1. Moving averages [10 pts]

by Roumen Guha on Sunday, February 26th, 2017

There are many ways to model the relationship between an input sequence $\{u_1, u_2, \ldots\}$ and an output sequence $\{y_1, y_2, \ldots\}$. In class, we saw the *moving average* (MA) model, where each output is approximated by a linear combination of the k most recent inputs:

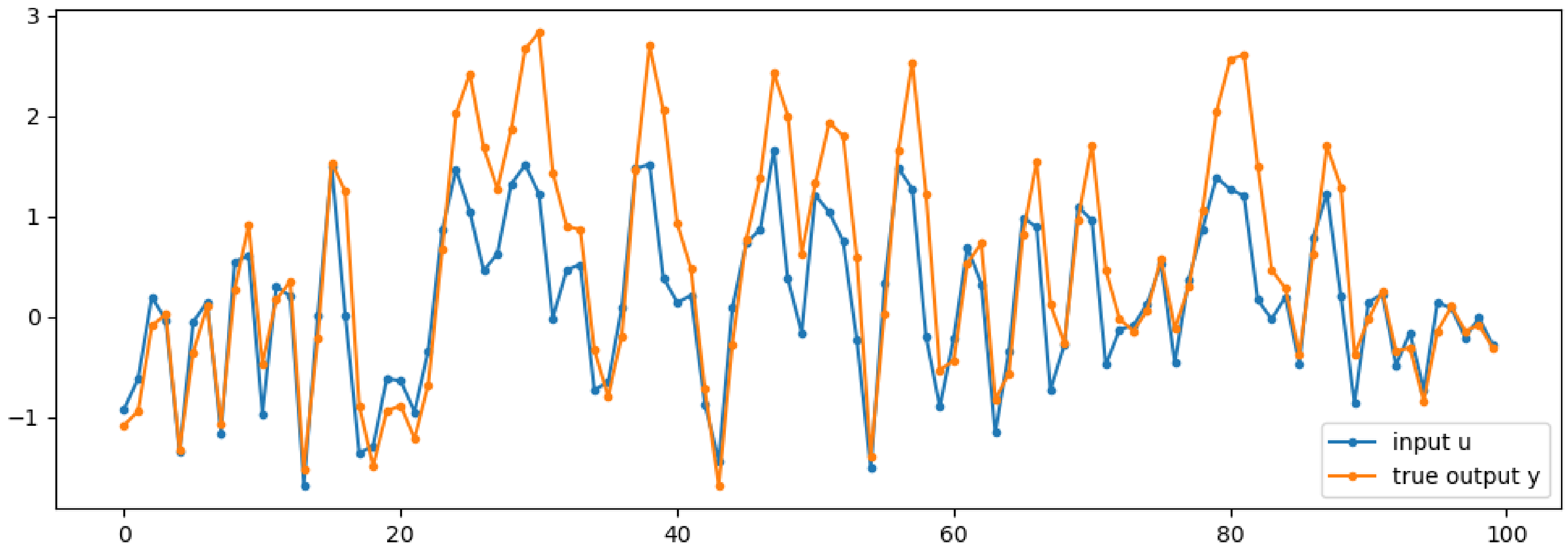
$$\text{MA:} \qquad y_t \approx b_1 u_t + b_2 u_{t-1} + \cdots + b_k u_{t-k+1}$$

We then used least-squares to find the coefficients $b_1, \ldots, b_k$. What if we didn't have access to the inputs at all, and we were asked to predict future y values based *only* on the previous y values? One way to do this is by using an *autoregressive* (AR) model, where each output is approximated by a linear combination of the l most recent outputs (excluding the present one):

$$\text{AR:} \qquad y_t \approx a_1 y_{t-1} + a_2 y_{t-2} + \cdots + a_\ell y_{t-\ell}$$

Of course, if the inputs contain pertinent information, we shouldn't expect the AR method to outperform the MA method!

In [139]:
```julia
# Load the data file (ref: Boyd/263)
raw = readcsv("uy_data.csv");
u = raw[:,1]; # inputs
y = raw[:,2]; # true output
T = length(u)

# plot the u and y data
using PyPlot
figure(figsize=(12,4))
plot([u y],".-");
legend(["input u", "true output y"], loc="lower right");
```



a) Using the same dataset from class **uy_data.csv**, plot the true y, and on the same axes, also plot the estimated $\hat{y}$ using the MA model and the estimated $\hat{y}$ using the AR model. Use $k = l = 5$ for both models. To quantify the difference between estimates, also compute $\|y - \hat{y}\|$ for both cases.

