**Contents**

1 Geometric Programming 101  
1.1 What is a GP?  
1.2 Why are GPs special?  
1.3 What are Signomials / Signomial Programs?  
1.4 Where can I learn more?  

2 Installation  
2.1 Installing MOSEK  
2.2 Debugging your installation  
2.3 Bleeding-edge installations  

3 Getting Started  
3.1 Declaring Variables  
3.2 Creating Monomials and Posynomials  
3.3 Declaring Constraints  
3.4 Formulating a Model  
3.5 Solving the Model  
3.6 Printing Results  
3.7 Sensitivities and dual variables  

4 Debugging Models  
4.1 Potential errors and warnings  
4.2 Dual Infeasibility  
4.3 Primal Infeasibility  

5 Visualization and Interaction  
5.1 Sankey Diagrams  
5.2 Plotting a 1D Sweep  

6 Building Complex Models  
6.1 Checking for result changes  
6.2 Inheriting from Model  
6.3 Accessing Variables in Models  
6.4 Vectorization  
6.5 Multipoint analysis modeling  

7 Advanced Commands  

---

1 Geometric Programming 101  
1.1 What is a GP?  
1.2 Why are GPs special?  
1.3 What are Signomials / Signomial Programs?  
1.4 Where can I learn more?  

2 Installation  
2.1 Installing MOSEK  
2.2 Debugging your installation  
2.3 Bleeding-edge installations  

3 Getting Started  
3.1 Declaring Variables  
3.2 Creating Monomials and Posynomials  
3.3 Declaring Constraints  
3.4 Formulating a Model  
3.5 Solving the Model  
3.6 Printing Results  
3.7 Sensitivities and dual variables  

4 Debugging Models  
4.1 Potential errors and warnings  
4.2 Dual Infeasibility  
4.3 Primal Infeasibility  

5 Visualization and Interaction  
5.1 Sankey Diagrams  
5.2 Plotting a 1D Sweep  

6 Building Complex Models  
6.1 Checking for result changes  
6.2 Inheriting from Model  
6.3 Accessing Variables in Models  
6.4 Vectorization  
6.5 Multipoint analysis modeling  

7 Advanced Commands
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Derived Variables</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Sweeps</td>
<td>38</td>
</tr>
<tr>
<td>7.3</td>
<td>Tight ConstraintSets</td>
<td>39</td>
</tr>
<tr>
<td>7.4</td>
<td>Loose ConstraintSets</td>
<td>40</td>
</tr>
<tr>
<td>7.5</td>
<td>Substitutions</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Signomial Programming</td>
<td>43</td>
</tr>
<tr>
<td>8.1</td>
<td>Example Usage</td>
<td>43</td>
</tr>
<tr>
<td>8.2</td>
<td>Sequential Geometric Programs</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>Examples</td>
<td>47</td>
</tr>
<tr>
<td>9.1</td>
<td>iPython Notebook Examples</td>
<td>47</td>
</tr>
<tr>
<td>9.2</td>
<td>A Trivial GP</td>
<td>47</td>
</tr>
<tr>
<td>9.3</td>
<td>Maximizing the Volume of a Box</td>
<td>48</td>
</tr>
<tr>
<td>9.4</td>
<td>Water Tank</td>
<td>49</td>
</tr>
<tr>
<td>9.5</td>
<td>Simple Wing</td>
<td>50</td>
</tr>
<tr>
<td>9.6</td>
<td>Simple Beam</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>Glossary</td>
<td>57</td>
</tr>
<tr>
<td>10.1</td>
<td>gpkit package</td>
<td>57</td>
</tr>
<tr>
<td>11</td>
<td>Citing GPkit</td>
<td>61</td>
</tr>
<tr>
<td>12</td>
<td>Acknowledgements</td>
<td>63</td>
</tr>
<tr>
<td>13</td>
<td>Release Notes</td>
<td>65</td>
</tr>
</tbody>
</table>
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.
1.1 What is a GP?

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

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

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

\[ f(x) = c x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n} \]

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:

\[ g(x) = \sum_{k=1}^{K} c_k x_1^{a_{1,k}} x_2^{a_{2,k}} \ldots x_n^{a_{n,k}} \]

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:

\[
\begin{align*}
\text{minimize} & \quad g_0(x) \\
\text{subject to} & \quad f_i(x) = 1, \quad i = 1, \ldots, m \\
& \quad g_i(x) \leq 1, \quad i = 1, \ldots, n
\end{align*}
\]

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

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

1.2 Why are GPs special?

Geometric programs have several powerful properties:

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

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

1.3 What are Signomials / Signomial Programs?

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

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

1.4 Where can I learn more?

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

- A tutorial on geometric programming, by S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi.
- Convex optimization, by S. Boyd and L. Vandenberghe.
CHAPTER 2

Installation

1. If you are on Mac or Windows, we recommend installing Anaconda. Alternatively, install pip and create a virtual environment.

2. (optional) Install the MOSEK solver as directed below

3. Run `pip install gpkit` in the appropriate terminal or command prompt.

4. Open a Python prompt and run `import gpkit` to finish installation and run unit tests.

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

2.1 Installing MOSEK

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

Mac OS X

- If `which gcc` does not return anything, install the Apple Command Line Tools.

