LLS Documentation

Release 0.2

David Zeng, Keegan Go, Karanveer Mohan, Jenny Hong

Nov 05, 2017

Contents

1 In Depth Docs **3**

1.1 The Math Behind LLS 3
1.2 LinearLeastSquares.jl Tutorial 4
1.3 LineaLeastSquares.jl Examples 8
1.4 Credits 25

LinearLeastSquares, or LLS for short, is a library that makes it easy to formulate and solve least squares optimization problems with linear equality constraints. With LLS, these types of problems can be created using a natural syntax that mirrors standard mathematical notation.

LinearLeastSquares is a software package developed for the course, Introduction to Matrix Methods (EE103), taught by Professor Stephen Boyd at Stanford University. The accompanying text for LinearLeastSquares is Vectors, Matrices, and Least Squares.

For example, the classic problem of finding the least norm solution to an underdetermined system can be easily setup and solved with the following code:

```
using LinearLeastSquares

# Problem data
p = 20;
n = 30;
C = randn(p, n);
d = randn(p, 1);

# Build the components of the problem
x = Variable(n);
objective = sum_squares(x);
constraint = C * x == d;

# Solve the problem
optimal_value = minimize!(objective, constraint)
```

This example showcases the Julia implementation of LLS; other implementations include Python.

CHAPTER 1

In Depth Docs

1.1 The Math Behind LLS

1.1.1 Linearly Constrained Least Squares

LLS solves **linearly constrained least squares** (or LCLS) problems, which have the form:

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & Cx = d \end{array}$$

where the unknown variable x is a vector of size n. The values for A, b, C, and d are given and have sizes $m \times n$, m, $p \times n$, and p, respectively. LLS finds a value for x that satisfies the linear equality constraints $Cx = d$ and minimizes the objective, the sum of the squares of the entries of $Ax - b$.

When there are no equality constraints, LCLS reduces to the simple unconstrained least squares problem (LS):

$$\text{minimize} \quad \|Ax - b\|_2^2 \ .$$

When the objective is absent, LCLS reduces to finding x that satisfies $Cx = d$, i.e., solving a set of linear equations.

1.1.2 Solving LCLS

There is a unique solution to the LCLS problem if and only if there is a unique solution to the following system of linear equations in the variable x and a new variable z:

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} 2A^T b \\ d \end{bmatrix},$$

i.e., the matrix on the left is invertible. This occurs when the matrix C has independent rows, and the matrix $\begin{bmatrix} A \\ C \end{bmatrix}$ has indepedent columns.

When there are no equality constraints, the unconstrained least squares problem has a unique solution if and only if the system of linear equations:

$$2A^T Ax = 2A^T b$$

has a unique solution, which occurs when $A^T A$ is invertible, i.e., the columns of A are independent.

When the objective is absent, the system of linear equations $Cx = d$ has a unique solution if and only if C is invertible.

LLS allows you to specify an LCLS problem in a natural way. It translates your specification into the general form in this section, and then solves the appropriate set of linear equations.

1.2 LinearLeastSquares.jl Tutorial

The Julia package that implements LLS goes by the name of LinearLeastSquares.jl. We'll refer to LinearLeastSquares.jl as LLS throughout this tutorial.

1.2.1 Installing LLS

LLS requires Julia 0.3 or higher. For those new to Julia, the official Julia docs are a good way to get acquainted with the language.

To install LLS, simply open up a Julia terminal and run the commands:

```
Pkg.update()
Pkg.add("LinearLeastSquares")
```

To use LLS in Julia, run the following command to import the library:

```
using LinearLeastSquares
```

The same line of code can also be used in Julia scripts to import the LinearLeastSquares.jl package.

1.2.2 Variables and Constants

To declare variables in LLS, use the following syntax to specify their size:

```
x = Variable()      # A scalar variable
y = Variable(3)     # A vector variable with 3 rows and 1 column
z = Variable(10, 4) # A matrix variable that has 10 rows and 4 columns
```

LLS currently only supports variables up to 2 dimensions in size, i.e., scalars, vectors, and matrices. Variables have no value upon creation, but after solving a problem, LLS will populate all variables in the problem with optimal values. These values can be accessed using the evaluate function:

```
# x is a variable with value populated
x_value = evaluate(x)
```

Constants refer to any numerical scalars, vectors, or matrices of fixed value. Together with variables, they serve as the building blocks for more complex expressions.

1.2.3 Affine Expressions

Affine expressions are linear functions of variables plus a constant. Variables are themselves affine expressions. The most basic way to build more affine expressions from variables is to use the overloaded binary arithmetic operators `+, -, *, /`. The following operations are supported:

1. Addition or subtraction of two affine expressions, provided they are the same size or one is scalar.
2. Addition or subtraction of an affine expression and a constant, provided they are the same size or one is scalar.
3. Scalar or matrix multiplication between an affine expression and a constant.
4. Division of an affine expression by a scalar, nonzero constant.

Here are some examples of using binary operators to construct affine expressions:

```
w = Variable()     # scalar
x = Variable(3)    # 3-by-1 vector
Y = Variable(2, 3) # 2-by-3 matrix
z = Variable()     # scalar
b = [1 2 3]        # 1-by-3 matrix
C = randn(3, 4)    # 3-by-4 matrix

affine1 = w + b * x / 1.3 - 6.1   # scalar
affine2 = (affine1 - Y) * C       # 2-by-4 matrix
affine3 = affine2 - affine1       # 2-by-4 matrix
```

An affine expression can be evaluated to a numerical value if all variables the affine expression depends on have been populated with values. For example, the following code prints the value of the affine expression `affine1`, assuming both `w` and `x` have been populated with values:

```
println(evaluate(affine1))
```

Affine expressions support indexing and slicing using Julia's native syntax:

```
x = Variable(4)
a = x[3]              # third component of x
y = x[1:2]            # first two components of x
X = Variable(4, 5)
Y = X[3:4, 4:5]       # bottom right 2-by-2 submatrix of X
T = X[2:end, :]       # all but the first row of X
Z = 2 * x[1] + X
b = Z[1, 2]           # entry in first row and second column of Z
```

Affine expressions may also be stacked vertically and horizontally using Julia's native syntax:

```
x = Variable()
y = Variable(1, 3)
z = Variable(3)
T = Variable(3, 3)
horizontal_stack = [x y]  # 1-by-4 matrix
vertical_stack = [z; x]   # 4-by-1 vector
horizontal_and_vertical_stack = [x y; z T]  # 4-by-4 matrix
```

A few other functions also alter the shapes and sizes of affine expressions:

```
x = Variable(3, 1)
T = Variable(4, 4)

y = x' # transpose of x
```

```
X = diagm(x)     # create a diagonal matrix from a vector x
t = diag(T)      # extract the main diagonal of T as a column vector
t1 = diag(T, 1)  # extract the diagonal one right of the main diagonal of T
t2 = diag(T, -1) # extract the diagonal one left of the main diagonal of T

S = reshape(T, 8, 2) # reshape T as an 8-by-2 matrix
s = vec(S)           # reshape S as a 16-by-1 vector

x_rep = repmat(x, 2, 3) # tiles x twice vertically and three times horizontally to␣
↪form a 6-by-3 matrix
```

The sum and mean of the entries of an affine expression can be constructed:

```
X = Variable(2, 3)
sum_of_entries = sum(X)     # sums all entries of X
sum_of_columns = sum(X, 1) # sums along the first dimension of X, creating a row␣
↪vector
sum_of_rows = sum(X, 2)     # sums along the second dimension of X, creating a column␣
↪vector
mean_of_entries = mean(X)
mean_of_columns = mean(X, 1)
mean_of_rows = mean(X, 2)
```

1.2.4 Linear Equality Constraints

In LLS, a linear equality constraint is formed between an affine expression and a constant, or two affine expressions, using the == operator. Note that the == operator has been overloaded to no longer return a boolean, but rather an object representing the linear equality constraint. A linear equality constraint is only valid if the left hand side and the right hand side of the == have the same size, or if one is scalar. Here are some examples of linear equality constraints:

```
x = Variable(3)
A = randn(4, 3)
constraint1 = A * x == randn(4, 1)
constraint2 = 3 == x[1:2]
```

Lists of constraints can also be created. Additional constraints can be appended to a list using the + operator.

```
constraint_list = [A * x == randn(4, 1), 3 == x[1:2]]
constraint_list += x[3] == 1.6
```

An empty list of constraints can be created with `[]`. You can add to an empty list with the same syntax.

```
new_list = []
new_list += x[2] == 1.2
```

1.2.5 The `solve!` Function

LLS can solve a system of linear equations using the `solve!` funciton. The exclamation point after `solve` is a Julia convention signifying that this function will have side effects; specifically, it will assign values to variables after solving. After that, the values of the variables, and any expressions that depend on them, can be accessed.

```
x = Variable()
y = Variable()
```

```
solve!(x + 3 * y == 2, x - y == 1)
println(evaluate(x))
println(evaluate(y))
```

The arguments to the `solve!` function are either one or more comma separated linear equality constraints or a list of linear equality constraints. Only systems with unique solutions can be solved by LLS; see the *Solving LCLS* section for detailed conditions. The `solve!` function will issue an error if these conditions are not satisfied.

