasible, it will return a dual infeasibility certificate. Otherwise, it may finish with a solution status of `unknown`.

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

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

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

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

x = Variable("x")

constraints = [x >= 1]

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

Upon viewing the printed output,

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

Cost
----
 1e-30

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

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

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.

## 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 unknown 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 RuntimeError unknown on cvxopt and RuntimeError
#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.

## 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 RuntimeError!

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

Cost
----
  2

Free Variables
-----
x : 1

  | Relax.1
C : [ 1      2      ]

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

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

With constants relaxed individually
=====

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

```

```
x_min <= x_min_Relax.2*x_min_{before}_Relax.2
x_max_Relax.2 >= 1
x_max >= x_max_Relax.2**(-1)*x_max_{before}_Relax.2
x_max <= x_max_Relax.2*x_max_{before}_Relax.2

Cost
----
2

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

      | Relax.2
x_max : 1
x_min : 2

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

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



---

## Visualization and Interaction

---

### Interactive Control Panel

A model can be manipulated and visualized in Jupyter Notebook by calling `model.controlpanel()`. By default this creates a slider for every constant in the model and gives them automatic ranges, but variables and/or ranges can be changed in the Settings tab or specified in the first argument to `controlpanel()`.

Besides the default behaviour shown above, the control panel can also display custom analyses and plots via the `fn_of_sol` argument, which accepts a function (or list of functions) that take the solution as their input.

### Plotting a 1D Sweep

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

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

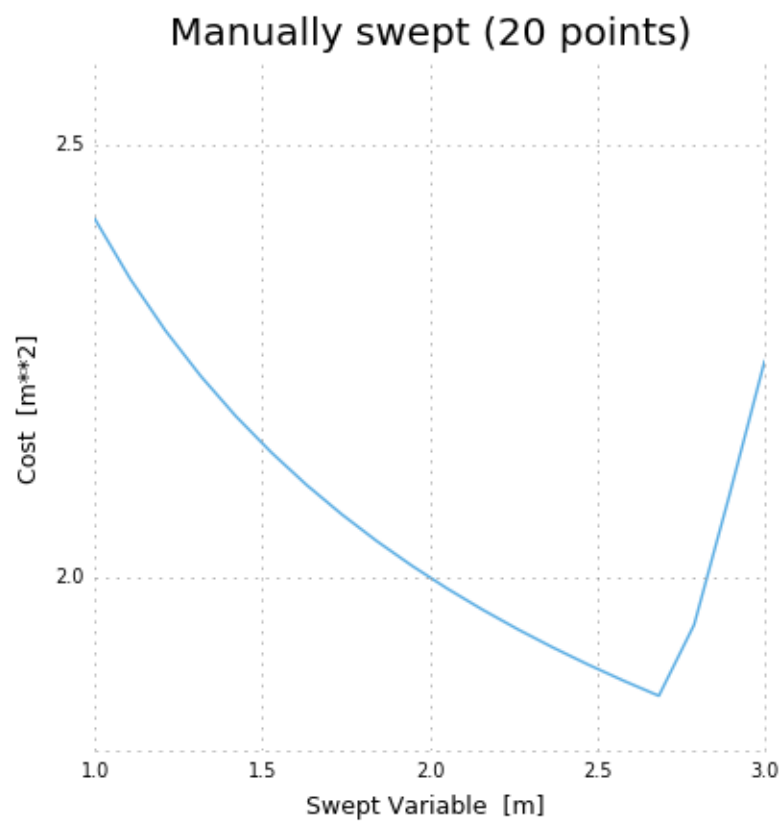
x = Variable("x", "m", "Swept Variable")
y = Variable("y", "m^2", "Cost")
m = Model(y, [y >= (x/2)**-0.5 * units.m**2.5 + 1*units.m**2, y >= (x/2)**2])

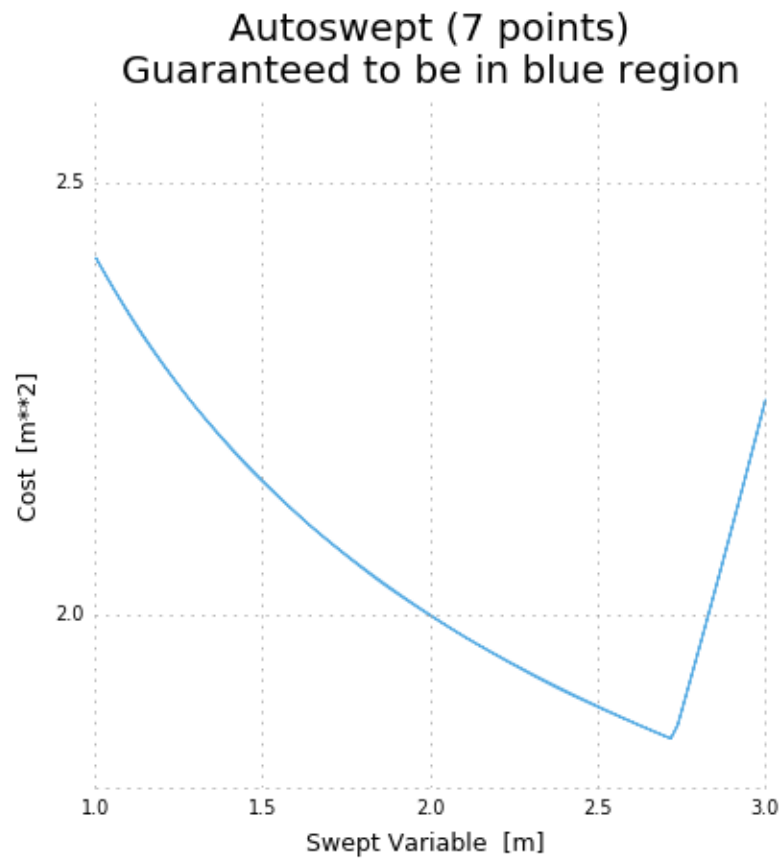
# arguments are: model, swept: values, posnomial for y-axis
sol = m.sweep({x: np.linspace(1, 3, 20)}, verbosity=0)
```

```
f, ax = sol.plot(y)
ax.set_title("Manually swept (20 points)")
f.show()
f.savefig("plot_sweep1d.png")

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

Which results in:









---

## Building Complex Models

---

### 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.

### Accessing Variables in Models

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

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

These methods are illustrated in the following example.

```

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

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

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

class PowerSystem(Model):
    "A battery powering a motor"
    def setup(self):
        components = [Battery(), Motor()]
        m = Variable("m", "lb", "mass")
        return [components,
                m >= sum(comp.topvar("m") for comp in components)]

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

```

```

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

```

## 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:

```

"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).summary()

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

```

```

SCALAR

Cost
----
1

Free Variables
-----
x : 1

-----

VECTORIZED

Cost
----
2

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

```

## Multipoint analysis modeling

In many engineering models, there is a physical object that is operated in multiple conditions. Some variables correspond to the design of the object (size, weight, construction) while others are vectorized over

the different conditions (speed, temperature, altitude). By combining named models and vectorization we can create intuitive representations of these systems while maintaining modularity and interoperability.

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

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

```

"""Modular aircraft concept"""
import 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")

        return self.components, [
            W >= sum(c.topvar("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.topvar("W") + 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
            self.fs = FlightSegment(aircraft)

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

            return self.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)

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

```

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

```

Cost
----
1.943 [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
D : [ 4.87     4.86     4.85     4.85     ] [lbf] drag force

Sensitivities
-----
      | Aircraft/Fuselage
W : +0.25                                weight

      | Aircraft/Wing
S : +0.68                                surface area
\rho : +0.48                              areal density
A : -0.31                                aspect ratio

Next Largest Sensitivities
-----
      | Mission/FlightSegment/AircraftP/WingAero
e : [ -0.093    -0.092    -0.092    -0.092    ] Oswald efficiency

```

```
| Mission/FlightSegment/FlightState
V : [ +0.1    +0.1    +0.1    +0.1    ] true airspeed
\rho : [ +0.034  +0.034  +0.035  +0.035  ] air density
```





## Derived Variables

### Evaluated Fixed Variables

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

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

### Evaluated Free Variables

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

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

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

## Sweeps

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

## Sweep Substitutions

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

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

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

## 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.

## 1D Autosweeps

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

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

```
"Show autosweep_1d functionality"
import numpy as np
import gpkit
from gpkit import units, Variable, Model
from gpkit.tools.autosweep import autosweep_1d
from gpkit.small_scripts import mag

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

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

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

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

## Tight ConstraintSets

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

given model, but representing it as an equality would be non-convex.

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

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

## Substitutions

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

### 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.

### 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`.

## 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).

## Updating ConstraintSet substitutions

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

## Substituting with replacement

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

## 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 is 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`.

## Freeing Fixed Variables

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

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

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

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

## Composite Objectives

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

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

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

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

```
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.

---

## Signomial Programming

---

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

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

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

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

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

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

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

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

### 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)
print m.localsolve(verbosity=0).summary()
assert abs(m.solution(x) - 0.9) < 1e-6
```

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

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

## Sequential Geometric Programs

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

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

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

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

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

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

objective = y

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

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

```
Cost
----
0.7854

Free Variables
-----
```



```
x : 0.7854
y : 0.7854
```

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

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

x = Variable("x")

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

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

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

```
Cost
----
0.7854

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

            # Call external code
            res = external_code(x_star)

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

```

and replace the incompatible constraint in our GP:

```

"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)
print m.localsolve(verbosity=0).summary()

```

```
Cost
----
0.7854

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

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

---

### iPython Notebook Examples

More examples, including some with in-depth explanations and interactive visualizations, can be seen on [nbviewer](#).