- Download MOSEK 8, then:
  - Move the `mosek` folder to your home directory
  - Follow these steps for Mac.
  - Request an academic license file and put it in `~/mosek/`

Linux

- Download MOSEK 8, then:
  - Move the `mosek` folder to your home directory
  - Follow these steps for Linux.
  - Request an academic license file and put it in `~/mosek/`
Windows

- **Download MOSEK 8**, then:
  - Follow these steps for Windows.
  - Request an academic license file and put it in C:\Users\(your_username)\mosek\n  - Make sure gcc is on your system path.
    * To do this, type gcc into a command prompt.
    * If you get executable not found, then install the 64-bit version (x86_64 installer architecture dropdown option) with GCC version 6.4.0 or older of mingw.
    * In an Anaconda command prompt (or equivalent), run cd C:\Program Files\mingw-w64\x86_64-6.4.0-posix-seh-rt_v5-rev0\ (or whatever corresponds to the correct installation directory; note that if mingw is in Program Files (x86) instead of Program Files you've installed the 32-bit version by mistake)
    * Run mingw-64 to add it to your executable path. For step 3 of the install process you'll need to run pip install gpkit from this prompt.

2.2 Debugging your installation

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

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

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

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

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

2.3 Bleeding-edge installations

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

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

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

### 3.1 Declaring Variables

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

#### 3.1.1 Free Variables

```python
# 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")
```

#### 3.1.2 Fixed Variables

To declare a variable with a constant value, use the `Variable` class, as above, but put a number before the units:
# Declare \( \rho \) equal to 1.225 kg/m\(^3\).
# NOTE: in python string literals, backslashes must be doubled
rho = Variable("\rho", 1.225, "kg/m\(^3\)", "Density of air at sea level")

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

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

## 3.1.3 Vector Variables

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

```python
# 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")
```

## 3.2 Creating Monomials and Posynomials

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

```python
# create a Monomial term \( xy^2/z \)
x = Variable("x")
y = Variable("y")
z = Variable("z")
m = x * y**2 / z
```

```python
# create a Posynomial expression \( x + xy^2 \)
x = Variable("x")
y = Variable("y")
p = x + x * y**2
```

## 3.3 Declaring Constraints

Constraint objects represent constraints of the form Monomial \( \geq \) Posynomial or Monomial \( \leq \) Monomial (which are the forms required for GP-compatibility).

Note that constraints must be formed using \( \leq \), \( \geq \), or \( \leq \) operators, not \( < \) or \( > \).

```python
# 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)
```

```python
```

```python
```
3.4 Formulating a Model

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

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

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

3.5 Solving the Model

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

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

3.6 Printing Results

The solution object can represent itself as a table:

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

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.59 [1/m**3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Free Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x : 0.5774 [m]</td>
</tr>
<tr>
<td>y : 0.2887 [m]</td>
</tr>
<tr>
<td>z : 0.3849 [m]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>S : 1 [m**2]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sensitivities</th>
</tr>
</thead>
<tbody>
<tr>
<td>S : -1.5</td>
</tr>
</tbody>
</table>

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

```python
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.

3.7 Sensitivities and dual variables

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

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

3.7.1 Using variable sensitivities

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

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

```python
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"] ["variables"] [x_squared_min] == 2
```
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 \( m \texttt{.debug()} \), which will search for changes that would make the model feasible:

```python
"Debug examples"
from gpkit import Variable, Model, units

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

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

print("# Now let's try a model unsolvable with relaxed constants
")

m2 = Model(x, [x <= units("inch"), x >= units("yard")])
m2.debug()

print("# And one that's only unbounded
")

# the value of x_min was used up in the previous model!
x_min = Variable("x_min", 2, "ft")
m3 = Model(x/y, [x >= x_min])
m3.debug()
```
< DEBUGGING >
> Trying with bounded variables and relaxed constants:

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

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

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

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

Solves with these constraints relaxed:
  1: 3500% relaxed, from x [ft] >= 1 [yd]
      to 36·x [ft] >= 1 [yd]

>> Success!
# And one that's only unbounded

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

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

>> Success!

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

### 4.1 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 Run**

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

* **RuntimeWarning**: Primal solution violates constraint: 1.000149786 is greater than 1

  - 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.

## 4.2 Dual Infeasibility

In some cases a model will not solve because the optimal value of one or more variables is 0 or infinity (negative or positive infinity in logspace). Such a problem is dual infeasible because the GP’s dual problem, which determines the optimal values of the 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 slide off to the bounds, notes this in the solution dictionary and prints a warning (if verbosity is greater than 0).

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

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

x = Variable("x")
constraints = [x >= 1]

m = Model(1/x, constraints)  # MOSEK returns DUAL_INFEAS_CER on .solve()
m = Model(1/x, Bounded(constraints))
# by default, prints bounds warning during solve
sol = m.solve(verbosity=0)
print(sol.summary())
# but they can also be accessed from the solution:
assert (sol["boundedness"]['value near upper bound']
  == sol["boundedness"]['sensitive to upper bound'])
```

Upon viewing the printed output,

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

(continues on next page)
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.

## 4.3 Primal Infeasibility

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

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

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

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

# raises UnknownInfeasible on cvxopt, PrimalInfeasible on mosek
# m.solve()
```

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

x = Variable("x")
y = Variable("y", 2)

constraints = [
    x >= 1,
    0.5 <= x*y,
    x*y <= 1.5
]
```

(continues on next page)
Since $y$ is now set to 2 and $x$ can be no less than 1, it is again impossible for $x \cdot y$ 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.