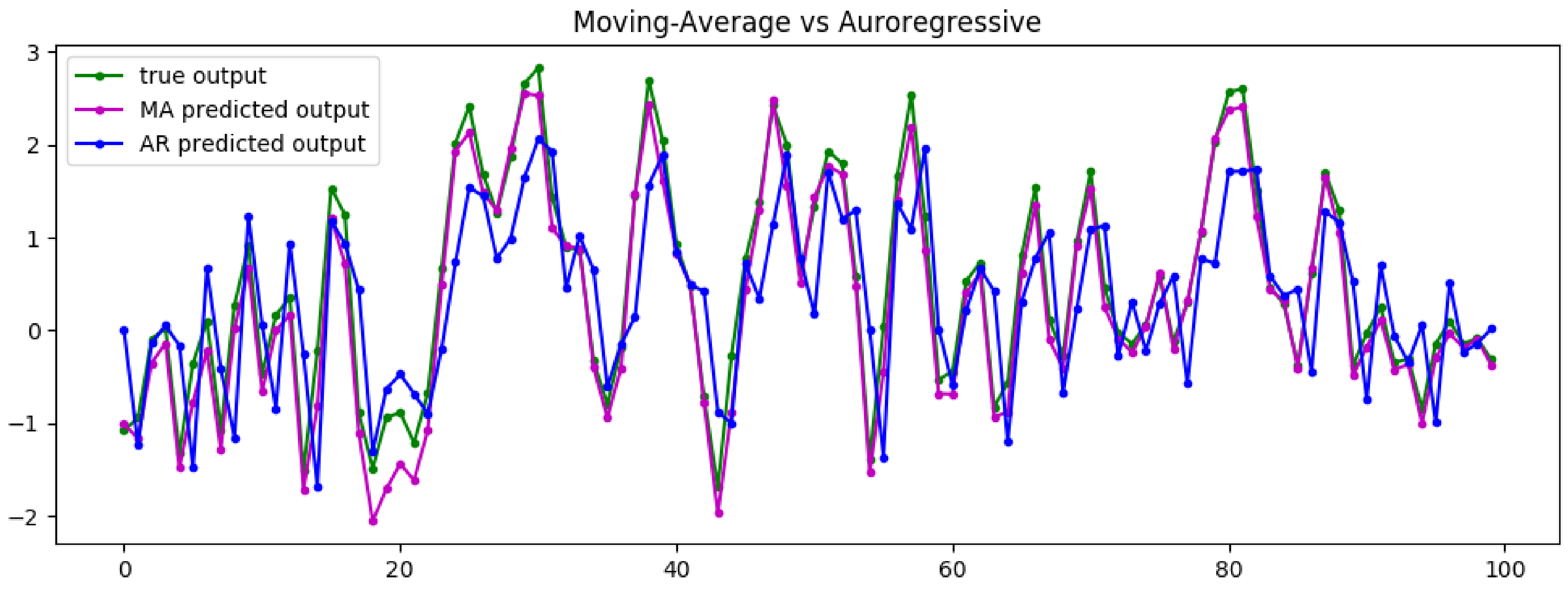
```
In [140]: # Moving-average (MA)
k = 5
A_MA = zeros(T, k)

for i = 1:k
    A_MA[i:end, i] = u[1:end-i+1]
end

wopt_MA = A_MA\y
yest_MA = A_MA*wopt_MA
println(wopt_MA)

# compute the error that the moving-average model makes
MaxWidth = 40
err_MA = zeros(MaxWidth)
for width = 1:MaxWidth
    AMA = zeros(T,width)
    for i = 1:width
        AMA[i:end,i] = u[1:end-i+1]
    end
    wMA = AMA\y
    yMAest = AMA*wMA
    err_MA[width] = norm(y-yMAest)
end
```

```
[1.1012,0.528947,0.262297,0.0521686,0.00421062]
```

In [141]:
```
# Autoregressive (AR)
l = 5
A_AR = zeros(T, l)

for i = 1:l
    A_AR[i+1:end, i] = y[1:end-i]
end

wopt_AR = A_AR\y
yest_AR = A_AR*wopt_AR
println(wopt_AR)

# compute the error that the autoregressive model makes
MaxWidth = 40
err_AR = zeros(MaxWidth)
for width = 1:MaxWidth
    AAR = zeros(T,width)
    for i = 1:width
        AAR[i+1:end, i] = y[1:end-i]
    end
    wAR = AAR\y
    yARest = AAR*wAR
    err_AR[width] = norm(y-yARest)
end
```

```
[1.1482,-0.876969,0.620235,-0.289534,0.139]
```

In [142]:
```
using PyPlot

figure(figsize=(12,4))
plot(y, "g.-", yest_MA, "m.-", yest_AR, "b.-")
legend(["true output", "MA predicted output", "AR predicted output"], loc="top left");
title("Moving-Average vs Auroregressive")

println("MA estimated error: ", norm(yest_MA - y))
println("AR estimated error: ", norm(yest_AR - y))
```



```
MA estimated error: 2.460854388269911
AR estimated error: 7.436691765656793
```

b) Yet another possible modeling choice is to combine both AR and MA. Unsurprisingly, this is called the autoregressive moving average (ARMA) model:

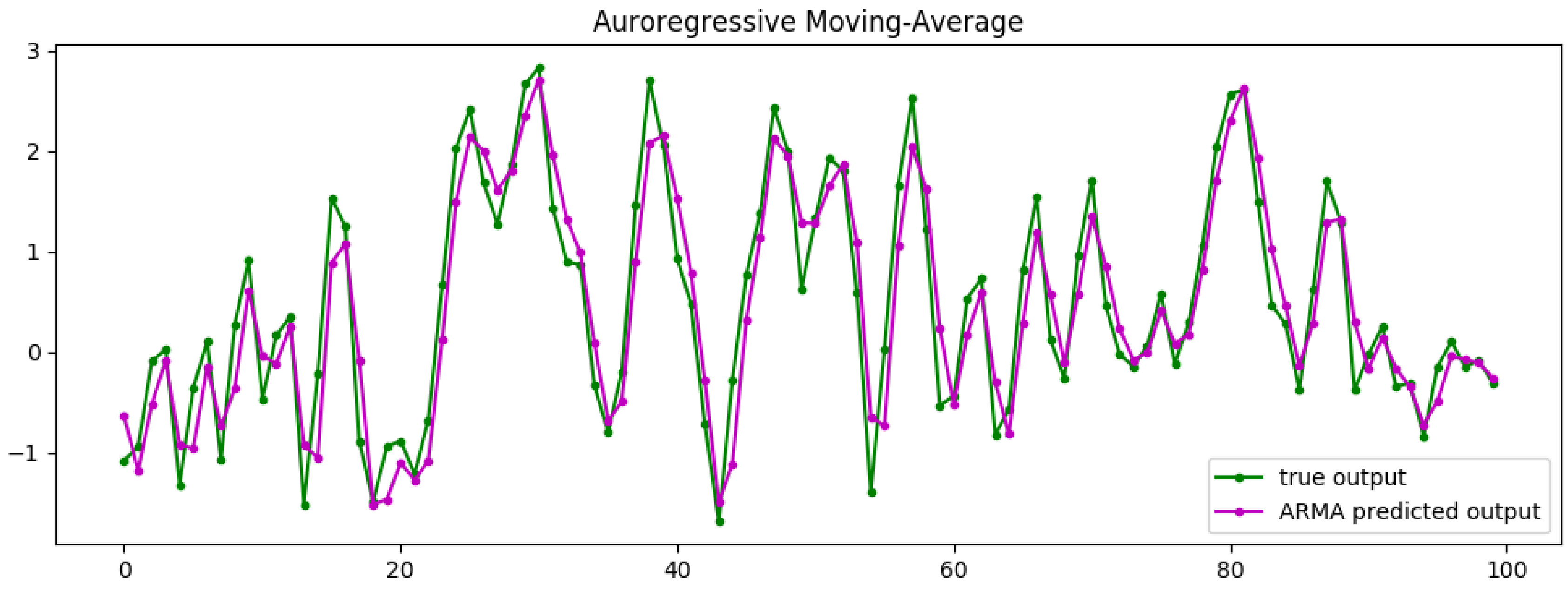
$$\text{ARMA:} \qquad y_t \approx a_1 y_{t-1} + a_2 y_{t-2} + \cdots + a_\ell y_{t-\ell} + b_1 u_t + b_2 u_{t-1} + \cdots + b_k u_{t-k+1}$$

