Supporting Information for

Kinetics of $NO + H^+ + NO_3^- \rightarrow NO_2 + HNO_2$ on BaNa-Y: Evidence for a diffusion limited $A + B \rightarrow 0$ Reaction on a Surface.

*Aditya Savara and Eric Weitz**

Department of Chemistry and the Institute for Catalysis in Energy Processes,

Northwestern University, 2145 Sheridan Road, Evanston, IL 60208

* To whom correspondence should be addressed. E-mail: weitz@northwestern.edu

Outline:

I. Nitrate Peak Integration

The surface nitrate peaks on Na-Y and BaNa-Y, which have been identified based on prior assignments reported in the literature, absorb in the region from 1500 to 1300 cm^{-1} and are centered at approximately 1400 cm^{-1}.[1-6] For BaNa-Y, the nitrate peak was integrated from 1554 to 1241 cm^{-1}, with the baseline defined by drawing a line from 1554 to 1241 cm^{-1}. A second approach was used for BaNa-Y to verify that the choice of baseline did not affect the rate constants obtained from the fitting procedure: The "right"

portion of the nitrate peak was integrated from 1342 cm^{-1} to 1262 cm^{-1}, with the baseline defined using a line drawn from 1342 cm^{-1} to 1262 cm^{-1}. For Na-Y, the nitrate peak was integrated from 1554 to 1360 cm^{-1}, with a baseline defined from 1554 to 1153 cm^{-1}. The integrated areas for these nitrate peaks were divided by the integrated area of a zeolite framework absorption , determined by integrating this latter absorption band from 1263 to 989 cm^{-1}, with a baseline determined by a line drawn from 1263 to 985 cm^{-1}. By dividing the integrated area of the nitrate peaks by the integrated area of the zeolite framework absorption, the values thus obtained were quantitatively proportional to the absolute concentration of surface nitrates, and allowed kinetic data to be compared between samples.

II. Experimental Conditions:

Experiments were conducted at temperatures from 100 to 300 °C, with varied initial nitrate coverages, $[NO_3^-]_0$, and NO pressures that varied by up to a factor of 10. The full range of conditions studied is shown in Table S.1.

Table S.1. Experimental Conditions Tested

Experiment #	Sample Type	Temperature (°C)	$[NO_3^-]_0$, a.u.	[NO], torr
16	BaNa-Y-1	200	8.597623	0.96
17		200	8.430677	0.75
18		200	8.508355	1.34
19		200	8.743717	1.13
21		200	8.062429	1.0
22b		200	8.121394	1.18
22f		200	6.35009	0.784
30		200	5.497148	1.176
22c		100	11.2937	0.862
22e		100	9.524985	0.791
22d		150	10.84154	0.785
60		250	14.72096	1.15
61		250	13.51955	1.08
62		250	12.51471	1.32
73		250	11.10562	13.4
74		250	8.089615	0.3

75		250	8.887324	12.6
77		250	2.500379	0.01
79		250	2.289777	0.02
80		250	3.400929	0.03
63		300	13.93502	1.37
71		300	3.024014	0.2
72		300	7.606979	5
153	BaNa-Y-2	250	4.876794	9.92
154		250	3.317197	0.82
155		250	9.745098	1.806
158		250	4.02568	2.04
159		250	9.014925	0.914
160		250	5.971154	1.9
165		250	3.654883	1.928
161		300	6.390459	1.8778
162		300	7.823741	2.37
163		300	5.251259	1.4
166	Na-Y	150	3.001852	1.796
167		150	6.634371	2.232
168		150	7.916979	3.32

There are several possible sample-related sources of scatter for the quantity of nitrates remaining at the end time (or the end-time itself). As mentioned in Section III.A of the article, the Ba^{2+} and Na^{+} ions in the zeolite are the binding sites for the nitrates.[7] These nitrates likely do not all have the same binding energy, but rather are bound to sites which exhibit a distribution of binding energies.[7] As discussed in Section V.B of the article, the distribution of nitrate binding energies could potentially affect the quantity of nitrates remaining at the end of reaction (as discussed in Section V.B of the article). Also, if the nitrate distribution affects the spatial distribution of the reactants or the diffusion coefficient for H^{+},[8] this could potentially lead to scatter in the end-times of reaction observed based on the relations in Table 1 of the article (factors which govern the end-time of reaction are discussed in Sections V.A to V.D of the article).

In these experiments, there are also two conditions which might lead to changes in the surface fractal dimension between experiments. Prior to every experiment, the sample is calcined (heated to high temperatures) to remove water. We have found that

the normalized saturation coverage of the nitrates in the beam path decreases on the order of ~5% with every calcination. We interpret this observation as indicating that the sample slightly dealuminates after each calcination, as nitric acid is known to leach *at least* extraframework aluminum from zeolites.[9-11] Dealumination may result in slightly different surface fractal dimensions for different experiments.[12,13] Additionally, the adsorbed nitrates may also change the surface fractal dimension as a function of coverage, as they likely occupy a volume on the same scale as NH_3/NH_4^+ -- and exchange of Na^+ with NH_4^+ has been shown to change the fractal dimension.[13] However, as is shown in the article, although there is scatter in the data for the fitted end-times of reaction, approximating the surface fractal dimension as remaining constant during a single experiment and between experiments still yields relatively good agreement with theoretical predictions.

III.A First-Order and Second-Order Data Fitting.

The rate expressions that were fitted are presented in Table S.2. In each case, it was necessary to assume that some nitrates were unreactive to obtain reasonable fits. More details on the fitting are given in Section III.C. Arrhenius plots using the average values for k_{obsA}, k_{obsB}, $k_{obsA}/[NO]$ and $k_{obsB}/[NO]$ are shown in Figure S.1. The slope of linear regression is the activation energy, E_a, in Joules, while the y-intercept is the natural log of the pre-exponential, A, of the rate constant. The values obtained from these linear fits for $\ln(k_{obsA})$ and $\ln(k_{obsA}/[NO])$ are given in Table S.1. There were not enough data points to provide a reliable fit for k_{obsB}.

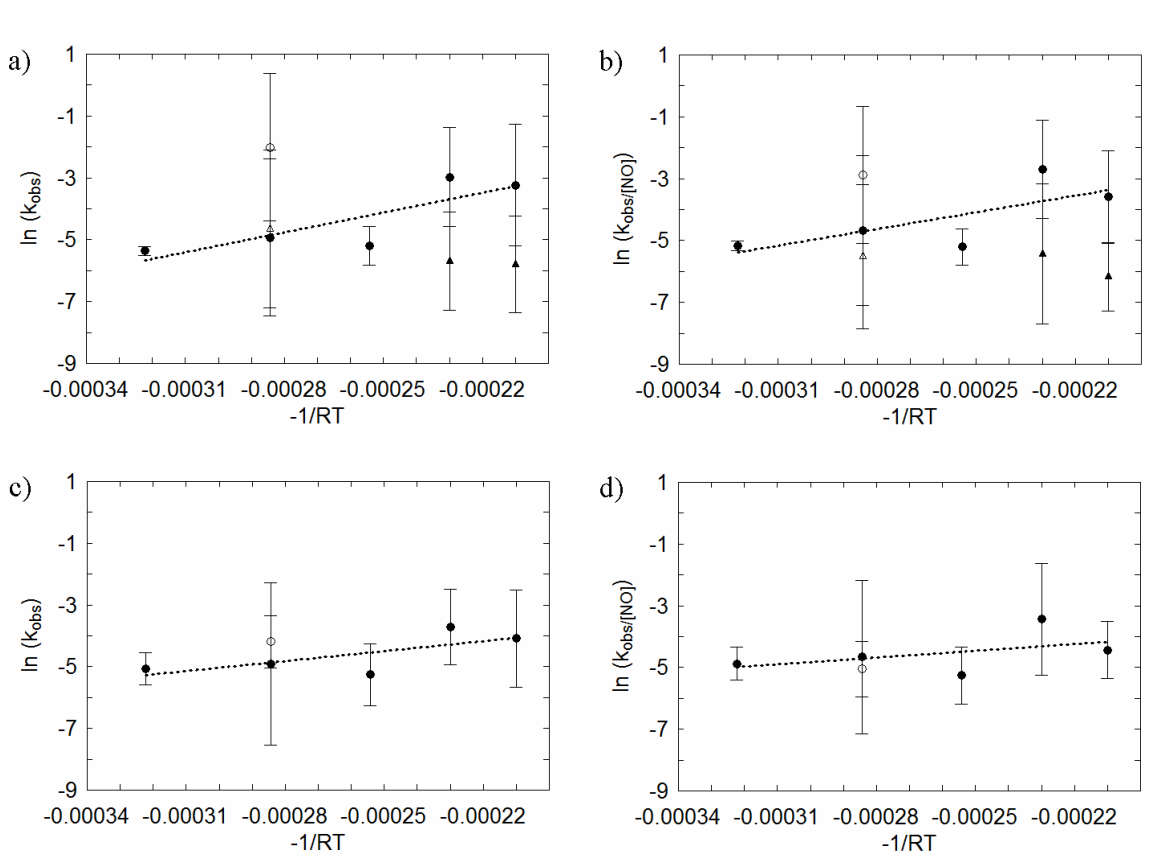


Figure S.1: Arrhenius plots. a) first-order k_{obs}, b) first-order k_{obs}/[NO], c) second-order k_{obs}, d) second-order k_{obs}/[NO]. Symbols: ● BaNa-Y k_{obsA}, ▲ BaNa-Y k_{obsB}, ○ Na-Y k_{obsA}, △ Na-Y k_{obsB}. Averaged values are plotted. The error bars shown are one standard deviation.

Table S.2 Parameters for k_{obsA} calculated from Arrhenius plots.

$[NO_3^-]$ order	Rate Form, $d[NO_3^-]/dt$ =	k_A=	A	E_a, kJ/mol
First-order	$k_A[(NO_3^-)_A] + k_B[(NO_3^-)_B]$	k_{ObsA}	4×10^0 s^{-1}	20
First-order	$k_A[(NO_3^-)_A][NO] + k_B[(NO_3^-)_B][NO]$	k_{ObsA}/[NO]	1×10^0 s^{-1}	20
Second-order	$k_A[NO_3^-]^2$	k_{ObsA}	2×10^{-1} $a.u.^{-1}$ s^{-1}	10
Second-order	$k_A[NO_3^-]^2[NO]$	k_{ObsA}/[NO]	7×10^{-2} $a.u.^{-1}$ s^{-1}	10

The standard deviations are quite large for the observed rate constants obtained from an average at each temperature. This is true both when NO is included in the rate equation as well as when NO is excluded from the rate equation. Despite the large

number of experiments performed, and the variety of conditions tested, none of the rate equations fit the data obviously better than when the data was fit to other rate equations: Instead, all rate equations tested resulted in large standard deviations for the rate constants extracted from multiple experimental runs, i.e. all rate constants that were obtained from fits gave similar error limits for the fitted values. In addition to large scatter in the rate constants obtained for these models, the data showed a decrease in the observed rate constant with time (i.e. --progression of reaction) for every model studied – that is, a plot of nitrate depletion versus time exhibited a functional form referred to as a "stretched exponential" (not shown). Further, the Arrhenius plots for all of the observed rate constants display a very low apparent activation energy (< 40 kJ/mol) with unrealistically low pre-exponentials: For a surface reaction which is not diffusion limited a realistic pre-exponential is expected to be on the order of 10^{13} to 10^{19} s^{-1}.[14,15]

The low apparent activation energies would in principle be consistent with the rate limiting step being desorption / decomposition of HNO_2 (the expected product), -- as the desorption / decomposion of HNO_2 on BaNa-Y has an activation energy of < 25 kJ/mol.[16,17] However, no HNO_2 intermediate is observed at these temperatures, and such a mechanism would not, by itself, explain the stretched exponential behavior for the depletion of nitrates, or the apparent low pre-exponentials for the effective rate constant.

The apparent low activation energy and "stretched exponential" behavior could be indications of a diffusion limited reaction. The two obvious possibilities for the source of the diffusion limitation are the diffusion of NO into the zeolite crystals, or surface diffusion of the adsorbates. We showed previously that diffusion limited transport of NO gas molecules into the zeolite crystals (approximated as mass transport limited Fickian

diffusion into a sphere[18]), was an unlikely explanation for the observed kinetics, and also would not explain the depletion of nitrates observed on long time scales.[19] The possibility of a reaction limited by diffusion of surface species is explored in the main article. For thoroughness, a Taylor-Weinberg type analysis was also performed, as described in the following section.

III.B. Taylor-Weinberg Analysis for First and Second-Order Models

Given the different kinetic regimes observed and the range of the initial coverages and NO pressures encompassed by the data, a Taylor-Weinberg type analysis was performed.[20-22] This analysis assumes that the rate constant is a function of coverage, and allows the pre-exponential and the activation energy to vary as functions of nitrate coverage.

For each experiment the data for the nitrate concentration as a function of time was fit using the function $Ae^{-bt} + Ce^{-dt} + Ft + G$ (with an average R^2 of 0.95), and the rate of nitrate depletion was evaluated as a function of coverage from these fits, at discrete coverages separated by intervals of 0.01 normalized integrated absorbance units (the units for nitrate coverage). Using these rates, k was calculated as a function of nitrate coverage for kinetic models that assumed either a first-order or second-order dependence on nitrate coverage and that either included or excluded [NO] in the rate equation. These models were applied to the data from all substrates (BaNa-Y-1, BaNa-Y-2, and Na-Y). The fitted rate constants, as a function of absolute coverage, were found to have overlapping ranges for all substrates studied, so the data for all substrates were treated as one large data set for the Taylor-Weinberg analysis for each kinetic model.

Figures S.2a and S.2b show the coverage dependent pre-exponential and activation energy curves obtained for k for the rate form $k[NO][NO_3^-]^2$, by using the Taylor-Weinberg analysis. The pre-exponential and activation energy curves do not have a monotonic dependence on the nitrate coverage – similar non-monotonic dependencies of the pre-exponentail and activation energy on the coverage were obtained for the other three rate forms (see previous paragraph). The non-monotonic behavior in Figures S.2a and S.2b indicate that these models are unable to describe the data. Figure S.3 shows the poor R^2 obtained from linear fits of the Arrhenius equation to the values obtained for k from the $k[NO][NO_3^-]^2$ rate equation, as a function of nitrate coverage. The rate of depletion is more strongly correlated with the time elapsed from the start of reaction -- rather than with the nitrate coverage. This behavior is as expected for a DLAB0 reaction, and is discussed in the main article.