### 4.3.1 Relaxation

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

```
"Relaxation examples"

from gpkit import Variable, Model
x = Variable("x")
x_min = Variable("x_min", 2)
x_max = Variable("x_max", 1)
m = Model(x, [x <= x_max, x >= x_min])
print("Original model")
print("=") # m.solve() # raises a RuntimeWarning!
print("With constraints relaxed equally")
print("=") from gpkit.constraints.relax import ConstraintsRelaxedEqually
allrelaxed = ConstraintsRelaxedEqually(m)
mr1 = Model(allrelaxed.relaxvar, allrelaxed)
print(mr1)
print(mr1.solve(verbosity=0).table()) # solves with an x of 1.414
print(""
print("With constraints relaxed individually")
print("=") from gpkit.constraints.relax import ConstraintsRelaxed
constraintsrelaxed = ConstraintsRelaxed(m)
mr2 = Model(constraintsrelaxed.relaxvars.prod() * m.cost**0.01,
            # add a bit of the original cost in constraintsrelaxed)
print(mr2)
print(mr2.solve(verbosity=0).table()) # solves with an x of 1.0
print(""
print("With constants relaxed individually")
print("=")
```

(continues on next page)
from gpkit.constraints.relax import ConstantsRelaxed
constantsrelaxed = ConstantsRelaxed(m)
m3 = Model(constantsrelaxed.relaxvars.prod() * m.cost**0.01,
    # add a bit of the original cost in
    constantsrelaxed)
print(m3)
print(m3.solve(verbosity=0).table())  # brings x_min down to 1.0
print(''

Original model
==============

Cost-----
x

Constraints-------------
  x <= x_max
  x >= x_min

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

Cost-----
  C

Constraints-------------
  "minimum relaxation":
    C >= 1
  "relaxed constraints":
    x <= C·x_max
    x_min <= C·x

Optimal Cost--------------
    1.414

Free Variables--------------
    x : 1.414
        | Relax
    C : 1.414

Fixed Variables-------------
    x_max : 1
    x_min : 2

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

(continues on next page)
Most Sensitive Constraints
--------------------------
+0.5 : x <= C·x_max
+0.5 : x_min <= C·x

With constraints relaxed individually
=======================================

Cost
----
C[:].prod()·x^0.01

Constraints
----------
"minimum relaxation":
C[:] >= 1
"relaxed constraints":
  x <= C[0]·x_max
  x_min <= C[1]·x

Optimal Cost
------------
2

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

  | Relax1
  C : [ 1 2 ]

Fixed Variables
---------------
  x_max : 1
  x_min : 2

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

Most Sensitive Constraints
--------------------------
  +1 : x_min <= C[1]·x
  +0.99 : x <= C[0]·x_max
  +0.01 : C[0] >= 1

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

Cost
----
[Relax2.x_max, Relax2.x_min].prod()·x^0.01

(continues on next page)
Constraints
----------

Relax2
"original constraints":
  x <= x_max
  x >= x_min
"relaxation constraints":
"x_max":
  Relax2.x_max >= 1
  x_max >= Relax2.OriginalValues.x_max/Relax2.x_max
  x_max <= Relax2.OriginalValues.x_max·Relax2.x_max
"x_min":
  Relax2.x_min >= 1
  x_min >= Relax2.OriginalValues.x_min/Relax2.x_min
  x_min <= Relax2.OriginalValues.x_min·Relax2.x_min

Optimal Cost
-----------
2

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

  | Relax2
  x_max : 1
  x_min : 2

Fixed Variables
---------------
  | Relax2.OriginalValues
  x_max : 1
  x_min : 2

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

Most Sensitive Constraints
--------------------------
  +1 : x >= x_min
  +1 : x_min >= Relax2.OriginalValues.x_min/Relax2.x_min
  +0.99 : x <= x_max
  +0.99 : x_max <= Relax2.OriginalValues.x_max·Relax2.x_max
CHAPTER 5

Visualization and Interaction

5.1 Sankey Diagrams

5.1.1 Requirements

• jupyter notebook
• ipysankeywidget

5.1.2 Example

Code in this section uses the CE solar model

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

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

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

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

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

(continues on next page)
5.1.3 Explanation

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

5.1.4 Usage

Variables

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

Free

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

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

Fixed

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

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

Equivalent Variables

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

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

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

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

5.1.5 Syntax

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

5.2 Plotting a 1D Sweep

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

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

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,
    Tight([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")
sol.save()

# 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:
5.2. Plotting a 1D Sweep

Autoswept (7 points)
Guaranteed to be in blue region

Cost [m^2]

Swept Variable [m]

1
2
3

2.4
2.2
2.0
CHAPTER 6

Building Complex Models

6.1 Checking for result changes

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

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

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

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

6.2 Inheriting from Model

GPkit encourages an object-oriented modeling approach, where the modeler creates objects that inherit from Model to break large systems down into subsystems and analysis domains. The benefits of this approach include modularity, reusability, and the ability to more closely follow mental models of system hierarchy. For example: two different models for a simple beam, designed by different modelers, should be able to be used interchangeably inside another subsystem (such as an aircraft wing) without either modeler having to write specifically with that use in mind.
When you create a class that inherits from Model, write a `.setup()` method to create the model’s variables and return its constraints. `GPkit.Model.__init__` will call that method and automatically add your model’s name and unique ID to any created variables.

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

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

### 6.3 Accessing Variables in Models

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

- using `Model.variablesbyname(key)`. This returns all Variables in the Model, as well as in any submodels, that match the key.
- 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.