1.2.6 Sum of Squares Expressions

In LLS, a sum of squares expression is the sum of squares of the entries of a scalar, vector, or matrix. The most basic way to create such an expression is to call the `sum_squares` function on an affine expression argument. For example, `sum_squares(A * x - b)` is the LLS representation of $\|Ax - b\|_2^2$. To create other sum of squares expressions, the `+`, `*`, and `/` operators can be used in conjunction with the following rules:

1. Two sum of squares expressions can be added
2. A sum of squares expression can be multiplied or divided by a postive, scalar constant.
3. A nonnegative scalar constant may be added to a sum of squares expression.

Note that sum of squares expression cannot be subtracted from each other, or multiplied or divided by a negative number. LLS will issue an error message if the user attempts any of these. Here are some examples of building sum of squares expressions:

```
A = randn(4, 3)
b = randn(4, 1)
x = Variable(3)
c = 0.1
reg_least_squares = sum_squares(A * x - b) + c * sum_squares(x)
```

Similar to an affine expression, a sum of squares expression can be evaluated to a numerical value if all variables the sum of squares expression depends on have been populated with values. For example, the following code prints the value of the sum of squares expression `reg_least_squares`, assuming `x` has been populated with a value:

```
println(evaluate(reg_least_squares))
```

Often you'll find it useful to first initialize a sum of squares expression to `0` and then add on more sum of squares expressions in a for loop.

```
error_term = 0
for i in 1:3
  error_term += rand() * sum_squares(A[i, :] * x + b[i])
end
```

The variance of the entries of an affine expression `X` can be expressed as `sum_squares(mean(X) - X) / (m * n)`, where `m` and `n` are the number of rows and number of columns of `X`, respectively. For convenience, the function `var` can be used to directly create this sum of squares expression for variance.

```
X = Variable(3, 4)
variance = var(X)
```

1.2.7 The `minimize!` Function

LLS can also solve a linearly constrained least squares problem using the `minimize!` function:

```
A = randn(3, 2)
b = randn(2, 1)
x = Variable(3)
objective = sum_squares(x)
constraint = A * x == b
optimal_value = minimize!(objective, constraint)
println(evaluate(x))
```

The first argument, or objective, of `minimize!` must be a sum of squares expression. The remaining arguments are for constraints, and can be zero or more comma separated linear equality constraints, or a list of linear equality constraints. The `minimize!` function will return the optimal value of the sum of squares expression, while populating all variables with optimal values. Here are some usage examples:

```
x = Variable(3)
C = randn(2, 3)
d = randn(2)
A = randn(4, 3)
b = randn(4)

# no constraints
objective2 = sum_squares(A * x - b)
optimum_value_2 = minimize!(objective2)
println(evaluate(x))

# list of constraints
objective1 = sum_squares(x)
constraints = [C * x == d, x[1] == 0]
optimum_value_1 = minimize!(objective1, constraints)
println(evaluate(x))
```

A linearly constrained least squares can only be solved if it satisfies the conditions in the *Solving LCLS* section. The `minimize!` function will issue an error these conditions are not satisfied.

1.3 LineaLeastSquares.jl Examples

This tutorial showcases LinearLeastSquares.jl through a few involved examples of linearly constrained least squares problems. We'll refer to LinearLeastSquares.jl as "LLS" throughout. The plots generated by the following examples use the Gadfly package. The documentation on how to use Gadfly can be found here, but by no means is it necessary to read unless you would like to create plots for yourself. To install Gadfly, run the following command in a Julia shell

```
Pkg.add("Gadfly")
```

Some of the examples will also use data files, which can all be found here.

1.3.1 Regression

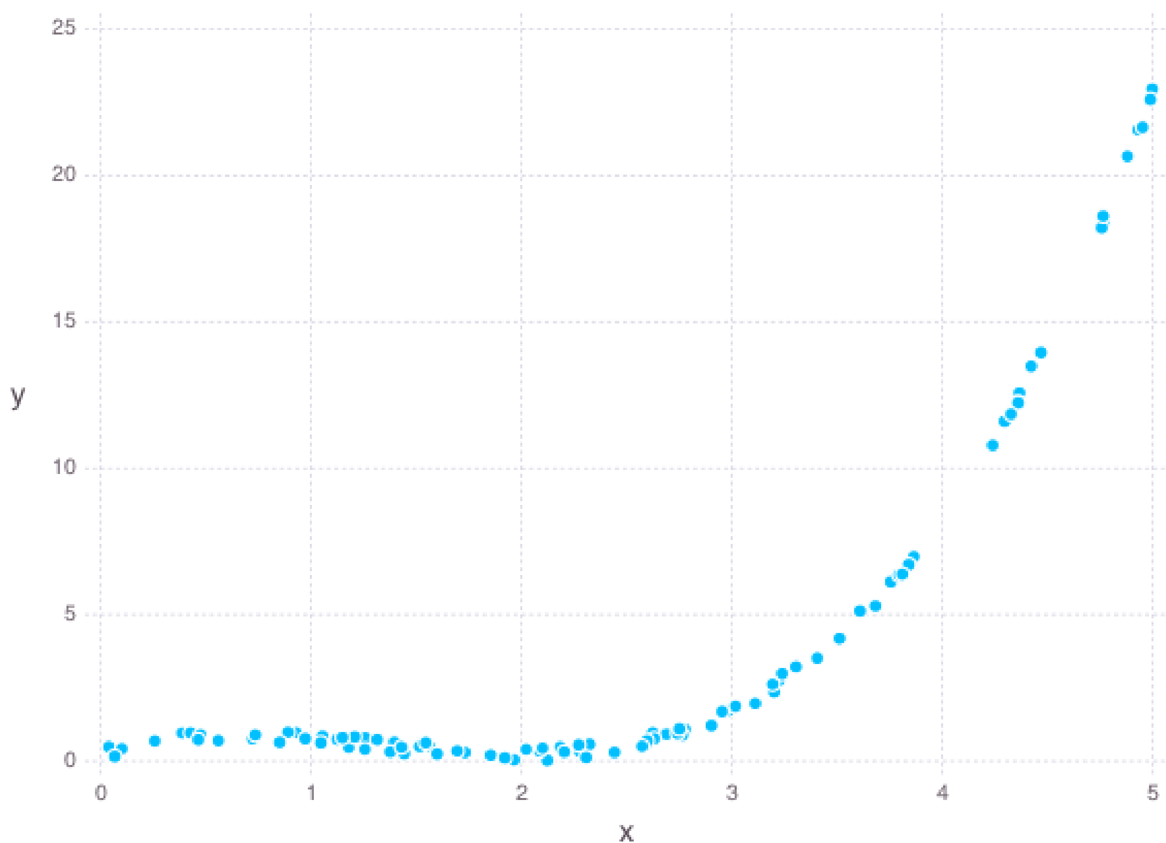
Regression is the problem of trying to fit a function to some data. In this example, we will frame some simple regression problems as unconstrained least squares problems for LLS to solve.

We are given n points, represented by two n-by-1 vectors, `x_data` and `y_data`. The x and y coordinates of the i-th point are given by the i-th entries of `x_data` and `y_data`, respectively.