### 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
```

## 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
print m.solve(verbosity=0).table()
```

The output is

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

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

```

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

Sensitivities
-----
A_{wall}  : -1.5     upper limit, wall area
  alpha   : +0.5     lower limit, wall aspect ratio
  gamma   : +0.0003  lower limit, floor aspect ratio
A_{floor} : -5.7e-09 upper limit, floor area
  beta    : -1.4e-09 upper limit, wall aspect ratio
  delta   : -1.4e-09 upper limit, floor aspect ratio

```

## Water Tank

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

```

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

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

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

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

```

The output is

```

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

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

Sensitivities

```

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

## 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
pi = np.pi

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

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

```

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 coefficient of wing
 C_f : 0.003599   skin friction coefficient
  D : 303.1       [N] total drag force
 Re : 3.675e+06   Reynold's number
  S : 16.44       [m**2] total wing area
  V : 38.15      [m/s] cruising speed
  W : 7341       [N] total aircraft weight
 W_w : 2401      [N] wing weight

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

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 coefficient 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

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

```

## Simple Beam

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

```

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

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.summary()
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

Most Sensitive
-----
L : +4 Overall beam length
EI : -1
q : [ +0.0072  +0.042  +0.12  +0.23  ... ] Distributed load at each_
↔point

```

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

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

## Subpackages

### `gpkit.constraints` package

#### Submodules

#### `gpkit.constraints.array` module

Implements `ArrayConstraint`

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

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

A `ConstraintSet` for prettier array-constraint printing.

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

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

**subinplace** (*subs*)

Substitutes in place, updating `self.substitutions` accordingly.

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

### gpkit.constraints.bounded module

Implements Bounded

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

Bases: `gpkit.constraints.set.ConstraintSet`

Bounds contained variables so as to ensure dual feasibility.

**constraints** [iterable] constraints whose varkeys will be bounded

**substitutions** [dict] as in `ConstraintSet.__init__`

**verbosity** [int]

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

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

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

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

**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*,  
*add\_cost\_values\_to\_substitutions=True*)

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** ()

Resets varkeys to what is in the cost and constraints

**rootconstr\_latex** (*excluded=None*)

Latex showing cost, to be used when this is the top constraint

**rootconstr\_str** (*excluded=None*)

String showing cost, to be used when this is the top constraint

**subinplace** (*subs*)

Substitutes in place.

## gpkit.constraints.geometric\_program module

Implement the GeometricProgram class

```
class gpkit.constraints.geometric_program.GeometricProgram(cost, constraints,
                                                         substitu-
                                                         tions=None,
                                                         ver-
                                                         bosity=1)
```

Bases: `gpkit.constraints.costed.CostedConstraintSet`, `gpkit.nomials.data.NomialData`

Standard mathematical representation of a GP.

**cost** [Constraint] Posynomial to minimize when solving

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

instead use two inequality constraints (e.g.  $x \leq 1$ ,  $1/x \leq 1$ )

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

*solver\_out* and *solver\_log* are set during a solve *result* is set at the end of a solve if solution status is optimal

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

**check\_solution** (*cost, primal, nu, la, tol=0.001, abstol=1e-20*)

Run a series of checks to mathematically confirm sol solves this GP

**cost: float** cost returned by solver

**primal: list** primal solution returned by solver

**nu**: `numpy.ndarray` monomial lagrange multiplier

**la**: `numpy.ndarray` posynomial lagrange multiplier

RuntimeWarning, if any problems are found

**gen** (*verbosity=1*)

Generates nomial and solve data (A, p\_idx) from self.posynomials

**solve** (*solver=None, verbosity=1, warn\_on\_check=False, \*args, \*\*kwargs*)

Solves a GeometricProgram and returns the solution.

**solver** [str or function (optional)] By default uses one of the solvers found during installation.

If set to “mosek”, “mosek\_cli”, or “cvxopt”, uses that solver. If set to a function, passes that function cs, A, p\_idx, and k.

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

**\*args, \*\*kwargs** : Passed to solver constructor and solver function.

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

**cost** [float] The value of the objective at the solution.

**variables** [dict] The value of each variable at the solution.

**sensitivities** [dict]

**monomials** [array of floats] Each monomial’s dual variable value at the solution.

**posynomials** [array of floats] Each posynomials’s dual variable value at the solution.

`gpkit.constraints.geometric_program.genA(exps, varlocs)`

Generates A matrix from exps and varlocs

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

**varlocs** [dict] Locations of each variable in exps

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

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

## gpkit.constraints.linked module

## 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

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

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

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

**gp** (*verbosity=1*, *constants=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

**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, constants=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

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

**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, skip\_sweepfailures, constants, sweep, linked\_sweep, 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.

**process\_result** (*result*)

**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.ConstraintRelaxedEqually` (*constraints*)  
 Bases: `gpkIt.constraints.set.ConstraintSet`

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

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

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

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

## gpkIt.constraints.set module

Implements `ConstraintSet`

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

Recursive container for `ConstraintSets` and `Inequalities`

**append** (*value*)

**as\_gpconstr** (*x0*, *substitutions=None*)

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  $\leq 1$

**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** ()

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\_posyslst1*

**nus**: list of lists Each posynomial's monomial sensitivities

**constraint\_sens** [dict] The interesting and computable sensitivities of this constraint

**var\_senss** [dict] The variable sensitivities of this constraint

**str\_without** (*excluded=None*)

String representation of a ConstraintSet.

**subconstr\_latex** (*excluded=None*)

The collapsed appearance of a ConstraintSet

**subconstr\_str** (*excluded=None*)

The collapsed appearance of a ConstraintSet

**subinplace** (*subs*)

Substitutes in place, updating self.substitutions accordingly.

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

**topvar** (*key*)

If a variable by a given name exists in the top model, return it

**unique\_varkeys = frozenset([])**

**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  $\leq 1$ )

**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()
```

**firstgp** (*x0*, *substitutions*)

Generates a simplified GP representation for later modification

**gp** (*x0=None*, *verbosity=1*, *modifylastgp=False*)

The GP approximation of this SP at *x0*.

**localsolve** (*solver=None*, *verbosity=1*, *x0=None*, *reltol=0.0001*, *iteration\_limit=50*, *modifylastgp=True*, *\*\*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\_idx*s, and *k*.

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

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

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

**iteration\_limit** [int] Maximum GP iterations allowed.

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

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

## gpkit.constraints.single\_equation module

Implements SingleEquationConstraint

```
class gpkIt.constraints.single_equation.SingleEquationConstraint (left,  
oper,  
right)  
  
Bases: object  
Constraint expressible in a single equation.  
func_ops = {'<=': <built-in function le>, '=': <built-in function eq>, '>=': <built-in function ge>}  
latex (excluded=None)  
    Latex representation without attributes in excluded list  
latex_ops = {'<=': '\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
```

## Module contents

Contains ConstraintSet and related classes and objects

## gpkIt.interactive package

### Submodules

#### gpkIt.interactive.chartjs module

### gpkit.interactive.linking\_diagram module

Module for creating diagrams illustrating variables shared between submodels.

`gpkit.interactive.linking_diagram.linking_diagram` (*topmodel*, *subsystems*,  
*filename*)

Method to create a latex diagram illustrating how variables are linked between a parent model and its submodels

*topmodel* - a model object, the parent model of all submodels  
*subsystems* - a list of model objects, each a submodel of the *topmodel*  
*filename* - a string which is the name of the file latex output will be written to

note: the following packages must be used in the latex file `usepackage{tikz}` `usetikzlibrary{backgrounds}`

### gpkit.interactive.plot\_sweep module

Implements `plot_sweep1d` function

`gpkit.interactive.plot_sweep.assign_axes` (*var*, *posys*, *axes*)

Assigns axes to *posys*, creating and formatting if necessary

`gpkit.interactive.plot_sweep.format_and_label_axes` (*var*, *posys*, *axes*, *ylabel=True*)

Formats and labels axes

`gpkit.interactive.plot_sweep.plot_1dsweepgrid` (*model*, *sweeps*, *posys*,  
*origsol=None*, *tol=0.01*,  
*\*\*solveargs*)

Creates and plots a sweep from an existing model

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

### gpkit.interactive.plotting module

Plotting methods

`gpkit.interactive.plotting.compare` (*models*, *sweeps*, *posys*, *tol=0.001*)

Compares the values of *posys* over a sweep of several models.

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

Currently only supports a single *sweepvar*.

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

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

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

Plots the convergence of a signomial programming model

**model: Model** Signomial programming model that has already been solved

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

### gpkit.interactive.ractor module

Implements Ractor-based interactive CADtoons

```
gpkit.interactive.ractor.ractorjs(title, model, update_py, ranges, con-  
straint_js='')
```

Creates Javascript/HTML for CADtoon interaction without installing GPKit.

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

Creates interactive iPython widget for controlling a CADtoon

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

Displays cadtoon as iPython HTML

### gpkit.interactive.sensitivity\_map module

Implements heatmapped equations to highlight sensitivities.

```
class gpkit.interactive.sensitivity_map.SensitivityMap(model,  
paintby='constants')
```

Bases: object

Latex representations of a model heatmapped by its latest sensitivities

**model** [Model] The Model object that the Map will be based on

**paintby** [string] The unit of colouring. Must be one of “constants”, “monomials”, or “posynomials”.

```
from IPython.display import display for key in m.solution[“sensitivities”]:
```

```
    print key display(SensitivityMap(m, paintby=key))
```

```
constraint_latex_list(paintby)
```

Generates LaTeX for constraints.

**latex**

LaTeX representation.

**solution**

Gets solution, indexing into a sweep if necessary.

```
gpkit.interactive.sensitivity_map.colorfn_gen(scale, power=0.66)
```

Generates color gradient of a given power law.

