Cassowary Documentation

Release 0.5.1

Russell Keith-Magee

May 07, 2019

Contents

A pure Python implementation of the Cassowary constraint-solving algorithm. Cassowary is the algorithm that forms the core of the OS X and iOS visual layout mechanism.

CHAPTER 1

Quickstart

Cassowary is compatible with both Python 2 or Python 3. To install Cassowary in your virtualenv, run:

```
$ pip install cassowary
```

Then, in your Python code, you can create and solve constraint systems. See *the documentation* for examples of what this looks like in practice.

CHAPTER 2

Documentation

2.1 Solving constraint systems

Constraint solving systems are an algorithmic approach to solving Linear Programming problems. A linear programming problem is a mathematical problem where you have a set of non- negative, real valued variables (`x[1], x[2], ... x[n]`), and a series of linear constraints (i.e, no exponential terms) on those variables. These constraints are expressed as a set of equations of the form:

```
a[1]x[1] + ... + a[n]x[n] = b,
```

```
a[1]x[1] + ... + a[n]x[n] <= b, or
```

```
a[1]x[1] + ... + a[n]x[n] >= b,
```

Given these constraints, the problem is to find the values of `x[i]` that minimize or maximize the value of an **objective function**:

```
c + d[1]x[1] + ... + d[n]x[n]
```

Cassowary is an algorithm designed to solve linear programming problems of this type. Published in 1997, it now forms the basis for the UI layout tools in OS X Lion, and iOS 6+ (the approach known as Auto Layout). The Cassowary algorithm (and this implementation of it) provides the tools to describe a set of constraints, and then find an optimal solution for that set of constraints.

2.1.1 Variables

At the core of the constraint problem are the variables in the system. In the formal mathematical system, these are the `x[n]` terms; in Python, these are rendered as instances of the *`Variable`* class.

Each variable is named, and can accept a default value. To create a variable, instantiate an instance of *`Variable`*:

```
from cassowary import Variable

# Create a variable with a default value.
x1 = Variable('x1')
```

(continues on next page)

(continued from previous page)

```
# Create a variable with a specific value
x2 = Variable('x1', 42.0)
```

Any value provided for the variable is just a starting point. When constraints are imposed, this value can and will change, subject to the requirements of the constraints. However, providing an initial value may affect the search process; if there's an ambiguity in the constraints (i.e., there's more than one possible solution), the initial value provided to variables will affect the solution on which the system converges.

2.1.2 Constraints

A constraint is a mathematical equality or inequality that defines the linear programming system.

A constraint is declared by providing the Python expression that encompasses the logic described by the constraint. The syntax looks essentially the same as the raw mathematical expression:

```
from cassowary import Variable

# Create a variable with a default value.
x1 = Variable('x1')
x2 = Variable('x2')
x3 = Variable('x4')

# Define the constraint
constraint = x1 + 3 * x2 <= 4 * x3 + 2
```

In this example, *constraint* holds the definition for the constraint system. Although the statement uses the Python comparison operator <=, the result is *not* a boolean. The comparison operators <=, <, >=, >, and == have been overridden for instances of *Variable* to enable you to easily define constraints.

2.1.3 Solvers

The solver is the engine that resolves the linear constraints into a solution. There are many approaches to this problem, and the development of algorithmic approaches has been the subject of math and computer science research for over 70 years. Cassowary provides one implementation – a *SimplexSolver*, implementing the Simplex algorithm defined by Dantzig in the 1940s.

The solver takes no arguments during constructions; once constructed, you simply add constraints to the system.

As a simple example, let's solve the problem posed in Section 2 of the Badros & Borning's paper on Cassowary. In this problem, we have a 1 dimensional number line spanning from 0 to 100. There are three points on it (left, middle and right), with the following constraints:

- The middle point must be halfway between the left and right point;
- The left point must be at least 10 to the left of the right point;
- All points must fall in the range 0-100.

This system can be defined in Python as follows:

```
from cassowary import SimplexSolver, Variable

solver = SimplexSolver()
```

(continues on next page)

(continued from previous page)

```
left = Variable('left')
middle = Variable('middle')
right = Variable('right')

solver.add_constraint(middle == (left + right) / 2)
solver.add_constraint(right == left + 10)
solver.add_constraint(right <= 100)
solver.add_constraint(left >= 0)
```

There are an infinite number of possible solutions to this system; if we interrogate the variables, you'll see that the solver has provided one possible solution:

```
>>> left.value
90.0
>>> middle.value
95.0
>>> right.value
100.0
```

2.1.4 Stay constraints

If we want a particular solution to our left/right/middle problem, we need to fix a value somewhere. To do this, we add a *Stay* - a special constraint that says that the value should not be altered.