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

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

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

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

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

class Motor(Model):
    """Electric motor

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

    Lower Unbounded
    ---------------
    Pmax"
```

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

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

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

Getting the only var 'E': PowerSystem.Battery.E [MJ]
The top-level var 'm': PowerSystem.m [lb]
All the variables 'm': [gpkit.Variable(PowerSystem.Battery.m [lb]), gpkit.
                        →Variable(PowerSystem.Motor.m [lb]), gpkit.Variable(PowerSystem.m [lb])]

6.4 Vectorization

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

"from gpkit/tests/t_vars.py"

def test_shapes(self):
    with gpkit.Vectorize(3):
        with gpkit.Vectorize(5):
            y = gpkit.Variable("y")
            x = gpkit.VectorVariable(2, "x")
            z = gpkit.VectorVariable(7, "z")
            self.assertEqual(y.shape, (5, 3))

(continues on next page)
This allows models written with scalar constraints to be created with vector constraints:

```python
"""Vectorization demonstration"
from gpkit import Model, Variable, Vectorize
class Test(Model):
    """A simple scalar model
    Upper Unbounded
    ---------------
    x
    """
def setup(self):
x = self.x = Variable("x")
    return [x >= 1]

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

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

SCALAR

Optimal Cost
-------------
1

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

Most Sensitive Constraints
--------------------------
  +1 : x >= 1

VECTORIZED

Optimal Cost
-------------
2

Free Variables
--------------

(continues on next page)
6.5 Multipoint analysis modeling

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

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

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

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

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

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

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

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

""
@parse_variables(__doc__, globals())
def setup(self, aircraft, state):
    self.aircraft = aircraft
    self.state = state
    self.wing_aero = aircraft.wing.dynamic(aircraft.wing, state)
```

(continues on next page)
self.perf_models = [self.wing_aero]

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

V = state.V
rho = state.rho

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

return {
    "lift":
        W + Wfuel <= 0.5*rho*CL*S*V**2,
    "fuel burn rate":
        Wburn >= 0.1*D,
    "performance":
        self.perf_models}

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

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

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

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

        return {
            "definition of W":
                W >= sum(c.W for c in self.components),
            "components":
                self.components
        }

dynamic = AircraftP

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

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

""
@parse_variables(__doc__, globals())
def setup(self):
    pass

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

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

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

    ""
def setup(self, aircraft):
    self.aircraft = aircraft
    self.flightstate = FlightState()
    self.aircraftp = aircraft.dynamic(aircraft, self.flightstate)
    self.Wburn = self.aircraftp.Wburn
    self.Wfuel = self.aircraftp.Wfuel
    return {
        "flightstate": self.flightstate,
        "aircraft performance": self.aircraftp
    }

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

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

    Lower Unbounded
    ---------------
    aircraft.W

    ""
def setup(self, aircraft):
    self.aircraft = aircraft
    with Vectorize(4):  # four flight segments
        self.fs = FlightSegment(aircraft)
        Wburn = self.fs.aircraftp.Wburn
        Wfuel = self.fs.aircraftp.Wfuel
        self.takeoff_fuel = Wfuel[0]
    return {
        "definition of Wburn":
        Wfuel[:-1] >= Wfuel[1:] + Wburn[:-1],
    }

6.5. Multipoint analysis modeling
"require fuel for the last leg":
Wfuel[-1] >= Wburn[-1],
"flight segment":
self.fs}

class WingAero(Model):
    """Wing aerodynamics

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

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

Lower Unbounded
--------------
CL, wing.S, state.mu, state.rho, state.V
""
@parse_variables(__doc__, globals())
def setup(self, wing, state):
    self.wing = wing
    self.state = state
    c = wing.c
    A = wing.A
    S = wing.S
    rho = state.rho
    V = state.V
    mu = state.mu

    return {
        "drag model":
            CD >= 0.074/Re**0.2 + CL**2/np.pi/A/e,
        "definition of Re":
            Re == rho*V*c/mu,
        "definition of D":
            D >= 0.5*rho*V**2*CD*S
    }

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

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

Lower Unbounded
--------------
$c, S$

```python
@parse_variables(__doc__, locals(), globals())
def setup(self):
    return
        {"parametrization of wing weight":
            W >= S*rho,
        "definition of mean chord":
            c == (S/A)**0.5}

dynamic = WingAero

class Fuselage(Model):
    """The thing that carries the fuel, engine, and payload
A full model is left as an exercise for the reader.