```
gpkit.interactive.sensitivity_map.signomial_print(sig, sol, colorfn,  
paintby='constants',  
idx=None)
```

For pretty printing with Sympy

### gpkit.interactive.widgets module

#### Module contents

Module for the interactive and plotting functions of GPKit



## gpkit.nomials package

### Submodules

#### gpkit.nomials.array module

Module for creating NomialArray instances.

#### Example

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

#### class gpkit.nomials.array.NomialArray

Bases: `numpy.ndarray`

A Numpy array with elementwise inequalities and substitutions.

input\_array : array-like

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

#### **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 exponents, coefficients, varlocs data – common to nomials and programs

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

Object for holding coefficients, exponents, 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

**prod** ()  
Return self for compatibility with NomialArray

**str\_without** (*excluded=None*)  
String representation excluding fields ('units', varkey attributes)

**sub = None**

**sum** ()  
Return self for compatibility with NomialArray

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

**unitstr** (*units, into='%s', options='~', dimless=''*)  
Returns the string corresponding to an object's units.

**value**  
Self, with values substituted for variables that have values  
  
float, if no symbolic variables remain after substitution (Monomial, Posynomial, or Nomial), otherwise.

## gpkit.nomials.nomial\_math module

Signomial, Posynomial, Monomial, Constraint, & MonoEQCOnstraint classes

```
class gpkit.nomials.nomial_math.Monomial (exps=None,          cs=1,          re-
                                         quire_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,          re-
                                             quire_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  $\geq$  posynomial Stored in the `posylt1_rep` attribute as a single Posynomial (self  $\leq$  1) Usually initialized via operator overloading, e.g. `cc = (y**2  $\geq$  1 + x)`

**as\_gpconstr** (*x0, substitutions*)

GP version of a Posynomial constraint is itself

**as\_posylt1** (*substitutions=None*)

Returns the posys  $\leq$  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,  $c \leq 0$  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 substituted 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  $\geq$  posynomial Stored internally (exps, cs) as a single Signomial ( $0 \geq$  self) Usually initialized via operator overloading, e.g.  $cc = (y**2 \geq 1 + x - y)$  Additionally retains input format (lhs vs rhs) in self.left and self.right Form is self.left  $\geq$  self.right.

**as\_approxsgt** (*x0*)  
 Returns monomial-greater-than sides, to be called after as\_approxlt1

**as\_approxslt** ()  
 Returns posynomial-less-than sides of a signomial constraint

**as\_gpconstr** (*x0, substitutions=None*)  
 Returns GP approximation of an SP constraint at x0

**as\_posyslt1** (*substitutions=None*)  
 Returns the posys  $\leq 1$  representation of this constraint.

**class** gpkit.nomials.nomial\_math.**SingleSignomialEquality** (*left, right*)  
 Bases: *gpkit.nomials.nomial\_math.SingomialInequality*

A constraint of the general form posynomial  $==$  posynomial

**as\_approxsgt** (*x0*)  
 Returns monomial-greater-than sides, to be called after as\_approxlt1

**as\_approxslt** ()  
 Returns posynomial-less-than sides of a signomial constraint

**as\_gpconstr** (*x0, substitutions=None*)  
 Returns GP approximation of an SP constraint at x0

**as\_posyslt1** (*substitutions=None*)  
 Returns the posys  $\leq 1$  representation of this constraint.

**gpkit.nomials.nomial\_math.non\_dimensionalize** (*posy*)  
 Non-dimensionalize a posy (warning: mutates posy)

### 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)

**to** (arg)

Create new Signomial converted to new units

**class** gpkit.nomials.variables.**VectorizableVariable** (\*args, \*\*descr)

Bases: `gpkit.nomials.variables.Variable`, `gpkit.nomials.variables.ArrayVariable`

A Variable outside a vectorized environment, an ArrayVariable within.

## Module contents

Contains nomials, inequalities, and arrays

## gpkit.tests package

### Submodules

#### gpkit.tests.diff\_output module

Function to diff example output and allow small numerical errors

`gpkit.tests.diff_output.diff` (output1, output2, tol=0.001)  
check that output1 and output2 are same up to small errors in numbers

#### gpkit.tests.from\_paths module

Runs each file listed in pwd/TESTS as a test

**class** gpkit.tests.from\_paths.**TestFiles** (methodName=‘runTest’)  
Bases: `unittest.case.TestCase`

Stub to be filled with files in \$pwd/TESTS

`gpkit.tests.from_paths.add_filetest` (testclass, path)  
Add test that imports the given path and runs its test() function

`gpkit.tests.from_paths.clean` (string)  
Parses string into valid python variable name

<https://stackoverflow.com/questions/3303312/how-do-i-convert-a-string-to-a-valid-variable-name-in-python>

`gpkit.tests.from_paths.newtest_fn` (*name, solver, import\_dict, path*)  
Doubly nested callbacks to run the test with `getattr(self, name)()`

`gpkit.tests.from_paths.run` (*filename='TESTS', xmloutput=False, skipsolvers=None*)  
Parse and run paths from a given file for each solver

### **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()
        I can't be greater than (1 + something positive)

    test_bad_elements()

    test_constraintget()

    test_equality_relaxation()

    test_evalfn()

    test_init()
        Test Constraint __init__

    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_autosweep (example)
test_beam (example)
test_debug (example)
test_external_sp (example)
test_external_sp2 (example)
test_model_var_access (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_examples.assert_logtol (first, second, logtol=1e-06)
    Asserts that the logs of two arrays have a given abstol

```

### 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_sensitivities ()  
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  $xy \geq 1$   
    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_bounded ()
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 No-
    mialArray
    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_substitution ()
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.TestCootMatrix (methodName='runTest')
    Bases: unittest.case.TestCase

    TestCase for the CootMatrix class

test_shape()

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\_units()**

**test\_call\_vector()**

**test\_key\_options()**

**test\_subinto()**

**test\_table()**

**test\_units\_sub()**

### **gpkIt.tests.t\_sub module**

Test substitution capability across gpkIt

**class** `gpkIt.tests.t_sub.TestModelSubs` (*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test substitution for Model objects

**test\_bad\_gp\_sub()**

**test\_bad\_subinplace()**

**test\_calconst()**

**test\_getkey()**

**test\_model\_composition\_units()**

**test\_model\_recursion()**

**test\_persistence()**

**test\_phantoms()**

**test\_quantity\_sub()**

**test\_skipfailures()**



```

test_united_sub_sweep ()
test_vector_init ()
test_vector_sub ()
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_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.TestTools (methodName='runTest')
    Bases: unittest.case.TestCase
    TestCase for math models
test_binary_sweep_tree ()
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_tools.assert_logtol (first, second, logtol=1e-06)
    Asserts that the logs of two arrays have a given abstol

```

### 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_to ()

    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 ()`

### gpkit.tests.test\_repo module

Implements tests for all external repositories.

`gpkit.tests.test_repo.call_and_retry (cmd, max_iterations=5, delay=5)`

Tries `max_iterations` times (waiting `d` each time) to run a command

`gpkit.tests.test_repo.get_settings ()`

Gets settings from a TESTCONFIG file

`gpkit.tests.test_repo.git_clone (repo, branch='master')`

Tries several times to clone a given repository

`gpkit.tests.test_repo.pip_install (package, local=False)`

Tries several times to install a pip package

`gpkit.tests.test_repo.test_repo (repo='.', xmloutput=False)`

Test repository.

If no repo name given, runs in current directory. Otherwise, assumes is in directory above the repo with a shared gpkit-models repository.

`gpkit.tests.test_repo.test_repos (repos=None, xmloutput=False)`

Get the list of external repos to test, and test.

### Module contents

GPkit testing module

## gpkit.tools package

### Submodules

#### gpkit.tools.autosweep module

Tools for optimal fits to GP sweeps

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

Bases: `object`

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

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

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

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

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

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

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

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

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

**add\_splitcost** (*splitval, splitlb, splitub*)  
 Adds a splitval, lower bound, and upper bound

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

**min\_bst** (*value*)  
 Returns smallest bst around value.

**posy\_at** (*posy, value*)  
 Logspace interpolates between sols to get posynomial values.  
 No guarantees, just like a regular sweep.

**sample\_at** (*values*)  
 Creates a SolutionOracle at a given range of values

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

**sollist**  
 Returns a list of all the solutions in an autosweep

**class** `gpkIt.tools.autosweep.SolutionOracle` (*bst, sampled\_at*)  
 Bases: `object`

Acts like a SolutionArray for autosweeps

**cost\_lb** ()  
 Gets cost lower bounds from the BST and units them

**cost\_ub** ()  
 Gets cost upper bounds from the BST and units them

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

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

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

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

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

## gpkIt.tools.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.spdata module

Implements SPData class

```
class gpkit.tools.spdata.SPData(model)
Bases: gpkit.nomials.data.NomialData
```

Generates matrices describing an SP.

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

**save** (filename)  
Save spdata to an h5 file.

### gpkit.tools.tools module

Non-application-specific convenience methods for GPKit

```
gpkit.tools.tools.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  $e^{\text{posy}} - 1$ 
```

**posy** [gpkit.Posynomial] Variable or expression to exponentiate

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

**gpkit.Posynomial** Taylor expansion of  $e^{\text{posy}} - 1$ , 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

## Submodules

### 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 mskexopt.

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.