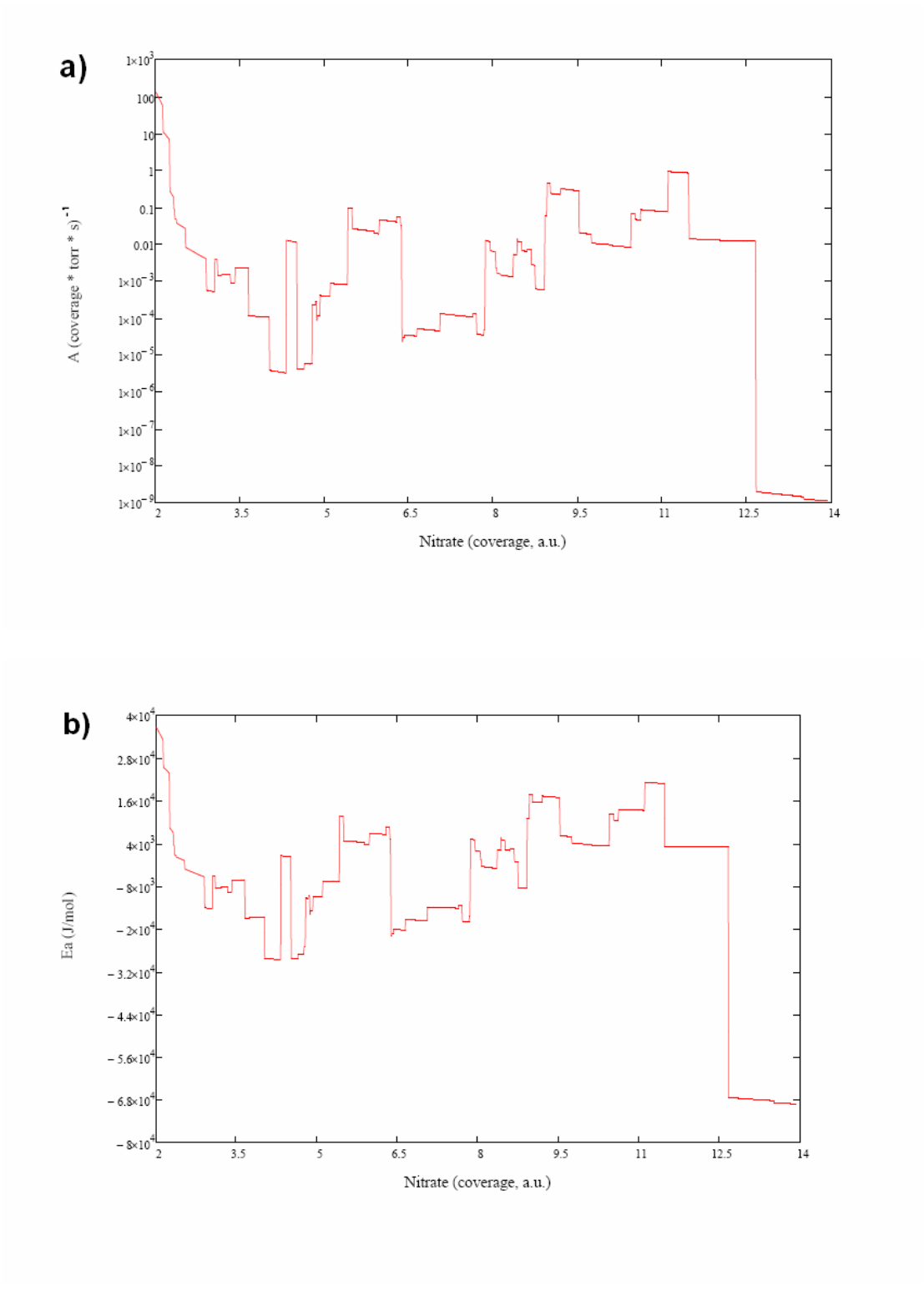


Figure S.2 Pre-exponential and Activation energy of reaction as a function of coverage based on a Taylor-Weinberg analysis with the rate equation $d[NO_3^-]/dt=k[NO][NO_3^-]^2$

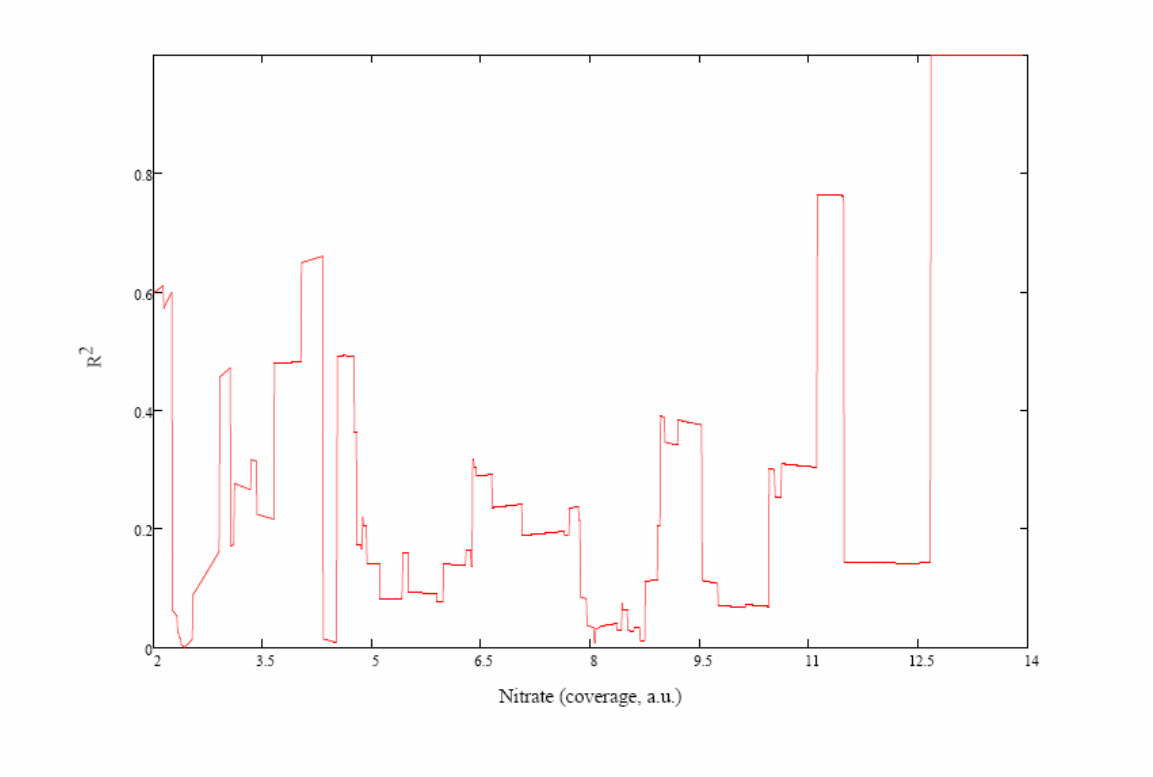


Figure S.3 R^2 as a function of coverage for the linear fits to the Arrhenius equation during the Taylor-Weinberg analysis with the rate equation $d[NO_3^-]/dt=k[NO][NO_3^-]^2$

III.C. Details on Fitting to the First-Order and Second-Order Models:

Surface nitrates had a rapid initial depletion (< 60 s) followed by a slower depletion at longer times (up to 10,000 s). To obtain rate constants, the nitrate depletion was fitted with modified versions of the following rate forms, as discussed in more detail below:

$$d[NO_3^-]/dt = k_{obsA}[(NO_3^-)_A] \quad (1)$$

$$d[NO_3^-]/dt = k_{obsA}[(NO_3^-)_A] + k_{obsB}[(NO_3^-)_B] \quad (2)$$

$$d[NO_3^-]/dt = k_{obsA}[NO_3^-]^2 \quad (3)$$

Eqs 1 to 3 were modified to include the assumption that some nitrates are completely unreactive on exposure to NO. Thus, the following integrated rate equations were used for fitting the nitrate depletion data, with Eqs. 4,5,6 corresponding to Eqs. 1,2,3, respectively:

$$[(NO_3^-)] = [(NO_3^-)_A]_0 e^{-k_{obsA}t} + [(NO_3^-)_U]_0 \quad (4)$$

$$[(NO_3^-)] = [(NO_3^-)_A]_0 e^{-k_{obsA}t} + [(NO_3^-)_B]_0 e^{-k_{obsB}t} + [(NO_3^-)_U]_0 \quad (5)$$

$$[(NO_3^-)] = 1/\left([(NO_3^-)_A]_0^{-1} + k_{obsA}t\right) + [(NO_3^-)_U]_0 \quad (6)$$

$[(NO_3)^-_U]_0$ represents nitrates which are unreactive. On BaNa-Y at lower temperatures ($\leq$ 200 °C), Eqs. 4 and 6 yielded adequate fits for a first-order and second-order dependence on nitrate coverage, respectively. For the experiments performed at high temperatures ($\geq$250 °C) on BaNa-Y, and for the experiments at 150 °C on Na-Y, only Eqs 5 and 6 yielded adequate fits for first-order and second-order nitrate dependence, respectively. A fit was deemed inadequate if the curve fitting routine in Sigmaplot returned any of the following: a non-convergence error, extraordinarily large standard fitting errors for any parameter (i.e., a magnitude greater than the parameter itself, in this case errors >10,000), or a fitting dependency of >0.99 on any parameter. Each of these

are indications of an inaccurate and/or imprecise fit, as described in the Sigmaplot programming guide.[23]

In addition to fitting the nitrate depletion data directly, k_{obsA} was also obtained from the corresponding half-life equations derived from integrated rate laws, with the experimental "total" half-depletion value – the point at which half of all reactive nitrates are depleted – given by $[NO_3^-]_{1/2} = ([NO_3^-]_{initial}-[NO_3^-]_{final})/2$. This approximation again assumes that unreactive nitrates remain after the rest of the nitrates have reacted. In the case of the first-order fits described in the previous paragraph (with Equation 5), it was assumed that there were two parallel first-order processes, yielding two observed rate constants, k_{obsA} and k_{obsB}. During the first-order half-life analysis, the rate constant was taken to be reflective of only the faster first-order process, k_{obsA}, with the effects of k_{obsB} approximated as having a negligible effect on the "total half-life" obtained from the time at which the experimental nitrates concentration is equal to $[NO_3^-]_{1/2}$ (as defined above). This approximation, that k_{obsB} would have a negligible contribution to the rate constant obtained from the "total half-life", is reasonable given that the total nitrate depletion observed is dominated by the much faster initial reaction: $k_{obsA} >> k_{obsB}$ and $[(NO_3\text{-})_A] > [(NO_3\text{-})_B]$. As can be seen by comparing Figures S.4 to S.7, nearly all of the values obtained from the half-life analysis are within one standard deviation of the values obtained from fitting equations 4 through 6. Thus, the values for k_{obsA} from the half-life analysis (this paragraph) and from the fitting analysis (previous paragraph) were averaged, and the final values are presented in Table S.2 (the values in Table S.2 have also been averaged across both of the nitrate peak areas as defined in Section I).

The units for the first-order rate constants are s^{-1}, and the units for the second-order rate constants are $a.u.^{-1}\ s^{-1}$, where a.u. are the normalized arbitrary units which arise from dividing the integrated area of the surface nitrate infrared peak by the integrated area of the zeolite framework infrared peak, as described in Section S.I. The observed rate constants and parameters obtained from fitting the data are included in Appendix S.1. The relations between these "observed rate constants" and the final "calculated rate constants" are shown in Table S.2 in Section III.A. For second-order rate constants, the observed rate constants obtained from monitoring the "right" BaNa-Y nitrate peak were scaled to those of the "full" BaNa-Y nitrate peak using a factor of 10.5. This scaling was necessary for a direct comparison of the observed second-order rate constants, as prior to this scaling the observed second-order rate constants obtained from the "right" BaNa-Y nitrate peak and "full" BaNa-Y nitrate peak were not normalized to the same concentration units. The scaling factor is based on the average quantity of reactive nitrates obtained from a second-order fit. This factor was obtained by dividing the average of the second column of Table S.4d by the average of the second column of Table S.4e. As shown in Figures S.4 to S.7, for each rate equation, the values obtained for the rate constants were within one standard deviation regardless of the method used. Thus, the values presented in Table S.2 are averages of all the values obtained by both methods (half-life analysis and fitting) and for both peak definitions for k_{obsA}, and, k_{obsB}, $k_{obsA}/[NO]$, and $k_{obsB}/[NO]$.

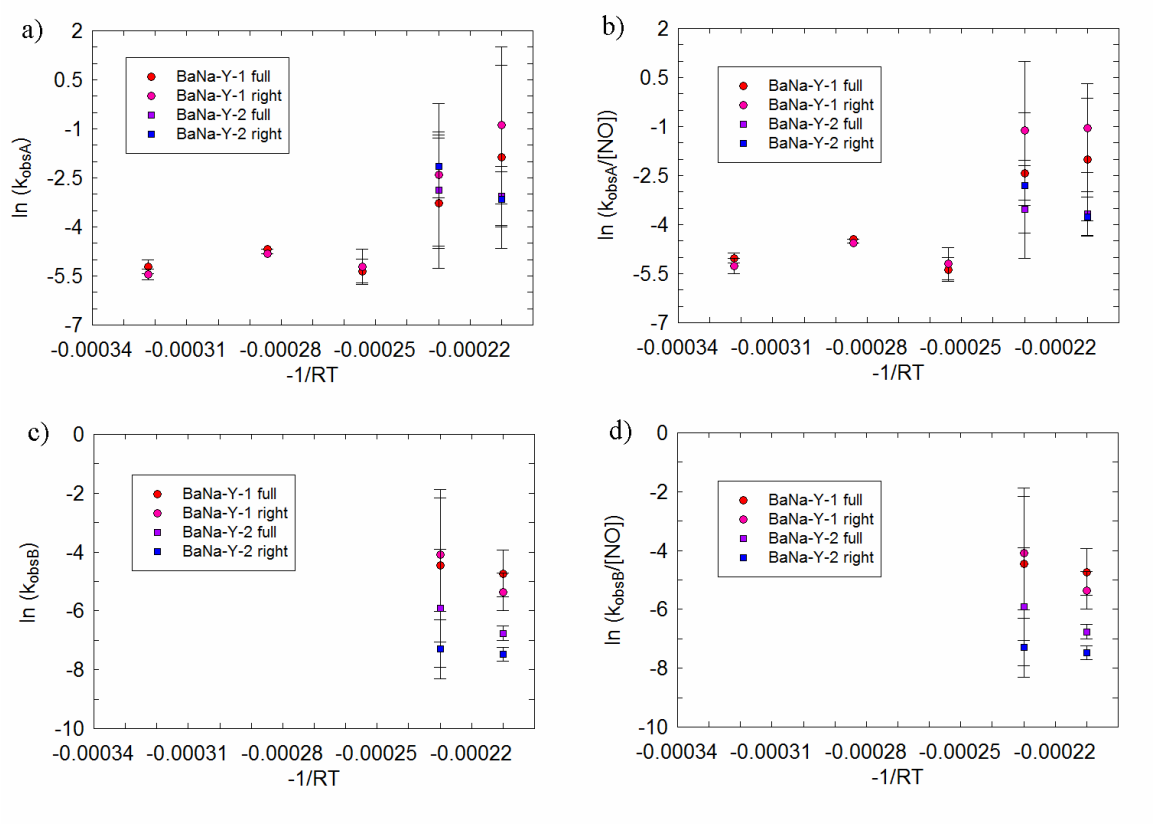


Figure S.4 Arrhenius plots for observed rate constants obtained from first-order fitting. Points are averaged values, error bars shown are one standard deviation.

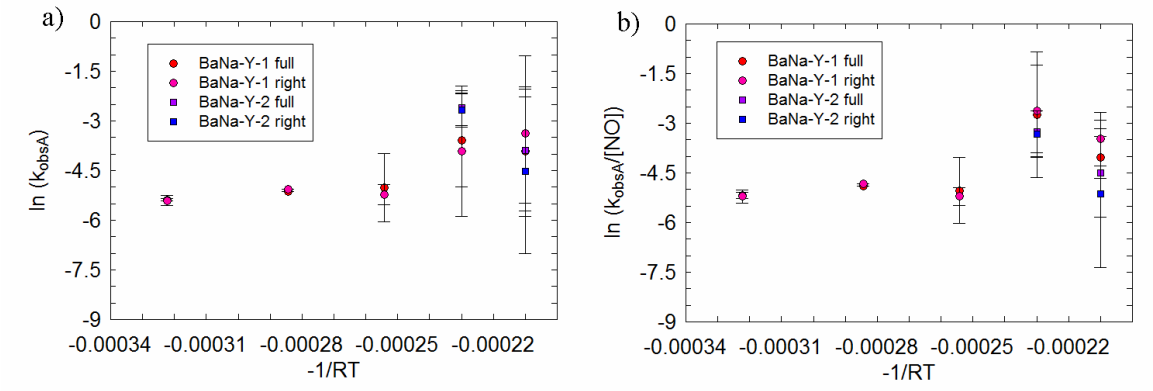


Figure S.5 Arrhenius plots for observed rate constants obtained from first-order half-life analysis.

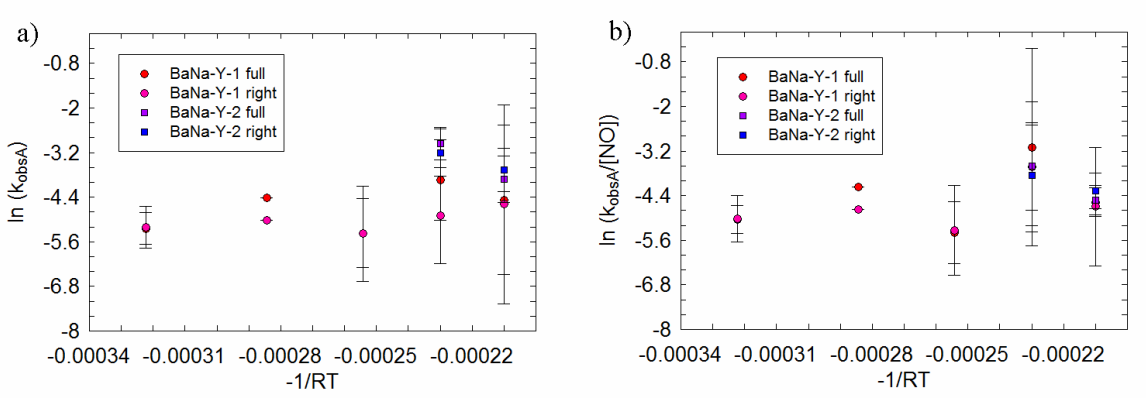


Figure S.6 Arrhenius plots for observed rate constants obtained from second-order fitting. Points are averaged values, error bars shown are one standard deviation.