We'll start by generating and visualizing some data to get a better sense of the problem at hand:

```
# Set the random seed to get consistent data
srand(1)

# Number of examples to use
n = 100

# Specify the true value of the variable
true_coeffs = [2; -2; 0.5]

# Generate data
x_data = rand(n, 1) * 5
x_data_expanded = hcat([x_data .^ i for i in 1 : 3]...)
y_data = x_data_expanded * true_coeffs + 0.5 * rand(n, 1)


p = plot(
  x=x_data, y=y_data, Geom.point,
  Theme(panel_fill=color("white"))
)
```

The following graph of the data will appear



Linear Regression

We will first try to fit a line to the data. A general function for a line is

$$f(x) = \alpha + \beta x$$

where α is the offset and β is the slope. We would like to pick α and β so that our data points lie "close" to our line. For a point with coordinates x and y the residual between the point and our line is defined as

$$r(x, y) = f(x) - y.$$

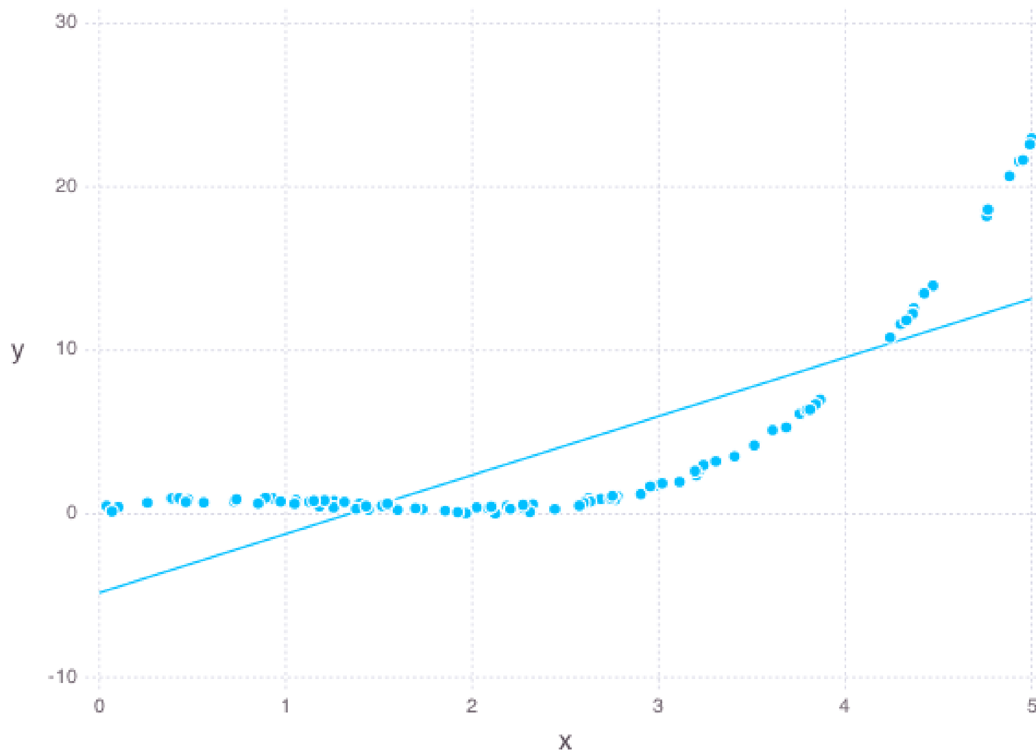
One reasonable way to measure the how different line is from the data is to sum the squares of the residuals between each point in the data and the line:

$$E(\alpha, \beta) = \sum_{i=1}^{n} (r(x_i, y_i))^2 = \sum_{i=1}^{n} (\alpha + \beta x_i - y_i)^2.$$

We would like to choose α and β to minimize this error. We can now frame this problem in Julia code and solve our problem using LLS:

```
slope = Variable()
offset = Variable()
line = offset + x_data * slope
residuals = line - y_data
fit_error = sum_squares(residuals)
optval = minimize!(fit_error)

# plot the data and the line
t = [0; 5; 0.1]
p = plot(
  layer(x=x_data, y=y_data, Geom.point),
  layer(x=t, y=evaluate(slope) * t + evaluate(offset), Geom.line),
  Theme(panel_fill=color("white"))
)
```

The line of best fit on our data is shown below:



Quadratic Regression

A line is probably not the best function to fit to this data. Instead, let's try to fit a quadratic function, which has the form:

$$f(x) = \alpha + \beta x + \gamma x^2$$

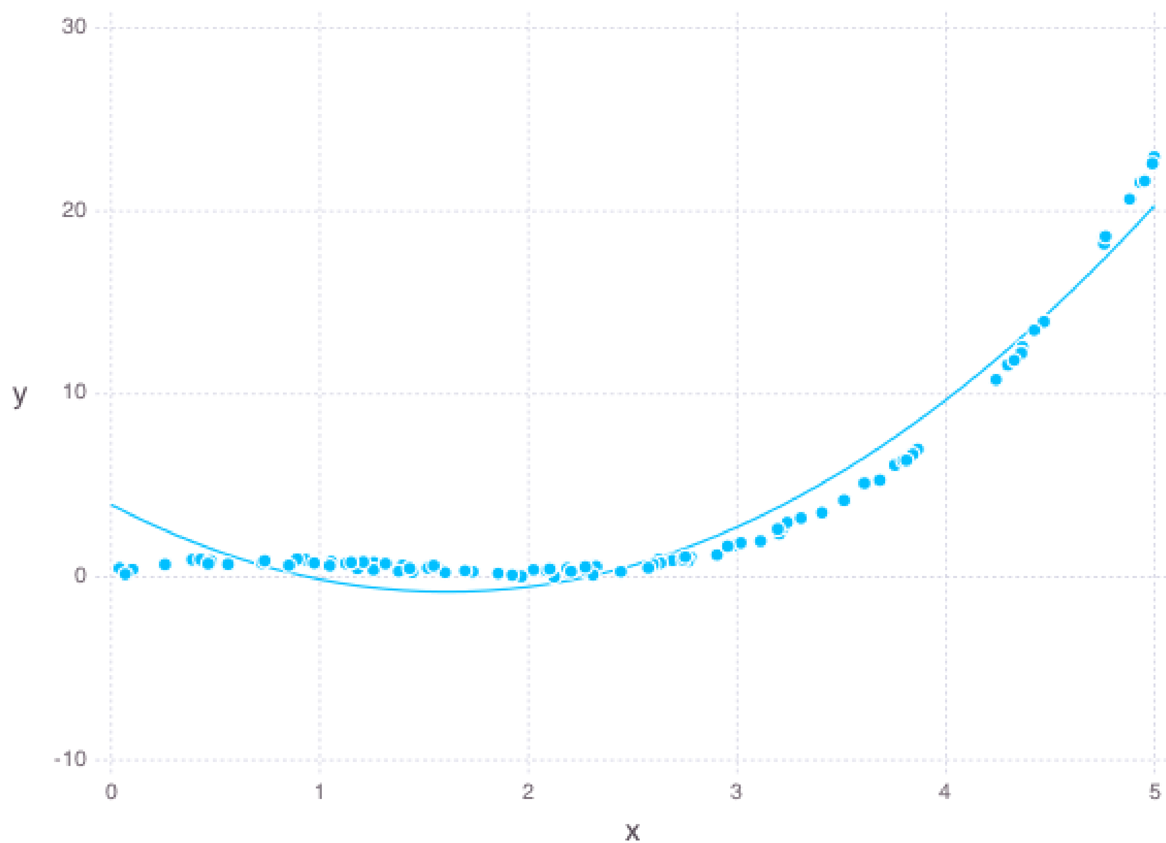
where the new coefficient γ corresponds to the quadratic term A similar residual function from the linear regression example can be used here; we measure the error of our quadratic fit by summing the squares of the residuals

$$E(\alpha, \beta, \gamma) = \sum_{i=1}^{n} (r(x_i, y_i))^2 = \sum_{i=1}^{n} (\alpha + \beta x_i + \gamma x_i^2 - y_i)^2.$$

Again, we pick our coefficients to minimize the error. Here is the Julia code to solve this problem using LLS and plot the quadratic:

```
quadratic_coeff = Variable()
slope = Variable()
offset = Variable()
quadratic = offset + x_data * slope + quadratic_coeff * x_data .^ 2
residuals = quadratic - y_data
fit_error = sum_squares(residuals)
optval = minimize!(fit_error)

# Create some evenly spaced points for plotting, again replicate powers
t = reshape([0 : 0.1 : 5], length([0 : 0.1 : 5]), 1)
t_squared = t .^ 2
```

```
# Plot our regressed function
p = plot(
  layer(x=x_data, y=y_data, Geom.point),
  layer(x=t, y=evaluate(offset) + t * evaluate(slope) + t_squared *␣
↪evaluate(quadratic_coeff), Geom.line),
  Theme(panel_fill=color("white"))
)
```



A much better fit than the line!