```

## 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

```

## gpkIt.keydict module

Implements KeyDict and KeySet classes

```

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

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

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

```

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

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

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

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

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

See also: `gpkit/tests/t_keydict.py`.

**`collapse_arrays = True`**

**`get`** (*key*, *alternative*=<type 'exceptions.KeyError'>)

**`keymapping = True`**

**`parse_and_index`** (*key*)

Returns key if key had one, and `veckey/idx` for indexed `veckey`s.

**`update`** (*\*args*, *\*\*kwargs*)

Iterates through the dictionary created by `args` and `kwargs`

**`update_keymap`** ()

Updates the keymap with the keys in `_unmapped_keys`

**`class gpkit.keydict.KeySet`** (*\*args*, *\*\*kwargs*)

Bases: `gpkit.keydict.KeyDict`

`KeyDict`s that don't collapse arrays or store values.

**`add`** (*item*)

Adds an item to the keyset

**`collapse_arrays = False`**

**`update`** (*\*args*, *\*\*kwargs*)

Iterates through the dictionary created by `args` and `kwargs`

`gpkit.keydict.clean_value` (*key*, *value*)

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

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



## gpkit.modified\_ctypesgen module

## gpkit.repr\_conventions module

Repository for representation standards

`gpkit.repr_conventions.unitstr (units, into='%s', options='~', dimless='')`  
Returns the string corresponding to an object's units.

## gpkit.small\_classes module

Miscellaneous small classes

**class** `gpkit.small_classes.CootMatrix (*args, **kwargs)`  
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.

**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.Count`  
Bases: `object`

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

**next** ()  
Increment `self.count` and return it

**class** `gpkit.small_classes.DictOfLists`  
Bases: `dict`

A hierarchy of dictionaries, 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.

## 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.veckeyed` (*key*)  
Return a veckey version of a VarKey

## 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)
```

**plot** (*posys=None, axes=None*)

Plots a sweep for each posy

**program** = None

**subinto** (*posy*)

Returns NomialArray of each solution substituted into posy.

**summary** (*showvars=(), ntopsenss=5*)

Print summary table, showing top sensitivities and no constants

**table** (*showvars=(), tables=('cost', 'sweepvariables', 'freevariables', 'constants', 'sensitivities'), \*\*kwargs*)

A table representation of this SolutionArray

**tables: Iterable**

**Which to print of ("cost", "sweepvariables", "freevariables", "constants", "sensitivities")**

**fixedcols**: If true, print vectors in fixed-width format latex: int

If > 0, return latex format (options 1-3); otherwise plain text

**included\_models: Iterable of strings** If specified, the models (by name) to include

**excluded\_models: Iterable of strings** If specified, model names to exclude

str

**table\_titles** = {'variables': 'Variables', 'freevariables': 'Free Variables', 'sweepvariables': 'Sweep Variables',

`gpkit.solution_array.insenss_table` (*data, \_, maxval=0.1, \*\*kwargs*)

Returns insensitivity table lines

`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  $\text{all}(\text{abs}(\text{value})) < \text{minval}$

**printunits:** bool **fixedcols:** bool

if True, print rhs (val, units, label) in fixed-width cols

**varfmt:** string format for variable names

**valfmt:** string format for scalar values

**vecfmt:** string format for vector values

**latex:** int If > 0, return latex format (options 1-3); otherwise plain text

**included\_models:** Iterable of strings If specified, the models (by name) to include

**excluded\_models:** Iterable of strings If specified, model names to exclude

**sortbyvals** [boolean] If true, rows are sorted by their average value instead of by name.

`gpkIt.solution_array.senss_table` (*data*, *showvars=()*, *title='Sensitivities'*, *\*\*kwargs*)

Returns sensitivity table lines

`gpkIt.solution_array.topsenss_filter` (*data*, *showvars*, *nvars=5*)

Filters sensitivities down to top N vars

`gpkIt.solution_array.topsenss_table` (*data*, *showvars*, *nvars=5*, *\*\*kwargs*)

Returns top sensitivity table lines

## gpkIt.varkey module

Defines the VarKey class

**class** `gpkIt.varkey.VarKey` (*name=None*, *\*\*kwargs*)

Bases: object

An object to correspond to each 'variable name'.

**name** [str, VarKey, or Monomial] Name of this Variable, or object to derive this Variable from.

**\*\*kwargs** : Any additional attributes, which become the descr attribute (a dict).

VarKey with the given name and descr.

**eq\_ignores** = frozenset(['units', 'value'])

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

Returns latex representation.

**latex\_unitstr** ()  
Returns latex unitstr

**naming**  
Returns this varkey's naming tuple

**new\_unnamed\_id** ()  
Increment self.count and return it

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

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

**unitstr** (*units, into='%s', options='~', dimless=''*)  
Returns the string corresponding to an object's units.

## Module contents

GP and SP Modeling Package

For examples please see the examples folder.