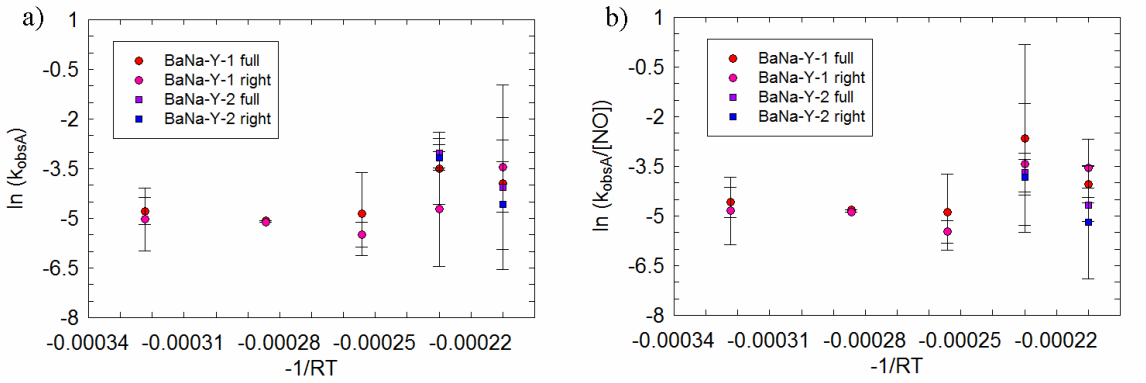


Figure S.7 Arrhenius plots for observed rate constants obtained from second-order half-life analysis.

IV. Spline-Fitting of Alpha Plots

Alpha in equation 9 of the published article is obtained from the slope of a plot of $\log(1/[NO_3^-]-1/[NO_3^-]_0)$ vs. log(time). As described in Section III.C of the published article the theoretical value for alpha on BaNa-Y and Na-Y is expected to be $\alpha \approx 0.5$. The alpha plots for the different experiments are shown in Appendix S.3. Determining the time region of the Zeldovich regime is a difficult spline-fitting problem – particularly due to the fact that the noise varies as a function of time (as mentioned in section IV of the article), and due to the fact that the piece-wise nature of the data is expected to change as an undetermined non-linear function of the experimental conditions (e.g., NO concentration, distribution of sites occupied, and Temperature). For more information about the difficulties in spline-fitting data in the presence of noise, see for example "Local Linear Regression" in reference 23.[24] Custom spline-fitting programs written in Python were used to find the onset and end-time of the Zeldovich regime. These programs are included in Appendix S.2, and each requires the data to be provided in a comma-separated values file with no column headings. The custom written program named Nt2 finds the rolling slope of log(time) vs. $1/[NO_3^-]-1/[NO_3^-]_0$, and was used first to determine at which time the slope was best approximated as 0.5. The custom written programs Ntf7 and Ntf8 were given this time as an input, and both programs then spline-fit the data, returning the data points corresponding to the onset and end-time. Ntf7 fits the linear region corresponding to the Zeldovich regime with a fixed slope (0.5) while Ntf8 allows the slope to vary during fitting (typically 0.4 to 0.6 after fitting). The true endpoints can be generally expected to fall between these two cases. The results from fitting the data are shown in Table S.3. In some experiments, the depletion of surface

nitrates was not monitored until the end of the Zeldovich regime, as it was not possible to gauge when the end-time would occur during the experiments. Thus, in some cases, the true τ_f is $\geq$ the recorded τ_f. Large fitting errors result from the noise (scatter) and fluctuation ("waviness") in the data (shown in the alpha plots in Appendix S.3). The fitting errors for τ_f cannot be readily quantified as the magnitude of the noise and fluctuation varies between experiments, as well across time in the same experiment. However, the noise and fluctuations do not appear to affect the slopes of the alpha plots on the scale of several orders of magnitude of time (note that x-axis in alpha plots is the log of time) and given that the time at which τ_f occurs is determined based on a change in slope in the alpha plots, we estimate that the fits to determine τ_f are accurate to an order-of-magnitude in time, which is consistent with the scatter observed in Figures 2 and 3 of the published article. The τ_f values obtained from both ntf7 and ntf8 were used to produce the plots shown in the published article, as well as for Figure S.8 below. In Figure S.8, the x and y axes have the same scale, and it is clear that $\log(\tau_f)$ does not correlate directly with $[NO_3^-]_0$. The values for $[NO_3^-]$ at τ_f shown in Figure 4 of the published article are the values of $[NO_3^-]$ at τ_f from ntf7, though the values obtained for $[NO_3^-]$ at τ_f from ntf8 were very similar and are only omitted from the bar graph presented in Figure 4 of the published article for clarity.

Table S.3. Fitting Results for τ_f

Experiment #	Sample Type	τ_f from ntf7	τ_f from ntf8	$[NO_3^-]$ at τ_f from ntf7	$[NO_3^-]$ at τ_f from ntf8
16	BaNa-Y-1	2.11	2.39	6.826779	6.459968
17		2.54	2.56	6.81372	6.791179
18		2.77	2.77	7.839163	7.851512
19		2.91	2.91	8.009894	8.063157
21		2.86	2.68	6.272539	6.464358
22b		2.92	2.91	6.688625	6.540824
22f		2.41	2.75	10.47487	10.06647
22c		3.09	3	9.399727	9.435376
22e		2.82	2.77	8.959245	8.773412
22d		2.47	2.47	6.175931	1.546477
60		1.8	2.49	8.717328	8.428954
61		1.74	1.74	8.351575	7.725496
62		2.09	1.89	7.060601	7.172736
73		0.99	0.89	10.30436	10.76006
74		2.02	2.43	1.600681	2.393133
75		2.02	1.92	6.631469	6.534404
77		2.47	2.45	8.517967	8.776183
79		2.71	2.71	8.016592	7.753558
80		2.74	2.37	7.90324	7.972297
63		1.785	1.785	2.26879	2.235628
71		3.4	2.57	2.205491	1.622716
72		1.48	1.8	2.69206	2.810496
153	BaNa-Y-2	0.47	1.81	3.679977	2.934522
154		1.92	1.89	2.281852	2.138169
155		1.26	1.78	5.352548	5.265791
158		1.81	1.8	2.51506	2.420105
159		1.69	1.72	5.308052	5.408987
160		1.76	1.86	5.016666	5.030267
165		1.73	1.73	5.103187	4.966929
161		2.17	1.94	5.31874	5.396961
162		1.72	1.72	4.791448	4.67547
163		2.24	2.92	2.023837	2.005926
166	Na-Y	1.85	1.92	2.641772	2.63429
167		1.72	1.72	3.024705	3.025137
168		1.92	2.39	3.076942	6.067994

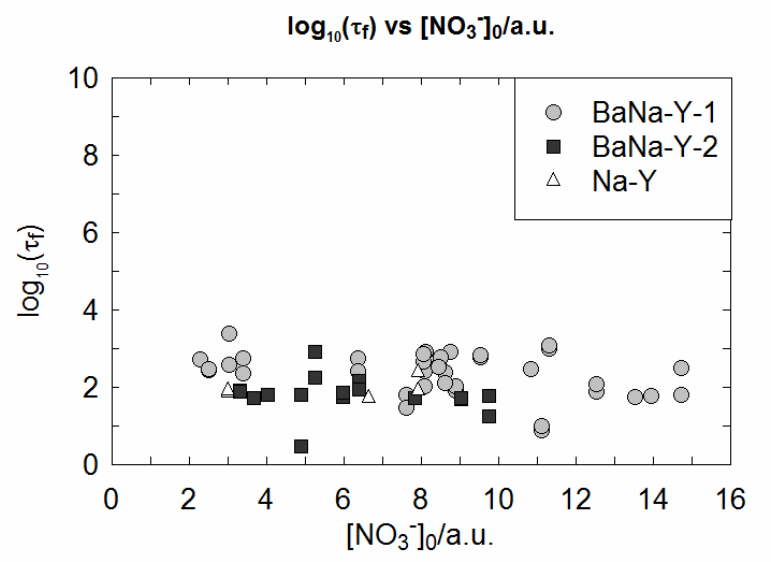


Fig. S.8. log(τ_f) vs. $[NO_3^-]_0$ with equal scaling for the x-axis and y-axis. It is clear that log(τ_f) does not correlate directly with $[NO_3^-]_0$

V. Estimating the Activation Energy for Diffusion

The temperature dependence of the calculated diffusion coefficients was analyzed using Relation 1 of the published article, which indicates that the end-time of the Zeldovich regime, τ_f, is inversely proportional to the diffusion coefficient. Simulations by Argyrakis et al.[25] have found that for a two dimensional surface, an order of magnitude estimate can be made for Rel. 1 in Table 1 of the published article, of $\tau_f \sim (0.14)^2L^2/D$. However, this estimate assumes every encounter of A and B leads to reaction (p_{eff}=1). Thus, in an actual reaction, where p_{eff}<1, the end-times would be expected to be longer (as discussed in the published article, Section V.C), and consequently the values we obtain for D from the above relation are expected to correspond to a lower limit for the actual value of D. As shown in Section VII of the SI, the surface area per crystallite used is ~≤ 36 cm^2, which we take for L^2 in Rel. 1 in Table 1 of the published article.

As discussed in Section V.C of the published article, τ_f is inversely proportional to [NO]. Thus, to obtain the lower limit for the diffusion coefficient from the experimental values of τ_f, it was first necessary to normalize the τ_f values by [NO]. The relationship of [NO] vs. τ_f was fit to a power law, based on the dependence observed in the log-log plot of Figure 3 of the published article, with the result being τ_f = ([NO]/(23.4±5.1))^(-0.51±0.07), where the error bars represent the standard errors from fitting. We do not ascribe any physical significance to this functional form, we use it only for empirical scaling between [NO] and τ_f -- which is required to compare the effective diffusion coefficient between different experiments. Using the fitted relationship above, all τ_f values were normalized to the fitted value for τ_f with [NO]=10 Torr. The value of 10 Torr was used as it is the highest "scale" for [NO] exposure used during experiments -- and it showed the fastest nitrate depletion rate – thus it provides our *lowest observed limit* for the effective diffusion coefficient at each temperature. From these normalized τ_f values, the effective diffusion coefficient was calculated for each of the data points in Figure 3 using the relation $\tau_f \sim (0.14)^2 L^2/D$. An Arrhenius plot was constructed from the calculated effective diffusion coefficients (Figure S.9, shown on the following page), yielding $D_0 = 10^{6\pm7}$ cm^2 s^{-1} and $E_a = 30 \pm 30$ kJ/mol where the errors represent fitting errors yielded by Origin's standard error-weighted fit (this weights the importance of each point used in the linear regression by the inverse square of the vertical error bars associated with that point). The error bars for D_0 and E_a are large due to the scatter of the data and because the temperature range where these experiments are possible sufficiently narrow (a range of <200K) to make it difficult to obtain a more precise value for D_0 given the low level of precision in determining τ_f.

As will be discussed in the next section, many of the individual data points for the diffusion coefficient (in Figure S.9) are too high to be realistic – being two orders of magnitude greater than 10^{-2} cm^2 s^{-1}. However, all raw data points -- before scaling -- yielded diffusion coefficients of realistic orders of magnitude, $<10^{-2}$ cm^2 s^{-1}. The high values for the diffusion coefficient obtained from some experiments may then be artifacts arising from the [NO] scaling procedure: the [NO] scaling function was determined empirically from data with high scatter, and thus may have resulted in inaccurate scaling, which could in turn lead to artificially high calculated diffusion coefficients. Nonetheless, the [NO] scaling procedure is necessary to obtain effective diffusion coefficients from multiple experiments due to the relationship between τ_f and [NO] which is discussed in Section V.C of the published article.

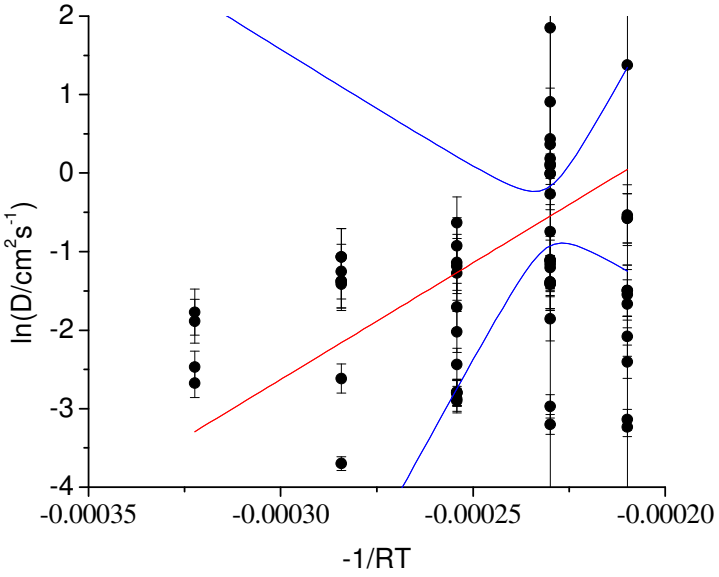


Figure S.9 Arrhenius plot for diffusion coefficients obtained from normalized τ_f values for each experiment. τ_f values were scaled to the fitted value of τ_f for [NO]=10 Torr.

VI. Discussion of the Values Obtained for the Diffusion Coefficient

Using the procedure outlined in the previous section it was determined that, $D_0 = 10^{6\pm7}$ cm^2 s^{-1} and $E_a = 30 \pm 30$ kJ/mol. As described in section III.C of the published article the value obtained for the activation energy is, as expected, near the lower end of the range for the activation energies for the hopping of acidic protons in zeolites,. However, as discussed below, the correlation between the value for D_0 obtained from these experiments and a specific physical process is not clear.

The value obtained for D_0 is higher than would be expected for an atom or molecule diffusing on a surface, as D_0 for such processes on surfaces is typically 10^{-2} to 10^{-3} cm^2 s^{-1}.[26,27,28] However, a possible explanation for the high value of D_0 obtained might be found in the mechanism for proton migration in zeolites. The low activation energy route for proton migration in zeolites arises from several parallel mechanisms for water-assisted proton-migration.[29-31] Thus, proton migration in our study is not expected to involve only a single elementary process and, therefore, the diffusion coefficient would not be expected to be accurately described by an Arrhenius equation over a wide temperature range: Different mechanisms could dominate as the temperature changes over a sufficiently wide range. Consequently, it is not surprising that our extrapolation to obtain D_0 from an Arrhenius plot does not yield a realistic pre-exponential for the intrinsic diffusion coefficient. Our extrapolation instead yields the effective pre-exponential for an empirically observed mass-transfer diffusion coefficient. While this is the most likely explanation for the high D_0 observed, we will also note other plausible explanations below.