1.3.2 Control

A simple control problem on a system usually involves a variable $x(t)$ that denotes the state of the system over time, and a variable $u(t)$ that denotes the input into the system over time. Linear constraints are used to capture the evolution of the system over time:

$$x(t) = Ax(t-1) + Bu(t), \text{ for } t = 1, \ldots, T,$$

where the numerical matrices A and B are called the dynamics and input matrices, respectively.

The goal of the control problem is to find a sequence of inputs $u(t)$ that will allow the state $x(t)$ to achieve specified

values at certain times. For example, we can specify initial and final states of the system:

$$x(0) = x_i$$
$$x(T) = x_f$$

Additional states between the initial and final states can also be specified. These are known as waypoint constraints. Often, the input and state of the system will have physical meaning, so we often want to find a sequence inputs that also minimizes a least squares objective like the following:

$$\sum_{t=0}^{T} \|Fx(t)\|_2^2 + \sum_{t=1}^{T} \|Gu(t)\|_2^2,$$

where F and G are numerical matrices.

We'll now apply the basic format of the control problem to an example of controlling the motion of an object in a fluid over T intervals, each of h seconds. The state of the system at time interval t will be given by the position and the velocity of the object, denoted $p(t)$ and $v(t)$, while the input will be forces applied to the object, denoted by $f(t)$. By the basic laws of physics, the relationship between force, velocity, and position must satisfy:

$$p(t+1) = p(t) + hv(t)$$
$$v(t+1) = v(t) + ha(t)$$

Here, $a(t)$ denotes the acceleration at time t, for which we we use $a(t) = f(t)/m + g - dv(t)$, where m, d, g are constants for the mass of the object, the drag coefficient of the fluid, and the acceleration from gravity, respectively.

Additionally, we have our initial/final position/velocity conditions:

$$p(1) = p_i$$
$$v(1) = v_i$$
$$p(T+1) = p_f$$
$$v(T+1) = 0$$

One reasonable objective to minimize would be

$$\text{objective} = \mu \sum_{t=1}^{T+1} (v(t))^2 + \sum_{t=1}^{T} (f(t))^2$$

We would like to keep both the forces small to perhaps save fuel, and keep the velocities small for safety concerns. Here μ serves as a parameter to control which part of the objective we deem more important, keeping the velocity small or keeping the force small.

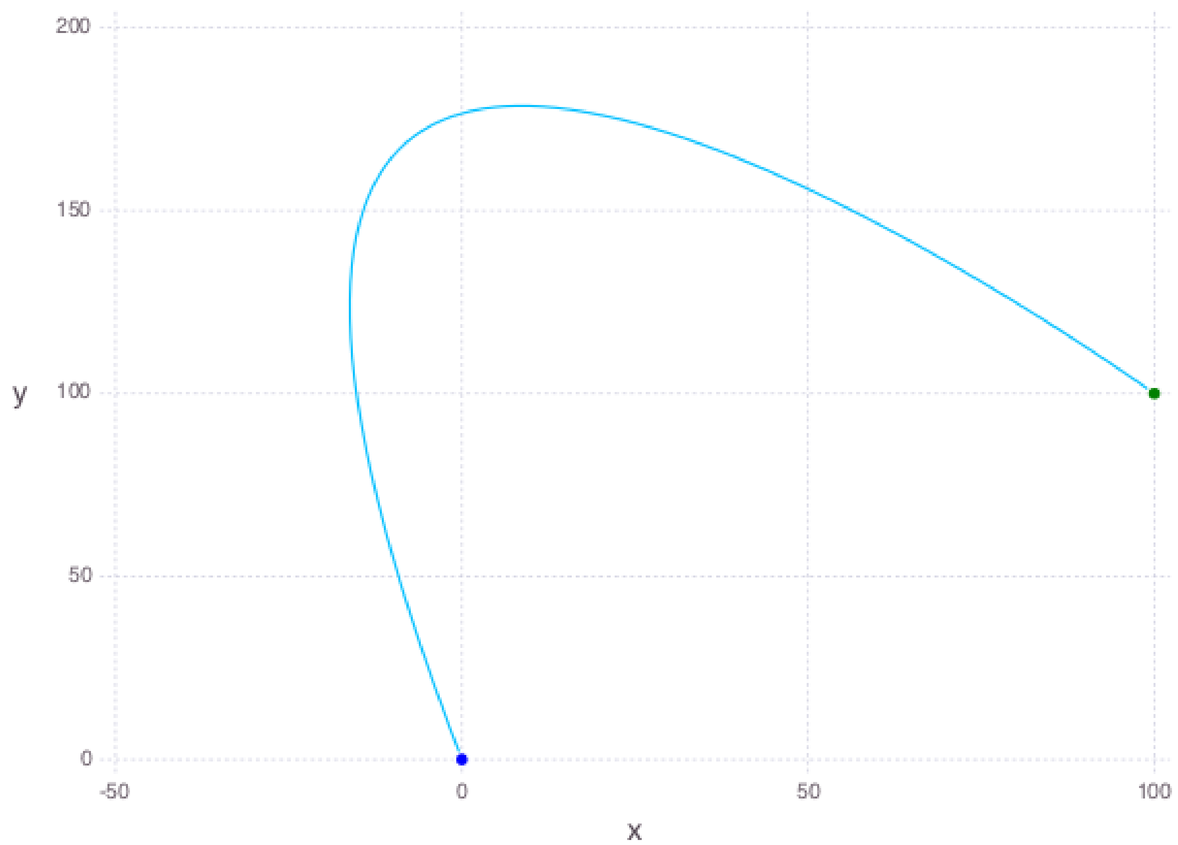
The following code builds and solves our control example:

```
# Some constraints on our motion
# The object should start from the origin, and end at rest
initial_velocity = [-20; 100]
final_position = [100; 100]

T = 100 # The number of timesteps
h = 0.1 # The time between time intervals
mass = 1 # Mass of object
drag = 0.1 # Drag on object
g = [0, -9.8] # Gravity on object

# Declare the variables we need
position = Variable(2, T)
velocity = Variable(2, T)
```

```
force = Variable(2, T - 1)

# Create a problem instance
mu = 1
constraints = []

# Add constraints on our variables
for i in 1 : T - 1
  constraints += position[:, i + 1] == position[:, i] + h * velocity[:, i]
end

for i in 1 : T - 1
  acceleration = force[:, i]/mass + g - drag * velocity[:, i]
  constraints += velocity[:, i + 1] == velocity[:, i] + h * acceleration
end

# Add position constraints
constraints += position[:, 1] == 0
constraints += position[:, T] == final_position

# Add velocity constraints
constraints += velocity[:, 1] == initial_velocity
constraints += velocity[:, T] == 0

# Solve the problem
optval = minimize!(sum_squares(force), constraints)
```

