In [26]:
```
#Matrix specifying connections in the network
network = [ 0 1 1 0 0 0 0 0 0 0 1;
            1 0 1 0 0 0 0 1 1 0 0;
            1 1 0 1 0 0 0 0 1 1 1;
            0 0 1 0 1 1 0 0 0 0 1;
            0 0 0 1 0 0 0 0 1 0 1;
            0 0 0 1 0 0 1 0 1 1 0;
            0 0 0 0 0 1 0 1 0 1 0;
            0 1 0 0 0 0 1 0 0 1 0;
            0 1 1 0 1 1 0 0 0 1 0;
            0 0 1 0 0 1 1 1 1 0 0;
            1 0 1 1 1 0 0 0 0 0 0 ]

n = size(network)[2]

using JuMP
m = Model()

@defVar(m, flow[1:n,1:n] >= 0)   #flow[i, j] is amount going from node i to j

@addConstraint(m, sum(flow[10,:]) >= 1)   #Initial amount on starting node
#Only 1 unit of flow may pass through intermediate nodes
@addConstraint(m, inflowCap[c in 1:n-1], sum(flow[:,c]) <= 1)
@addConstraint(m, outflow[i in 1:n-2], sum(flow[i,:]) <= sum(flow[:,i]))   #Flow balance
#There is flow only through existing connections
@addConstraint(m, linkExists[r in 1:n, c in 1:n], network[r,c] - flow[r, c] >= 0)

@setObjective(m, Max, sum(flow[:, n]))    #Maximize flow towards end node

solve(m)
println("Number of disjoint paths: ", getObjectiveValue(m))
```

```
Number of disjoint paths: 4.0
```

The network is represented with a matrix, with 1 and 0 representing a connection and no connection respectively between two node. The decision variable is a matrix of a similar size, with the numbers representing amount of unit of flow between two nodes. The horizontal sum of column n is the total outflow from node n, and the vertical sum in column m is the total inflow into node m. The model maximizes the sum on column 11. Row 10 is allowed to have more than one unit. Only 1 unit may pass through the intermediate nodes, thus the amount of flow that ends up in column 11 is the number of disjoint paths between node 10 and 11.

The models here chose a, b that minimizes the total L2 error. In the first model all data are included, but in the second one x=3 and x=12 and the corresponding y-values are not included in the error calculation. Then the data and the linear fits are plotted. The second linear fit is lower than the first one, because the outlier y's being higher biases the fit upwards in order to minimize the error.

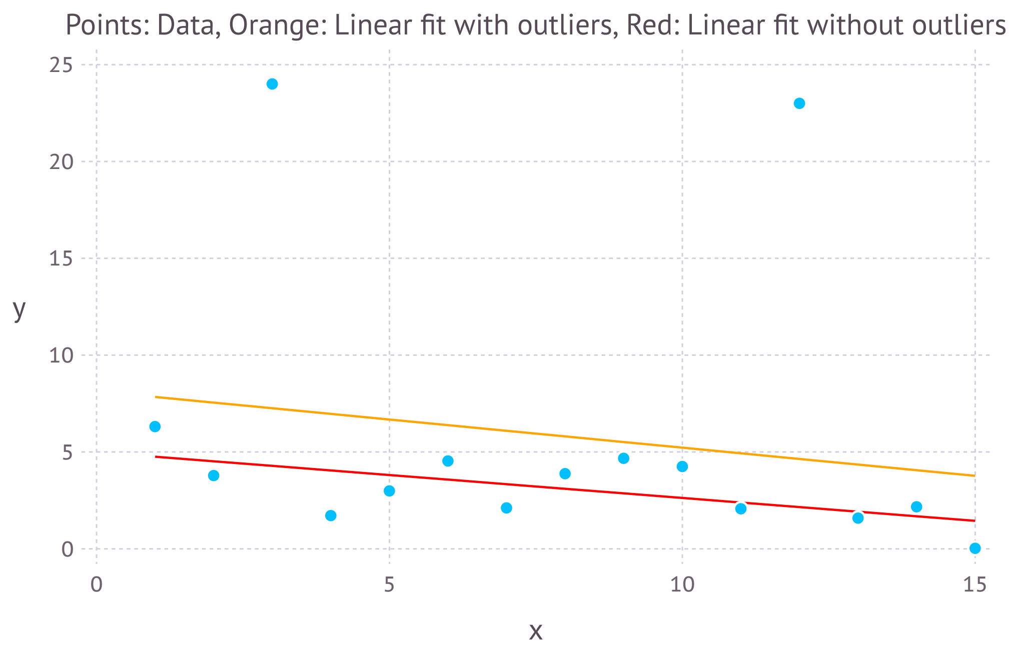
In [63]:
```
#x and y values
x = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
y = [6.31 3.78 24 1.71 2.99 4.53 2.11 3.88 4.67 4.25 2.06 23 1.58 2.17 0.02];
```