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

""
@parse_variables(__doc__, locals(), globals())
def setup(self):
    pass
```

AC = Aircraft()
MISSION = Mission(AC)
M = Model(MISSION.takeoff_fuel, [MISSION, AC])
print(M)
sol = M.solve(verbosity=0)
# save solution to some files
sol.savemat()
sol.savecsv()
sol.savetxt()
sol.save("solution.pkl")
# retrieve solution from a file
sol_loaded = pickle.load(open("solution.pkl", "rb"))

vars_of_interest = set(AC.varkeys)
# note that there's two ways to access submodels
assert (MISSION["flight_segment"])["aircraft performance"]
    is MISSION.fs.aircraftp
vars_of_interest.update(MISSION.fs.aircraftp.unique_varkeys)
vars_of_interest.add(M["D"])
print(sol.summary(vars_of_interest))
print(sol.table(tables="loose constraints"))
M.append(MISSION.fs.aircraftp.Wburn >= 0.2*MISSION.fs.aircraftp.wing_aero.D)
sol = M.solve(verbosity=0)
print(sol.diff("solution.pkl", showvars=vars_of_interest, sortbymodel=False))

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

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_{\text{fuel}}[0] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mission</strong></td>
</tr>
<tr>
<td>&quot;definition of ( W_{\text{burn}} ):&quot; ( W_{\text{fuel}}[:-1] \geq W_{\text{fuel}}[1:] + W_{\text{burn}}[:-1] )</td>
</tr>
<tr>
<td>&quot;require fuel for the last leg&quot;: ( W_{\text{fuel}}[3] \geq W_{\text{burn}}[3] )</td>
</tr>
</tbody>
</table>

| FlightSegment |
| AircraftP |
| "fuel burn rate": \( W_{\text{burn}}[:] \geq 0.1 \cdot D[:] \) |
| "lift": \( W_{\text{fuel}}[:] \leq 0.5 \cdot \rho[:] \cdot C_l[:] \cdot S \cdot V[:]^2 \) |
| "performance": \( W_{\text{Aero}} \) |
| "definition of \( D \)": \( D[:] \geq 0.5 \cdot \rho[:] \cdot V[:]^2 \cdot C_D[:] \cdot S \) |
| "definition of \( R_e \)": \( R_e[:] = \rho[:] \cdot V[:] \cdot c/mu[:] \) |
| "drag model": \( C_D[:] \geq 0.074/R_e[:]^{0.2} + C_l[:]^2/\pi A/e[:] \) |

| FlightState |
| (no constraints) |

| Aircraft |
| "definition of \( W \)": \( W_{\text{Aircraft}} \geq W_{\text{Fuselage}} + W_{\text{Wing}} \) |
| "components": |
| Fuselage |
| (no constraints) |

| Wing |
| "definition of mean chord": \( c = (S/A)^{0.5} \) |
| "parametrization of wing weight": \( W_{\text{Aircraft.Wing}} \geq S \cdot W_{\text{Aircraft.Wing.rho}} \) |

<table>
<thead>
<tr>
<th>Optimal Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.091</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Free Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft</td>
</tr>
<tr>
<td>( W : 144.1 ) [lbf] weight</td>
</tr>
<tr>
<td>Aircraft.Wing</td>
</tr>
<tr>
<td>( S : 44.14 ) [ft²] surface area</td>
</tr>
<tr>
<td>( W : 44.14 ) [lbf] weight</td>
</tr>
</tbody>
</table>

(continues on next page)
<table>
<thead>
<tr>
<th>Mission.FlightSegment.AircraftP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wburn : [ 0.274 0.273 0.272 0.272 ] [lbf] segment fuel burn</td>
</tr>
<tr>
<td>Wfuel : [ 1.09 0.817 0.544 0.272 ] [lbf] fuel weight</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mission.FlightSegment.AircraftP.WingAero</th>
</tr>
</thead>
<tbody>
<tr>
<td>D : [ 2.74 2.73 2.72 2.72 ] [lbf] drag force</td>
</tr>
</tbody>
</table>

Variable Sensitivities
----------------------
| Aircraft.Fuselage |
| W : +0.97 weight |

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

Next Most Sensitive Variables
-----------------------------
| Mission.FlightSegment.AircraftP.WingAero |
| e : [ -0.18 -0.18 -0.18 -0.18 ] Oswald efficiency |

| Mission.FlightSegment.FlightState |
| V : [ -0.22 -0.21 -0.21 -0.21 ] true airspeed |
| rho : [ -0.12 -0.11 -0.11 -0.11 ] air density |

Most Sensitive Constraints
--------------------------
| Aircraft |

| Mission |

| Aircraft.Wing |
| +0.43 : .W >= S·rho |

Insensitive Constraints |below +1e-05|
--------------------------------------
(none)

Solution Diff (for selected variables)
======================================
(positive means the argument is smaller)

Constraint Differences
************************
--- removed in argument
+++ added in argument
@@ -41,4 +41,3 @@
   c = (S/A)^0.5 "parametrization of wing weight": 
   Aircraft.Wing.W >= S*Aircraft.Wing.rho

(continues on next page)
- $W_{\text{burn}}[:] \geq 0.2 \cdot D[:]

********************
Relative Differences [above 1%]

W_{\text{burn}} : [ +102.1\% +101.6\% +101.1\% +100.5\% ] segment fuel burn
W_{\text{fuel}} : [ +101.3\% +101.1\% +100.8\% +100.5\% ] fuel weight
D : [ +1.1\% - - - ] drag force

Sensitivity Differences [above 0.1]

The largest is -0.00451643.
CHAPTER 7

Advanced Commands

7.1 Derived Variables

7.1.1 Evaluated Fixed Variables

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

```python
# code from t_GPSubs.test_calcconst in tests/t_sub.py
x = Variable("x", "hours")
t_day = Variable("t_{day}"), 12, "hours")
t_night = Variable("t_{night}"), lambda c: 24 - c[t_day], "hours")

# note that t_night has a function as its value
m = Model(x, [x >= t_day, x >= t_night])
sol = m.solve(verbosity=0)
self.assertAlmostEqual(sol(t_night)/gpkit.ureg.hours, 12)
m.substitutions.update({t_day: ("sweep", [8, 12, 16])})
sol = m.solve(verbosity=0)
sel.assertAlmostEqual(sol(t_night), 12)
self.assertEqual(len(sol["cost"]), 3)
npt.assert_allclose(sol(t_day) + sol(t_night), 24)
```

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

7.1.2 Evaluated Free Variables

Some free variables may be evaluated from the values of other (non-evaluated) free variables after the optimization is performed. For example, if the efficiency $\nu$ 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!

```python
# 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 *Sequential Geometric Programs*.

### 7.2 Sweeps

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

#### 7.2.1 Sweep Substitutions

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

Vector variables may also be substituted for: `{y: ('sweep', [[1, 2], [1, 2], [1, 2]])}` will sweep `y` over all `y_i ∈ {1, 2}`. These sweeps cannot be specified during Variable creation.

A Model with sweep substitutions will solve for all possible combinations: e.g., if there’s a variable `x` with value `('sweep', [1, 3])` and a variable `y` with value `('sweep', [14, 17])` then the gp will be solved four times, for `(x, y) ∈ {(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.