A second plausible explanation is that if the disappearance of nitrates becomes rate-limited by the depletion of the less reactive nitrates *prior* to the occurrence of finite size effects[25,32] (i.e., before the "natural" end of the Zeldovich regime) – then the end-time obtained from the fitting procedure would be shorter than that predicted by theory, leading to a larger than expected diffusion coefficient --as observed. This would be a qualitative deviation from the ideal case of the DLAB0 model, as the end-time for the Zeldovich regime would arise from a different kinetic process than expected for a DLABO model (i.e., a different rate limiting step). Alternatively, the high value of D_0 may be a result of deviations from the ideal case of the DLAB0 limited model which are minor enough to still retain the essence of the model in terms of the qualitative aspects of kinetics that are observed. Such deviations could arise from a variety of possible deviations from the ideal model. These include: multiple types of nitrates with differing stability,[33] regions of the surface that have different dimensionality, non-random initial conditions,[33,34] a $p_{eff} < 1$, unusual lattice symmetry, crystal inhomogeneities and deviations from true random-walk behaviour (such as due to particle interactions)[35]. All of these factors could potentially affect the growth of the segregated reactant regions and/or the proportionality constant of Rel. 1 in Table 1 of the published article (currently assumed to be $(0.14)^2$) -- and thus change the observed end-times (and consequently the estimated diffusion coefficient). Modifications of the current model are necessary to treat such effects and a more in depth understanding of the interplay among such effects could come from additional theoretical studies, especially given the difficulty in finding experimental systems that cleanly exhibit DLAB0 kinetics.[33]

VII. Estimation of Crystallite Surface Area

With knowledge of the crystallite size, it is possible to calculate the total surface area per zeolite crystallite. As mentioned earlier, the Na and Ba cations are small compared to the size of the zeolite supercages. Thus, the change in surface area between Na-Y and BaNa-Y crystallites would not be expected to be large. According to the supplier's specifications sheet (-100 mesh), the zeolite crystallites were $\leq 3.5 \times 10^{-6}$ cm^3 for >90% of the crystals. Na-Y has a cubic crystal structure, with a crystallographic unit cell side length of ~24.6 Å for the Si/Al ratios used,[36,37] leading to a crystallographic unit cell volume of ~15,000 $Å^3$. From the upper bound of the volume of the crystallites used, this gives $\leq 2.34 \times 10^{14}$ unit cells per crystallite. The sum of the number of framework atoms (Si and Al) per unit cell is 192.[38] Taking the ICP obtained Si/Al ratio of 2.62 (Section II of the published article), the density of the Na-Y used in this study is ~12700 atomic mass units per crystallographic unit cell. From the atomic mass per unit cell and the number of unit cells per crystallite, the weight per crystallite is $\leq 4.96 \times 10^{-6}$ g. A surface area of ~725 m^2 g^{-1} has been reported in the literature for a Na-Y zeolite with a similar Si/Al ratio as the one used in this study.[39] Given a surface area of ~725 m^2 g^{-1}, the surface area per crystallite is $\leq 3.6 \times 10^{-3}$ m^2, which is ≤ 36 cm^2.

References:

(1) Szanyi, J.; Kwak, J. H.; Moline, R. A.; Peden, C. H. F. *Physical Chemistry Chemical Physics* **2003**, *5*, 4045.

(2) Chao, C. C.; Lunsford, J. H. *Journal of the American Chemical Society* **1971**, *93*, 71.

(3) Szanyi, J.; Kwak, J. H.; Peden, C. H. F. *Journal of Physical Chemistry B* **2004**, *108*, 3746.

(4) Szanyi, J.; Kwak, J. H.; Burton, S.; Rodriguez, J. A.; Peden, C. H. F. *Journal of Electron Spectroscopy and Related Phenomena* **2006**, *150*, 164.

(5) Li, G. H.; Larsen, S. C.; Grassian, V. H. *Journal of Molecular Catalysis A* **2005**, *227*, 25.

(6) Li, G. H.; Jones, C. A.; Grassian, V. H.; Larsen, S. C. *Journal of Catalysis* **2005**, *234*, 401.

(7) Savara, A.; Sachtler, W. M. H.; Weitz, E. *Applied Catalysis B* **2009**, *90*, 120.

(8) Kärger, J.; Ruthven, D. M. *Diffusion in zeolites and other microporous solids*; Wiley: New York, 1992.

(9) Giudici, R.; Kouwenhoven, H. W.; Prins, R. *Applied Catalysis a-General* **2000**, *203*, 101.

(10) Gola, A.; Rebours, B.; Milazzo, E.; Lynch, J.; Benazzi, E.; Lacombe, S.; Delevoye, L.; Fernandez, C. *Microporous and Mesoporous Materials* **2000**, *40*, 73.

(11) Yoshida, A.; Nakamoto, H.; Okanishi, K.; Tsuru, T.; Takahashi, H. *Bulletin of the Chemical Society of Japan* **1982**, *55*, 581.

(12) Lima, E. J.; Bosch, P.; Lara, V. H.; Bulbulian, S. *Chemistry of Materials* **2004**, *16*, 2255.

(13) Erdem-Senatalar, A.; Tatlier, M. *Chaos Solitons & Fractals* **2000**, *11*, 953.

(14) Zhdanov, V. P.; Pavlicek, J.; Knor, Z. *Catalysis Reviews-Science and Engineering* **1988**, *30*, 501.

(15) Lombardo, S. J.; Bell, A. T. *Surface Science Reports* **1991**, *13*, 1.

(16) Yeom, Y. H.; Henao, J.; Li, M. J.; Sachtler, W. M. H.; Weitz, E. *Journal of Catalysis* **2005**, *231*, 181.

(17) Savara, A.; Sachtler, W. M. H.; Weitz, E. *Applied Catalysis B* **2008**, *Submitted*.

(18) Foley, N. J.; Thomas, K. M.; Forshaw, P. L.; Stanton, D.; Norman, P. R. *Langmuir* **1997**, *13*, 2083.

(19) Savara, A.; Weitz, E. *Abstracts of Papers of the American Chemical Society* **2006**, *231*.

(20) Taylor, J. L.; Weinberg, W. H. *Surface Science* **1978**, *78*, 259.

(21) Masel, R. I. *Principles of adsorption and reaction on solid surfaces*; Wiley: New York, 1996.

(22) King, D. A.; Madey, T. E.; Yates, J. T. *Journal of Chemical Physics* **1971**, *55*, 3236.

(23) *SigmaPlot 5.0 : programming guide*; SPSS Inc.: Chicago, Illinois, 1998.

(24) Takezawa, K. *Introduction to nonparametric regression*; Wiley-Interscience: Hoboken NJ, 2006.

(25) Argyrakis, P.; Kopelman, R.; Lindenberg, K. *Chemical Physics* **1993**, *177*, 693.

(26) Seebauer, E. G.; Allen, C. E. *Progress in Surface Science* **1995**, *49*, 265.

(27) Wang, X. R.; Xiao, X. D.; Zhang, Z. Y. *Surface Science* **2002**, *512*, L361.

(28) Karl J. Sladek, E. R. G., Raymond F. Baddour. *Industrial & Engineering Chemistry Fundamentals* **1974** *13*, 100.

(29) Ryder, J. A.; Chakraborty, A. K.; Bell, A. T. *Journal of Physical Chemistry B* **2000**, *104*, 6998.

(30) Ernsberger, F. M. *Journal of the American Ceramic Society* **1983**, *66*, 747.

(31) Martucci, A.; Parodi, I.; Simoncic, P.; Armbruster, T.; Alberti, A. *Microporous and Mesoporous Materials* **2009**, *123*, 15.

(32) Lin, A.; Kopelman, R.; Argyrakis, P. *Physical Review E* **1996**, *53*, 1502.

(33) Monson, E.; Kopelman, R. *Physical Review E* **2004**, *69*, 021103.

(34) Sancho, J. M.; Romero, A. H.; Lindenberg, K.; Sagues, F.; Reigada, R.; Lacasta, A. M. *Journal of Physical Chemistry* **1996**, *100*, 19066.

(35) Oshanin, G.; Sokolov, I. M.; Argyrakis, P.; Blumen, A. *Journal of Chemical Physics* **1996**, *105*, 6304.

(36) Dempsey, W.; Kuhl, G. H.; Olson, D. H. *Journal of Physical Chemistry* **1969**, *73*, 387.

(37) Hriljac, J. A.; Eddy, M. M.; Cheetham, A. K.; Donohue, J. A.; Ray, G. J. *Journal of Solid State Chemistry* **1993**, *106*, 66.

(38) Yeom, Y. H.; Jang, S. B.; Kim, Y.; Song, S. H.; Seff, K. *Journal of Physical Chemistry B* **1997**, *101*, 6914.

(39) Langmi, H. W.; Walton, A.; Al-Mamouri, M. M.; Johnson, S. R.; Book, D.; Speight, J. D.; Edwards, P. P.; Gameson, I.; Anderson, P. A.; Harris, I. R. *Journal of Alloys and Compounds* **2003**, *356*, 710.

Appendix S.1: Parameters obtained from first and second-order fits

Table S.4a First-order parameters from fitting to Eqs. 4 & 5, "full nitrate peak"

Experiment #	k_{ObsA}	$[(NO_3^-)_A]_0$	k_{ObsB}	$[(NO_3^-)_B]_0$	$[(NO_3^-)_U]_0$
16	0.0067	2.1825			5.9656
17	0.0055	1.834			6.4968
18	0.0057	0.587			7.8843
19	0.0048	0.5572			8.1156
21	0.0049	1.7704			6.3269
22b	0.0032	1.4933			6.5145
22f	0.0024	1.2369			5.1195
30	0.0066	0.425			4.8015
22c	0.0063	0.9141			10.5356
22e	0.0047	0.703			8.8557
22d	0.0092	1.1651			9.4498
60	0.0685	5.3464	0.0049	0.5914	8.5674
61	0.0496	5.2017			8.0119
62	0.8849	2.7903	0.0361	3.5179	7.5147
73	0.5743	2.7431	0.023	0.7108	8.1319
74	0.0042	0.403			7.6725
75	0.0575	0.5539	0.0072	0.7132	7.6378
77	0.0122	0.1027	0.012	0.1217	2.2591
79	0.0098	0.0448	0.0012	0.1184	0
80	0.0034	0.8391	0.0001	-0.7257	3.1808
63	0.0401	2.0996			10.6075
71	0.0058	0.3815	0.0008	0.9762	1.4847
72	1.3808	0.1101	0.043	0.054	0.6155
153	0.1627	1.1413	0.0028	0.8581	2.3263
154	0.0011	0.3838	0.0718	0.8736	2.0557
155	0.22	4.9697	0.0015	0.9159	4.5557
158	0.084	1.2153	0.0139	0.3862	2.4361
159	0.0746	3.3324			5.3781
160	0.0696	1.6535	0.0043	0.1327	1.9224
165	0.1021	0.635	0.0025	0.3763	4.8672
161	0.0341	1.0553	0.0017	0.6036	4.6859
162	0.1327	2.5769	0.0027	0.845	4.8799
163	0.0234	0.1921	0.0021	0.3249	4.6423
166	0.0321	0.6012	0.0005	1.0843	2.9495
167	0.9599	4.1862	0.0438	4.0392	3.2369
168	3.4797	22.5196	0.039	5.32	3.3723

Table S.4b First-order parameters from fitting to Eqs. 4 & 5, "right nitrate peak"

Experiment #	k_{ObsA}	$[(NO_3^-)_A]_0$	k_{ObsB}	$[(NO_3^-)_B]_0$	$[(NO_3^-)_U]_0$
16	0.0055	0.2173			0.8356
17	0.0055	0.1957			0.7214
18					
19	0.0033	0.1129			0.8126
21	0.0043	0.1369			0.7416
22b	0.0061	0.224			0.6815
22f	0.0033	0.1173			0.476
30	0.0159	0.0858			0.646
22c	0.0038	0.0923			1.0789
22e	0.0048	0.0466			1.0212
22d	0.0081	0.162			0.8902
60	0.071	0.2515	0.0039	0.1432	1.3924
61	0.0496	5.2017			8.0119
62	0.8016	0.6027	0.013	0.1469	0.7968
73					
74	0.8948	0.1328	0.0048	0.1429	0.714
75	0.6476	0.2239	0.0238	0.13	0.5263
77	0.1291	0.0617	0.006	0.1022	0.2409
79	0.0098	0.0448	0.0012	0.1184	0
80	0.0022	0.1428	0.0003	0.0541	0.0577
63	1.3145	1.1298			0.914
71	0.0113	0.039	0.001	0.1084	0.0891
72	1.3808	0.1101	0.043	0.054	0.6155
153	0.8044	0.191	0.0033	0.0936	0.3181
154	0.0997	0.0969	0.0011	0.035	0.2134
155	0.2058	0.5502	0.0015	0.1299	0.5294
158	0.0605	0.1244	0.0031	0.0275	0.3192
159	0.0713	0.2789	0.0001	0.2224	0.3915
160	0.0541	0.129	0.0034	0.018	0.1536
165	0.077	0.0475	0.0011	0.0433	0.564
161	0.0322	0.0741	0.0009	0.0588	0.388
162	0.1102	0.1413	0.0016	0.0601	0.4368
163	0.0221	0.0216	0	0.219	0.3388
166					
167					
168					

Table S.4c. First-order k_{obsA} from half-Life analysis, "full peak" & "right peak"

Experiment #	k_{ObsA} "full peak"	k_{ObsA} "right peak"
16	0.010756	0.006841
17	0.0059	0.005741
18	0.005636	
19	0.004633	0.004816
21	0.004648	0.004408
22b	0.003322	0.005693
22f	0.002185	0.003233
30	0.066774	0.008312
22c	0.004503	0.004077
22e	0.004722	0.004977
22d	0.005876	0.006329
60	0.059429	0.047377
61	0.05839	0.065568
62	0.047007	0.019804
73	0.4658	
74	0.006361	0.012853
75	0.025269	0.693147
77	0.012361	0.0168
79	0.012504	0.001995
80	0.005203	0.001473
63	0.049511	0.077016
71	0.002108	0.002442
72	0.075234	0.214511
153	0.138629	0.346574
154	0.048397	0.067152
155	0.1299	0.125798
158	0.056289	0.061054
159	0.073403	0.049012
160	0.061865	0.021517
165	0.053502	0.040463
161	0.022137	0.034657
162	0.098481	0.058649
163	0.004016	0.00063
166	0.004326	
167	0.141339	
168	0.088776	

Table S.4d Second-order parameters from fitting to Eqs. 4 & 5, "full nitrate peak"

Experiment #	k_{ObsA}	$[(NO_3^-)_A]_0$	$[(NO_3^-)_U]_0$
16	0.0044	2.599428	5.7623
17	0.0034	2.229654	6.2226
18	0.0115	0.738225	7.7969
19	0.0065	0.719994	7.9599
21	0.0033	2.148228	6.0815
22b	0.0014	2.05846	5.9822
22f	0.0019	1.530925	4.8894
30	0.0229	0.672179	4.7468
22c	0.0072	1.090394	10.3905
22e	0.0039	0.983768	8.5835
22d	0.0121	1.376273	9.3724
60	0.0184	6.69344	8.4802
61	0.0053	7.473842	5.8122
62	0.0176	5.102041	7.2212
73	0.1614	3.234153	8.2008
74	0.0072	0.332557	7.7276
75	0.0211	1.258336	7.6226
77	0.0342	0.298445	2.1865
79	0.0421	0.226572	2.0828
80	0.0066	0.884799	2.4618
63	0.014	2.862869	9.9433
71	0.0014	1.383892	1.4535
72	0.0763	1.196602	6.3834
153	0.0214	1.39821	2.439
154	0.0781	1.099989	2.2489
155	0.0602	6.153846	4.9074
158	0.0623	1.807011	2.3983
159	0.0409	4.253509	5.2027
160	0.0654	0.860289	4.9427
165	0.0635	2.110595	1.8764
161	0.0154	1.576044	4.7492
162	0.0407	3.206156	5.0744
163	0.0128	0.512426	4.6222
166	0.0376	0.914244	3.784
167	0.0231	6.535948	3.1018
168	0.0123	7.097232	3.0359

Table S.4e Second-order parameters from fitting to Eqs. 4 & 5, "right nitrate peak"

Experiment #	k_{ObsA}	$[(NO_3^-)_A]_0$	$[(NO_3^-)_U]_0$	k_{ObsA} Scaled*
16	0.0322	0.259727	0.7219	0.003203
17	0.0313	0.23758	0.6918	0.003985
18				
19	0.0177	0.156723	0.77	0.001496
21	0.0347	0.168175	0.7204	0.003313
22b	0.6249	0.286262	0.022	0.050565
22f	0.0321	0.14454	0.4596	0.003909
30				
22c	0.0384	0.114151	1.0609	0.004253
22e	0.0855	0.0579	1.0115	0.010321
22d	0.0694	0.189186	0.8769	0.008441
60	0.1632	0.356037	1.4292	0.01355
61	0.0964	0.305446	0.921	0.008523
62	0.1893	0.287679	0.7865	0.013693
73				
74	0.0346	0.177873	0.6895	0.011012
75	0.5552	0.266852	0.529	0.004207
77	0.103	0.130842	0.2375	0.983466
79	0.024	0.166492	0	0.114579
80	0.01	0.205141	0.052	0.031827
63	0.0259	0.327579	0.6601	0.001805
71	0.0198	0.144527	0.0892	0.009453
72	2.324	0.117849	0.6148	0.04438
153	0.1506	0.121202	0.3254	0.00145
154	1.2263	0.121683	0.2328	0.142792
155	0.4384	0.669792	0.582	0.023178
158	0.5158	0.164802	0.3203	0.024142
159	0.3092	0.351667	0.5722	0.032301
160	0.3906	0.066669	0.577	0.019629
165	0.5188	0.164258	0.1515	0.025693
161	0.1586	0.113421	0.4027	0.008064
162	0.5072	0.173792	0.4549	0.020434
163	0.2418	0.034638	0.5422	0.016491
166				
167				
168				

* The scaled k_{obsA} is obtained by dividing k_{obsA} by 10.5, see text for details.