## Requirements

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

## Attributes

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

**class** `gpkit.GPkitUnits`

Bases: object

Return monomials instead of Quantities

**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 extended 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/latest/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={2017},  
  note={Version 0.5.3}  
}
```





## 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.



This page lists the changes made in each point version of gpkit.

### Version 0.5.3

- faster SP solves (#1109)
- `LinkedConstraintSet` deprecated (#1110)
- Fixes to `autosweep`, `ConstraintSet`, `interactive`
- Solution time is now stored with solutions (including sweeps/SPs)
- Model strings are divided with slashes (e.g. `Airplane/Wing`)

### Version 0.5.2

- **Added new `sweep` and `autosweep` methods to `Model`**
  - Added `plot` routines to the results of those routines to make it easy to plot a 1D sweep.
- **Added new `summary` method to `solution_array`.**
  - It and `table` accept iterables of vars, will only print vars in that iterable (or, by default, all vars)
- **Cleaned up and documented the `interactive` submodule**
  - removed contour and sensitivity plots
  - added a 1D-sweep plotting function
  - added that plotting function as an option within the control panel interface
- **Overhauled and documented three types of variables whose value is determined by functions:**

- calculated constants
- post-solve calculated variables
- between-GP-solves calculated variables (for Sequential Geometric Programs)
- Fix Bounded and implement `debug()` for SPs
- Apply `subinplace` to substitutions dictionary as well
- Require GP substitutions to be Numbers only
- Extend Bounded to one-sided bounds
- Print model's numbers by default, unless "modelnums" in `exclude`
- Implement lazy keymapping, allowing GP/SP results to be KeyDicts
- Handle Signomial Inequalities that become Posynomial Inequalities after substitution
- Various documentation updates
- Various bug fixes

## 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*)

## 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*)

## 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.

## 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

## 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.

## 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

## 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.

## 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.

## 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)





## g

- gpkit, 97
- gpkit.build, 90
- gpkit.constraints, 66
  - gpkit.constraints.array, 57
  - gpkit.constraints.bounded, 58
  - gpkit.constraints.costed, 58
  - gpkit.constraints.geometric\_program, 59
  - gpkit.constraints.model, 60
  - gpkit.constraints.prog\_factories, 62
  - gpkit.constraints.relax, 62
  - gpkit.constraints.set, 63
  - gpkit.constraints.sigeq, 64
  - gpkit.constraints.signomial\_program, 64
  - gpkit.constraints.single\_equation, 65
  - gpkit.constraints.tight, 66
- gpkit.exceptions, 91
- gpkit.interactive, 68
  - gpkit.interactive.linking\_diagram, 67
  - gpkit.interactive.plot\_sweep, 67
  - gpkit.interactive.plotting, 67
  - gpkit.interactive.ractor, 68
  - gpkit.interactive.sensitivity\_map, 68
- gpkit.keydict, 91
- gpkit.nomials, 75
  - gpkit.nomials.array, 69
  - gpkit.nomials.data, 70
  - gpkit.nomials.nomial\_core, 70
  - gpkit.nomials.nomial\_math, 71
  - gpkit.nomials.substitution, 74
  - gpkit.nomials.variables, 74
- gpkit.repr\_conventions, 93
- gpkit.small\_classes, 93
- gpkit.small\_scripts, 94
- gpkit.solution\_array, 94
- gpkit.tests, 87
  - gpkit.tests.diff\_output, 75
  - gpkit.tests.from\_paths, 75
  - gpkit.tests.helpers, 76
  - gpkit.tests.run\_tests, 77
  - gpkit.tests.t\_constraints, 77
  - gpkit.tests.t\_examples, 78
  - gpkit.tests.t\_keydict, 79
  - gpkit.tests.t\_model, 79
  - gpkit.tests.t\_nomial\_array, 81
  - gpkit.tests.t\_nomials, 82
  - gpkit.tests.t\_small, 83
  - gpkit.tests.t\_solution\_array, 84
  - gpkit.tests.t\_sub, 84
  - gpkit.tests.t\_tools, 85
  - gpkit.tests.t\_vars, 86
  - gpkit.tests.test\_repo, 87
- gpkit.tools, 90
  - gpkit.tools.autosweep, 87
  - gpkit.tools.fmincon, 88
  - gpkit.tools.spdata, 89
  - gpkit.tools.tools, 89
- gpkit.varkey, 96



## A

add() (gpkit.keydict.KeySet method), 92  
 add\_filetest() (in module gpkit.tests.from\_paths), 75  
 add\_split() (gpkit.tools.autosweep.BinarySweepTree method), 87  
 add\_splitcost() (gpkit.tools.autosweep.BinarySweepTree method), 88  
 append() (gpkit.constraints.set.ConstraintSet method), 63  
 append() (gpkit.small\_classes.CootMatrix method), 93  
 append() (gpkit.small\_classes.DictOfLists method), 93  
 append\_sub() (in module gpkit.nomials.substitution), 74  
 array\_constraint() (in module gpkit.nomials.array), 69  
 ArrayConstraint (class in gpkit.constraints.array), 57  
 ArrayVariable (class in gpkit.nomials.variables), 74  
 as\_approxsgt() (gpkit.nomials.nomial\_math.SignomialInequality method), 73  
 as\_approxsgt() (gpkit.nomials.nomial\_math.SingleSignomialEquality method), 73  
 as\_approxslt() (gpkit.nomials.nomial\_math.SignomialInequality method), 73  
 as\_approxslt() (gpkit.nomials.nomial\_math.SingleSignomialEquality method), 73  
 as\_gpconstr() (gpkit.constraints.set.ConstraintSet method), 63  
 as\_gpconstr() (gpkit.nomials.nomial\_math.PosynomialInequality method), 72  
 as\_gpconstr() (gpkit.nomials.nomial\_math.SignomialInequality method), 73  
 as\_gpconstr() (gpkit.nomials.nomial\_math.SingleSignomialEquality method), 73  
 as\_posyslt1() (gpkit.constraints.set.ConstraintSet method), 63  
 as\_posyslt1() (gpkit.nomials.nomial\_math.PosynomialInequality method), 72  
 as\_posyslt1() (gpkit.nomials.nomial\_math.SignomialInequality method), 73  
 as\_posyslt1() (gpkit.nomials.nomial\_math.SingleSignomialEquality method), 73  
 assert\_logtol() (in module gpkit.tests.t\_examples), 79

assert\_logtol() (in module gpkit.tests.t\_tools), 85  
 assign\_axes() (in module gpkit.interactive.plot\_sweep), 67  
 atindex() (gpkit.small\_classes.DictOfLists method), 93  
 autosweep() (gpkit.constraints.model.Model method), 61  
 autosweep\_1d() (in module gpkit.tools.autosweep), 88

## B

begin\_variable\_naming() (in module gpkit), 98  
 BinarySweepTree (class in gpkit.tools.autosweep), 87  
 Bounded (class in gpkit.constraints.bounded), 58  
 Box (class in gpkit.tests.t\_model), 79  
 BoxAreaBounds (class in gpkit.tests.t\_model), 79  
 build (gpkit.build.SolverBackend attribute), 91  
 build() (gpkit.build.Mosek method), 90  
 build\_gpkit() (in module gpkit.build), 91

## C

call\_and\_retry() (in module gpkit.tests.test\_repo), 87  
 check\_solution() (gpkit.constraints.geometric\_program.GeometricProgram method), 59  
 clean() (in module gpkit.tests.from\_paths), 75  
 clean\_value() (in module gpkit.keydict), 92  
 close() (gpkit.tests.helpers.NullFile method), 76  
 collapse\_arrays (gpkit.keydict.KeyDict attribute), 92  
 collapse\_arrays (gpkit.keydict.KeySet attribute), 92  
 colorfn\_gen() (in module gpkit.interactive.sensitivity\_map), 68  
 compare() (in module gpkit.interactive.plotting), 67  
 composite\_objective() (in module gpkit.tools.tools), 89  
 ConstantsRelaxed (class in gpkit.constraints.relax), 62  
 constraint\_latex\_list() (gpkit.interactive.sensitivity\_map.SensitivityMap method), 68  
 ConstraintSet (class in gpkit.constraints.set), 63

- ConstraintsRelaxed (class in gpkit.constraints.relax), 62
  - ConstraintsRelaxedEqually (class in gpkit.constraints.relax), 63
  - controlpanel() (gpkit.constraints.costed.CostedConstraintSet method), 58
  - convert\_to() (gpkit.nomials.nomial\_core.Nomial method), 70
  - CootMatrix (class in gpkit.small\_classes), 93
  - CootMatrixTuple (in module gpkit.small\_classes), 93
  - cost\_at() (gpkit.tools.autosweep.BinarySweepTree method), 88
  - cost\_lb() (gpkit.tools.autosweep.SolutionOracle method), 88
  - cost\_ub() (gpkit.tools.autosweep.SolutionOracle method), 88
  - CostedConstraintSet (class in gpkit.constraints.costed), 58
  - Count (class in gpkit.small\_classes), 93
  - CVXopt (class in gpkit.build), 90
- ## D
- debug() (gpkit.constraints.model.Model method), 61
  - descr (gpkit.nomials.variables.Variable attribute), 75
  - DictOfLists (class in gpkit.small\_classes), 93
  - diff() (gpkit.nomials.data.NomialData method), 70
  - diff() (gpkit.nomials.nomial\_math.Signomial method), 72
  - diff() (in module gpkit.build), 91
  - diff() (in module gpkit.tests.diff\_output), 75
  - disable\_units() (in module gpkit), 98
  - dot() (gpkit.small\_classes.CootMatrix method), 93
- ## E
- enable\_units() (in module gpkit), 98
  - end\_variable\_naming() (in module gpkit), 98
  - eq\_ignores (gpkit.varkey.VarKey attribute), 96
- ## F
- firstgp() (gpkit.constraints.signomial\_program.SignomialProgram method), 65
  - flat() (gpkit.constraints.set.ConstraintSet method), 63
  - format\_and\_label\_axes() (in module gpkit.interactive.plot\_sweep), 67
  - func\_ops (gpkit.constraints.single\_equation.SingleEquationConstraint attribute), 66
- ## G
- gen() (gpkit.constraints.geometric\_program.GeometricProgram method), 60
  - genA() (in module gpkit.constraints.geometric\_program), 60
  - generate\_example\_tests() (in module gpkit.tests.helpers), 76
  - generate\_mfiles() (in module gpkit.tools.fmincon), 88
  - GeometricProgram (class in gpkit.constraints.geometric\_program), 59
  - get() (gpkit.keydict.KeyDict method), 92
  - get\_settings() (in module gpkit.tests.test\_repo), 87
  - get\_tol() (in module gpkit.tools.autosweep), 88
  - git\_clone() (in module gpkit.tests.test\_repo), 87
  - gp() (gpkit.constraints.model.Model method), 61