#### 7.2.2 1D Autosweeps

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

Accessing variable and cost values from an autosweep is slightly different, as can be seen in this example:
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])
toll = 1e-3
bst1 = autosweep_1d(m1, toll, l, [1, 10], verbosity=0)
print("Solved after \$2i \plus/-.3g\)\) % (bst1.nsols, bst1.
˓
→
)

# autosweep solution accessing
l_vals = np.linspace(1, 10, 10)
soll = bst1.sample_at(l_vals)
print("values of l:
˓
→
%)

values of A:
˓
→
%s
˓
→
\)\)

cost_estimate = soll["cost"]
cost_lb, cost_ub = soll.cost_lb(), soll.cost_ub()
print("cost lower bound:\n˓
→
%)

print("cost estimate:\n˓
→
%)

print("cost upper bound:\n˓
→
%)

# you can evaluate arbitrary posynomials
np.testing.assert_allclose(mag(2*soll(A)), mag(soll(2*A)))
assert (soll["cost"] == soll(A**2)).all()

# the cost estimate is the logspace mean of its upper and lower bounds
np.testing.assert_allclose((np.log(mag(cost_lb)) + np.log(mag(cost_ub)))/2,
˓
→
np.log(mag(cost_estimate)))

# save autosweep to a file and retrieve it
bst1.save("autosweep.pkl")
bst1_loaded = pickle.load(open("autosweep.pkl", "rb"))

# this problem is two intersecting lines in logspace
m2 = Model(A**2, [A >= (l/3)**2, A >= (l/3)**0.5 * units.m**1.5])
tol2 = {'mosek': 1e-12, "cvxopt": 1e-7,
˓
→
"mosek_cli": 1e-6,
˓
→
"mosek_conif": 1e-6}[gpkit.settings["default_solver"]]

# test Model method
sol2 = m2.autosweep({l: [1, 10]}, tol2, verbosity=0)
bst2 = sol2.bst
print("Solved after \$2i \plus/-.3g\)\) % (bst2.nsols, bst2.
˓
→

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).

7.3 Tight ConstraintSets

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

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

### 7.4 Loose ConstraintSets

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

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

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

### 7.5 Substitutions

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

#### 7.5.1 Substituting into Posynomials, NomialArrays, and GPs

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

```python
# 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.
7.5.2 Substituting multiple values

```python
# 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.

7.5.3 Substituting with nonnumeric values

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

```python
# 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).

7.5.4 Updating ConstraintSet substitutions

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

When a Model is created, any fixed Variables are used to form a dictionary: 
\{(var: var.descr['value']) for var in self.varlocs if "value" in var.descr\}. This dictionary 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`.

7.5.6 Freeing Fixed Variables

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

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

\[
\begin{align*}
\text{minimize} & \quad g_0(x) \\
\text{subject to} & \quad f_i(x) = 1, \quad i = 1, \ldots, m \\
& \quad g_i(x) - h_i(x) \leq 1, \quad i = 1, \ldots, n
\end{align*}
\]

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.

### 8.1 Example Usage

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

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

(continues on next page)
# must enable signomials for subtraction

```python
with gpkit.SignomialsEnabled():
    constraints = [x >= 1-y, y <= 0.1]
```

# create and solve the SP

```python
m = gpkit.Model(x, constraints)
print(m.localsolve(verbosity=0).summary())
```

```python
assert abs(m.solution(x) - 0.9) < 1e-6
```

# full interim solutions are available

```python
print("x values of each GP solve (note convergence)")
print(", ".join("%.5f" % sol["freevariables"] for sol in m.program.
    results))
```

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 $x_k$ argument.

### 8.2 Sequential Geometric Programs

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

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

```
\begin{align*}
\text{minimize} & \quad y \\
\text{subject to} & \quad y \geq \sin(x) \\
& \quad \frac{\pi}{4} \leq x \leq \frac{\pi}{2}
\end{align*}
```

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"
```

```python
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())
```
Optimal Cost
-------------
0.7854

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

Most Sensitive Constraints
--------------------------
+1 : x >= 0.785
+1 : y >= x

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

```python
import numpy as np
def external_code(x):
    "Returns sin(x)"
    return np.sin(x)
```

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

```python
from external_function import external_code
class ExternalConstraint:
    "Class for external calling"
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def as_gpconstr(self, x0):
        # Creating a default constraint for the first solve
        if self.x not in x0:
            return (self.y >= self.x)
        # Otherwise calls external code at the current position...
        x_star = x0[self.x]
        res = external_code(x_star)
        # ...and returns a linearized posy <= 1
        return (self.y >= res * self.x/x_star)
```

and replace the incompatible constraint in our GP:

```python
(continues on next page)
```
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())

Optimal Cost
-------------
0.7071

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

Most Sensitive Constraints
---------------------------
+1 : <external_constraint.ExternalConstraint object>
+1 : x >= 0.785

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 \(x_0\) argument.
9.1 iPython Notebook Examples

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

9.2 A Trivial GP

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

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

# Decision variable
x = Variable("x")

# Constraint
costs = [x >= 1]

# Objective (to minimize)
objective = x

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

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

# print selected results
print("Optimal cost: \$%.4g\" % sol["cost"])
print("Optimal x val: \$%.4g\" % sol["variables"]

Of course, the optimal value is 1. Output:
Optimal cost: 1
Optimal x val: 1

## 9.3 Maximizing the Volume of a Box

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

"Maximizes box volume given area and aspect ratio constraints."