Solve the problem once more, this time using an ARMA model with $k = l = 1$. Plot y and $\hat{y}$ as before, and also compute the error $\|y - \hat{y}\|$.

```
In [143]: # Autoregressive Moving-average (ARMA)
k = l = 1
A_MA = zeros(T, k)
A_AR = zeros(T, l)

for i = 1:k
    A_MA[i:end, i] = u[1:end-i+1]
end

for i = 1:l
    A_AR[i+1:end, i] = y[1:end-i]
end

A_ARMA = A_AR + A_MA

wopt_ARMA = A_ARMA\y
yest_ARMA = A_ARMA*wopt_ARMA
println(wopt_ARMA)

# compute the error that the autoregressive moving-average model makes
MaxWidth = 40
err_ARMA = zeros(MaxWidth)
for width = 1:MaxWidth
    AMA = zeros(T,width)
    for i = 1:width
        AMA[i:end, i] = u[1:end-i+1]
    end
    AAR = zeros(T,width)
    for i = 1:width
        AAR[i+1:end, i] = y[1:end-i]
    end
    AARMA = AAR + AMA
    wARMA = AARMA\y
    yARMAest = AARMA*wARMA
    err_ARMA[width] = norm(y-yARMAest)
end
```

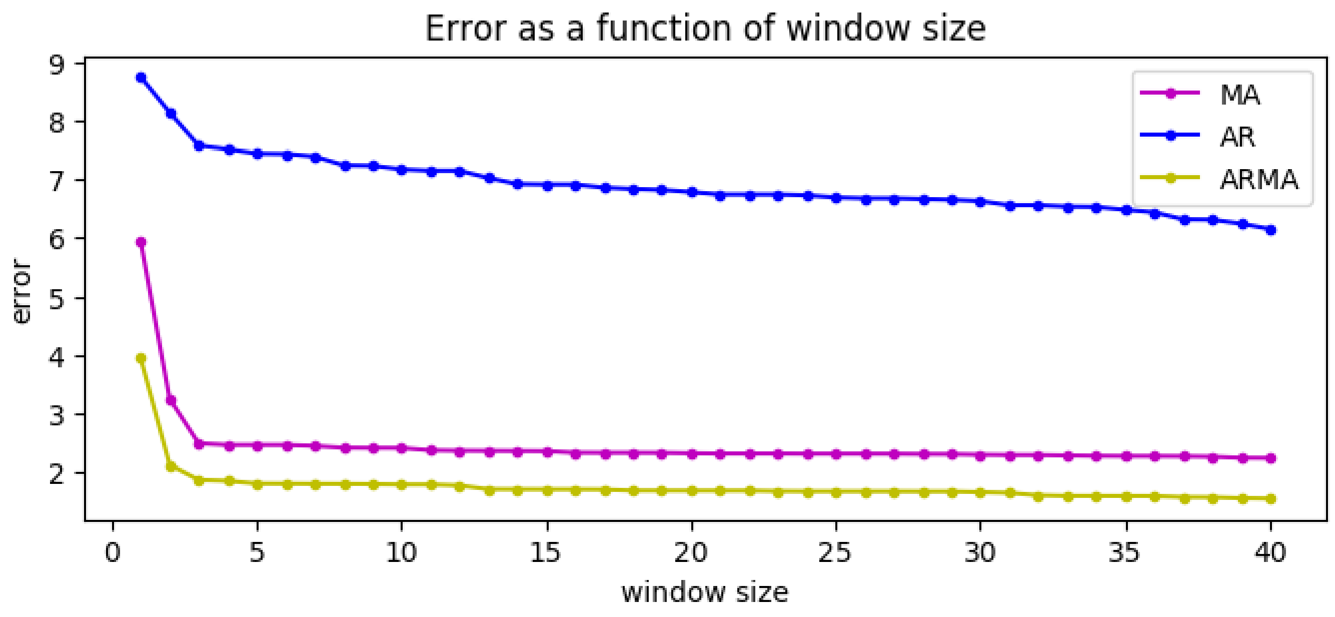
```
[0.696848]
```

```
In [144]: using PyPlot

          figure(figsize=(12,4))
          plot(y,"g.-",yest_ARMA,"m.-")
          legend(["true output", "ARMA predicted output"], loc="lower right");
          title("Auroregressive Moving-Average");
          println("ARMA estimated error: ", norm(yest_ARMA - y))
```



```
ARMA estimated error: 3.9412385247036528
```