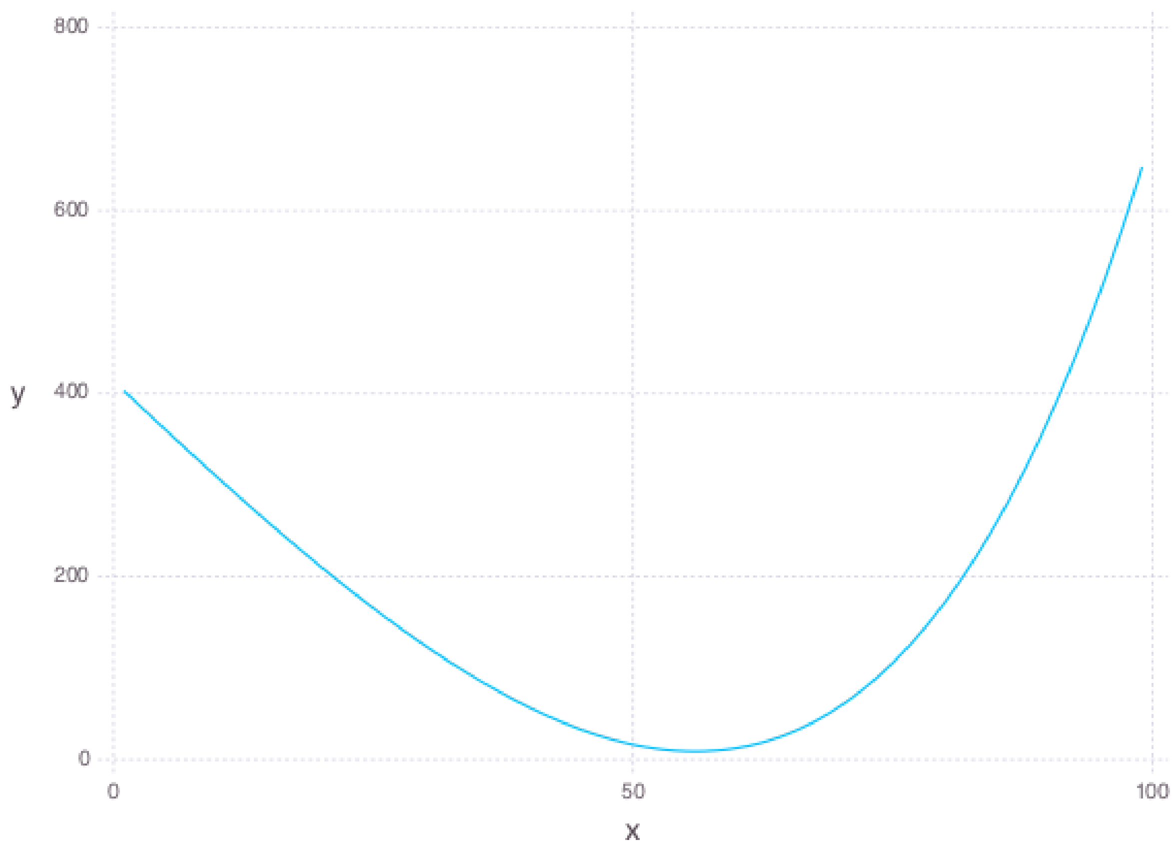
We can plot the trajectory taken by the object. The blue point denotes the initial position, and the green point denotes the final position.

```
pos = evaluate(position)
p = plot(
  layer(x=[pos[1, 1]], y=[pos[2, 1]], Geom.point, Theme(default_color=color("blue"))),
  layer(x=[pos[1, T]], y=[pos[2, T]], Geom.point, Theme(default_color=color("green
↪"))),
  layer(x=pos[1, :], y=pos[2, :], Geom.line(preserve_order=true)),
  Theme(panel_fill=color("white"))
)
```



We can also see how the magnitude of the force changes over time.

```
p = plot(x=1:T, y=sum(evaluate(force).^2, 1), Geom.line, Theme(panel_fill=color("white
↪")))
```



1.3.3 Image Processing

Tomography

Tomography is the process of reconstructing a density distribution from given integrals over sections of the distribution. In our example, we will work with tomography on black and white images. Suppose x be the vector of n pixel densities, with x_j denoting how white pixel j is. Let y be the vector of m line integrals over the image, with y_i denoting the integral for line i. We can define a matrix A to describe the geometry of the lines. Entry A_{ij} describes how much of pixel j is intersected by line i. Assuming our measurements of the line integrals are perfect, we have the relationship that

$$y = Ax$$

However, anytime we have measurements, there are usually small errors that occur. Therefore it makes sense to try to minimize

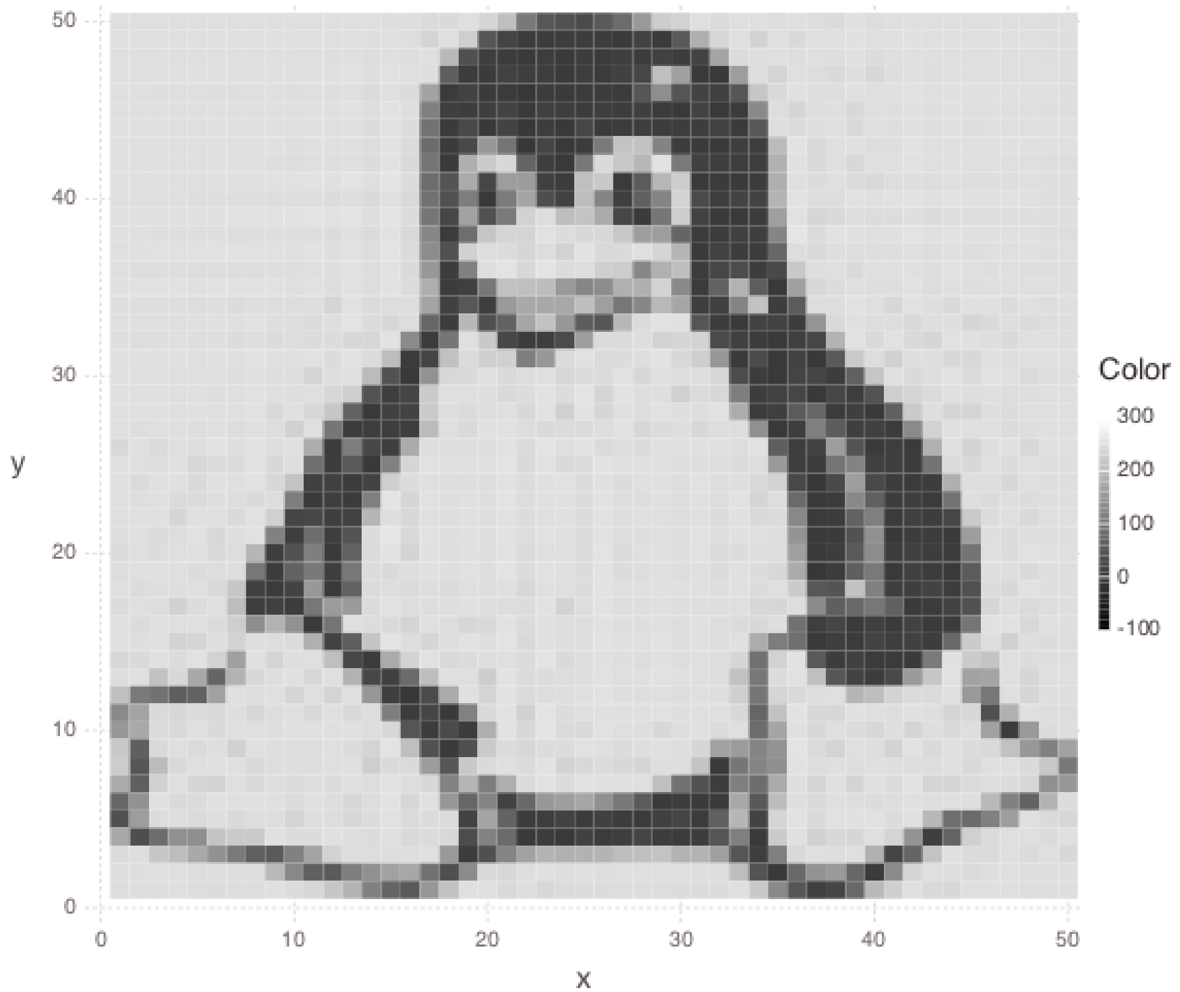
$$\|y - Ax\|_2^2.$$

This is simply an unconstrained least squares problem; something we can readily solve in LLS!

```
line_mat_x = readdlm("tux_sparse_x.txt")
line_mat_y = readdlm("tux_sparse_y.txt")
line_mat_val = readdlm("tux_sparse_val.txt")
```

```
line_vals = readdlm("tux_sparse_lines.txt")

# Form the sparse matrix from the data
# Image is 50 x 50
img_size = 50
# The number of pixels in the image
num_pixels = img_size * img_size

line_mat = spzeros(3300, num_pixels)

num_vals = length(line_mat_val)

for i in 1:num_vals
  x = int(line_mat_x[i])
  y = int(line_mat_y[i])
  line_mat[x + 1, y + 1] = line_mat_val[i]
end

x = Variable(num_pixels)
objective = sum_squares(line_mat * x - line_vals)
optval = minimize!(objective)

rows = zeros(img_size*img_size)
cols = zeros(img_size*img_size)
for i = 1:img_size
  for j = 1:img_size
    rows[(i-1)*img_size + j] = i
    cols[(i-1)*img_size + j] = img_size + 1 - j
  end
end

p = plot(
  x=rows, y=cols, color=reshape(evaluate(x), img_size, img_size), Geom.rectbin,
  Scale.ContinuousColorScale(Scale.lab_gradient(color("black"), color("white")))
)
```

The final result of the tomography will look something like



1.3.4 Machine Learning

Binary Classification

One common problem found in machine learning is the classification of a group of objects into two subgroups. In this example, we will try to separate sports articles from other texts in a collection of documents.