from gpkit import Variable, Model

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

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

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

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

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

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

The output is

Optimal Cost
-------------
0.003674

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

Fixed Variables

(continues on next page)
9.4 Water Tank

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

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

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

# because its units are incorrect the line below will print a warning
bad_monomial_equality = (M == V)

              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

```
Infeasible monomial equality: Cannot convert from 'V [m^3]' to 'M [kg]'

Optimal Cost
-----------
1.293

Free Variables
-----------
```
**9.5 Simple Wing**

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

"Minimizes airplane drag for a simple drag and structure model."

```python
import pickle
import numpy as np
from gpkit import Variable, Model
pi = np.pi

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

(continues on next page)
C_f = Variable("C_f", ",", "skin friction coefficient")
W_w = Variable("W_w", ",", "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())

# save solution to a file and retrieve it
sol.save("solution.pkl")
print(sol.diff("solution.pkl"))

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

The output is

SINGLE

======

Optimal Cost

-------------

303.1

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

(continues on next page)
D : 303.1  [N]  total drag force
Re : 3.675e+06  Reynolds 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 Variables
------------------------
W_0 : +1 aircraft weight excluding wing
e : -0.48 Oswald efficiency factor
(S/S_{wet}) : +0.43 wetted area ratio
k : +0.43 form factor
V_{min} : -0.37 takeoff speed

Most Sensitive Constraints
--------------------------
+1.3 : W >= W_0 + W_w
+1 : C_D >= (CDA0)/S + k·C_f·(S/S_{wet}) + C_L^2/(\pi·A·e)
+1 : D >= 0.5·\rho·S·C_D·V^2
+0.96 : W <= 0.5·\rho·S·C_L·V^2
+0.43 : C_f >= 0.074/Re^0.2

Solution Diff
=============  
(positive means the argument is smaller)
** no constraint differences **

Relative Differences |above 1%|
----------------------
The largest is +0%.

Sensitivity Differences |above 0.1|
--------------------------------
The largest is +0.

SWEEP
=====  

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

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

Free Variables
-------------
A : [ 6.2  4.77  8.84  7.16 ] aspect ratio
C_D : [ 0.0146  0.0123  0.0196  0.0157 ] Drag coefficient of wing
C_L : [ 0.296  0.198  0.463  0.31 ] Lift coefficient of wing
C_f : [ 0.00333  0.00314  0.00361  0.00342 ] skin friction coefficient
Most Sensitive Variables
------------------------

<table>
<thead>
<tr>
<th>Variable</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_0$</td>
<td>$+0.92$</td>
</tr>
<tr>
<td>$V_{\text{min}}$</td>
<td>$-0.82$</td>
</tr>
<tr>
<td>$V$</td>
<td>$+0.59$</td>
</tr>
<tr>
<td>$(\frac{S}{S_{\text{wet}}})$</td>
<td>$+0.56$</td>
</tr>
<tr>
<td>$k$</td>
<td>$+0.56$</td>
</tr>
</tbody>
</table>

Most Sensitive Constraints (in last sweep)
------------------------------------------

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+1$</td>
<td>$C_D \geq \frac{(CDA0)/S + k \cdot C_f \cdot (\frac{S}{S_{\text{wet}}}) + C_L}{\pi \cdot A \cdot e}$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$D \geq 0.5 \cdot \rho \cdot S \cdot C_D \cdot V^2$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$W \geq W_0 + W_w$</td>
</tr>
<tr>
<td>$+0.57$</td>
<td>$W \leq 0.5 \cdot \rho \cdot S \cdot C_L \cdot V^2$</td>
</tr>
<tr>
<td>$+0.54$</td>
<td>$C_f \geq 0.074/Re^{0.2}$</td>
</tr>
</tbody>
</table>

Solution Diff
==============

(positive means the argument is smaller)

** no constraint differences **

Relative Differences |above 1%|
----------------------|

<table>
<thead>
<tr>
<th>Variable</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Re$</td>
<td>$+46.4%$</td>
</tr>
<tr>
<td>$C_L$</td>
<td>$-40.6%$</td>
</tr>
<tr>
<td>$V$</td>
<td>$+18.0%$</td>
</tr>
<tr>
<td>$W_w$</td>
<td>$-20.7%$</td>
</tr>
<tr>
<td>$C_D$</td>
<td>$-29.0%$</td>
</tr>
<tr>
<td>$A$</td>
<td>$-26.7%$</td>
</tr>
<tr>
<td>$S$</td>
<td>$+12.8%$</td>
</tr>
<tr>
<td>$D$</td>
<td>$+11.5%$</td>
</tr>
<tr>
<td>$V_{\text{min}}$</td>
<td>$-9.1%$</td>
</tr>
<tr>
<td>$W$</td>
<td>$-6.8%$</td>
</tr>
<tr>
<td>$C_f$</td>
<td>$-7.3%$</td>
</tr>
</tbody>
</table>

Absolute Differences |above 0|
---------------------|

<table>
<thead>
<tr>
<th>Variable</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Re$</td>
<td>$+1.7e+06$</td>
</tr>
<tr>
<td>$W$</td>
<td>$-5e+02$</td>
</tr>
<tr>
<td>$W_w$</td>
<td>$-5e+02$</td>
</tr>
<tr>
<td>$D$</td>
<td>$+35$</td>
</tr>
<tr>
<td>$V$</td>
<td>$+6.8$</td>
</tr>
<tr>
<td>$S$</td>
<td>$+2.1$</td>
</tr>
<tr>
<td>$V_{\text{min}}$</td>
<td>$-2$</td>
</tr>
</tbody>
</table>

9.5. Simple Wing

(continues on next page)
9.6 Simple Beam

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

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

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

class Beam(Model):
    """Discretization of the Euler beam equations for a distributed load.
    """
    Variables
    ---------
    EI [N*m^2] Bending stiffness
dx [m] Length of an element
    L 5 [m] Overall beam length
```