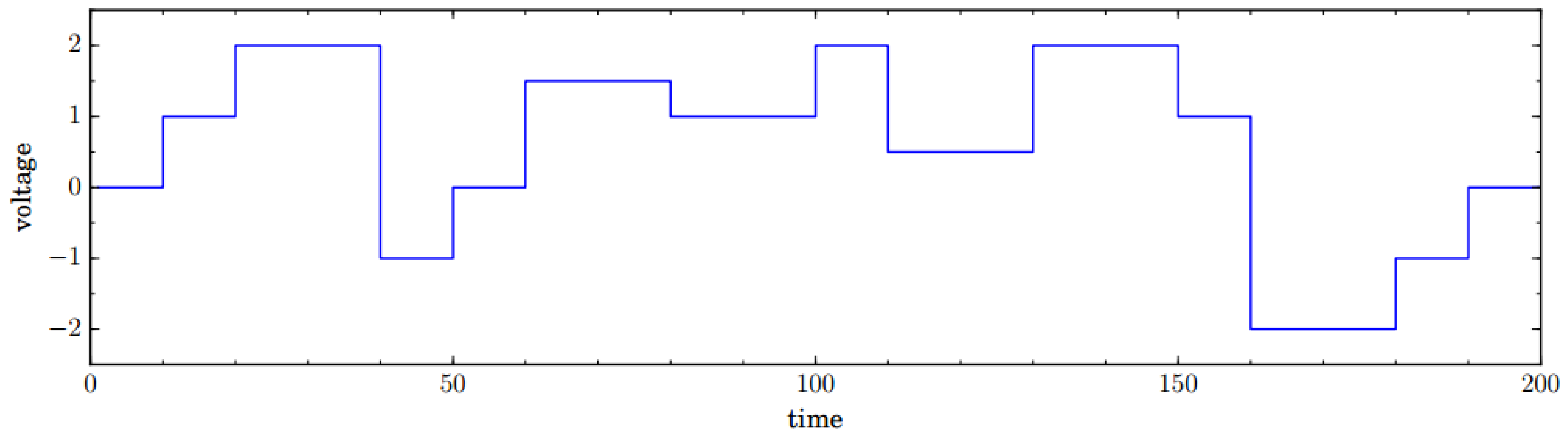
In [145]:
```
figure(figsize=(8,3))
title("Error as a function of window size")
plot(1:MaxWidth, err_MA, "m.-")
plot(1:MaxWidth, err_AR, "b.-")
plot(1:MaxWidth, err_ARMA, "y.-")
xlabel("window size")
ylabel("error")
legend(["MA", "AR", "ARMA"],loc="top right",fontsize=10)
;
```



2. Voltage smoothing [10 pts]

by Roumen Guha, on Sunday, February 26th, 2017

We would like to send a sequence of voltage inputs to the manipulator arm of a robot. The desired signal is shown in the plot below (also available in **voltages.csv**)

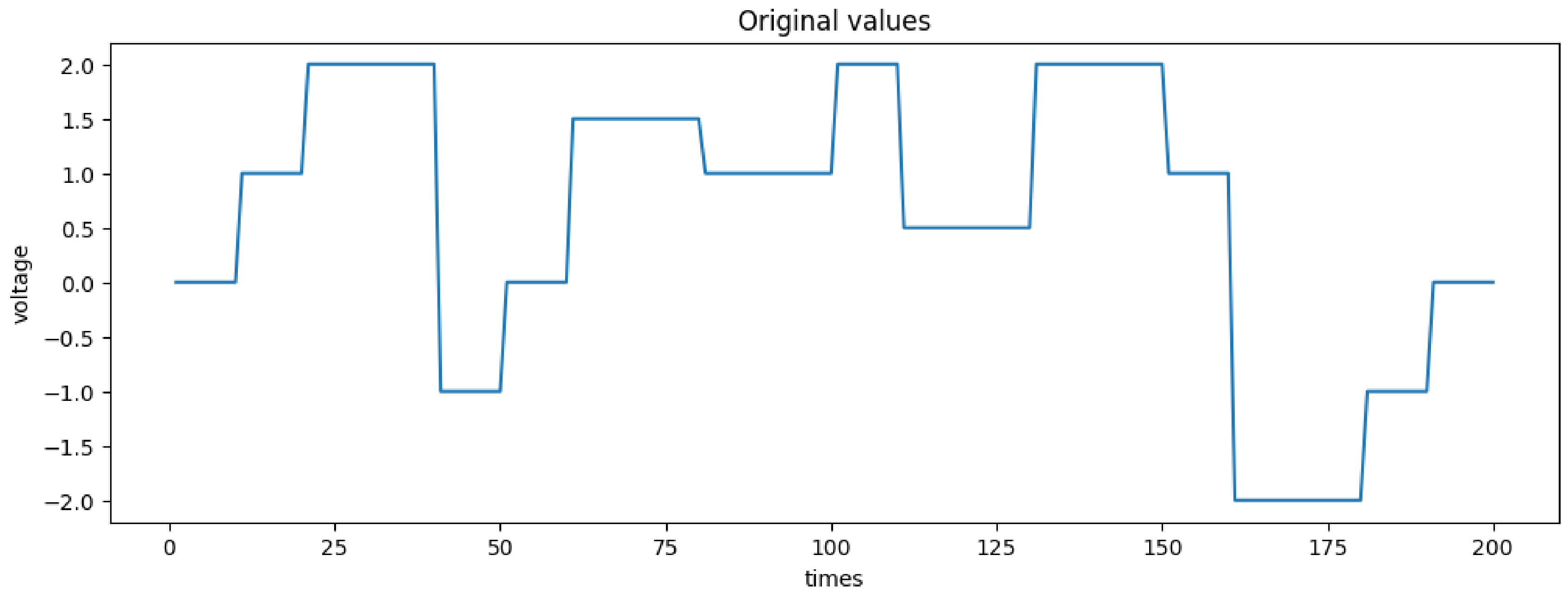


Unfortunately, abrupt changes in voltage cause undue wear and tear on the motors over time, so we would like to modify the signal so that the transitions are smoother. If the voltages above are given by $v_1, v_2, \ldots, v_{200}$, one way to characterize smoothness is via the sum of squared differences:

$$R(v) = (v_2 - v_1)^2 + (v_3 - v_2)^2 + \cdots + (v_{200} - v_{199})^2$$

When $R(v)$ is smaller, the voltage is smoother. Solve a regularized least squares problem that explores the tradeoff between matching the desired signal above and making the signal smooth. Explain your reasoning, and include a plot comparing the desired voltages with your smoothed voltages.

```
In [195]: raw = readcsv("voltages.csv");
          voltages = raw[:,1]; # inputs
          times = [1:length(voltages);]

          using PyPlot
          figure(figsize=(12,4))
          plot(times, voltages)
          title("Original values")
          xlabel("times")
          ylabel("voltage")
          ;
```

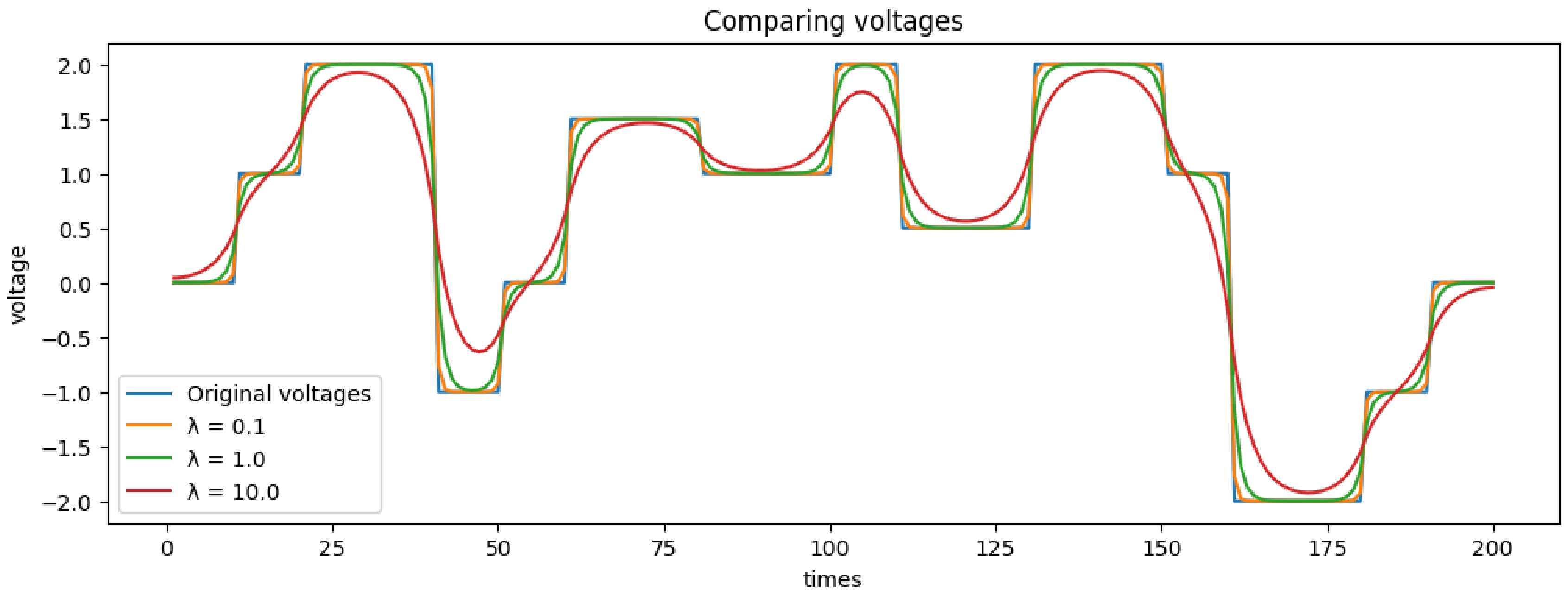


```
In [242]: using JuMP, Mosek, Gurobi

          function optimalVoltages(λ)

              m = Model(solver = GurobiSolver(OutputFlag=0))

              @variable(m, v[1:200])

              @objective(m, Min, sum((voltages[i] - v[i]).^2 for i in 1:times[end]) + λ*sum((v[i + 1] - v[i]).^2 for i in 

              solve(m)

              return (getvalue(v))
          end;
```

```
WARNING: Method definition optimalVoltages(Any) in module Main at In[240]:5 overwritten at In[242]:5.
```

The objective of this model is where the magic happens. I created a variable for the model, v, and in the objective stated that I'd like the difference between v and the intended voltages to be as small as possible. I also introduced an expression that wants the differences between the adjacent v values to be as small as possible so that changes in voltages are not too abrupt, with this expression being multiplied by λ, to be used as a tradeoff paramater. Below is a plot with 3 values of λ, at 0.1, 1.0, and 10.0.