  - gp() (gpkit.constraints.signomial\_program.SignomialProgram method), 65
  - gpkit (module), 97
  - gpkit.build (module), 90
  - gpkit.constraints (module), 66
  - gpkit.constraints.array (module), 57
  - gpkit.constraints.bounded (module), 58
  - gpkit.constraints.costed (module), 58
  - gpkit.constraints.geometric\_program (module), 59
  - gpkit.constraints.model (module), 60
  - gpkit.constraints.prog\_factories (module), 62
  - gpkit.constraints.relax (module), 62
  - gpkit.constraints.set (module), 63
  - gpkit.constraints.sigeq (module), 64
  - gpkit.constraints.signomial\_program (module), 64
  - gpkit.constraints.single\_equation (module), 65
  - gpkit.constraints.tight (module), 66
  - gpkit.exceptions (module), 91
  - gpkit.interactive (module), 68
  - gpkit.interactive.linking\_diagram (module), 67
  - gpkit.interactive.plot\_sweep (module), 67
  - gpkit.interactive.plotting (module), 67
  - gpkit.interactive.ractor (module), 68
  - gpkit.interactive.sensitivity\_map (module), 68
  - gpkit.keydict (module), 91
  - gpkit.nomials (module), 75
  - gpkit.nomials.array (module), 69
  - gpkit.nomials.data (module), 70
  - gpkit.nomials.nomial\_core (module), 70
  - gpkit.nomials.nomial\_math (module), 71
  - gpkit.nomials.substitution (module), 74
  - gpkit.nomials.variables (module), 74
  - gpkit.repr\_conventions (module), 93
  - gpkit.small\_classes (module), 93
  - gpkit.small\_scripts (module), 94
  - gpkit.solution\_array (module), 94
  - gpkit.tests (module), 87
  - gpkit.tests.diff\_output (module), 75
  - gpkit.tests.from\_paths (module), 75
  - gpkit.tests.helpers (module), 76
  - gpkit.tests.run\_tests (module), 77
  - gpkit.tests.t\_constraints (module), 77
  - gpkit.tests.t\_examples (module), 78
  - gpkit.tests.t\_keydict (module), 79
  - gpkit.tests.t\_model (module), 79
  - gpkit.tests.t\_nomial\_array (module), 81
  - gpkit.tests.t\_nomials (module), 82

gpkit.tests.t\_small (module), 83  
 gpkit.tests.t\_solution\_array (module), 84  
 gpkit.tests.t\_sub (module), 84  
 gpkit.tests.t\_tools (module), 85  
 gpkit.tests.t\_vars (module), 86  
 gpkit.tests.test\_repo (module), 87  
 gpkit.tools (module), 90  
 gpkit.tools.autosweep (module), 87  
 gpkit.tools.fmincon (module), 88  
 gpkit.tools.spdata (module), 89  
 gpkit.tools.tools (module), 89  
 gpkit.varkey (module), 96  
 GPkitUnits (class in gpkit), 97

## H

HashVector (class in gpkit.small\_classes), 94

## I

import\_tests() (in module gpkit.tests.run\_tests), 77  
 init\_from\_nomials() (gpkit.nomials.data.NomialData method), 70  
 insenss\_table() (in module gpkit.solution\_array), 95  
 installed (gpkit.build.SolverBackend attribute), 91  
 interact() (gpkit.constraints.costed.CostedConstraintSet method), 58  
 InvalidGPConstraint, 91  
 is\_sweepvar() (in module gpkit.small\_scripts), 94  
 isfile() (in module gpkit.build), 91

## K

key (gpkit.nomials.variables.Variable attribute), 75  
 KeyDict (class in gpkit.keydict), 91  
 keymapping (gpkit.keydict.KeyDict attribute), 92  
 KeySet (class in gpkit.keydict), 92

## L

latex (gpkit.interactive.sensitivity\_map.SensitivityMap attribute), 68  
 latex() (gpkit.constraints.set.ConstraintSet method), 63  
 latex() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 66  
 latex() (gpkit.nomials.array.NomialArray method), 69  
 latex() (gpkit.nomials.nomial\_core.Nomial method), 70  
 latex() (gpkit.varkey.VarKey method), 96  
 latex\_num() (in module gpkit.small\_scripts), 94  
 latex\_opsers (gpkit.constraints.single\_equation.SingleEquationConstraint attribute), 66  
 latex\_unitstr() (gpkit.varkey.VarKey method), 96  
 left (gpkit.nomials.array.NomialArray attribute), 69  
 linking\_diagram() (in module gpkit.interactive.linking\_diagram), 67  
 load\_settings() (in module gpkit), 98  
 localsolve() (gpkit.constraints.model.Model method), 61

localsolve() (gpkit.constraints.signomial\_program.SignomialProgram method), 65  
 log() (in module gpkit.build), 91  
 logged\_example\_testcase() (in module gpkit.tests.helpers), 76  
 look (gpkit.build.SolverBackend attribute), 91  
 look() (gpkit.build.CVXopt method), 90  
 look() (gpkit.build.Mosek method), 90  
 look() (gpkit.build.MosekCLI method), 90

## M

mag() (in module gpkit.small\_scripts), 94  
 make\_initial\_guess() (in module gpkit.tools.fmincon), 89  
 matrix\_converter() (in module gpkit.small\_classes), 94  
 mdmake() (in module gpkit.tools.tools), 89  
 mdparse() (in module gpkit.tools.tools), 89  
 min\_bst() (gpkit.tools.autosweep.BinarySweepTree method), 88  
 Model (class in gpkit.constraints.model), 60  
 mono\_approximation() (gpkit.nomials.nomial\_math.Monomial method), 71  
 mono\_approximation() (gpkit.nomials.nomial\_math.Signomial method), 72  
 mono\_lower\_bound() (gpkit.nomials.nomial\_math.Posynomial method), 71  
 Monomial (class in gpkit.nomials.nomial\_math), 71  
 MonomialEquality (class in gpkit.nomials.nomial\_math), 71  
 Mosek (class in gpkit.build), 90  
 MosekCLI (class in gpkit.build), 90

## N

name (gpkit.build.CVXopt attribute), 90  
 name (gpkit.build.Mosek attribute), 90  
 name (gpkit.build.MosekCLI attribute), 91  
 name (gpkit.build.SolverBackend attribute), 91  
 name (gpkit.constraints.model.Model attribute), 61  
 name (gpkit.tests.t\_model.TestGP attribute), 79  
 name (gpkit.tests.t\_model.TestSP attribute), 80  
 NamedVariables (class in gpkit), 97  
 naming (gpkit.constraints.model.Model attribute), 61  
 naming (gpkit.varkey.VarKey attribute), 97  
 ndig (gpkit.tests.t\_model.TestGP attribute), 79  
 ndig (gpkit.tests.t\_model.TestSP attribute), 81  
 new\_test() (in module gpkit.tests.helpers), 76  
 new\_unnamed\_id() (gpkit.varkey.VarKey method), 97  
 NewDefaultSolver (class in gpkit.tests.helpers), 76  
 newtest\_fn() (in module gpkit.tests.from\_paths), 75  
 next() (gpkit.small\_classes.Count method), 93  
 Nomial (class in gpkit.nomials.nomial\_core), 70  
 nomial\_latex\_helper() (in module gpkit.small\_scripts), 94

- NomialArray (class in gpkit.nomials.array), 69  
 NomialData (class in gpkit.nomials.data), 70  
 nomials (gpkit.nomials.nomial\_math.ScalarSingleEquationConstraint attribute), 72  
 non\_dimensionalize() (in module gpkit.nomials.nomial\_math), 73  
 NullFile (class in gpkit.tests.helpers), 76  
 num (gpkit.constraints.model.Model attribute), 61
- O**
- outer() (gpkit.nomials.array.NomialArray method), 69
- P**
- padleft() (gpkit.nomials.array.NomialArray method), 69  
 padright() (gpkit.nomials.array.NomialArray method), 69  
 parse\_and\_index() (gpkit.keydict.KeyDict method), 92  
 parse\_subs() (in module gpkit.nomials.substitution), 74  
 patches (gpkit.build.Mosek attribute), 90  
 pathjoin() (in module gpkit.build), 91  
 pip\_install() (in module gpkit.tests.test\_repo), 87  
 plot() (gpkit.solution\_array.SolutionArray method), 95  
 plot() (gpkit.tools.autosweep.SolutionOracle method), 88  
 plot\_1dsweepgrid() (in module gpkit.interactive.plot\_sweep), 67  
 plot\_convergence() (in module gpkit.interactive.plotting), 67  
 posy\_at() (gpkit.tools.autosweep.BinarySweepTree method), 88  
 posy\_negy() (gpkit.nomials.nomial\_math.Signomial method), 72  
 Posynomial (class in gpkit.nomials.nomial\_math), 71  
 PosynomialInequality (class in gpkit.nomials.nomial\_math), 72  
 process\_result() (gpkit.constraints.bounded.Bounded method), 58  
 process\_result() (gpkit.constraints.relax.ConstantsRelaxed method), 62  
 process\_result() (gpkit.constraints.set.ConstraintSet method), 63  
 process\_result() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 72  
 process\_result() (gpkit.constraints.tight.Tight method), 66  
 prod() (gpkit.nomials.array.NomialArray method), 69  
 prod() (gpkit.nomials.nomial\_core.Nomial method), 71  
 program (gpkit.constraints.model.Model attribute), 61  
 program (gpkit.solution\_array.SolutionArray attribute), 95
- R**
- ractorjs() (in module gpkit.interactive.ractor), 68  
 ractorpy() (in module gpkit.interactive.ractor), 68  
 raise\_badelement() (in module gpkit.constraints.set), 64  
 raise\_elementhasnumpybools() (in module gpkit.constraints.set), 64  
 raise\_exception() (in module gpkit.build), 91  
 recurse\_splits() (in module gpkit.tools.autosweep), 88  
 reltol (gpkit.constraints.tight.Tight attribute), 66  
 replacedir() (in module gpkit.build), 91  
 reset\_varkeys() (gpkit.constraints.costed.CostedConstraintSet method), 59  
 reset\_varkeys() (gpkit.constraints.set.ConstraintSet method), 63  
 results\_table() (in module gpkit.solution\_array), 95  
 right (gpkit.nomials.array.NomialArray attribute), 69  
 rootconstr\_latex() (gpkit.constraints.costed.CostedConstraintSet method), 59  
 rootconstr\_latex() (gpkit.constraints.set.ConstraintSet method), 63  
 rootconstr\_str() (gpkit.constraints.costed.CostedConstraintSet method), 59  
 rootconstr\_str() (gpkit.constraints.set.ConstraintSet method), 63  
 run() (in module gpkit.tests.from\_paths), 76  
 run() (in module gpkit.tests.run\_tests), 77  
 run\_sweep() (in module gpkit.constraints.prog\_factories), 62  
 run\_tests() (in module gpkit.tests.helpers), 76
- S**
- sample\_at() (gpkit.tools.autosweep.BinarySweepTree method), 88  
 save() (gpkit.tools.spdata.SPData method), 89  
 ScalarSingleEquationConstraint (class in gpkit.nomials.nomial\_math), 72  
 sens\_from\_dual() (gpkit.constraints.bounded.Bounded method), 58  
 sens\_from\_dual() (gpkit.constraints.set.ConstraintSet method), 64  
 sens\_from\_dual() (gpkit.nomials.nomial\_math.MonomialEquality method), 71  
 sens\_from\_dual() (gpkit.nomials.nomial\_math.PosynomialInequality method), 72  
 SensitivityMap (class in gpkit.interactive.sensitivity\_map), 68  
 senss\_table() (in module gpkit.solution\_array), 96  
 setup() (gpkit.tests.t\_model.Box method), 79  
 setup() (gpkit.tests.t\_model.BoxAreaBounds method), 79  