(continues on next page)
Boundary Condition Variables
-------------------------------
\[ V_{\text{tip}} \] \[ \text{eps \ [N]} \] Tip loading
\[ M_{\text{tip}} \] \[ \text{eps \ [N*m]} \] Tip moment
\[ \text{th}_{\text{base}} \] \[ \text{eps \ [-]} \] Base angle
\[ w_{\text{base}} \] \[ \text{eps \ [m]} \] Base deflection

Node Variables of length \( N \)
--------------------------
\[ q \] \[ 100*\text{np.ones}(N) \] \[ \text{[N/m]} \] Distributed load
\[ V \] \[ \text{[N]} \] Internal shear
\[ M \] \[ \text{[N*m]} \] Internal moment
\[ \text{th} \] \[ \text{[-]} \] Slope
\[ w \] \[ \text{[m]} \] Displacement

Upper Unbounded
---------------
\[ w_{\text{tip}} \]

```python
@parse_variables(__doc__, globals())
def setup(self, N=4):
    # minimize tip displacement (the last w)
    self.cost = self.w_tip = w[-1]
    return

    # 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 integration":
        V[:-1] >= V[1:] + 0.5*dx*(q[:-1] + q[1:]),
    # moment integration:
        M[:-1] >= M[1:] + 0.5*dx*(V[:-1] + V[1:]),
    # slope and displacement increase from base to tip (right > left)
    "theta integration":
        th[1:] >= th[:-1] + 0.5*dx*(M[1:] + M[:-1])/EI,
    # displacement integration:
        w[1:] >= w[:-1] + 0.5*dx*(th[1:] + th[:-1])
```

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

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

(continues on next page)
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

Optimal Cost
------------
1.621

Free Variables
--------------

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dx</td>
<td>1.2</td>
</tr>
<tr>
<td>M</td>
<td>[1.98e+03, 1.27e+03, 713, 317, 79.2, 0.0002] [N·m]</td>
</tr>
<tr>
<td>V</td>
<td>[660, 528, 396, 264, 132, 0.0002] [N]</td>
</tr>
<tr>
<td>th</td>
<td>[0.0002, 0.177, 0.285, 0.341, 0.363, 0.367]</td>
</tr>
<tr>
<td>w</td>
<td>[0.0002, 0.107, 0.384, 0.76, 1.18, 1.62] [m]</td>
</tr>
</tbody>
</table>

Most Sensitive Variables
-------------------------

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>+4 Overall</td>
</tr>
<tr>
<td>EI</td>
<td>-1 Bending</td>
</tr>
<tr>
<td>q</td>
<td>[+0.0072, +0.042, +0.12, +0.23, +0.37, +0.22]</td>
</tr>
</tbody>
</table>

Most Sensitive Constraints
---------------------------

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4 : L = 5·dx</td>
<td></td>
</tr>
</tbody>
</table>

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

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

10.1 gpkit package

10.1.1 Subpackages
gpkit.constraints package

Submodules
gpkit.constraints.array module
gpkit.constraints.bounded module
gpkit.constraints.costed module
gpkit.constraints.gp module
gpkit.constraints.model module
gpkit.constraints.prog_factories module
gpkit.constraints.relax module
gpkit.constraints.set module
gpkit Documentation, Release 0.9.9

gpkit.constraints.sgp module

gpkit.constraints.sigeq module

gpkit.constraints.single_equation module

gpkit.constraints.tight module

Module contents

gpkit.interactive package

Submodules

gpkit.interactive.chartjs module

gpkit.interactive.plot_sweep module

gpkit.interactive.plotting module

gpkit.interactive.ractor module

gpkit.interactive.sankey module

gpkit.interactive.widgets module

Module contents

gpkit.nomials package

Submodules

gpkit.nomials.array module

gpkit.nomials.core module

gpkit.nomials.data module

gpkit.nomials.map module

gpkit.nomials.math module

gpkit.nomials.substitution module

gpkit.nomials.variables module
Module contents

gpkit.tools package

Submodules

gpkit.tools.autosweep module

gpkit.tools.docstring module

gpkit.tools.fmincon module

gpkit.tools.spdata module

gpkit.tools.tools module

Module contents

10.1.2 Submodules

10.1.3 gpkit.build module

10.1.4 gpkit.exceptions module

10.1.5 gpkit.globals module

10.1.6 gpkit.keydict module

10.1.7 gpkit.repr_conventions module

10.1.8 gpkit.small_classes module

10.1.9 gpkit.small_scripts module

10.1.10 gpkit.solution_array module

10.1.11 gpkit.varkey module

10.1.12 Module contents
Citing GPkit

If you use GPkit please cite it with the following bibtex:

```latex
@inproceedings{burnell2020gpkit,
    author={Burnell, Edward and Damen, Nicole B and Hoburg, Warren},
    title={\hbox{GPkit}: A Human-Centered Approach to Convex Optimization in Engineering Design},
    year={2020},
    doi={10.1145/3313831.3376412}
}
```

(and you can read that paper, which describes some of GPkit’s design philosophy, here.)
Acknowledgements

We thank the following contributors for helping to improve GPkit:

- Marshall Galbraith for setting up continuous integration.
- Stephen Boyd for inspiration and suggestions.
- Kirsten Bray for designing the GPkit logo.
CHAPTER 13

Release Notes

Release notes are available on Github