For example, we might want to enforce the fact that the middle value should stay at a value of 45. We construct the system as before, but add:

```
middle.value = 45.0
solver.add_stay(middle)
```

Now when we interrogate the solver, we'll get values that reflect this fixed point:

```
>>> left.value
40.0
>>> middle.value
45.0
>>> right.value
50.0
```

2.1.5 Constraint strength

Not all constraints are equal. Some are absolute requirements - for example, a requirement that all values remain in a specific range. However, other constraints may be suggestions, rather than hard requirements.

To accommodate this, Cassowary allows all constraints to have a **strength**. Strength can be one of:

- `REQUIRED`
- `STRONG`
- `MEDIUM`
- `WEAK`

`REQUIRED` constraints **must** be satisfied; the remaining strengths will be satisfied with declining priority.

To define a strength, provide the strength value as an argument when adding the constraint (or stay):

```
from cassowary import SimplexSolver, Variable, STRONG, WEAK

solver = SimplexSolver()
x = Variable('x')

# Define some non-required constraints
solver.add_constraint(x <= 100, strength=STRONG)
solver.add_stay(x, strength=WEAK)
```

Unless otherwise specified, all constraints are `REQUIRED`.

2.1.6 Constraint weight

If you have multiple constraints of the same strength, you may want to have a tie-breaker between them. To do this, you can set a **weight**, in addition to a strength:

```
from cassowary import SimplexSolver, Variable, STRONG

solver = SimplexSolver()
x = Variable('x')

# Define some non-required constraints
solver.add_constraint(x <= 100, strength=STRONG, weight=10)
solver.add_constraint(x >= 50, strength=STRONG, weight=20)
```

2.1.7 Editing constraints

Any constraint can be removed from a system; just retain the reference provided when you add the constraint:

```
from cassowary import SimplexSolver, Variable

solver = SimplexSolver()
x = Variable('x')

# Define a constraint
constraint = solver.add_constraint(x <= 100)

# Remove it again
solver.remove_constraint(constraint)
```

Once a constraint is removed, the system will be automatically re-evaluated, with the possible side effect that the values in the system will change.

But what if you want to change a variable's value without introducing a new constraint? In this case, you can use an edit context.

Here's an example of an edit context in practice:

```
from cassowary import SimplexSolver, Variable

solver = SimplexSolver()
x = Variable('x')

# Add a stay to x - that is, don't change the value.
solver.add_stay(x)
```

(continues on next page)

(continued from previous page)

```
# Now, mark x as being editable...
solver.add_edit_variable(x)

# ... start and edit context...
with solver.edit():
    # ... and suggest a new value for the variable.

    solver.suggest_value(x, 42.0)
```

When the edit context exits, the system will re-evaluate itself, and the variable will have the new value. However, the variable isn't guaranteed to have the value you suggested - in this case it will, but if your constraint system has other constraints, they may affect the value of the variable after the suggestion has been applied.

All variables in the system will be re-evaluated when you leave the edit context; however, if you need to force a re-evaluation in the middle of an edit context, you can do so by calling `resolve()`.

2.2 Examples

The following examples demonstrate the use of Cassowary in practical constraint-solving problems.

2.2.1 Quadrilaterals

The "Bounded Quadrilateral" demo is the online example provided for Cassowary. The online example is implemented in JavaScript, but the implementation doesn't alter the way the Cassowary algorithm is used.

The Bounded quadrilateral problem starts with a bounded, two dimensional canvas. We want to draw a quadrilateral on this plane, subject to a number of constraints.