 setup() (gpkit.tests.t\_model.Thing method), 81  
 showcadtoon() (in module gpkit.interactive.ractor), 68  
 Signomial (class in gpkit.nomials.nomial\_math), 72  
 signomial\_print() (in module gpkit.interactive.sensitivity\_map), 68  
 SignomialEquality (class in gpkit.constraints.sigeq), 64  
 SignomialInequality (class in gpkit.nomials.nomial\_math), 73



- SignomialProgram (class in gpkit.constraints.signomial\_program), 64
- SignomialsEnabled (class in gpkit), 97
- simplify\_exps\_and\_cs() (in module gpkit.nomials.data), 70
- SingleEquationConstraint (class in gpkit.constraints.single\_equation), 65
- SingleSignomialEquality (class in gpkit.nomials.nomial\_math), 73
- solarray (gpkit.tools.autosweep.BinarySweepTree attribute), 88
- solarray (gpkit.tools.autosweep.SolutionOracle attribute), 88
- sollist (gpkit.tools.autosweep.BinarySweepTree attribute), 88
- solution (gpkit.constraints.model.Model attribute), 61
- solution (gpkit.interactive.sensitivity\_map.SensitivityMap attribute), 68
- SolutionArray (class in gpkit.solution\_array), 94
- SolutionOracle (class in gpkit.tools.autosweep), 88
- solve() (gpkit.constraints.geometric\_program.GeometricProgram method), 60
- solve() (gpkit.constraints.model.Model method), 61
- solver (gpkit.tests.t\_model.TestGP attribute), 79
- solver (gpkit.tests.t\_model.TestSP attribute), 81
- SolverBackend (class in gpkit.build), 91
- SolverLog (class in gpkit.small\_classes), 94
- sp() (gpkit.constraints.model.Model method), 61
- SPData (class in gpkit.tools.spdata), 89
- StdoutCaptured (class in gpkit.tests.helpers), 76
- str\_without() (gpkit.constraints.set.ConstraintSet method), 64
- str\_without() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 66
- str\_without() (gpkit.nomials.array.NomialArray method), 69
- str\_without() (gpkit.nomials.nomial\_core.Nomial method), 71
- str\_without() (gpkit.varkey.VarKey method), 97
- sub (gpkit.nomials.nomial\_core.Nomial attribute), 71
- sub() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 66
- sub() (gpkit.nomials.array.NomialArray method), 69
- sub() (gpkit.nomials.nomial\_math.Signomial method), 73
- sub() (gpkit.nomials.variables.Variable method), 75
- subconstr\_latex() (gpkit.constraints.model.Model method), 62
- subconstr\_latex() (gpkit.constraints.set.ConstraintSet method), 64
- subconstr\_latex() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 66
- subconstr\_str() (gpkit.constraints.model.Model method), 62
- subconstr\_str() (gpkit.constraints.set.ConstraintSet method), 64
- subconstr\_str() (gpkit.constraints.single\_equation.SingleEquationConstraint method), 66
- subinplace() (gpkit.constraints.array.ArrayConstraint method), 57
- subinplace() (gpkit.constraints.costed.CostedConstraintSet method), 59
- subinplace() (gpkit.constraints.set.ConstraintSet method), 64
- subinplace() (gpkit.nomials.nomial\_math.ScalarSingleEquationConstraint method), 72
- subinplace() (gpkit.nomials.nomial\_math.Signomial method), 73
- subinto() (gpkit.solution\_array.SolutionArray method), 95
- subscripts (gpkit.varkey.VarKey attribute), 97
- substitution() (in module gpkit.nomials.substitution), 74
- sum() (gpkit.nomials.array.NomialArray method), 69
- sum() (gpkit.nomials.nomial\_core.Nomial method), 71
- summary() (gpkit.solution\_array.SolutionArray method), 95
- sweep() (gpkit.constraints.model.Model method), 62
- ## T
- table() (gpkit.solution\_array.SolutionArray method), 95
- table\_titles (gpkit.solution\_array.SolutionArray attribute), 95
- te\_exp\_minus1() (in module gpkit.tools.tools), 89
- te\_secant() (in module gpkit.tools.tools), 89
- te\_tangent() (in module gpkit.tools.tools), 90
- test (in module gpkit.tests.t\_model), 81
- test\_601() (gpkit.tests.t\_model.TestGP method), 80
- test\_additive\_constants() (gpkit.tests.t\_model.TestGP method), 80
- test\_additive\_scalar() (gpkit.tests.t\_constraints.TestConstraint method), 77
- test\_additive\_scalar\_gt1() (gpkit.tests.t\_constraints.TestConstraint method), 77
- test\_array\_mult() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81
- test\_autosweep() (gpkit.tests.t\_examples.TestExamples method), 78
- test\_bad\_elements() (gpkit.tests.t\_constraints.TestConstraint method), 77
- test\_bad\_gp\_sub() (gpkit.tests.t\_sub.TestModelSubs method), 84
- test\_bad\_subinplace() (gpkit.tests.t\_sub.TestModelSubs method), 84
- test\_basic() (gpkit.tests.t\_sub.TestNomialSubs method), 85

test\_beam() (gpkit.tests.t\_examples.TestExamples method), 78

test\_binary\_sweep\_tree() (gpkit.tests.t\_tools.TestTools method), 85

test\_calconst() (gpkit.tests.t\_sub.TestModelSubs method), 84

test\_call() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84

test\_call\_units() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84

test\_call\_vector() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84

test\_composite\_objective() (gpkit.tests.t\_tools.TestTools method), 85

test\_constants\_in\_objective\_1() (gpkit.tests.t\_model.TestGP method), 80

test\_constants\_in\_objective\_2() (gpkit.tests.t\_model.TestGP method), 80

test\_constraint\_creation\_units() (gpkit.tests.t\_vars.TestVectorVariable method), 86

test\_constraint\_gen() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81

test\_constraint\_gen() (gpkit.tests.t\_nomials.TestPosynomial method), 82

test\_constraintget() (gpkit.tests.t\_constraints.TestConstraint method), 77

test\_cost\_freeing() (gpkit.tests.t\_model.TestGP method), 80

test\_cvxopt\_kwargs() (gpkit.tests.t\_model.TestModelSolverSpecific method), 80

test\_debug() (gpkit.tests.t\_examples.TestExamples method), 78

test\_dict\_key() (gpkit.tests.t\_vars.TestVarKey method), 86

test\_dictlike() (gpkit.tests.t\_keydict.TestKeyDict method), 79

test\_diff() (gpkit.tests.t\_nomials.TestPosynomial method), 82

test\_dimensionless\_units() (gpkit.tests.t\_sub.TestNomialSubs method), 85

test\_div() (gpkit.tests.t\_nomials.TestMonomial method), 82

test\_elementwise\_mult() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81

test\_empty() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81

test\_eq() (gpkit.tests.t\_nomials.TestPosynomial method), 82

test\_eq\_ne() (gpkit.tests.t\_nomials.TestMonomial method), 82

test\_eq\_ne() (gpkit.tests.t\_nomials.TestSignomial method), 83

test\_eq\_neq() (gpkit.tests.t\_vars.TestVarKey method), 86

test\_eq\_units() (gpkit.tests.t\_nomials.TestPosynomial method), 83

test\_equality\_relaxation() (gpkit.tests.t\_constraints.TestConstraint method), 77

test\_evalfn() (gpkit.tests.t\_constraints.TestConstraint method), 77

test\_exps\_is\_tuple() (gpkit.tests.t\_model.TestGP method), 80

test\_external\_sp() (gpkit.tests.t\_examples.TestExamples method), 78

test\_external\_sp2() (gpkit.tests.t\_examples.TestExamples method), 78

test\_failed\_getattr() (gpkit.tests.t\_keydict.TestKeyDict method), 79

test\_fmincon\_generator() (gpkit.tests.t\_tools.TestTools method), 85

test\_getattr() (gpkit.tests.t\_keydict.TestKeyDict method), 79

test\_getitem() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81

test\_getkey() (gpkit.tests.t\_sub.TestModelSubs method), 84

test\_hash() (gpkit.tests.t\_vars.TestVariable method), 86

test\_inheritance() (gpkit.tests.t\_constraints.TestMonomialEquality method), 77

test\_init() (gpkit.tests.t\_constraints.TestConstraint method), 77

test\_init() (gpkit.tests.t\_constraints.TestMonomialEquality method), 77

test\_init() (gpkit.tests.t\_constraints.TestSignomialInequality method), 78

test\_init() (gpkit.tests.t\_nomials.TestMonomial method), 82

test\_init() (gpkit.tests.t\_nomials.TestPosynomial method), 83

test\_init() (gpkit.tests.t\_nomials.TestSignomial method), 83

test\_init() (gpkit.tests.t\_small.TestHashVector method), 83

test\_init() (gpkit.tests.t\_vars.TestVariable method), 86

test\_init() (gpkit.tests.t\_vars.TestVarKey method), 86

test\_init() (gpkit.tests.t\_vars.TestVectorVariable method), 86

test\_initially\_infeasible() (gpkit.tests.t\_model.TestSP method), 81

test\_integer\_division() (gpkit.tests.t\_nomials.TestPosynomial method), 83



83  
test\_is\_vector\_variable() (gpkit.tests.t\_vars.TestArrayVariable method), 86  
test\_issue180() (gpkit.tests.t\_model.TestSP method), 81  
test\_key\_options() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84  
test\_latex() (gpkit.tests.t\_nomials.TestMonomial method), 82  
test\_left\_right() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 81  
test\_mdd\_example() (gpkit.tests.t\_model.TestGP method), 80  
test\_model\_composition\_units() (gpkit.tests.t\_sub.TestModelSubs method), 84  
test\_model\_recursion() (gpkit.tests.t\_sub.TestModelSubs method), 84  
test\_model\_var\_access() (gpkit.tests.t\_examples.TestExamples method), 78  
test\_modelname\_added() (gpkit.tests.t\_model.TestModelNoSolve method), 80  
test\_mono\_lower\_bound() (gpkit.tests.t\_nomials.TestPosynomial method), 83  
test\_mul() (gpkit.tests.t\_nomials.TestMonomial method), 82  
test\_mul\_add() (gpkit.tests.t\_small.TestHashVector method), 83  
test\_nan\_printing() (gpkit.tests.t\_solution\_array.TestResultsTable method), 84  
test\_ndim() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82  
test\_neg() (gpkit.tests.t\_small.TestHashVector method), 83  
test\_no\_naming\_on\_var\_access() (gpkit.tests.t\_model.TestModelNoSolve method), 80  
test\_non\_monomial() (gpkit.tests.t\_constraints.TestMonomialEquality method), 77  
test\_numeric() (gpkit.tests.t\_sub.TestNomialSubs method), 85  
test\_numerical\_precision() (gpkit.tests.t\_nomials.TestMonomial method), 82  
test\_oper\_overload() (gpkit.tests.t\_constraints.TestConstraint method), 77  
test\_outer() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82  
test\_partial\_sub\_signomial() (gpkit.tests.t\_model.TestSP method), 81  