Table S.4f. Second-order k_{obsA} from half-life analysis, "full peak" & "right peak"

Experiment #	k_{ObsA} "full peak"	k_{ObsA} "right peak"	k_{ObsA} "right peak" scaled*
16	0.00571	0.03787	0.003616
17	0.004246	0.039396	0.003762
18	0.012054		
19	0.009523	0.044693	0.004267
21	0.003769	0.03573	0.003412
22b	0.003694	0.035542	0.003394
22f	0.002533	0.033151	0.003165
30	0.130179	0.099725	0.009522
22c	0.006305	0.034722	0.003315
22e	0.011159	0.134002	0.012795
22d	0.006338	0.062846	0.006001
60	0.013474	0.109156	0.010422
61	0.016538	0.209689	0.020022
62	0.01243	0.042984	0.004104
73	0.219404		
74	0.021402	0.080698	0.007705
75	0.029134	2.753061	0.262868
77	0.073391	0.170043	0.016236
79	0.092399	0.016071	0.001534
80	0.008492	0.010478	0.001001
63	0.040816	0.584795	0.055837
71	0.002003	0.022084	0.002109
72	0.088546	2.831193	0.270328
153	0.076277	1.699187	0.162242
154	0.052501	0.618294	0.059036
155	0.034606	0.270245	0.025804
158	0.049186	0.481904	0.046013
159	0.025884	0.191202	0.018256
160	0.079449	0.318383	0.0304
165	0.0443	0.40427	0.038601
161	0.018333	0.26095	0.024916
162	0.036572	0.416322	0.039751
163	0.007857	0.011319	0.001081
166	0.003457		
167	0.026435		
168	0.012404		

* The scaled k_{obsA} is obtained by dividing k_{obsA} by 10.5, see text for details.

Appendix S.2: Programs used in spline-fitting alpha plots.

Program nt2. Example command: python nt2.py -s 0.01 -i 0.5 -f 16.csv -o 16nt2.csv

```
#finds the rolling slope w/ points evenly distributed across time after
import sys
import getopt
import math


class Point:
	def __init__(self, line, xp, yp):
		self.line = line
		self.x = xp
		self.y = yp
		self.xs = str(self.x)
		self.ys = str(self.y)


def findYintercept(points, slope):
	''' return the y-intercept

		yintercept = Avg(y) - slope*Avg(x)

	'''
	n = len(points)
	avgx = 0
	avgy = 0
	for i in points:
		avgx = avgx + i.x
		avgy = avgy + i.y
	avgx = avgx / n
```

```
        avgy = avgy / n

        yintercept = avgy - (slope * avgx)

        #print " ".join([point.xs + ", " + point.ys for point in points])
        #print "Avgx: " + str(avgx) + "    Avgy: " + str(avgy) + "    Slope: " + str(slope)
        #print "yint: " + str(yintercept)
        #print "-----------------------"

        return yintercept


def findslope(points):
        ''' returns the slope from the following

                slope = n(Summation(x * y)) - Summation(x)Summation(y)
                                / n(Summation(x**2)) - (Summation(x))**2

        '''
        n = len(points)

        crossSum = 0            # Summation(x * y)
        xSum = 0                # Summation(x)
        ySum = 0                # Summation(y)
        xsquareSum = 0          # Summation(x**2)
        for i in points:
                crossSum = crossSum + (i.x * i.y)
                xSum = xSum + i.x
```

```
                ySum = ySum + i.y
                xsquareSum = xsquareSum + pow(i.x,2)

        #print " ".join([point.xs + ", " + point.ys for point in points])
        slope = ( (n * crossSum) - (xSum * ySum) ) / ( (n * xsquareSum) - pow(xSum,2) )



        #print slope
        #print ""
        return slope

def printOutput(outfile, lines):
        delim = ","

        toprint = []
        toprint.append( delim.join(["LogTime", "Slope", "Y-Intercept"]) )
        for i in lines:
                toprint.append( delim.join([str(i[0]), str(i[1]), str(i[2])]) )

        if outfile == "":
                for line in toprint:
                        print line
        else:
                file = open(outfile, "w")
                for line in toprint:
                        file.write(line + "\n")
                file.close()
```

```
def main(argv):
        filename = ""
        fitsize = 0
        outfile = "outDefault.txt"
        FirstPoint = 0
        extraPoints = 3

        try:
                opts, args = getopt.getopt(argv, "f:i:s:o:", ["file=","interval=","step=","output="])
        except getopt.GetoptError:
                usage()
                sys.exit(2)

        for opt, arg in opts:
                if opt in ("-f", "--file"):
                        filename = arg
                elif opt in ("-i", "--interval"):
                        interval = float(arg)
                elif opt in ("-s", "--step"):
                        step = float(arg)
                elif opt in ("-o", "--output"):
                        outfile = arg
                else:
                        print "You need to specify an option"
                        sys.exit()
```

```
file = open(filename)
lines = file.readlines()

data = []
output = []

for index, line in enumerate(lines):
        line = line.rstrip("\n")
        pointsText = line.split(",")
        #pointsFloat = [float(pointsText[0]), float(pointsText[1])]
        data.append(Point(index, float(pointsText[0]), float(pointsText[1])))

CurrentTimeMin = 0 - (interval/2)
CurrentTimeMax = CurrentTimeMin + interval
#first time range centers around 0

while extraPoints > 1:
        FirstPoint = 0
        fitsize = 0
        extraPointCounter = 0
        #FirstPoint & fitsize start at zero, and are reset with every iteration
        for index, point in enumerate(data):
                if data[index].x < CurrentTimeMin:
                        FirstPoint = FirstPoint + 1
                        #The starting of the linear fit will be increased by 1 point for every
point before it.
                if data[index].x < CurrentTimeMax:
                        if data[index].x > CurrentTimeMin:
```

```
                        fitsize = fitsize + 1
                if data[index].x == CurrentTimeMax:
                        if data[index].x > CurrentTimeMin:
                                fitsize = fitsize + 1
                        #if the point is in the range, then the fitsize is increased by 1 -
indicating that a point is added.
                if data[index].x > CurrentTimeMax - (interval/2):
                        extraPointCounter = extraPointCounter + 1
                                #the above finds the correct fitsize when starting at first point,
and then adds 1 for every point
                                #so if only 2 points were in the first interval, it would go from
fitsize = 0 to fitsize = 2
        #after finding the fitsize, the program has to find the slope & y-intercept and add them to
the output array
        if fitsize > 1:
                slope = findslope(data[FirstPoint:FirstPoint+fitsize])
                yintercept = findYintercept(data[FirstPoint:FirstPoint+fitsize],slope)
                CurrentTime = CurrentTimeMin + interval/2
                output.append([CurrentTime, slope, yintercept,])
                #only find slope etc. if there are enough points
        CurrentTimeMin = CurrentTimeMin + step
        CurrentTimeMax = CurrentTimeMin + interval
        #now the min and max for the time interval being looked at are increased by the step.
        extraPoints = extraPointCounter
                #then the program has to loop back to searching in every point w/in
new interval,
                #unless the extraPointCounter is not greater than 1, which is checked
here.
```

```
        printOutput(outfile, output)


if __name__ == "__main__":
        main(sys.argv[1:])
```

Program ntf7 Example command:

python ntf7.py -s 0.01 -i 0.5 -f 16.csv -o 16ntf7.csv -g 1.85

```
#finds the range of the linear region, assuming slope 0.5
#includes having to provide a guess of the middle of the range, and an initial interval size,
#Based on next points (via interval) having to be w/in 2 local linear fit ResidualStDev of range's fit
#has new "Final Max" and "Final Min" functions.
#("Final Max" & Min are when total residuals double so fit starts getting really bad.)
import sys
import getopt
import math


class Point:
        def __init__(self, line, xp, yp):
                self.line = line
                self.x = xp
                self.y = yp
                self.xs = str(self.x)
                self.ys = str(self.y)


def findYintercept(points, slope):
        ''' return the y-intercept

                yintercept = Avg(y) - slope*Avg(x)

        '''
        n = len(points)
        avgx = 0
```

```
        avgy = 0
        for i in points:
                avgx = avgx + i.x
                avgy = avgy + i.y
        avgx = avgx / n
        avgy = avgy / n

        yintercept = avgy - (slope * avgx)

        #print " ".join([point.xs + ", " + point.ys for point in points])
        #print "Avgx: " + str(avgx) + "    Avgy: " + str(avgy) + "    Slope: " + str(slope)
        #print "yint: " + str(yintercept)
        #print "-----------------------"

        return yintercept


def findslope(points):
        ''' returns the slope from the following

                slope = n(Summation(x * y)) - Summation(x)Summation(y)
                                / n(Summation(x**2)) - (Summation(x))**2

        '''
        n = len(points)

        crossSum = 0                # Summation(x * y)
        xSum = 0                                        # Summation(x)
```

```
    ySum = 0                                    # Summation(y)
    xsquareSum = 0              # Summation(x**2)
    for i in points:
        crossSum = crossSum + (i.x * i.y)
        xSum = xSum + i.x
        ySum = ySum + i.y
        xsquareSum = xsquareSum + pow(i.x,2)

    #print " ".join([point.xs + ", " + point.ys for point in points])
    slope = ( (n * crossSum) - (xSum * ySum) ) / ( (n * xsquareSum) - pow(xSum,2) )

    #print slope
    #print ""
    return slope

def findCorrelation(points):
    ''' returns the corrrelation of the best fit line
    '''
    n = len(points)
    crossSum = 0                # Summation(x * y)
    xSum = 0                                    # Summation(x)
    ySum = 0                                    # Summation(y)
    xsquareSum = 0              # Summation(x**2)
    xVariance = 0
    yVariance = 0
    for i in points:
        crossSum = crossSum + ( i.x * i.y )
        xSum = xSum + i.x
```

```
            ySum = ySum + i.y
        #Note: below, xSum/n is just the average of x. Same for y.
        for i in points:
            xVariance = xVariance + ( i.x - xSum/n )**2
            yVariance = yVariance + ( i.y - ySum/n )**2
            CrossVariance = ( i.x - xSum/n)*( i.y - ySum/n)
        correlation = (CrossVariance**2/(xVariance*yVariance))**0.5
        return correlation


def findResidualStDev(points, slope, yintercept):
        ''' returns the corrrelation of the best fit line
        '''
        n = len(points)

        crossSum = 0            # Summation(x * y)
        xSum = 0                # Summation(x)
        ySum = 0                # Summation(y)
        xsquareSum = 0          # Summation(x**2)
        yResidualSumSq = 0.0
        #Below finds the sum of the y residuals, necessary to find the standard deviation of the residuals.
        #the "average" value is the predicted point of the line, given by slope*x + b, or slope*i.x+yintercept
        #the squaring and square rooting is to get the absolute value.
        for i in points:
            yResidualSumSq = yResidualSumSq + (i.y - (slope*i.x + yintercept))**2

        ResidualStDev = (yResidualSumSq/(n-1))**0.5
        return ResidualStDev
```

```
def checkInterval(IntervalPoints, LocalResidualStDev,slope,yintercept):
        ''' checks the points given versus the line fed,
                fits the interval to a linear fit, and assumes at least 1 point of the interval must be within
                2 ResidualStDev of the local area.
        '''
        Counter=1
        for i in IntervalPoints:
                if 2*LocalResidualStDev > ((i.y - (slope*i.x + yintercept))**2)**0.5:
                        Counter = 0
        #If any of the interval's points have residuals within 2 local Standard Deviations of the fed line
(line to compare)
        #the Counter will be set to 0 (indicating that the interval is okay).
        return Counter


def printOutput(outfile, lines):
        delim = ","


        toprint = []
        toprint.append( delim.join(["Onset", "Endset", "slope", "Y-Intercept", "StDev", "R^2" ]) )
        for i in lines:
                toprint.append( delim.join([str(i[0]), str(i[1]), str(i[2]),str(i[3]), str(i[4]), str(i[5])]) )


        if outfile == "":
                for line in toprint:
                        print line
        else:
                file = open(outfile, "w")
```

```
        for line in toprint:
            file.write(line + "\n")
        file.close()



def main(argv):
        filename = ""
        fitsize = 0
        outfile = "outDefault.txt"
        FirstPoint = 0
        extraPoints = 3
        PointsToCheck = 5
        #EndMax and EndMin are counters, when they reach 1, the extrapoint counter will be turned off for that end.
        EndMax = 0
        EndMin = 0

        try:
            opts, args = getopt.getopt(argv, "f:g:i:s:o:", ["file=","guess=","interval=","step=","output="])
        except getopt.GetoptError:
            usage()
            sys.exit(2)

        for opt, arg in opts:
            if opt in ("-f", "--file"):
                filename = arg
```

```
        elif opt in ("-i", "--interval"):
                interval = float(arg)
        elif opt in ("-g", "--guess"):
                guess = float(arg)
        elif opt in ("-s", "--step"):
                step = float(arg)
        elif opt in ("-o", "--output"):
                outfile = arg
        else:
                print "You need to specify an option"
                sys.exit()

file = open(filename)
lines = file.readlines()

data = []
output = []

for index, line in enumerate(lines):
        line = line.rstrip("\n")
        pointsText = line.split(",")
        #pointsFloat = [float(pointsText[0]), float(pointsText[1])]
        data.append(Point(index, float(pointsText[0]), float(pointsText[1])))

CurrentTimeMin = guess - (interval/2)
CurrentTimeMax = CurrentTimeMin + interval
#first time range centers around 0
```

```
while extraPoints > 0:
        FirstPoint = 0
        fitsize = 0
        extraPointCounter = 0
        PointsBeforeMin = 0
        PointsAfterMin = 0
        PointsBeforeMax = 0
        PointsAfterMax = 0
        #FirstPoint & fitsize start at zero, and are reset with every iteration
        for index, point in enumerate(data):
            if data[index].x < CurrentTimeMin:
                FirstPoint = FirstPoint + 1
                #The starting of the linear fit will be increased by 1 point for every point before it.
            if data[index].x < CurrentTimeMax:
                if data[index].x > CurrentTimeMin:
                    fitsize = fitsize + 1
            if data[index].x == CurrentTimeMax:
                fitsize = fitsize + 1
                #if the point is in the range, then the fitsize is increased by 1 - indicating that a point is added.
            if data[index].x > CurrentTimeMax:
                if EndMax == 0:
                    extraPointCounter = extraPointCounter + 1
            if data[index].x < CurrentTimeMin:
                if EndMin == 0:
                    extraPointCounter = extraPointCounter + 1
```

```
                                        #the above finds the correct fitsize when starting at first point,
and then adds 1 for every point
                                        #so if only 2 points were in the first interval, it would go from
fitsize = 0 to fitsize = 2
                        if data[index].x > CurrentTimeMin - interval/2:
                                if data[index].x < CurrentTimeMin:
                                        PointsBeforeMin = PointsBeforeMin + 1
                        if data[index].x < CurrentTimeMin + interval/2:
                                if data[index].x >= CurrentTimeMin:
                                        PointsAfterMin = PointsAfterMin + 1
                        if data[index].x > CurrentTimeMax - interval/2:
                                if data[index].x <= CurrentTimeMax :
                                        PointsBeforeMax = PointsBeforeMax + 1
                        if data[index].x < CurrentTimeMax + interval/2:
                                if data[index].x > CurrentTimeMax :
                                        PointsAfterMax = PointsAfterMax + 1
                        #The above counters are used to determine the ranges for fitting to check the
upcoming ranges
                                        #before the Min Time and after the Max Time
                slope = 0.5
                yintercept = findYintercept(data[FirstPoint:(FirstPoint+fitsize)+1],slope)
                ResidualStDev = findResidualStDev(data[FirstPoint:(FirstPoint+fitsize)+1], slope,
yintercept)
                # for defined point ranges, upper point of range doesn't count, but if passing "points" then
doesn't matter.
                # So for the point ranges defined above, there is always an extra +1 for the upper point.
                #The "Final" max and min are if the total residuals would more than double.
                PotentialMinRange = data[FirstPoint-PointsBeforeMin:(FirstPoint+fitsize)+1]
```

```
                FinalEndMin = 2*ResidualStDev - findResidualStDev(PotentialMinRange,
findslope(PotentialMinRange), findYintercept(PotentialMinRange,findslope(PotentialMinRange)))
                PotentialMaxRange = data[FirstPoint:(FirstPoint+fitsize+PointsAfterMax)+1]
                FinalEndMax = 2*ResidualStDev - findResidualStDev(PotentialMaxRange,
findslope(PotentialMaxRange), findYintercept(PotentialMaxRange,findslope(PotentialMaxRange)))

                if FinalEndMax < 0:
                        EndMax = 0
                if FinalEndMin < 0:
                        EndMin=0

                #below checks upperpoints and lower points, then changes Min & Max accordingly
(assuming Final Max & Min not reached)
                if FinalEndMin > 0:
                        LocalRange = data[FirstPoint-PointsBeforeMin:(FirstPoint+PointsAfterMin)+1]
                        LocalResidualStDev = findResidualStDev(LocalRange, findslope(LocalRange),
findYintercept(LocalRange,findslope(LocalRange)))
                        EndMin=checkInterval(data[FirstPoint-
PointsBeforeMin:(FirstPoint)+1],LocalResidualStDev, slope,yintercept)
                if FinalEndMax > 0:
                        LocalRange = data[FirstPoint+fitsize-
PointsBeforeMax:(FirstPoint+fitsize+PointsAfterMax)+1]
                        LocalResidualStDev = findResidualStDev(LocalRange, findslope(LocalRange),
findYintercept(LocalRange,findslope(LocalRange)))

        EndMax=checkInterval(data[FirstPoint+fitsize:(FirstPoint+fitsize+PointsAfterMax)+1],LocalResi
dualStDev,slope,yintercept)
```

```
                #above, the EndMax and EndMin are checked, to see if it should be increased or not (if it's not ended).
                        #Then, below it's either increased or not.
                if EndMax == 0:
                        CurrentTimeMax = CurrentTimeMax + step
                if EndMin == 0:
                        CurrentTimeMin = CurrentTimeMin - step
                #now the min and max for the time interval being looked at are increased by the step if their next points are okay
                #otherwise they have no change and the loop starts again.
                UpperPointInt = data[FirstPoint-PointsBeforeMin:(FirstPoint+PointsAfterMin)+1]
                LowerPointInt = data[FirstPoint+fitsize-PointsBeforeMax:(FirstPoint+fitsize+PointsAfterMax)+1]
                LowerPointsResidualStDev = findResidualStDev(LowerPointInt, findslope(LowerPointInt), findYintercept(LowerPointInt,findslope(LowerPointInt)))
                UpperPointsResidualStDev = findResidualStDev(UpperPointInt, findslope(UpperPointInt), findYintercept(UpperPointInt,findslope(UpperPointInt)))
                correlation = findCorrelation(data[FirstPoint:(FirstPoint+fitsize)+1])
                output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept, LowerPointsResidualStDev, UpperPointsResidualStDev])
                extraPoints = extraPointCounter
                                #then the program has to loop back to searching in every point w/in new interval,
                                #unless the extraPointCounter is not greater than 1, which is checked here.
        #The Current Time Min, fitsize & first point will be whereever they were on the final loop of above
        #The correlation and range will be determined from them.
```

```
            output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept, ResidualStDev,
correlation])
            slope = findslope(data[FirstPoint:(FirstPoint+fitsize)+1])
            correlation = findCorrelation(data[FirstPoint:(FirstPoint+fitsize)+1])
            yintercept = findYintercept(data[FirstPoint:(FirstPoint+fitsize)+1],slope)
            ResidualStDev = findResidualStDev(data[FirstPoint:(FirstPoint+fitsize)+1], slope, yintercept)
            #the final output consists of the Onset, Endset, the slope, y-intercept, the residual's standard
deviation, and the Correlation Coeff
            output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept, ResidualStDev,
correlation])
            printOutput(outfile, output)


if __name__ == "__main__":
            main(sys.argv[1:])
```