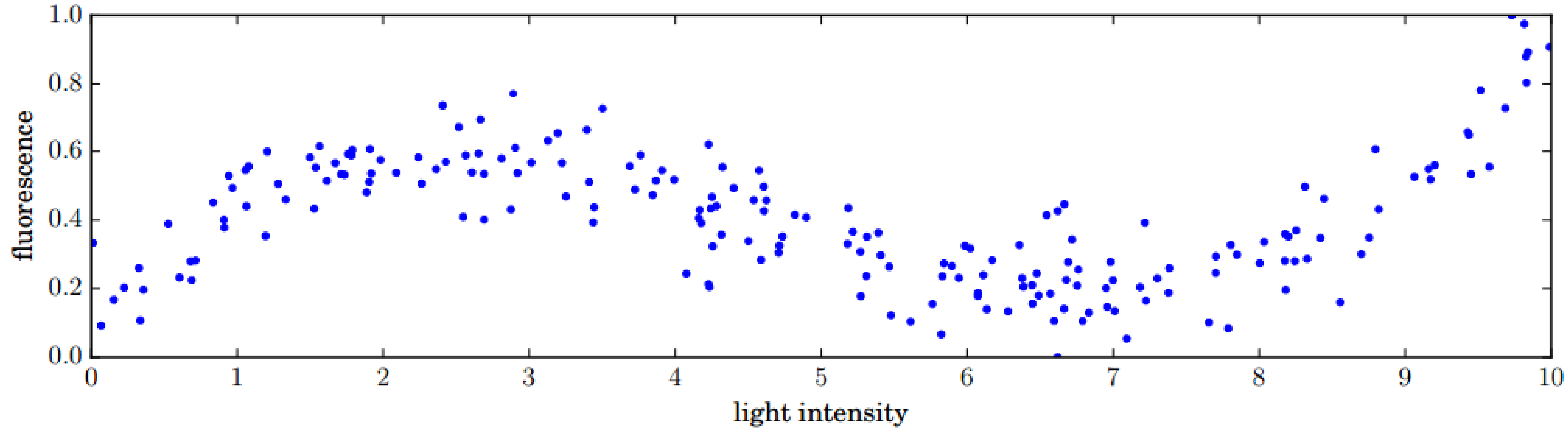
```
In [257]: using PyPlot
figure(figsize=(12,4))
plot(times, voltages)
plot(times, optimalVoltages(0.1))
plot(times, optimalVoltages(1))
plot(times, optimalVoltages(10))
title("Comparing voltages")
xlabel("times")
ylabel("voltage")
legend(["Original voltages", "λ = 0.1", "λ = 1.0", "λ = 10.0"], loc="lower left");
```



3. Spline fitting [10 pts]

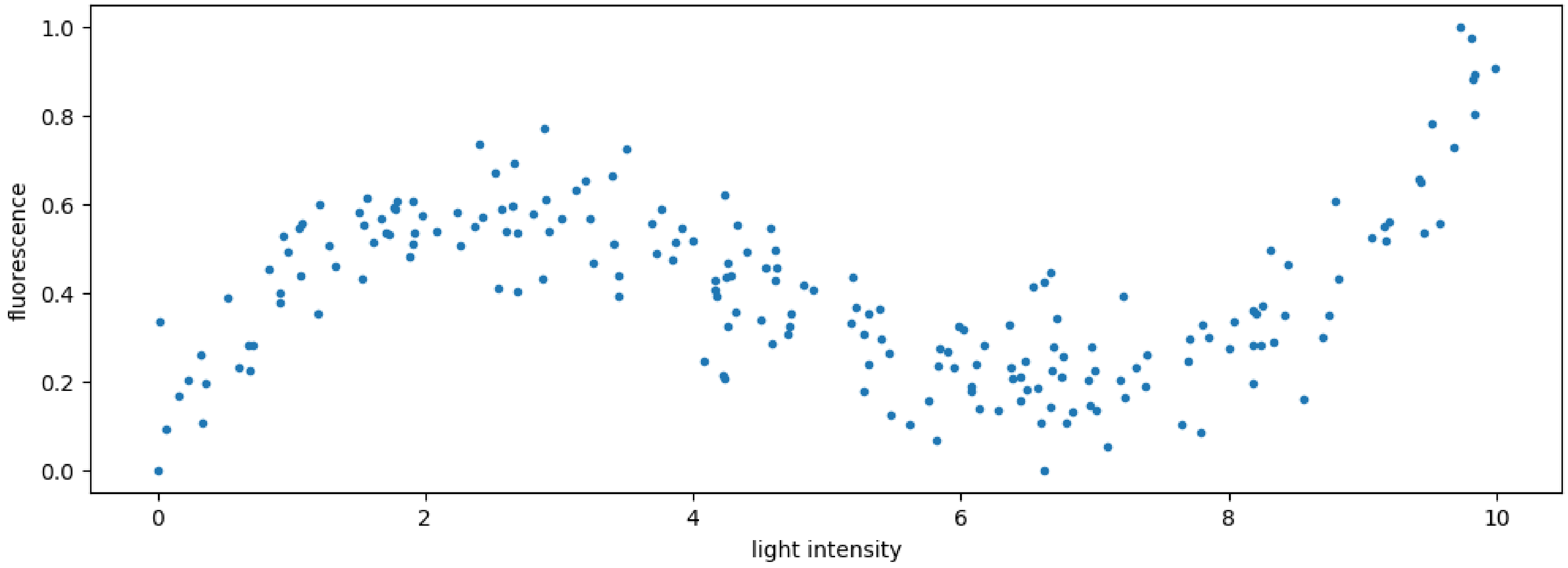
by Roumen Guha, on Sunday, February 26th, 2017

We are running a series of experiments to evaluate the properties of a new fluorescent material. As we vary the intensity of the incident light, the material should fluoresce different amounts. Unfortunately, the material isn't perfectly uniform and our method for measuring fluorescence is not very accurate. After testing 200 different intensities, we obtained the result below (also available in **xy_data.csv**). The intensities x_i and fluorescences y_i are recorded in the first and second columns of the data matrix, respectively.