test\_performance\_modeling() (gpkit.tests.t\_examples.TestExamples method), 78  
test\_persistence() (gpkit.tests.t\_sub.TestModelSubs method), 84  
test\_phantoms() (gpkit.tests.t\_sub.TestModelSubs method), 84  
test\_pint\_366() (gpkit.tests.t\_small.TestSmallScripts method), 83  
test\_posy\_simplification() (gpkit.tests.t\_model.TestGP method), 80  
test\_posyconstr\_in\_gp() (gpkit.tests.t\_constraints.TestTight method), 78  
test\_posyconstr\_in\_sp() (gpkit.tests.t\_constraints.TestTight method), 78  
test\_posyposy\_mult() (gpkit.tests.t\_nomials.TestPosynomial method), 83  
test\_posyslt1() (gpkit.tests.t\_constraints.TestSignomialInequality method), 78  
test\_pow() (gpkit.tests.t\_nomials.TestMonomial method), 82  
test\_pow() (gpkit.tests.t\_small.TestHashVector method), 83  
test\_primal\_infeasible\_ex1() (gpkit.tests.t\_examples.TestExamples method), 78  
test\_primal\_infeasible\_ex2() (gpkit.tests.t\_examples.TestExamples method), 78  
test\_prod() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82  
test\_quantity\_sub() (gpkit.tests.t\_sub.TestModelSubs method), 84  
test\_relaxation() (gpkit.tests.t\_examples.TestExamples method), 78  
test\_relaxation() (gpkit.tests.t\_model.TestSP method), 81  
test\_repo() (in module gpkit.tests.test\_repo), 87  
test\_repos() (in module gpkit.tests.test\_repo), 87  
test\_repr() (gpkit.tests.t\_nomials.TestMonomial method), 82  
test\_repr() (gpkit.tests.t\_vars.TestVarKey method), 86  
test\_result\_access() (gpkit.tests.t\_solution\_array.TestResultsTable method), 84  
test\_scalar\_units() (gpkit.tests.t\_sub.TestNomialSubs method), 85  
test\_sensitivities() (gpkit.tests.t\_model.TestGP method), 80  
test\_setattr() (gpkit.tests.t\_keydict.TestKeyDict method),

- 79
- test\_shape() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82
- test\_shape() (gpkit.tests.t\_small.TestCootMatrix method), 83
- test\_shapes() (gpkit.tests.t\_vars.TestVectorize method), 87
- test\_sigconstr\_in\_sp() (gpkit.tests.t\_constraints.TestTight method), 78
- test\_sigeq() (gpkit.tests.t\_model.TestGP method), 80
- test\_signomial() (gpkit.tests.t\_sub.TestNomialSubs method), 85
- test\_sigs\_not\_allowed\_in\_cost() (gpkit.tests.t\_model.TestSP method), 81
- test\_simple\_box() (gpkit.tests.t\_examples.TestExamples method), 78
- test\_simple\_sp() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_simple\_united\_gp() (gpkit.tests.t\_model.TestGP method), 80
- test\_simpleflight() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_simplification() (gpkit.tests.t\_nomials.TestPosynomial method), 83
- test\_sin\_approx\_example() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_singular() (gpkit.tests.t\_model.TestGP method), 80
- test\_skipfailures() (gpkit.tests.t\_sub.TestModelSubs method), 84
- test\_small\_named\_signomial() (gpkit.tests.t\_model.TestSP method), 81
- test\_sp\_bounded() (gpkit.tests.t\_model.TestSP method), 81
- test\_sp\_initial\_guess\_sub() (gpkit.tests.t\_model.TestSP method), 81
- test\_sp\_substitutions() (gpkit.tests.t\_model.TestSP method), 81
- test\_str() (gpkit.tests.t\_constraints.TestMonomialEquality method), 77
- test\_str() (gpkit.tests.t\_vars.TestArrayVariable method), 86
- test\_str\_with\_units() (gpkit.tests.t\_nomials.TestMonomial method), 82
- test\_string\_mutation() (gpkit.tests.t\_sub.TestNomialSubs method), 85
- test\_subinto() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84
- test\_substitution() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82
- test\_substitution\_issue905() (gpkit.tests.t\_constraints.TestBounded method), 85
- 77
- test\_sum() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82
- test\_table() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84
- test\_te\_exp\_minus1() (gpkit.tests.t\_tools.TestTools method), 85
- test\_te\_secant() (gpkit.tests.t\_tools.TestTools method), 85
- test\_te\_tangent() (gpkit.tests.t\_tools.TestTools method), 85
- test\_terminating\_constant\_() (gpkit.tests.t\_model.TestGP method), 80
- test\_to() (gpkit.tests.t\_vars.TestVariable method), 86
- test\_trivial\_gp() (gpkit.tests.t\_model.TestGP method), 80
- test\_trivial\_sp() (gpkit.tests.t\_model.TestSP method), 81
- test\_trivial\_sp2() (gpkit.tests.t\_model.TestSP method), 81
- test\_trivial\_vector\_gp() (gpkit.tests.t\_model.TestGP method), 80
- test\_unbounded() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_unbounded\_debugging() (gpkit.tests.t\_model.TestSP method), 81
- test\_unit\_parsing() (gpkit.tests.t\_vars.TestVariable method), 86
- test\_united\_sub\_sweep() (gpkit.tests.t\_sub.TestModelSubs method), 84
- test\_unitless\_monomial\_sub() (gpkit.tests.t\_sub.TestNomialSubs method), 85
- test\_units() (gpkit.tests.t\_nomial\_array.TestNomialArray method), 82
- test\_units() (gpkit.tests.t\_nomials.TestMonomial method), 82
- test\_units\_attr() (gpkit.tests.t\_vars.TestVarKey method), 86
- test\_units\_sub() (gpkit.tests.t\_solution\_array.TestSolutionArray method), 84
- test\_unitsstr() (gpkit.tests.t\_small.TestSmallScripts method), 84
- test\_value() (gpkit.tests.t\_vars.TestVariable method), 86
- test\_values\_vs\_subs() (gpkit.tests.t\_model.TestSP method), 81
- test\_variable() (gpkit.tests.t\_sub.TestNomialSubs method), 85
- test\_vector() (gpkit.tests.t\_keydict.TestKeyDict method), 79
- test\_vector() (gpkit.tests.t\_sub.TestNomialSubs method), 85
- test\_vector\_init() (gpkit.tests.t\_sub.TestModelSubs method), 85
- test\_vector\_sub() (gpkit.tests.t\_sub.TestModelSubs method), 85
- test\_vector\_sweep() (gpkit.tests.t\_sub.TestModelSubs method), 85

- test\_vectorize() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_water\_tank() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_x\_greaterthan\_1() (gpkit.tests.t\_examples.TestExamples method), 79
- test\_zero\_lower\_unbounded() (gpkit.tests.t\_model.TestGP method), 80
- test\_zeroing() (gpkit.tests.t\_model.TestGP method), 80
- TestArrayVariable (class in gpkit.tests.t\_vars), 86
- TestBounded (class in gpkit.tests.t\_constraints), 77
- testcase (in module gpkit.tests.t\_model), 81
- TestConstraint (class in gpkit.tests.t\_constraints), 77
- TestCootMatrix (class in gpkit.tests.t\_small), 83
- TestExamples (class in gpkit.tests.t\_examples), 78
- TestFiles (class in gpkit.tests.from\_paths), 75
- TestGP (class in gpkit.tests.t\_model), 79
- TestHashVector (class in gpkit.tests.t\_small), 83
- TestKeyDict (class in gpkit.tests.t\_keydict), 79
- TestModelNoSolve (class in gpkit.tests.t\_model), 80
- TestModelSolverSpecific (class in gpkit.tests.t\_model), 80
- TestModelSubs (class in gpkit.tests.t\_sub), 84
- TestMonomial (class in gpkit.tests.t\_nomials), 82
- TestMonomialEquality (class in gpkit.tests.t\_constraints), 77
- TestNomialArray (class in gpkit.tests.t\_nomial\_array), 81
- TestNomialSubs (class in gpkit.tests.t\_sub), 85
- TestPosynomial (class in gpkit.tests.t\_nomials), 82
- TestResultsTable (class in gpkit.tests.t\_solution\_array), 84
- TestSignomial (class in gpkit.tests.t\_nomials), 83
- TestSignomialInequality (class in gpkit.tests.t\_constraints), 77
- TestSmallScripts (class in gpkit.tests.t\_small), 83
- TestSolutionArray (class in gpkit.tests.t\_solution\_array), 84
- TestSP (class in gpkit.tests.t\_model), 80
- TestTight (class in gpkit.tests.t\_constraints), 78
- TestTools (class in gpkit.tests.t\_tools), 85
- TestVariable (class in gpkit.tests.t\_vars), 86
- TestVarKey (class in gpkit.tests.t\_vars), 86
- TestVectorize (class in gpkit.tests.t\_vars), 86
- TestVectorVariable (class in gpkit.tests.t\_vars), 86
- Thing (class in gpkit.tests.t\_model), 81
- Tight (class in gpkit.constraints.tight), 66
- to() (gpkit.nomials.nomial\_core.Nomial method), 71
- to() (gpkit.nomials.variables.Variable method), 75
- to\_united\_array() (gpkit.small\_classes.DictOfLists method), 94
- tocoo() (gpkit.small\_classes.CootMatrix method), 93
- tocsc() (gpkit.small\_classes.CootMatrix method), 93
- tocsr() (gpkit.small\_classes.CootMatrix method), 93
- todense() (gpkit.small\_classes.CootMatrix method), 93
- todia() (gpkit.small\_classes.CootMatrix method), 93
- todok() (gpkit.small\_classes.CootMatrix method), 93
- topsenss\_filter() (in module gpkit.solution\_array), 96
- topsenss\_table() (in module gpkit.solution\_array), 96
- topvar() (gpkit.constraints.set.ConstraintSet method), 64
- try\_str\_without() (in module gpkit.small\_scripts), 94
- trycall() (in module gpkit.constraints.single\_equation), 66
- ## U
- unique\_varkeys (gpkit.constraints.set.ConstraintSet attribute), 64
- units (gpkit.nomials.array.NomialArray attribute), 69
- unitstr() (gpkit.nomials.nomial\_core.Nomial method), 71
- unitstr() (gpkit.varkey.VarKey method), 97
- unitstr() (in module gpkit.repr\_conventions), 93
- update() (gpkit.keydict.KeyDict method), 92
- update() (gpkit.keydict.KeySet method), 92
- update\_keymap() (gpkit.keydict.KeyDict method), 92
- ## V
- value (gpkit.nomials.nomial\_core.Nomial attribute), 71
- values (gpkit.nomials.data.NomialData attribute), 70
- Variable (class in gpkit.nomials.variables), 74
- variables\_byname() (gpkit.constraints.set.ConstraintSet method), 64
- VarKey (class in gpkit.varkey), 96
- varkey\_bounds() (in module gpkit.constraints.bounded), 58
- varkeys (gpkit.constraints.set.ConstraintSet attribute), 64
- varkeys (gpkit.nomials.data.NomialData attribute), 70
- veckeved() (in module gpkit.small\_scripts), 94
- VectorizableVariable (class in gpkit.nomials.variables), 75
- Vectorize (class in gpkit), 97
- vectorize() (gpkit.nomials.array.NomialArray method), 69
- ## W
- write() (gpkit.small\_classes.SolverLog method), 94
- write() (gpkit.tests.helpers.NullFile method), 76
- ## Z
- zero\_lower\_unbounded\_variables() (gpkit.constraints.model.Model method), 62