Program ntf8. Example command:

python ntf8.py -s 0.01 -i 0.5 -f 16.csv -o 16ntf7.csv -g 1.85

```
#includes having to provide a guess of the middle of the range, and an initial interval size,
#Based on next points (via interval) having to be w/in 2 local linear fit ResidualStDev of range's fit
#has new "Final Max" and "Final Min" functions.
#("Final Max" & Min are when total residuals double so fit starts getting really bad.)
import sys
import getopt
import math


class Point:
        def __init__(self, line, xp, yp):
                self.line = line
                self.x = xp
                self.y = yp
                self.xs = str(self.x)
                self.ys = str(self.y)


def findYintercept(points, slope):
        ''' return the y-intercept

                yintercept = Avg(y) - slope*Avg(x)

        '''
        n = len(points)
        avgx = 0
        avgy = 0
        for i in points:
```

```
            avgx = avgx + i.x
            avgy = avgy + i.y
        avgx = avgx / n
        avgy = avgy / n

        yintercept = avgy - (slope * avgx)

        #print " ".join([point.xs + ", " + point.ys for point in points])
        #print "Avgx: " + str(avgx) + "    Avgy: " + str(avgy) + "    Slope: " + str(slope)
        #print "yint: " + str(yintercept)
        #print "-----------------------"

        return yintercept



def findslope(points):
        ''' returns the slope from the following

            slope = n(Summation(x * y)) - Summation(x)Summation(y)
                                / n(Summation(x**2)) - (Summation(x))**2

        '''
        n = len(points)

        crossSum = 0            # Summation(x * y)
        xSum = 0                # Summation(x)
        ySum = 0                # Summation(y)
        xsquareSum = 0          # Summation(x**2)
```

```
        for i in points:
                crossSum = crossSum + (i.x * i.y)
                xSum = xSum + i.x
                ySum = ySum + i.y
                xsquareSum = xsquareSum + pow(i.x,2)

        #print " ".join([point.xs + ", " + point.ys for point in points])
        slope = ( (n * crossSum) - (xSum * ySum) ) / ( (n * xsquareSum) - pow(xSum,2) )

        #print slope
        #print ""
        return slope

def findCorrelation(points):
        ''' returns the corrrelation of the best fit line
        '''
        n = len(points)
        crossSum = 0                # Summation(x * y)
        xSum = 0                                # Summation(x)
        ySum = 0                                # Summation(y)
        xsquareSum = 0              # Summation(x**2)
        xVariance = 0
        yVariance = 0
        for i in points:
                crossSum = crossSum + ( i.x * i.y )
                xSum = xSum + i.x
                ySum = ySum + i.y
        #Note: below, xSum/n is just the average of x. Same for y.
```

```
        for i in points:
                xVariance = xVariance + ( i.x - xSum/n )**2
                yVariance = yVariance + ( i.y - ySum/n )**2
                CrossVariance = ( i.x - xSum/n)*( i.y - ySum/n)
        correlation = (CrossVariance**2/(xVariance*yVariance))**0.5
        return correlation


def findResidualStDev(points, slope, yintercept):
        ''' returns the corrrelation of the best fit line
        '''
        n = len(points)

        crossSum = 0            # Summation(x * y)
        xSum = 0                        # Summation(x)
        ySum = 0                        # Summation(y)
        xsquareSum = 0          # Summation(x**2)
        yResidualSumSq = 0.0
        #Below finds the sum of the y residuals, necessary to find the standard deviation of the residuals.
        #the "average" value is the predicted point of the line, given by slope*x + b, or slope*i.x+yintercept
        #the squaring and square rooting is to get the absolute value.
        for i in points:
                yResidualSumSq = yResidualSumSq + (i.y - (slope*i.x + yintercept))**2

        ResidualStDev = (yResidualSumSq/(n-1))**0.5
        return ResidualStDev

def checkInterval(IntervalPoints, LocalResidualStDev,slope,yintercept):
```

```
        ''' checks the points given versus the line fed,
                fits the interval to a linear fit, and assumes at least 1 point of the interval must be within
                2 ResidualStDev of the local area.
        '''
        Counter=1
        for i in IntervalPoints:
                if 2*LocalResidualStDev > ((i.y - (slope*i.x + yintercept))**2)**0.5:
                        Counter = 0
        #If any of the interval's points have residuals within 2 local Standard Deviations of the fed line
(line to compare)
        #the Counter will be set to 0 (indicating that the interval is okay).
        return Counter


def printOutput(outfile, lines):
        delim = ","

        toprint = []
        toprint.append( delim.join(["Onset", "Endset", "slope", "Y-Intercept", "StDev", "R^2" ]) )
        for i in lines:
                toprint.append( delim.join([str(i[0]), str(i[1]), str(i[2]),str(i[3]), str(i[4]), str(i[5])]) )

        if outfile == "":
                for line in toprint:
                        print line
        else:
                file = open(outfile, "w")
                for line in toprint:
                        file.write(line + "\n")
```

```
        file.close()



def main(argv):
        filename = ""
        fitsize = 0
        outfile = "outDefault.txt"
        FirstPoint = 0
        extraPoints = 3
        PointsToCheck = 5
        #EndMax and EndMin are counters, when they reach 1, the extrapoint counter will be turned off for that end.
        EndMax = 0
        EndMin = 0


        try:
                opts, args = getopt.getopt(argv, "f:g:i:s:o:", ["file=","guess=","interval=","step=","output="])
        except getopt.GetoptError:
                usage()
                sys.exit(2)


        for opt, arg in opts:
                if opt in ("-f", "--file"):
                        filename = arg
                elif opt in ("-i", "--interval"):
                        interval = float(arg)
```

```
        elif opt in ("-g", "--guess"):
                guess = float(arg)
        elif opt in ("-s", "--step"):
                step = float(arg)
        elif opt in ("-o", "--output"):
                outfile = arg
        else:
                print "You need to specify an option"
                sys.exit()

file = open(filename)
lines = file.readlines()

data = []
output = []

for index, line in enumerate(lines):
        line = line.rstrip("\n")
        pointsText = line.split(",")
        #pointsFloat = [float(pointsText[0]), float(pointsText[1])]
        data.append(Point(index, float(pointsText[0]), float(pointsText[1])))

CurrentTimeMin = guess - (interval/2)
CurrentTimeMax = CurrentTimeMin + interval
#first time range centers around 0

while extraPoints > 0:
        FirstPoint = 0
```

```
fitsize = 0
extraPointCounter = 0
PointsBeforeMin = 0
PointsAfterMin = 0
PointsBeforeMax = 0
PointsAfterMax = 0
#FirstPoint & fitsize start at zero, and are reset with every iteration
for index, point in enumerate(data):
    if data[index].x < CurrentTimeMin:
        FirstPoint = FirstPoint + 1
        #The starting of the linear fit will be increased by 1 point for every point before it.
    if data[index].x < CurrentTimeMax:
        if data[index].x > CurrentTimeMin:
            fitsize = fitsize + 1
    if data[index].x == CurrentTimeMax:
        fitsize = fitsize + 1
        #if the point is in the range, then the fitsize is increased by 1 - indicating that a point is added.
    if data[index].x > CurrentTimeMax:
        if EndMax == 0:
            extraPointCounter = extraPointCounter + 1
    if data[index].x < CurrentTimeMin:
        if EndMin == 0:
            extraPointCounter = extraPointCounter + 1
            #the above finds the correct fitsize when starting at first point, and then adds 1 for every point
```

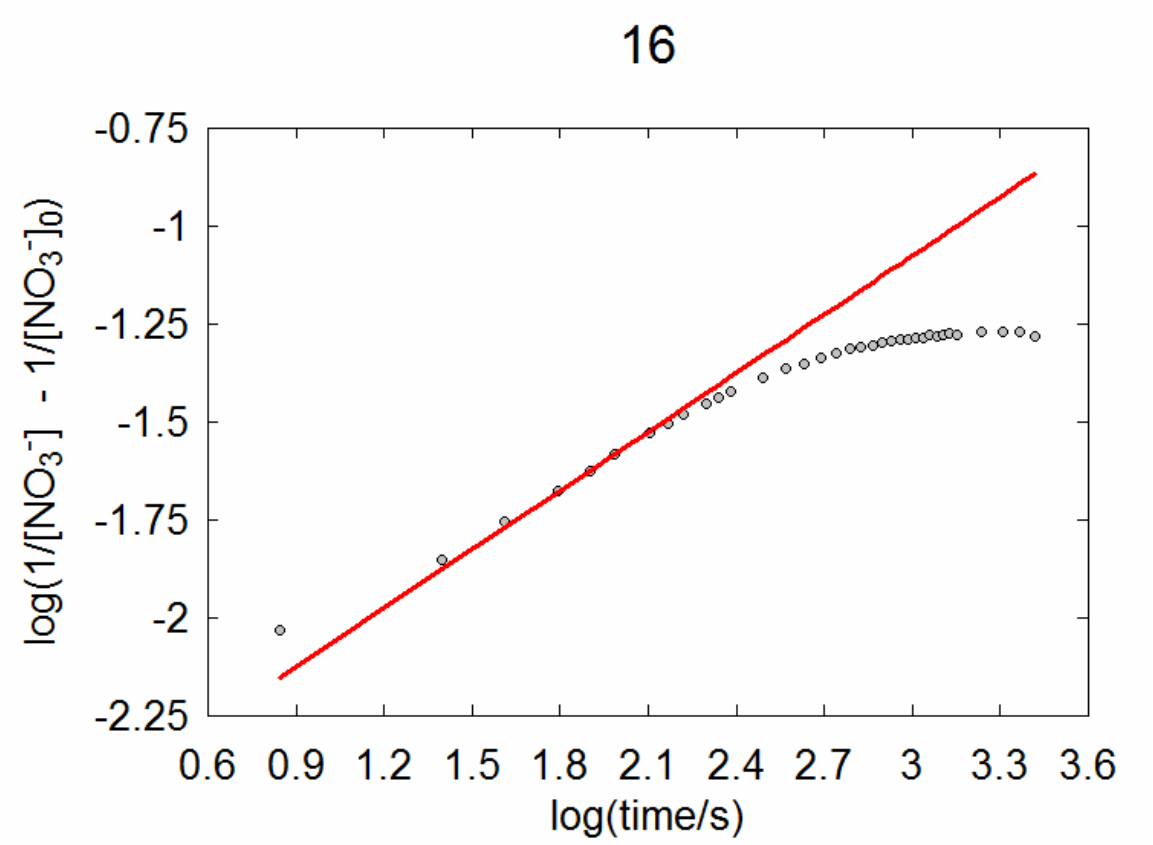
```
                                        #so if only 2 points were in the first interval, it would go from
fitsize = 0 to fitsize = 2
                        if data[index].x > CurrentTimeMin - interval/2:
                                if data[index].x < CurrentTimeMin:
                                        PointsBeforeMin = PointsBeforeMin + 1
                        if data[index].x < CurrentTimeMin + interval/2:
                                if data[index].x >= CurrentTimeMin:
                                        PointsAfterMin = PointsAfterMin + 1
                        if data[index].x > CurrentTimeMax - interval/2:
                                if data[index].x <= CurrentTimeMax :
                                        PointsBeforeMax = PointsBeforeMax + 1
                        if data[index].x < CurrentTimeMax + interval/2:
                                if data[index].x > CurrentTimeMax :
                                        PointsAfterMax = PointsAfterMax + 1
                        #The above counters are used to determine the ranges for fitting to check the
upcoming ranges
                                        #before the Min Time and after the Max Time
                slope = findslope(data[FirstPoint:(FirstPoint+fitsize)+1])
                yintercept = findYintercept(data[FirstPoint:(FirstPoint+fitsize)+1],slope)
                ResidualStDev = findResidualStDev(data[FirstPoint:(FirstPoint+fitsize)+1], slope,
yintercept)
                # for defined point ranges, upper point of range doesn't count, but if passing "points" then
doesn't matter.
                # So for the point ranges defined above, there is always an extra +1 for the upper point.
                #The "Final" max and min are if the total residuals would more than double.
                PotentialMinRange = data[FirstPoint-PointsBeforeMin:(FirstPoint+fitsize)+1]
                FinalEndMin = 2*ResidualStDev - findResidualStDev(PotentialMinRange,
findslope(PotentialMinRange), findYintercept(PotentialMinRange,findslope(PotentialMinRange)))
```