In [64]:
```
#Linear fit with all data
using JuMP
using Ipopt
m1 = Model(solver = IpoptSolver(print_level = 0))

@defVar(m1, a)   #Coefficient on x values
@defVar(m1, b)   #Constant term

@setObjective(m1, Min, sum((y - a*x - b).^2)) #Minimize l2 cost
solve(m1);
```

In [65]:
```
#Linear fit without outliers (x = 3, 12)
m2 = Model(solver = IpoptSolver(print_level = 0))
@defVar(m2, c)   #Coefficient on x values
@defVar(m2, d)   #Constant term
#Minimize l2 cost over included data
@setObjective(m2, Min, sum((y[1:2] - c*x[1:2] - d).^2) +  sum((y[4:11] - c*x[4:11] - d).^2) +
                       sum((y[13:15] - c*x[13:15] - d).^2))

solve(m2);
```

In [69]:
```
#Display data and linear fits
using Gadfly
plot(
layer(x = x, y = y, Geom.point),
layer(x = x, y = getValue(a)*x + getValue(b), Geom.line, Theme(default_color = colorant"orange")),
layer(x = x, y = getValue(c)*x + getValue(d), Geom.line, Theme(default_color = colorant"red")),
Guide.xlabel("x"), Guide.ylabel("y"),
Guide.title("Points: Data, Orange: Linear fit with outliers, Red: Linear fit without outliers")
)
```

Out[69]:

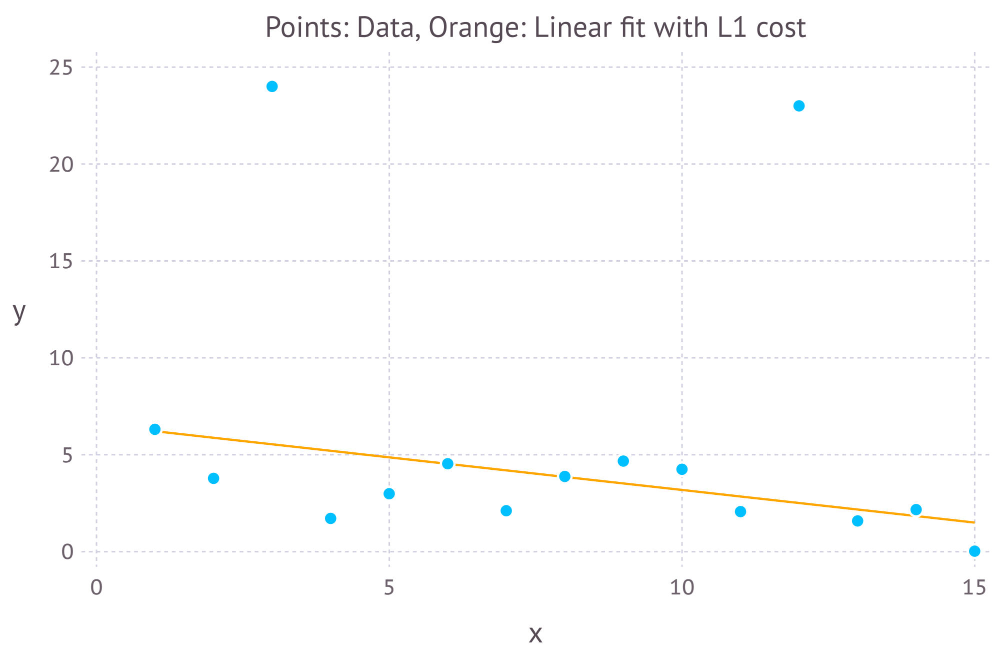


In [5]:
```
#x and y values
x = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
y = [6.31 3.78 24 1.71 2.99 4.53 2.11 3.88 4.67 4.25 2.06 23 1.58 2.17 0.02];
```

In [6]:
```
#Linear fit with all data using L1 cost
using JuMP
using Ipopt
m1 = Model(solver = IpoptSolver(print_level = 0))

#Absolute values of the differences between actual and predicted y's
@defVar(m1, absDiff[1:length(x)])
@defVar(m1, a)                          #Coefficient on x values
@defVar(m1, b)                          #Constant term

#Get the absolute values of the differences
@addConstraint(m1, posDiff[i in 1:length(x)], absDiff[i] >= y[i] - a*x[i] - b)
@addConstraint(m1, negDiff[i in 1:length(x)], absDiff[i] >= -(y[i] - a*x[i] - b) )

@setObjective(m1, Min, sum(absDiff))
solve(m1);
```