When classifying text documents, one of the most common techniques is to build a term-by-document frequency matrix F, where F_{ij} reflects the frequency of term j in document i.

The documents are then split into a training and testing set. For each document in the training example, we also label the document with a label. In this case, sports articles are labelled with a 1 and all other text documents are labelled with a -1. One reasonable approach to classify the documents is to model the label as an affine function of the term frequencies of the document:

$$\text{label}(i) = v + \sum_{j=1}^{n} w_j F_{ij}.$$

The goal now is to find a scalar v and a weight vector w, where w_j reflects how important term j is in determining the label of the document. In our context, a positive value means that the term is often seen in sports articles, while a negative value means the term is often seen in the other documents. One reasonable approach to finding the best w and v is to minimize the following objective:

$$\sum_{i=1}^{m}\left(\text{label}(i) - v - \sum_{j=1}^{n} w_j F_{ij}\right)^2 + \lambda \sum_{j=1}^{n} w_j^2$$

The first part of the objective is to ensure that our linear model actually closely reproduces the labels of our training documents. The second part of the objective ensures that the components of w are relatively small. Keeping w small allows our model to behave better on documents not in the training set. The regularization parameter λ is used to control how much we should prioritize keeping w small versus how close the affine function should fit the labels.

Here is the LLS code:

```
# read in the data
include("MatrixMarket.jl")
using MatrixMarket
A = full(MatrixMarket.mmread("largeCorpus.mtx"))

# extract the classes of each document
classes = A[:,1]
# TODO: modify classes so that 4 5 6 are 1 2 3
classes[classes .> 3] = classes[classes .> 3] - 3
A = A[:, 2:end]

# split into train/test
numData = size(A, 1)
data = randperm(numData)
ind = floor(numData*0.7)
training = data[1:ind]
test = data[ind+1:end]
trainDocuments = A[training,:]
trainClasses = classes[training,:]
testDocuments = A[test,:]
testClasses = classes[test,:]

# change all other than sports to -1 (sports is 1)
holdClass = 1
trainClasses[trainClasses .!= holdClass] = -1
trainClasses[trainClasses .== holdClass] = 1
testClasses[testClasses .!= holdClass] = -1
testClasses[testClasses .== holdClass] = 1

# build the problem and solve with LLS
lambda = 100
w = Variable(size(A, 2))
v = Variable()
objective = sum_squares(trainDocuments * w + v - trainClasses) + lambda * sum_
↪squares(w)
optval = minimize!(objective)
```

We can now sort our weight vector w to see which words were the most indicative of sports articles and which were most indicative of nonsports.

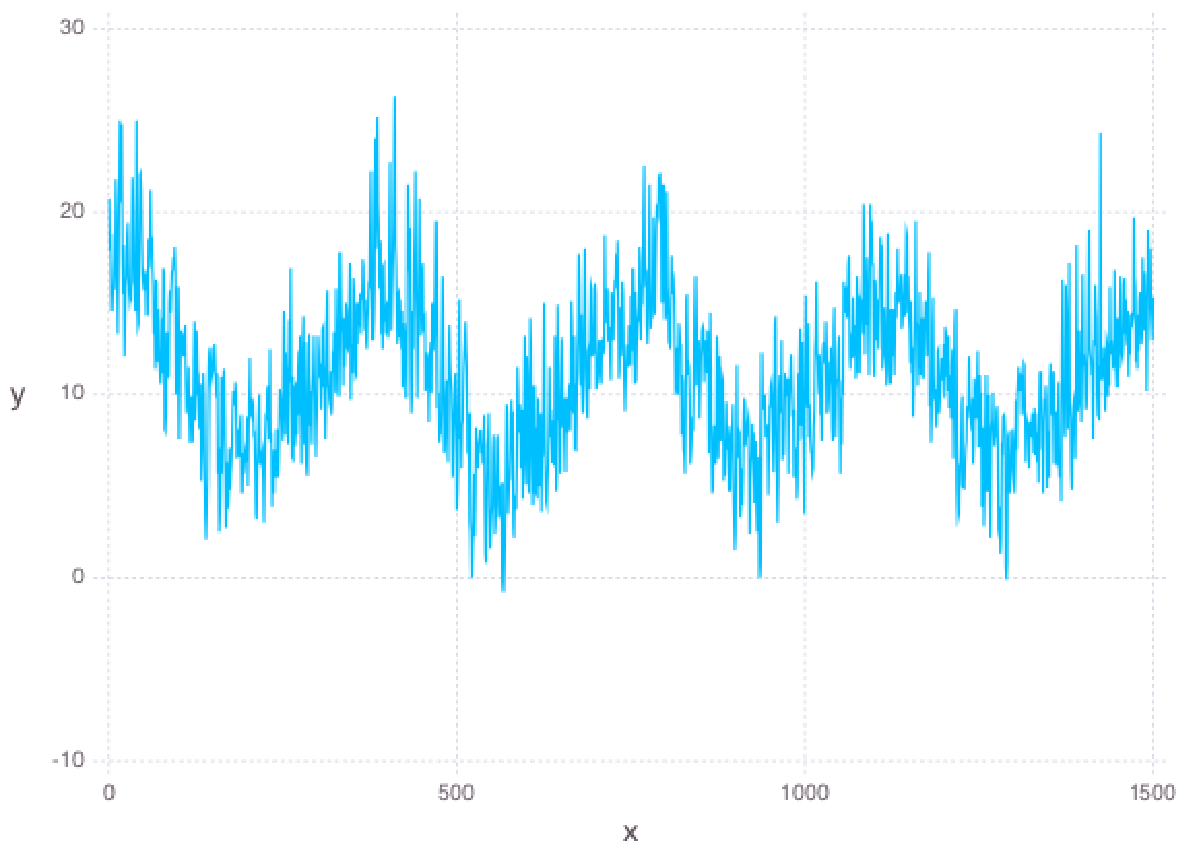
```
# print out the 5 words most indicative of sports and nonsports
words = String[]
```

```
f = open("largeCorpusfeatures.txt")
for i = 1:length(evaluate(w))
  push!(words, readline(f))
end
indices = sortperm(vec(evaluate(w)))
for i = 1:5
  print(words[indices[i]])
end
for i = 0:4
  print(words[indices[length(words) - i]])
end
```

Each run will yield different words, but it'll be clear which words come from sports articles.

1.3.5 Time Series Analysis

A time series is a sequence of data points, each associated with a time. In our example, we will work with a time series of daily temperatures in the city of Melbourne, Australia over a period of a few years. Let x be the vector of the time series, and x_i denote the temperature in Melbourne on day i. Here is a picture of the time series:



We can quickly compute the mean of the time series to be 11.2. If we were to always guess the mean as the temperature of Melbourne on a given day, the RMS error of our guesswork would be 4.1. We'll try to lower this RMS error by coming up with better ways to model the temperature than guessing the mean.

A simple way to model this time series would be to find a smooth curve that approximates the yearly ups and downs. We can represent this model as a vector s where s_i denotes the temperature on the i-th day. To force this trend to repeat yearly, we simply want

$$s_i = s_{i+365}$$

for each applicable i.