Firstly, we set up the solver system itself:

```
from cassowary import SimplexSolver, Variable
solver = SimplexSolver()
```

Then, we set up a convenience class for holding information about points on a 2D plane:

```
class Point(object):
    def __init__(self, identifier, x, y):
        self.x = Variable('x' + identifier, x)
        self.y = Variable('y' + identifier, y)

    def __repr__(self):
        return u'(%s, %s)' % (self.x.value, self.y.value)
```

Now we can set up a set of points to describe the initial location of our quadrilateral - a 190x190 square:

```
points = [
    Point('0', 10, 10),
    Point('1', 10, 200),
    Point('2', 200, 200),
    Point('3', 200, 10),

    Point('m0', 0, 0),
    Point('m1', 0, 0),
```

(continues on next page)

(continued from previous page)

```
    Point('m2', 0, 0),
    Point('m3', 0, 0),
]
midpoints = points[4:]
```

Note that even though we're drawing a quadrilateral, we have 8 points. We're tracking the position of the midpoints independent of the corners of our quadrilateral. However, we don't need to define the position of the midpoints. The position of the midpoints will be set by defining constraints.

Next, we set up some stays. A stay is a constraint that says that a particular variable shouldn't be modified unless it needs to be - that it should "stay" as is unless there is a reason not to. In this case, we're going to set a stay for each of the four corners - that is, don't move the corners unless you have to. These stays are defined as `WEAK` stays – so they'll have a very low priority in the constraint system. As a tie breaking mechanism, we're also going to set each stay to have a different weight - so the top left corner (point 1) will be moved in preference to the bottom left corner (point 2), and so on:

```
weight = 1.0
multiplier = 2.0
for point in points[:4]:
    solver.add_stay(point.x, strength=WEAK, weight=weight)
    solver.add_stay(point.y, strength=WEAK, weight=weight)
    weight = weight * multiplier
```

Now we can set up the constraints to define where the midpoints fall. By definition, each midpoint **must** fall exactly halfway between two points that form a line, and that's exactly what we describe - an expression that computes the position of the midpoint. This expression is used to construct a `Constraint`, describing that the value of the midpoint must equal the value of the expression. The `Constraint` is then added to the solver system:

```
for start, end in [(0, 1), (1, 2), (2, 3), (3, 0)]:
    cle = (points[start].x + points[end].x) / 2
    cleq = midpoints[start].x == cle
    solver.add_constraint(cleq)

    cle = (points[start].y + points[end].y) / 2
    cleq = midpoints[start].y == cle
    solver.add_constraint(cleq)
```

When we added these constraints, we didn't provide any arguments - that means that they will be added as `REQUIRED` constraints.

Next, lets add some constraints to ensure that the left side of the quadrilateral stays on the left, and the top stays on top:

```
solver.add_constraint(points[0].x + 20 <= points[2].x)
solver.add_constraint(points[0].x + 20 <= points[3].x)

solver.add_constraint(points[1].x + 20 <= points[2].x)
solver.add_constraint(points[1].x + 20 <= points[3].x)

solver.add_constraint(points[0].y + 20 <= points[1].y)
solver.add_constraint(points[0].y + 20 <= points[2].y)

solver.add_constraint(points[3].y + 20 <= points[1].y)
solver.add_constraint(points[3].y + 20 <= points[2].y)
```

Each of these constraints is posed as a `Constraint`. For example, the first expression describes a point 20 pixels to the right of the x coordinate of the top left point. This `Constraint` is then added as a constraint on the x coordinate

of the bottom right (point 2) and top right (point 3) corners - the x coordinate of these points must be at least 20 pixels greater than the x coordinate of the top left corner (point 0).

Lastly, we set the overall constraints – the constraints that limit how large our 2D canvas is. We'll constraint the canvas to be 500x500 pixels, and require that all points fall on that canvas:

```
for point in points:
    solver.add_constraint(point.x >= 0)
    solver.add_constraint(point.y >= 0)

    solver.add_constraint(point.x <= 500)
    solver.add_constraint(point.y <= 500)
```

This gives us a fully formed constraint system. Now we can use it to answer layout questions. The most obvious question – where are the midpoints?

```
>>> midpoints[0]
(10.0, 105.0)
>>> midpoints[1]
(105.0, 200.0)
>>> midpoints[2]
(200.0, 105.0)
>>> midpoints[3]
(105.0, 10.0)
```

You can see from this that the midpoints have been positioned exactly where you'd expect - half way between the corners - without having to explicitly specify their positions.