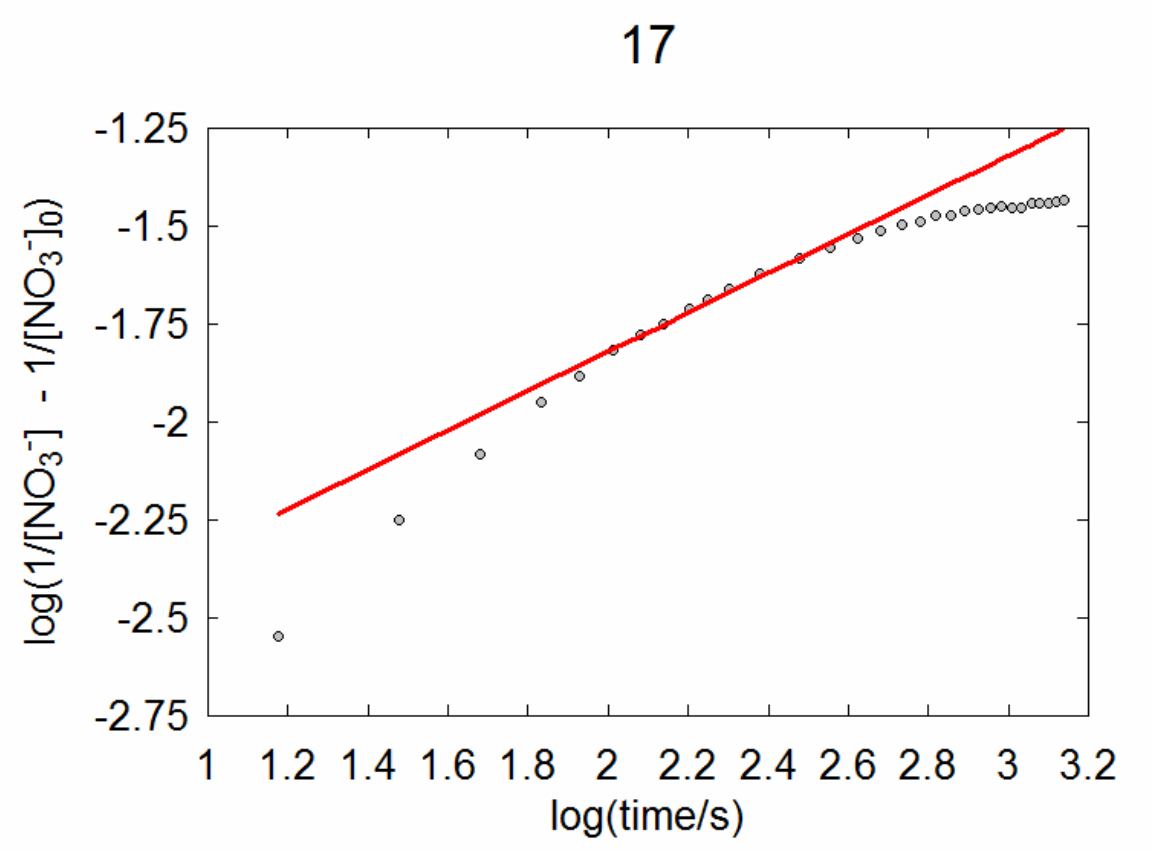
```
PotentialMaxRange = data[FirstPoint:(FirstPoint+fitsize+PointsAfterMax)+1]
FinalEndMax = 2*ResidualStDev - findResidualStDev(PotentialMaxRange, findslope(PotentialMaxRange), findYintercept(PotentialMaxRange,findslope(PotentialMaxRange)))

if FinalEndMax < 0:
    EndMax = 0
if FinalEndMin < 0:
    EndMin=0

#below checks upperpoints and lower points, then changes Min & Max accordingly (assuming Final Max & Min not reached)
if FinalEndMin > 0:
    LocalRange = data[FirstPoint-PointsBeforeMin:(FirstPoint+PointsAfterMin)+1]
    LocalResidualStDev = findResidualStDev(LocalRange, findslope(LocalRange), findYintercept(LocalRange,findslope(LocalRange)))
    EndMin=checkInterval(data[FirstPoint-PointsBeforeMin:(FirstPoint)+1],LocalResidualStDev, slope,yintercept)
if FinalEndMax > 0:
    LocalRange = data[FirstPoint+fitsize-PointsBeforeMax:(FirstPoint+fitsize+PointsAfterMax)+1]
    LocalResidualStDev = findResidualStDev(LocalRange, findslope(LocalRange), findYintercept(LocalRange,findslope(LocalRange)))

EndMax=checkInterval(data[FirstPoint+fitsize:(FirstPoint+fitsize+PointsAfterMax)+1],LocalResidualStDev,slope,yintercept)
#above, the EndMax and EndMin are checked, to see if it should be increased or not (if it's not ended).
    #Then, below it's either increased or not.
```

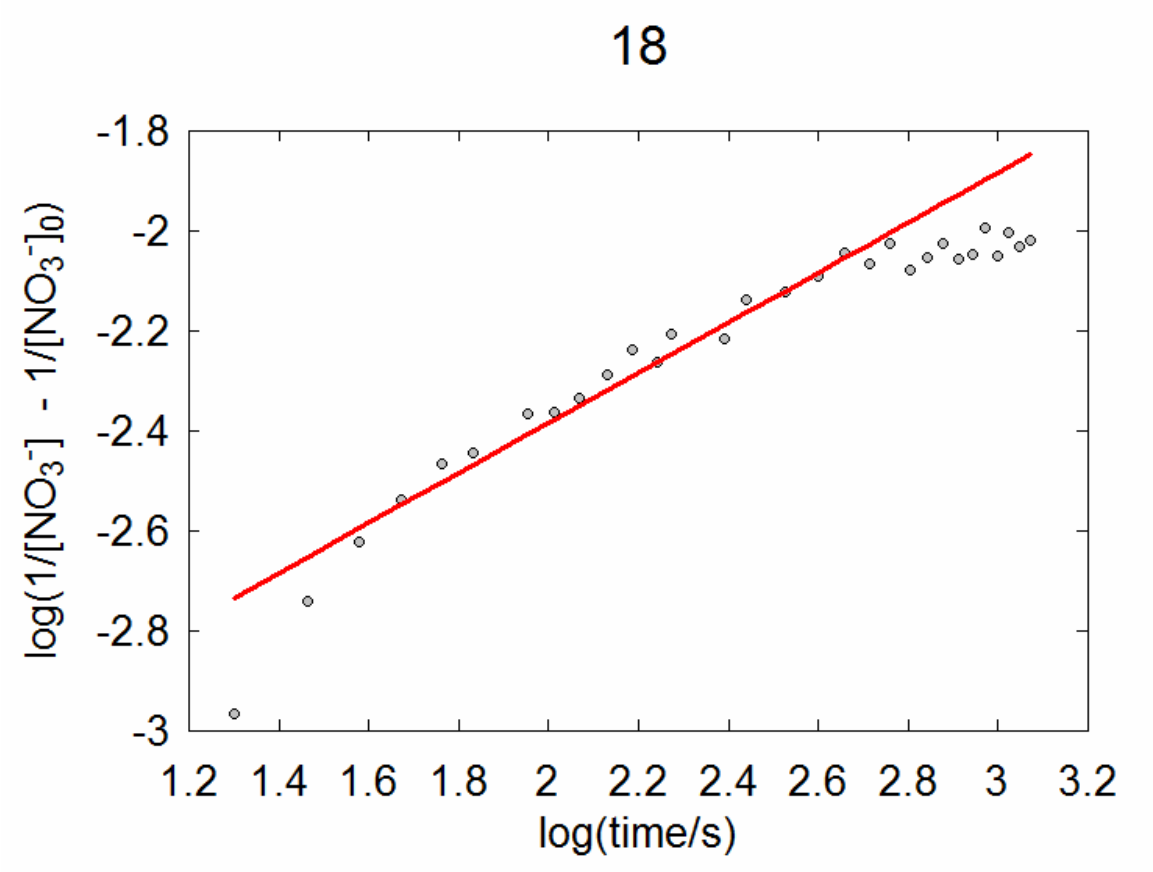
```
            if EndMax == 0:
                CurrentTimeMax = CurrentTimeMax + step
            if EndMin == 0:
                CurrentTimeMin = CurrentTimeMin - step
            #now the min and max for the time interval being looked at are increased by the step if
their next points are okay
            #otherwise they have no change and the loop starts again.
            UpperPointInt = data[FirstPoint-PointsBeforeMin:(FirstPoint+PointsAfterMin)+1]
            LowerPointInt = data[FirstPoint+fitsize-
PointsBeforeMax:(FirstPoint+fitsize+PointsAfterMax)+1]
            LowerPointsResidualStDev = findResidualStDev(LowerPointInt,
findslope(LowerPointInt), findYintercept(LowerPointInt,findslope(LowerPointInt)))
            UpperPointsResidualStDev = findResidualStDev(UpperPointInt,
findslope(UpperPointInt), findYintercept(UpperPointInt,findslope(UpperPointInt)))
            correlation = findCorrelation(data[FirstPoint:(FirstPoint+fitsize)+1])
            output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept,
LowerPointsResidualStDev, UpperPointsResidualStDev])
            extraPoints = extraPointCounter
                    #then the program has to loop back to searching in every point w/in
new interval,
                    #unless the extraPointCounter is not greater than 1, which is checked
here.
        #The Current Time Min, fitsize & first point will be whereever they were on the final loop of
above
        #The correlation and range will be determined from them.
        output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept, ResidualStDev,
correlation])
        slope = findslope(data[FirstPoint:(FirstPoint+fitsize)+1])
```

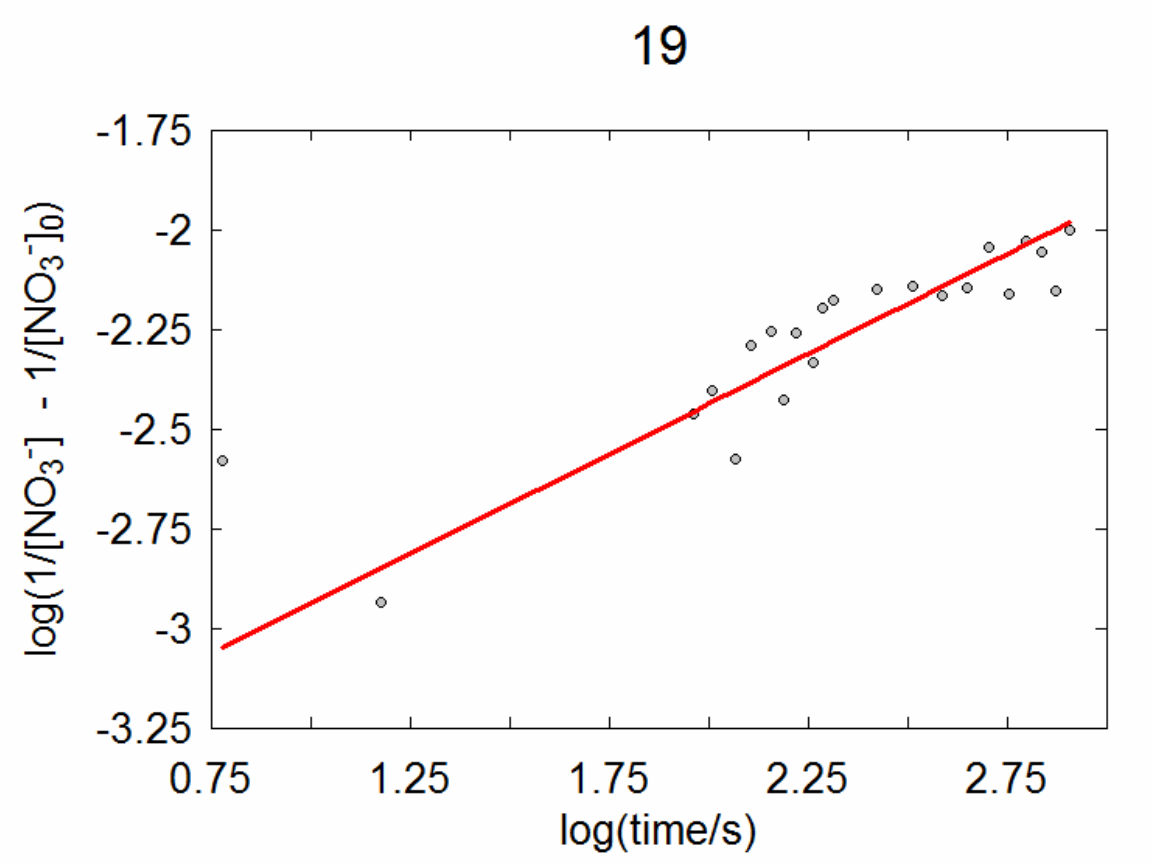
```
        correlation = findCorrelation(data[FirstPoint:(FirstPoint+fitsize)+1])
        yintercept = findYintercept(data[FirstPoint:(FirstPoint+fitsize)+1],slope)
        ResidualStDev = findResidualStDev(data[FirstPoint:(FirstPoint+fitsize)+1], slope, yintercept)
        #the final output consists of the Onset, Endset, the slope, y-intercept, the residual's standard deviation, and the Correlation Coeff
        output.append([CurrentTimeMin, CurrentTimeMax, slope, yintercept, ResidualStDev, correlation])
        printOutput(outfile, output)


if __name__ == "__main__":
        main(sys.argv[1:])
```

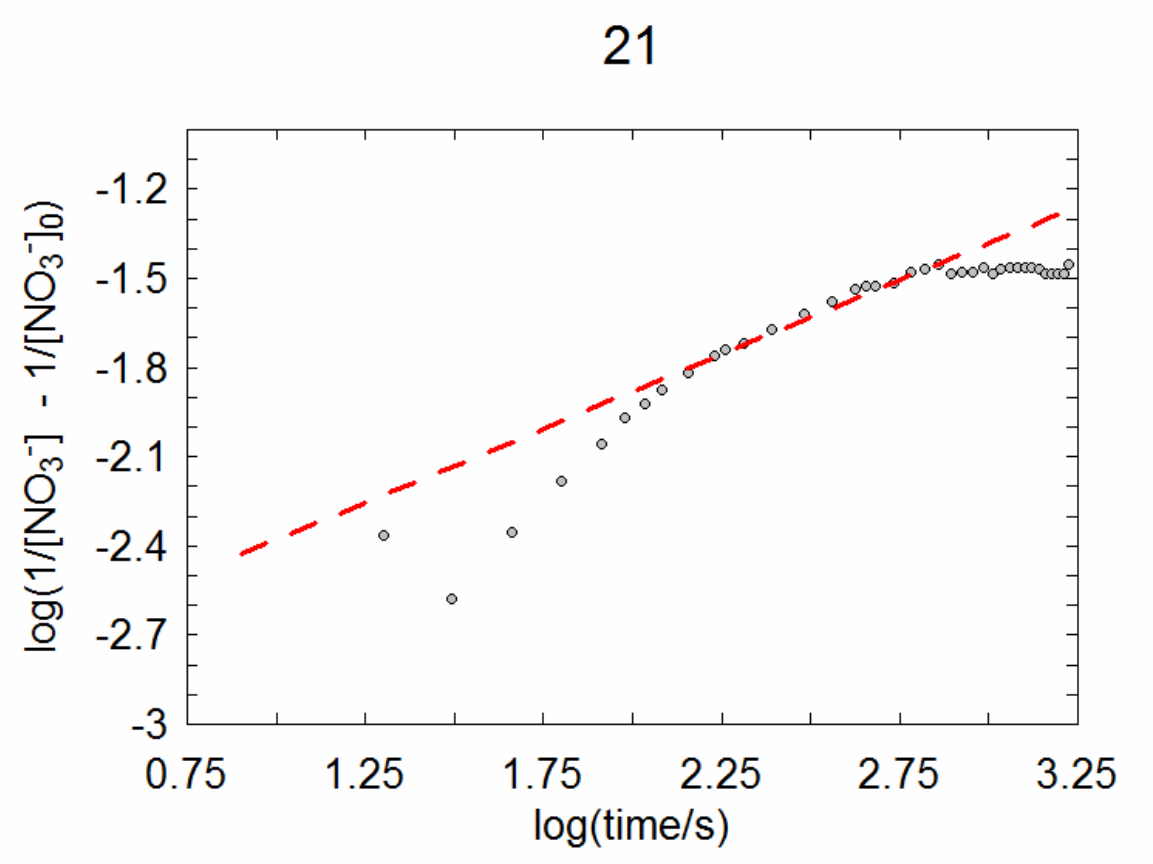
Appendix S.3: Alpha plots of experiments.