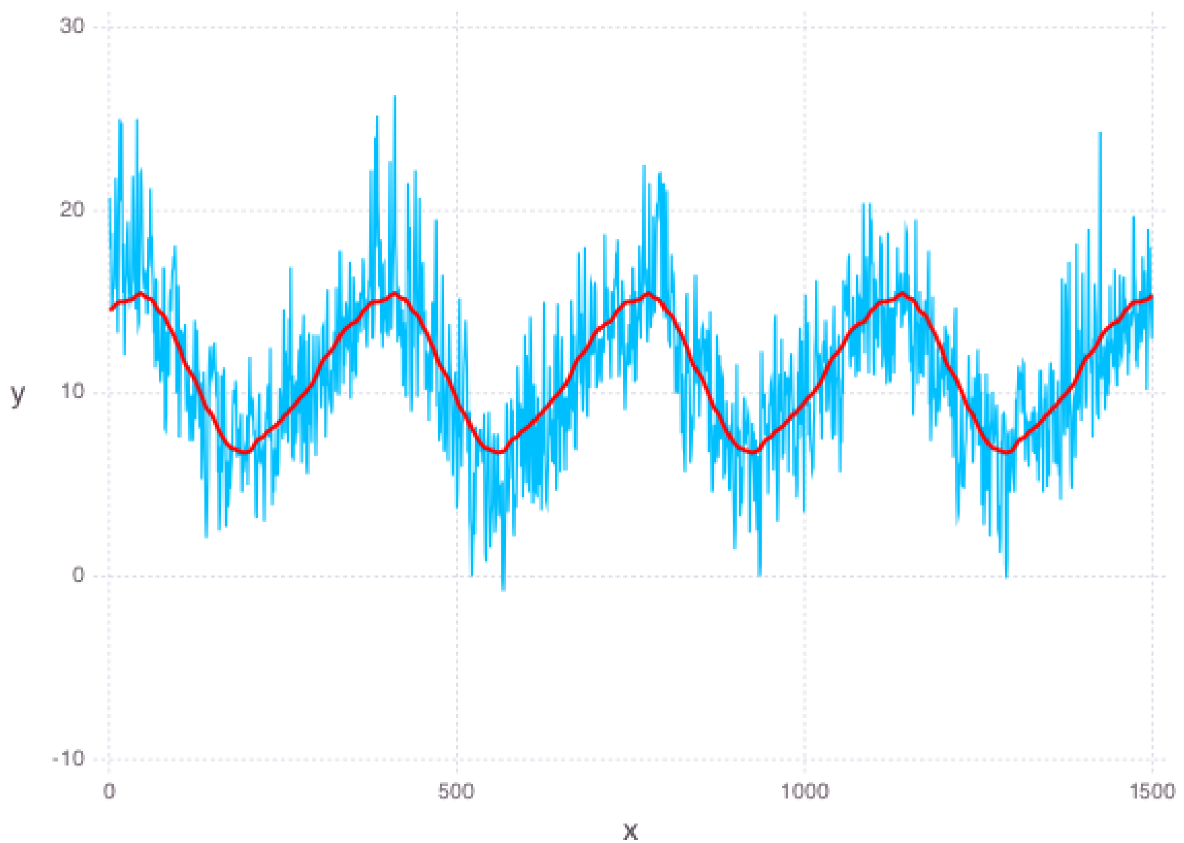
We also want our model to have two more properties. The first is that the temperature on each day in our model should be relatively close to the actual temperature of that day. The second is that our model needs to be smooth, so the change in temperature from day to day should be relatively small. The following objective would capture both properties:

$$\sum_{i=1}^{n}(s_i - x_i)^2 + \lambda \sum_{i=2}^{n}(s_i - s_{i-1})^2$$

where λ is the smoothing parameter. The larger λ is, the smoother our model will be.

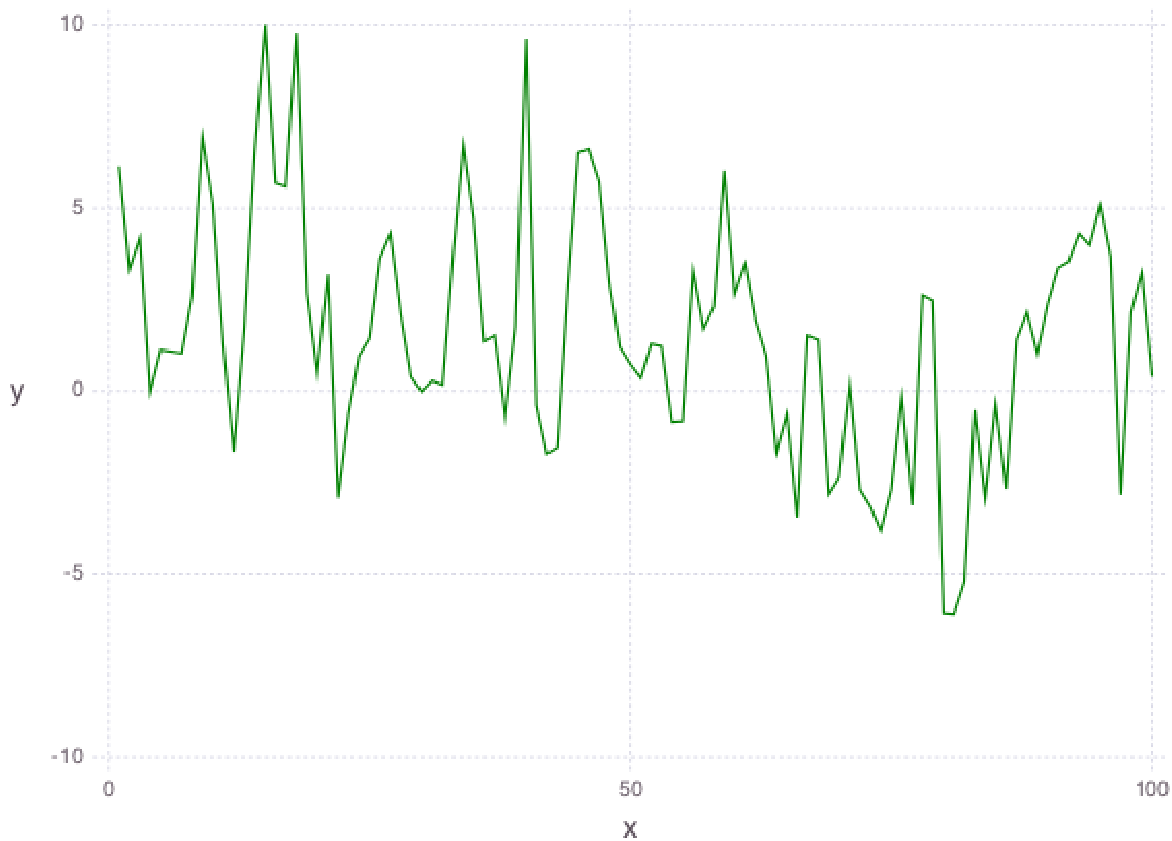
The following code uses LLS to find and plot the model:

```
temps = readdlm("melbourne_temps.txt", ',')
n = size(temps)[1]
p = plot(
  x=1:1500, y=temps[1:1500], Geom.line,
  Theme(panel_fill=color("white"))
)
# draw(PNG("melbourne.png", 16cm, 12cm), p)

yearly = Variable(n)
eq_constraints = []
for i in 365 + 1 : n
  eq_constraints += yearly[i] == yearly[i - 365]
end

smoothing = 100
smooth_objective = sum_squares(yearly[1 : n - 1] - yearly[2 : n])
optval = minimize!(sum_squares(temps - yearly) + smoothing * smooth_objective, eq_
↪constraints)
residuals = temps - evaluate(yearly)

# Plot smooth fit
p = plot(
  layer(x=1:1500, y=evaluate(yearly)[1:1500], Geom.line, Theme(default_color=color(
↪"red"), line_width=2px)),
  layer(x=1:1500, y=temps[1:1500], Geom.line),
  Theme(panel_fill=color("white"))
)
```



We can also plot the residual temperatures, r, define as $r = x - s$.

```
# Plot residuals for a few days
p = plot(
  x=1:100, y=residuals[1:100], Geom.line,
  Theme(default_color=color("green"), panel_fill=color("white"))
)
```



Our smooth model has a RMS error of 2.7, a significant improvement from just guessing the mean, but we can do better.