In [7]:
```
#Display data and linear fit
using Gadfly
plot(
layer(x = x, y = y, Geom.point),
layer(x = x, y = getValue(a)*x + getValue(b), Geom.line, Theme(default_color = colorant"orange")),
Guide.xlabel("x"), Guide.ylabel("y"), Guide.title("Points: Data, Orange: Linear fit with L1 cost")
)
```

Out[7]:



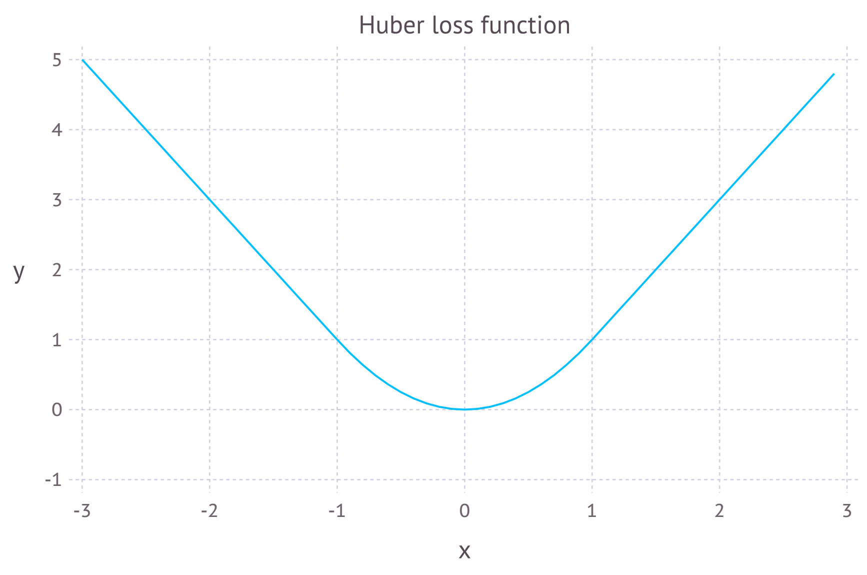
For getting the linear fit using L1 cost, an epigraph version of the previous model is used. An additional decision variable, an array AbsDiff, is used to store the absolute values of the differences between predicted and actual y values. Every value in this array is required to be greater than or equal to the difference at the corresponding x value. All those values are also required to be greater than the negative of the corresponding difference. The model minimizes the sum in this array, resulting it being populated with the absolute values of the differences.

The L1 cost handles outliers better than least squares does. This is because least squares square the differences. This exaggerates the large errors coming from outliers, thus biases the optimal linear fit in their direction.

In [3]:
```
using JuMP
using Ipopt

#Huber loss function for a given parameter M at point x
function HuberLoss(M, x, mod)
    m = mod
    @defVar(m, v >= 0)
    @defVar(m, w <= M)

    @addConstraint(m, x <= w + v)
    @addConstraint(m, -x <= w + v)

    @setObjective(m, Min, w^2 + 2*M*v)
    solve(m)
    return getObjectiveValue(m)
end;
```

In [4]:
```
#Display values of Huber loss for -3<= x <= 3
xHuber = [] #X-values
yHuber = [] #Y-values

i = -3.0
#Get resulting values of Huber loss function
while(i <= 3.0)
    m = Model(solver = IpoptSolver(print_level = 0))
    push!(xHuber, i)
    push!(yHuber, HuberLoss(1, i, m))
    i = i + 0.1
end

using Gadfly
plot(x = xHuber, y = yHuber, Geom.line, Guide.title("Huber loss function"))
```

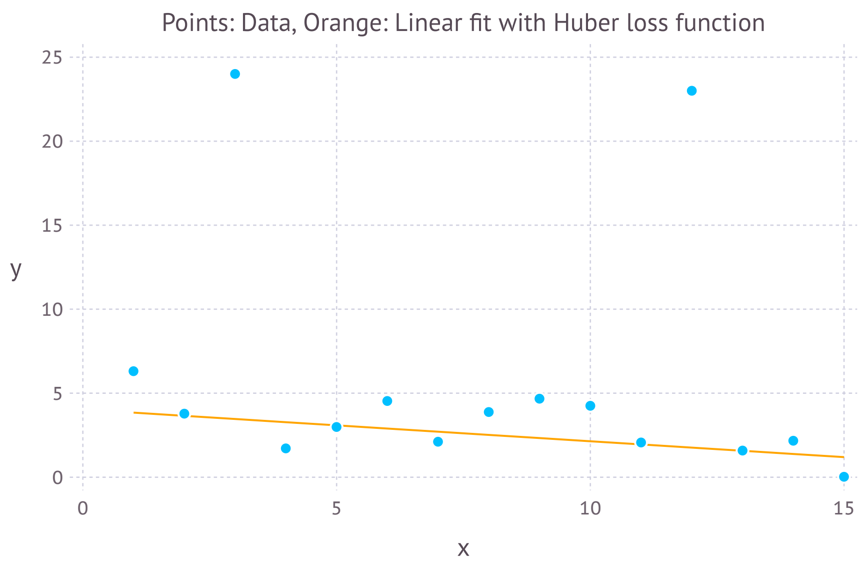
Out[4]:



In [6]:
```
#Data
x = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
y = [6.31 3.78 24 1.71 2.99 4.53 2.11 3.88 4.67 4.25 2.06 23 1.58 2.17 0.02]

#Finding the best fit for the data using Huber loss function
m2 = Model(solver = IpoptSolver(print_level = 0))

@defVar(m2, a)   #Coefficient on x values
@defVar(m2, b)   #Constant term

#Minimized total Huber loss
@setObjective(m2, Min, sum{HuberLoss(1, y[i] - a*x[i] - b, m2), i in 1:length(x)})
solve(m2);
```

In [7]:
```
#Display data and linear fit
using Gadfly
plot(
layer(x = x, y = y, Geom.point),
layer(x = x, y = getValue(a)*x + getValue(b), Geom.line, Theme(default_color = colorant"orange")),
Guide.xlabel("x"), Guide.ylabel("y"), Guide.title("Points: Data, Orange: Linear fit with Huber loss function")
)
```

Out[7]:



Here the Huber loss function is encoded as a Julia function. For the first task the x values are varied between -3 and 3 by 0.1 increments. For each increment the function is called with the current x value and the resulting Huber loss is stored. Then the Huber loss values in the range are plotted. For the second task a model passes the differences between predicted and actual y values into the function iteratively and the resulting losses are summed. The model minimizes the sum of Huber losses.

hw4_q3a

Geometric program:

Min $a_4 T^{-1} r^{-2}$

Subject to: $(a_1 T r w^{-1})/C_{max} + (a_2 r)/C_{max} + (a_3 r w)/C_{max} <= 1$
$T/T_{max} <= 1$, $T_{min} T^{-1} <= 1$
$r/r_{max} <= 1$, $r_{min} r^{-1} <= 1$
$w/w_{max} <= 1$, $w_{min} w^{-1} <= 1$
$w r^{-1}/0.1 <= 1$

Corresponding convex program:

Let x = log T, y = log r, z = log w.

Min $\log(a_4) - x - 2y$

Subject to $\log(e^{\log(a1/Cmax) + x + y - z} + e^{\log(a2/Cmax) + y} + e^{\log(a3/Cmax) + y + z}) <= 0$
$\log(T_{min}) <= x <= \log(T_{max})$
$\log(r_{min}) <= y <= \log(r_{max})$
$\log(w_{min}) <= z <= \log(w_{max})$
$z - y <= \log(0.1)$

This model is a JuMP version of the convex program shown in 3(a) with x, y, z as the log of the original variables T, r, w, as the decision variables. After the optimization is complete, x, y and z are raised by the natural exponent, e, to retrieve the value of original decision variable.

In [26]:
```
cMax = 500    #Maximum cost

a1 = a2 = a3 = a4 = 1  #Values of constants

using JuMP
using Ipopt
m = Model(solver = IpoptSolver(print_level=0))

@defVar(m, x)    #log of T, fluid temperature
@defVar(m, y)    #log of r, insulation thickness
@defVar(m, z)    #log of w, radius of pipe cross section

#Do not exceed maximum cost
@addNLConstraint(m, log(e^(log(a1/cMax) + x + y - z) +  e^(log(a2/cMax) + y)
                        + e^(log(a3/cMax) + y + z) ) <= 0 )
#Original variables are positive
@addNLConstraint(m, e^x >= 0)
@addNLConstraint(m, e^y >= 0)
@addNLConstraint(m, e^z >= 0)
@addConstraint(m, z - y <= log(0.1))       #w is smaller than r

@setObjective(m, Min, log(a4) - x - 2*y)   #Minimize total cost
solve(m)

println("Optimal variable values")
println("T: ", e^getValue(x))
println("r: ", e^getValue(y))
println("w: ", e^getValue(z))
```

```
Optimal variable values
T: 23.840239439054596
r: 46.39042809089772
w: 4.6390427937555
```