These relationships will be maintained if we then edit the position of the corners. Lets move the position of the bottom right corner (point 2). We mark the variables associated with that corner as being **Edit variables**:

```
solver.add_edit_var(points[2].x)
solver.add_edit_var(points[2].y)
```

Then, we start an edit, change the coordinates of the corner, and stop the edit:

```
with solver.edit():

    solver.suggest_value(points[2].x, 300)
    solver.suggest_value(points[2].y, 400)
```

As a result of this edit, the midpoints have automatically been updated:

```
>>> midpoints[0]
(10.0, 105.0)
>>> midpoints[1]
(155.0, 300.0)
>>> midpoints[2]
(250.0, 205.0)
>>> midpoints[3]
(105.0, 10.0)
```

If you want, you can now repeat the edit process for any of the points - including the midpoints.

2.2.2 GUI layout

The most common usage (by deployment count) of the Cassowary algorithm is as the Autolayout mechanism that underpins GUIs in OS X Lion and iOS6. Although there's lots of code required to make a full GUI toolkit work,

the layout problem is a relatively simple case of solving constraints regarding the size and position of widgets in a window.

In this example, we'll show a set of constraints used to determine the placement of a pair of buttons in a GUI. To simplify the problem, we'll only worry about the X coordinate; expanding the implementation to include the Y coordinate is a relatively simple exercise left for the reader.

When laying out a GUI, widgets have a width; however, widgets can also change size. To accommodate this, a widget has two size constraints in each dimension: a minimum size, and a preferred size. The minimum size is a `REQUIRED` constraint that must be met; the preferred size is a `STRONG` constraint that the solver should try to accommodate, but may break if necessary.

The GUI also needs to be concerned about the size of the window that is being laid out. The size of the window can be handled in two ways:

- a `REQUIRED` constraint – i.e., this *is* the size of the window; show me how to lay out the widgets; or
- a `WEAK` constraint – i.e., come up with a value for the window size that accommodates all the other widget constraints. This is the interpretation used to determine an initial window size.

As with the Quadrilateral demo, we start by creating the solver, and creating a storage mechanism to hold details about buttons:

```
from cassowary import SimplexSolver, Variable

solver = SimplexSolver()

class Button(object):
    def __init__(self, identifier):
        self.left = Variable('left' + identifier, 0)
        self.width = Variable('width' + identifier, 0)

    def __repr__(self):
        return u'(x=%s, width=%s)' % (self.left.value, self.width.value)
```

We then define our two buttons, and the variables describing the size of the window on which the buttons will be placed:

```
b1 = Button('b1')
b2 = Button('b2')
left_limit = Variable('left', 0)
right_limit = Variable('width', 0)

left_limit.value = 0
solver.add_stay(left_limit)
solver.add_stay(right_limit, WEAK)
```

The left limit is set as a `REQUIRED` constraint – the left border can't move from coordinate 0. However, the window can expand if necessary to accommodate the widgets it contains, so the right limit is a `WEAK` constraint.

Now we can define the constraints on the button layouts:

```
# The two buttons are the same width
solver.add_constraint(b1.width == b2.width)

# Button1 starts 50 from the left margin.
solver.add_constraint(b1.left == left_limit + 50)

# Button2 ends 50 from the right margin
```

(continues on next page)

(continued from previous page)

```
solver.add_constraint(left_limit + right_limit == b2.left + b2.width + 50)

# Button2 starts at least 100 from the end of Button1. This is the
# "elastic" constraint in the system that will absorb extra space
# in the layout.
solver.add_constraint(b2.left == b1.left + b1.width + 100)

# Button1 has a minimum width of 87
solver.add_constraint(b1.width >= 87)

# Button1's preferred width is 87
solver.add_constraint(b1.width == 87, strength=STRONG)

# Button2's minimum width is 113
solver.add_constraint(b2.width >= 113)

# Button2's preferred width is 113
solver.add_constraint(b2.width == 113, strength=STRONG)
```

Since we haven't imposed a hard constraint on the right hand side, the constraint system will give us the smallest window that will satisfy these constraints:

```
>>> b1
(x=50.0, width=113.0)
>>> b2
(x=263.0, width=113.0)

>>> right_limit.value
426.0
```

That is, the smallest window that can accommodate these constraints is 426 pixels wide. However, if the user makes the window larger, we can still lay out widgets. We impose a new `REQUIRED` constraint with the size of the window:

```
right_limit.value = 500
right_limit_stay = solver.add_constraint(right_limit, strength=REQUIRED)

>>> b1
(x=50.0, width=113.0)
>>> b2
(x=337.0, width=113.0)

>>> right_limit.value
500.0
```

That is - if the window size is 500 pixels, the layout will compensate by putting `button2` a little further to the right. The `WEAK` stay on the right limit that we established at the start is ignored in preference for the `REQUIRED` stay.