We now make the hypothesis that the residual temperature on a given day is some linear combination of the previous 5 days. Such a model is called autoregressive. We are essentially trying to fit the residuals as a function of other parts of the data itself. We want to find a vector of coefficients a such that

$$\mathrm{r}(i) \approx \sum_{j=1}^{5} a_j \mathrm{r}(i-j)$$

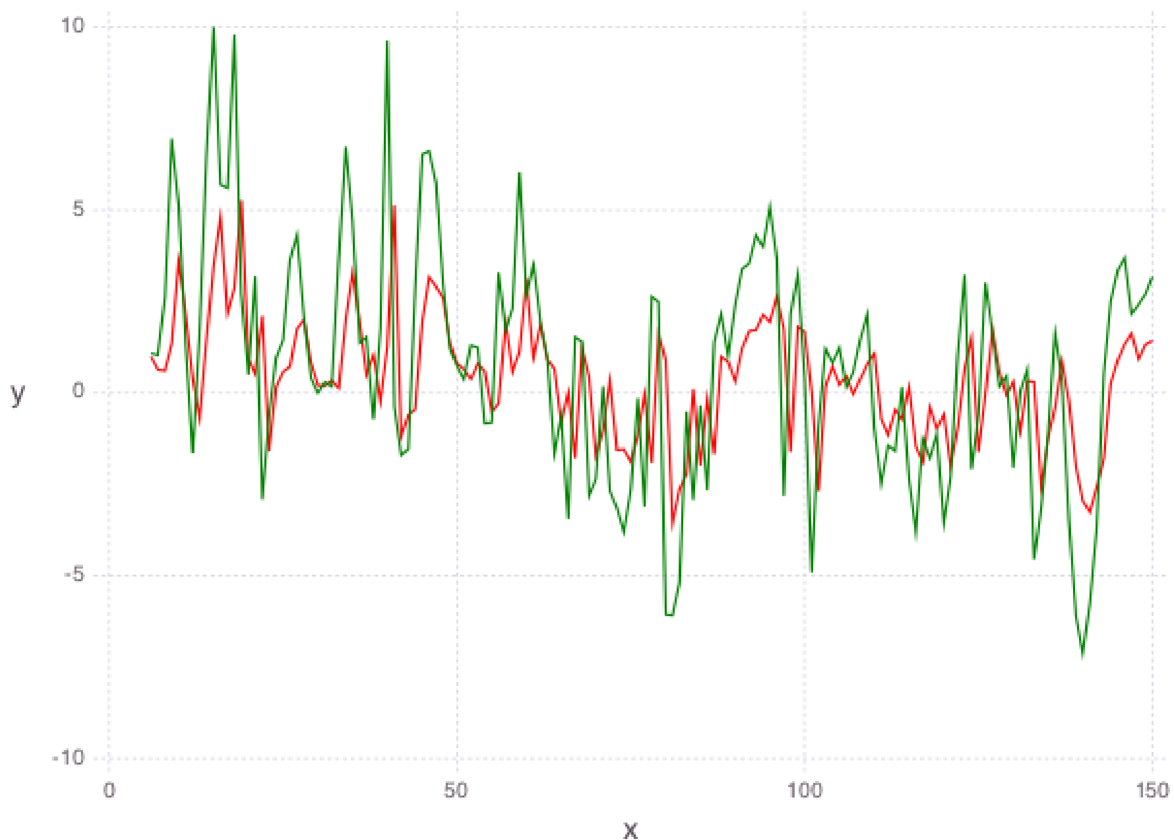
This can be done by simply minimizing the following sum of squares objective

$$\sum_{i=6}^{n} \left(\mathrm{r}(i) - \sum_{j=1}^{5} a_j \mathrm{r}(i-j) \right)^2$$

The following LLS code solves this problem and plots our autoregressive model against the actual residual temperatures:

```
# Generate the residuals matrix
ar_len = 5
residuals_mat = residuals[ar_len : n - 1]
for i = 1:ar_len - 1
  residuals_mat = [residuals_mat residuals[ar_len - i : n - i - 1]]
end
```

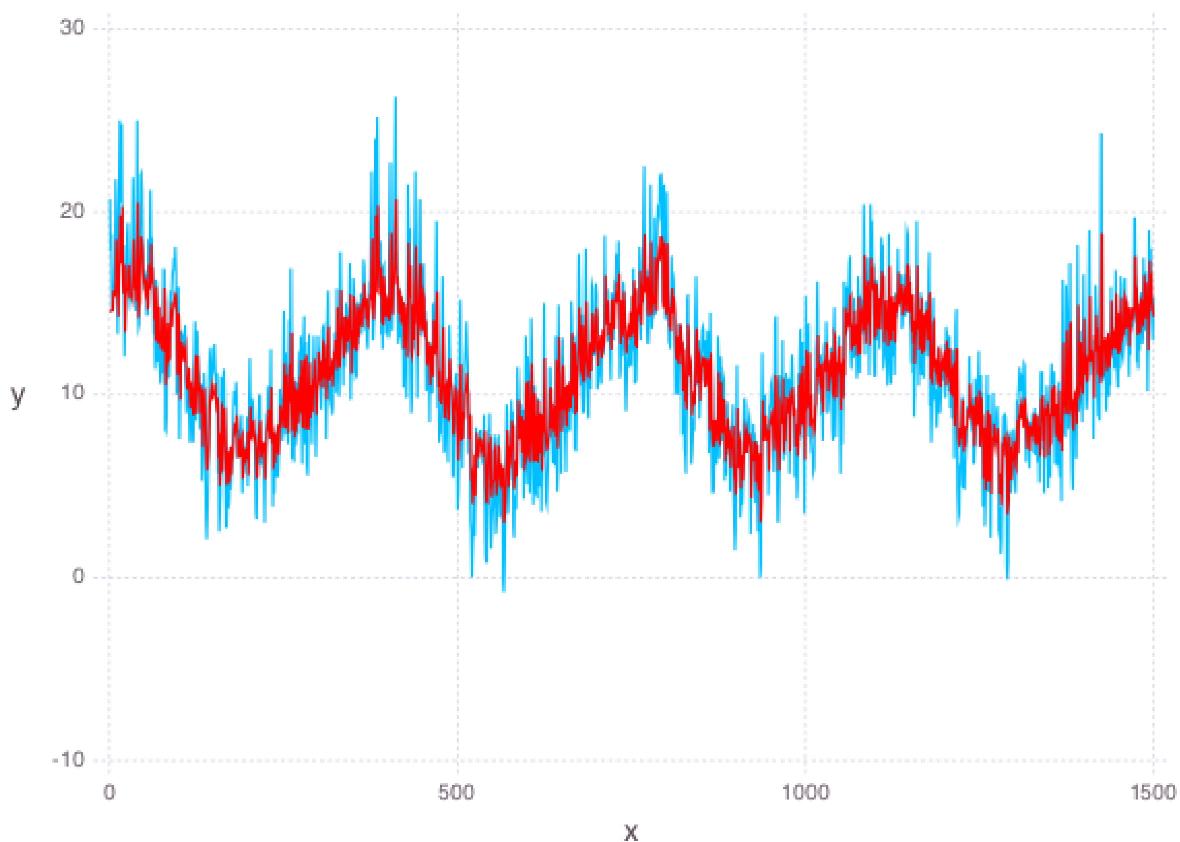
```
# Solve autoregressive problem
ar_coef = Variable(ar_len)
optval2 = minimize!(sum_squares(residuals_mat * ar_coef - residuals[ar_len + 1 :␣
↪end]))

# plot autoregressive fit of daily fluctuations for a few days
ar_range = 1:145
day_range = ar_range + ar_len
p = plot(
  layer(x=day_range, y=residuals[day_range], Geom.line, Theme(default_color=color(
↪"green"))),
  layer(x=day_range, y=residuals_mat[ar_range, :] * evaluate(ar_coef), Geom.line,␣
↪Theme(default_color=color("red"))),
  Theme(panel_fill=color("white"))
)
```



Now, we can add our autoregressive model for the residual temperatures to our smooth model to get an better fitting model for the daily temperatures in the city of Melbourne:

```
total_estimate = evaluate(yearly)
total_estimate[ar_len + 1 : end] += residuals_mat * evaluate(ar_coef)

# plot final fit of data
p = plot(
  layer(x=1:1500, y=total_estimate[1:1500], Geom.line, Theme(default_color=color("red
↪"))),
```

```
  layer(x=1:1500, y=temps[1:1500], Geom.line),
  Theme(panel_fill=color("white"))
)
```



The RMS error of this final model is 2.3.

1.4 Credits

LLS has been implemented in following languages:

- Julia: LinearLeastSquares.jl by David Zeng and Karanveer Mohan
- Python: lsqpy by Keegan Go

All implementations of LLS are released under the MIT license.

Much of the design of LLS was inspired by Convex.jl and Steven Diamond's CVXPY, similar software packages for solving the much more general class of convex optimization problems.

A huge thanks to Stephen Boyd for his feedback in both the design and documentation of LLS.