The material has interesting nonlinear properties, and we would like to characterize the relationship between intensity and fluorescence by using an approximate model that agrees well with the trend of our experimental data. Although there is noise in the data, we know from physics that ***the fluorescence must be zero when the intensity is zero.*** This fact must be reflected in all of our models!

```
In [3]: raw = readcsv("xy_data - modified.csv") # modified original file to include 0 fluorescence when light intensity
        x = raw[:,1]; # light intensity
        y = raw[:,2]; # fluorescence

        using PyPlot
        figure(figsize=(12,4))
        plot(x,y, ".")
        xlabel("light intensity")
        ylabel("fluorescence")

        ;
```



a) Polynomial fit. Find the best cubic polynomial fit to the data. In other words, look for a function of the form

$$y = a_1 x^3 + a_2 x^2 + a_3 x + a4$$

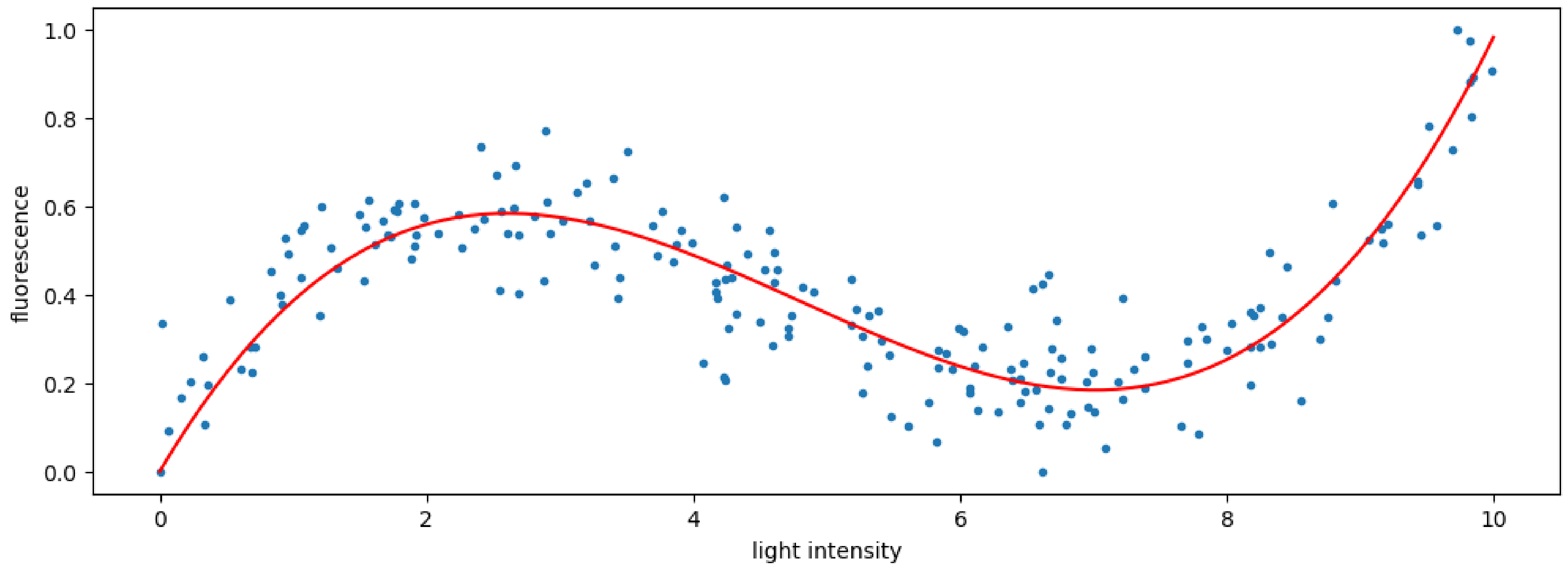
that has the best possible agreement with the data. Remember that the model should have zero fluorescence when the intensity is zero! Include a plot of the data along with your best-fit cubic on the same axes.

In [4]:
```
# order of polynomial to use
k = 3

# fit using a function of the form f(x) = u1 x^k + u2 x^(k-1) + ... + uk x + u{k+1}
n = length(x)
A = zeros(n,k+1)
for i = 1:n
    for j = 1:k+1
        A[i,j] = x[i]^(k+1-j)
    end
end
```

In [5]:
```
using JuMP, Gurobi, Mosek

m = Model(solver=MosekSolver(LOG=0))
#m = Model(solver=GurobiSolver(OutputFlag=0))

@variable(m, u[1:k+1])
@constraint(m, u[4] == 0)
@objective(m, Min, sum( (y - A*u).^2 ) )
status = solve(m)
uopt = getvalue(u)
println(status)
```

```
Optimal
```

In [6]:
```
using PyPlot

npts = 100
xfine = linspace(0,10,npts)
ffine = ones(npts)
for j = 1:k
    ffine = [ffine.*xfine ones(npts)]
end
yfine = ffine * uopt

figure(figsize=(12,4))
plot( x, y, ".")
plot( xfine, yfine, "r-")
xlabel("light intensity")
ylabel("fluorescence")

;
```



b) Spline fit. Instead of using a single cubic polynomial, we will look for a fit to the data using two quadratic polynomials. Specifically, we want to find coefficients p_i and q_i so that our data is well modeled by the piecewise quadratic function:

$$y = \begin{cases} p_1 x^2 + p_2 x + p_3 & \text{if } 0 \le x < 4 \\ q_1 x^2 + q_2 x + q_3 & \text{if } 4 \le x < 10 \end{cases}$$

These quadratic functions must be designed so that:

- as in the cubic model, there is zero fluorescence when the intensity is zero.
- both quadratic pieces have the same value at x = 4.
- both quadratic pieces have the same slope at x = 4.