If the window is then resized again, we can remove the 500 pixel limit, and impose a new limit:

```
solver.remove_constraint(right_limit_stay)

right_limit.value = 475
right_limit_stay = solver.add_constraint(right_limit, strength=REQUIRED)
solver.add_constraint(right_limit_stay)

>>> b1
(x=50.0, width=113.0)
```

(continues on next page)

(continued from previous page)

```
>>> b2
(x=312.0, width=113.0)

>>> right_limit.value
475.0
```

Again, `button2` has been moved, this time to the left, compensating for the space that was lost by the contracting window size.

2.3 Reference

2.3.1 Variables

class cassowary.**Variable**

Variable.**value**
The current value of the variable. If the variable is part of a constraint, the value will be updated as constraints are applied and changed.

Variable.**__init__**(*name*, *value=0.0*)
Define a new variable. Value is optional, but will affect the constraint solving process if multiple solutions are possible.

2.3.2 Solvers

class cassowary.**SimplexSolver**
A class for collecting constraints into a system and solving them.

SimplexSolver.**add_constraint**(*constraint*, *strength=REQUIRED*, *weight=1.0*)
Add a new constraint to the solver system. A constraint is a mathematical expression involving 1 or more variables, and an equality or inequality.

`strength` is optional; by default, all constraints are added as `REQUIRED` constraints.

`weight` is optional; by default, all constraints have an equal weight of 1.0.

Returns the constraint that was added.

SimplexSolver.**remove_constraint**(*var*)
Remove a new constraint to the solver system.

Returns the constraint that was added.

SimplexSolver.**add_stay**(*var*, *strength=REQUIRED*, *weight=1.0*)
Add a stay constraint to the solver system for the current value of the variable `var`.

`strength` is optional; by default, all constraints are added as `REQUIRED` constraints.

`weight` is optional; by default, all constraints have an equal weight of 1.0.

Returns the constraint that was added.

SimplexSolver.**add_edit_var**(*var*)
Mark a variable as being an edit variable. This allows you to suggest values for the variable once you start an edit context.

SimplexSolver.**remove_edit_var**(*var*)
Remove the variable from the list of edit variables.

SimplexSolver.**edit**()
Returns a context manager that can be used to manage the edit process.

SimplexSolver.**suggest_value**(*var*, *value*)
Suggest a new value for a edit variable.

This method can only be invoked while inside an edit context. `var` must be a variable that has been identified as an edit variable in the current edit context.

SimplexSolver.**resolve**()
Force a solver system to resolve any ambiguities. Useful when introducing edit constraints.

2.4 Contributing to Cassowary

If you experience problems with Cassowary, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

2.4.1 Setting up your development environment

The recommended way of setting up your development environment for Cassowary is to install a virtual environment, install the required dependencies and start coding. Assuming that you are using `virtualenvwrapper`, you only have to run:

```
$ git clone git@github.com:pybee/cassowary.git
$ cd cassowary
$ mkvirtualenv cassowary
```

Cassowary uses `unittest` (or `unittest2` for Python < 2.7) for its own test suite as well as additional helper modules for testing. If you are running a Python version `< 2.7` you will also need to `pip install unittest2`.

Now you are ready to start hacking! Have fun!

2.5 Cassowary Roadmap

Cassowary is a port to Python of existing implementations of the Cassowary algorithm. To that extent, it is "feature complete".

However, the test suite for the original implementation is not especially comprehensive. Contributions of additional tests (and any bug fixes revealed by those tests) are most welcome.

Documentation improvements are also most welcome.

2.6 Release History

2.6.1 0.5.0

Initial public release.

CHAPTER 3

Community

Cassowary is part of the BeeWare suite. You can talk to the community through:

- @pybeeware on Twitter
- The BeeWare Users Mailing list, for questions about how to use the BeeWare suite.
- The BeeWare Developers Mailing list, for discussing the development of new features in the BeeWare suite, and ideas for new tools for the suite.

CHAPTER 4

Contributing

If you experience problems with Cassowary, log them on GitHub. If you want to contribute code, please fork the code and submit a pull request.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

C

E

R

S

V