In other words, we are looking for a *smooth* piecewise quadratic. This is also known as a *spline* (this is just one type of spline, there are many other types!). Include a plot of the data along with your best-fit model.

In [7]:
```
k = 2

# fit using a function of the form f(x) = u1 x^k + u2 x^(k-1) + ... + uk x + u{k+1}
n1 = 78 # last index before x[78] which contains a value > 4.0
A = zeros(n1,k+1)
for i = 1:n1
    for j = 1:k+1
        A[i,j] = x[i]^(k+1-j)
    end
end
```

In [8]:
```
using JuMP, Gurobi, Mosek

m1 = Model(solver=MosekSolver(LOG=0))
#m = Model(solver=GurobiSolver(OutputFlag=0))

@variable(m1, u1[1:k+1])
@constraint(m1, u1[3] == 0)
@objective(m1, Min, sum( (y[1:n1] - A*u1).^2 ) )
status = solve(m1)
uopt1 = getvalue(u1)
println(status)

npts = 100
xfine1 = linspace(0,4,npts)
ffine1 = ones(npts)
for j = 1:k
    ffine1 = [ffine1.*xfine1 ones(npts)]
end
yfine1 = ffine1 * uopt1;
```

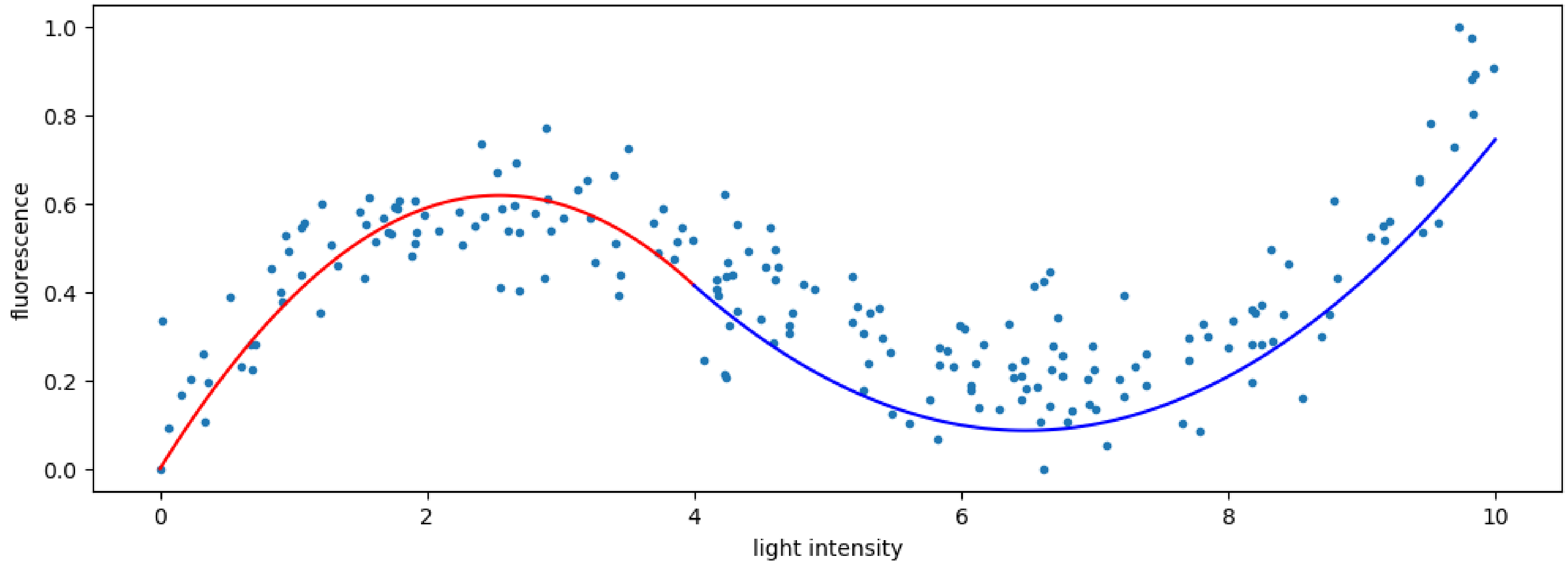
```
Optimal
```

In [9]:
```
k = 2

# fit using a function of the form f(x) = u1 x^k + u2 x^(k-1) + ... + uk x + u{k+1}
n2 = 78
A = zeros(length(x) - n1, k+1)
for i = 1:length(x)-n1
    for j = 1:k+1
        A[i,j] = x[i+n1]^(k+1-j)
    end
end
```

In [14]:
```
using JuMP, Gurobi, Mosek

m2 = Model(solver=MosekSolver(LOG=0))
#m = Model(solver=GurobiSolver(OutputFlag=0))

@variable(m2, u2[1:k+1])

@objective(m2, Min, sum( (y[n2 + 1:length(x)] - A*u2).^2 ))
status = solve(m2)
uopt2 = getvalue(u2)
println(status)

npts = 100
xfine2 = linspace(4,10,npts)
ffine2 = ones(npts)
for j = 1:k
    ffine2 = [ffine2.*xfine2 ones(npts)]
end
yfine2 = ffine2 * uopt2;
```

```
Optimal
```

In [19]:
```
using PyPlot

figure(figsize=(12,4))
plot( x, y, ".")
plot( xfine1, yfine1, "r-")
plot( xfine2, yfine2-(yfine2[1] - yfine1[end]), "b-")
xlabel("light intensity")
ylabel("fluorescence")

;
```


fluorescence
light intensity