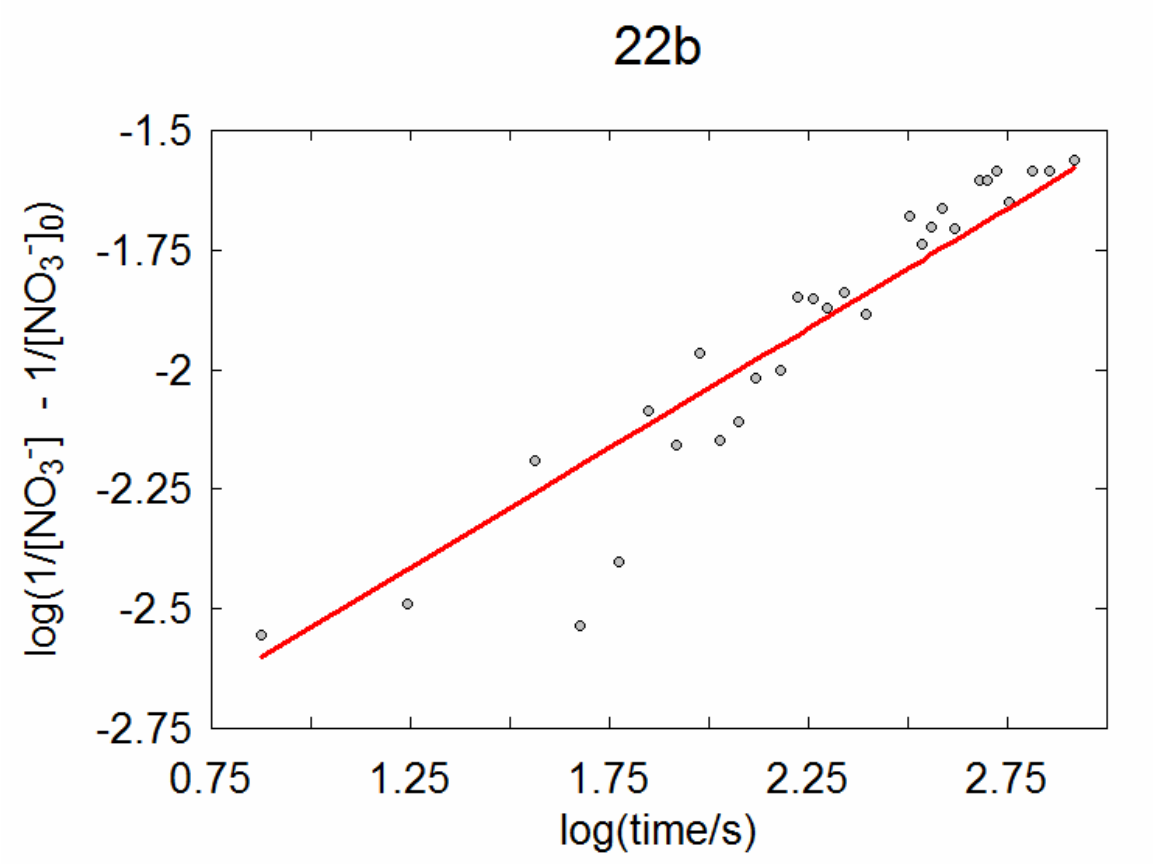


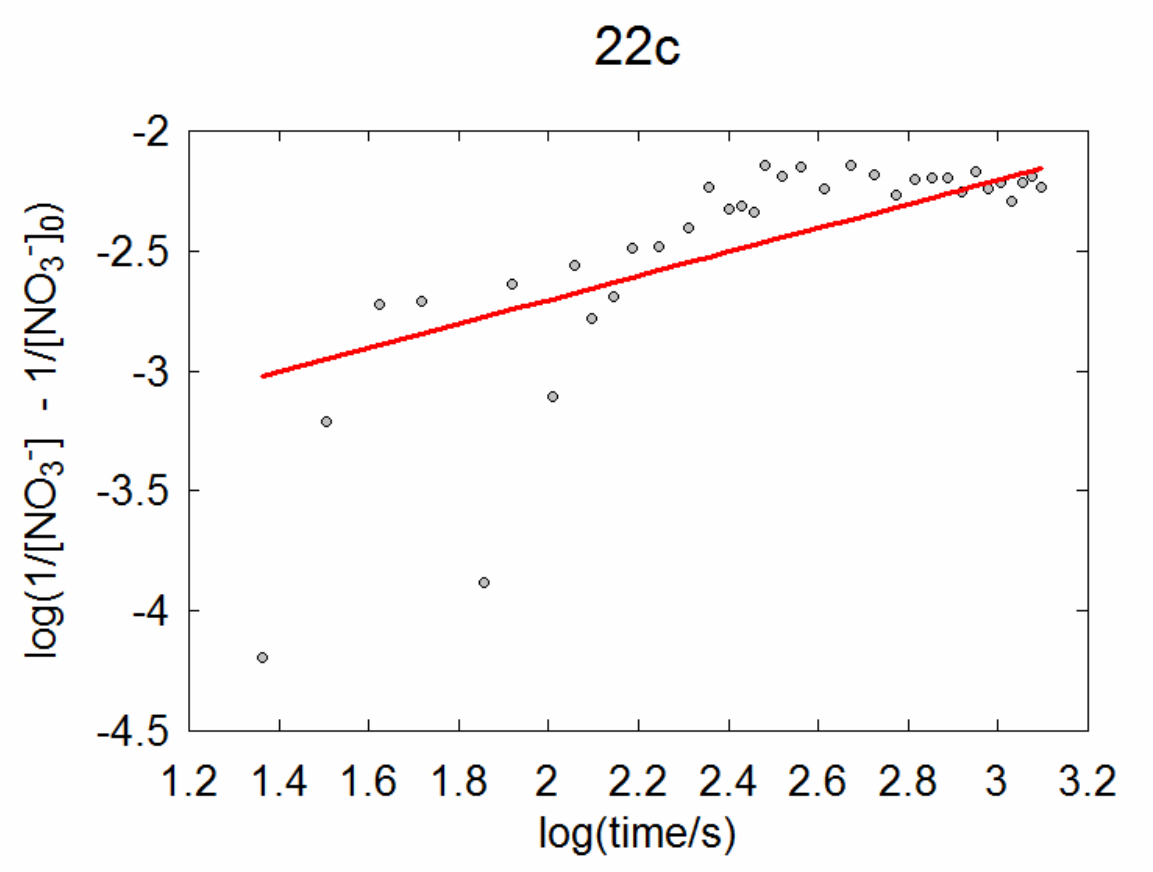

17
-1.25
-1.5
-1.75
-2
-2.25
-2.5
-2.75
log(1/[NO3-] - 1/[NO3-]0)
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2
log(time/s)

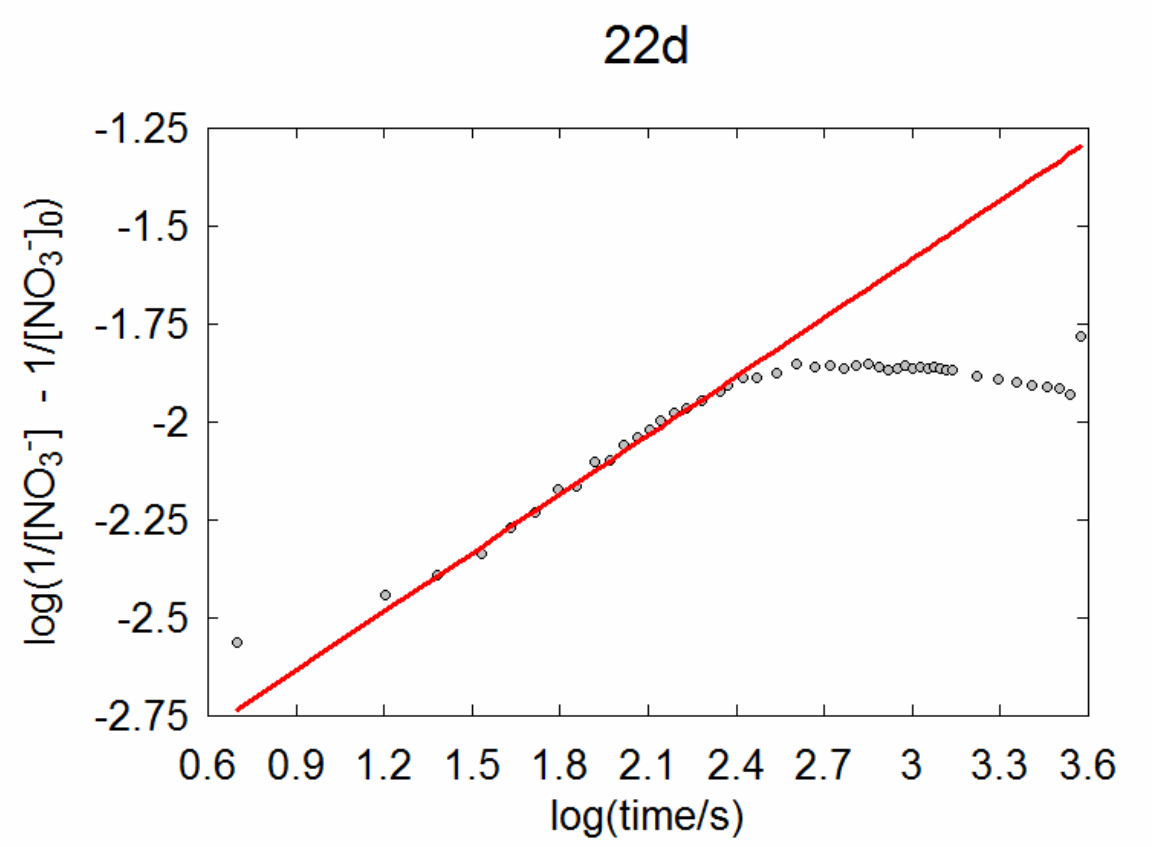

18
-1.8
-2
-2.2
-2.4
-2.6
-2.8
-3
log(1/[NO3-] - 1/[NO3-]0)
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2
log(time/s)

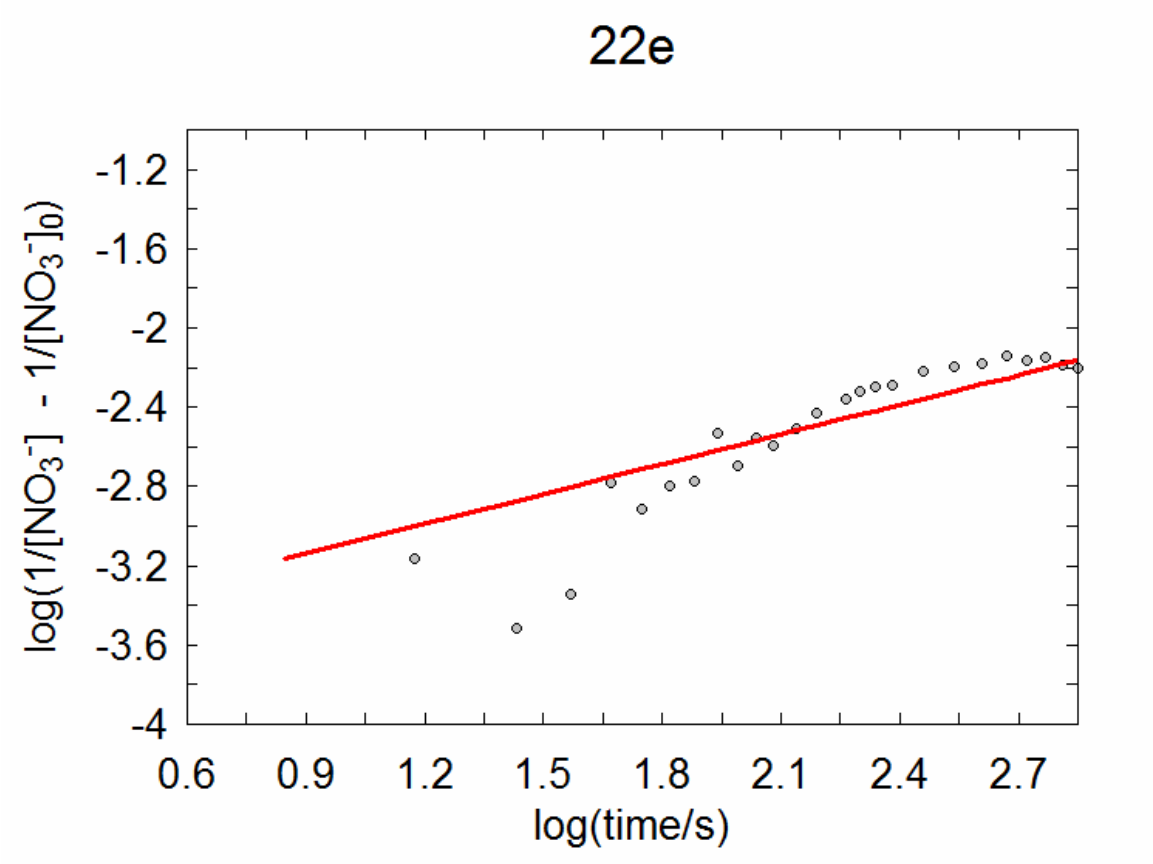

19
-1.75
-2
-2.25
-2.5
-2.75
-3
-3.25
log(1/[NO3-] - 1/[NO3-]0)
0.75
1.25
1.75
2.25
2.75
log(time/s)

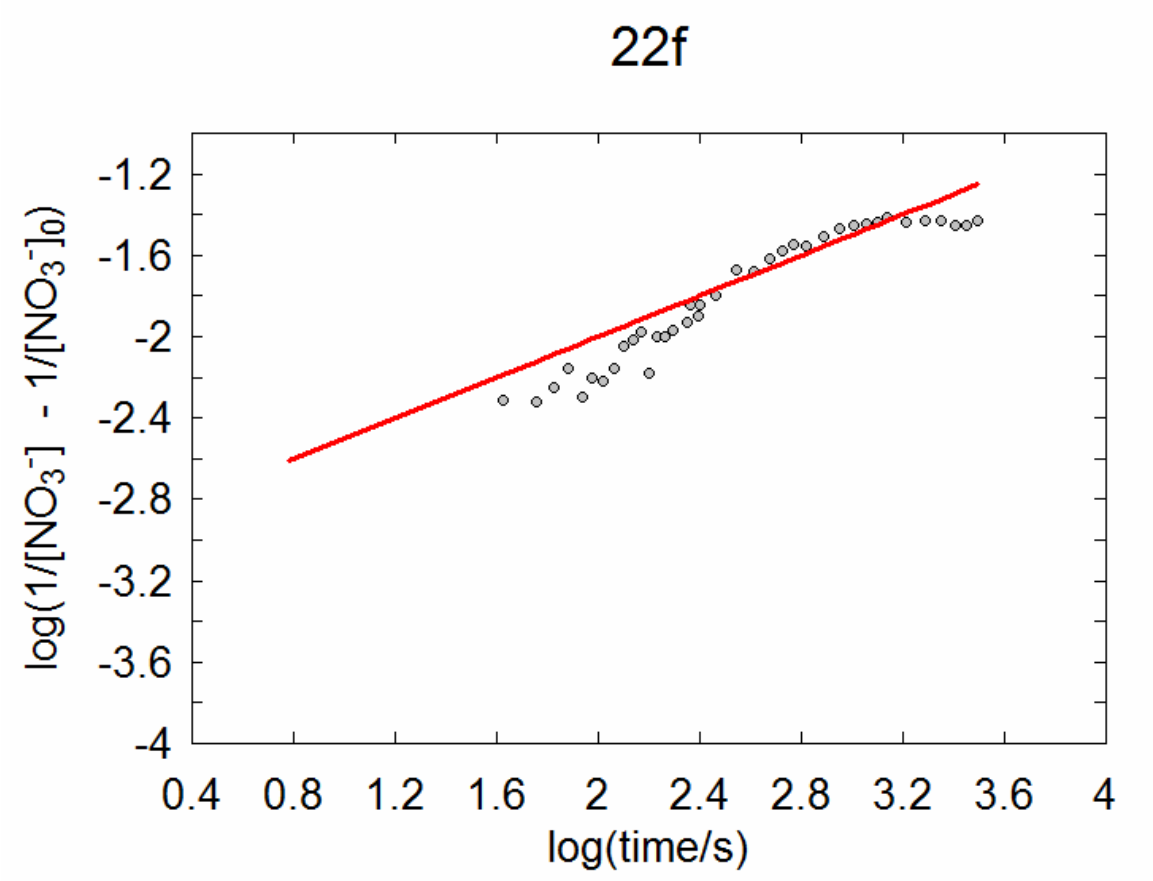

21
log(1/[NO3-] - 1/[NO3-]0)
-1.2
-1.5
-1.8
-2.1
-2.4
-2.7
-3
0.75
1.25
1.75
2.25
2.75
3.25
log(time/s)

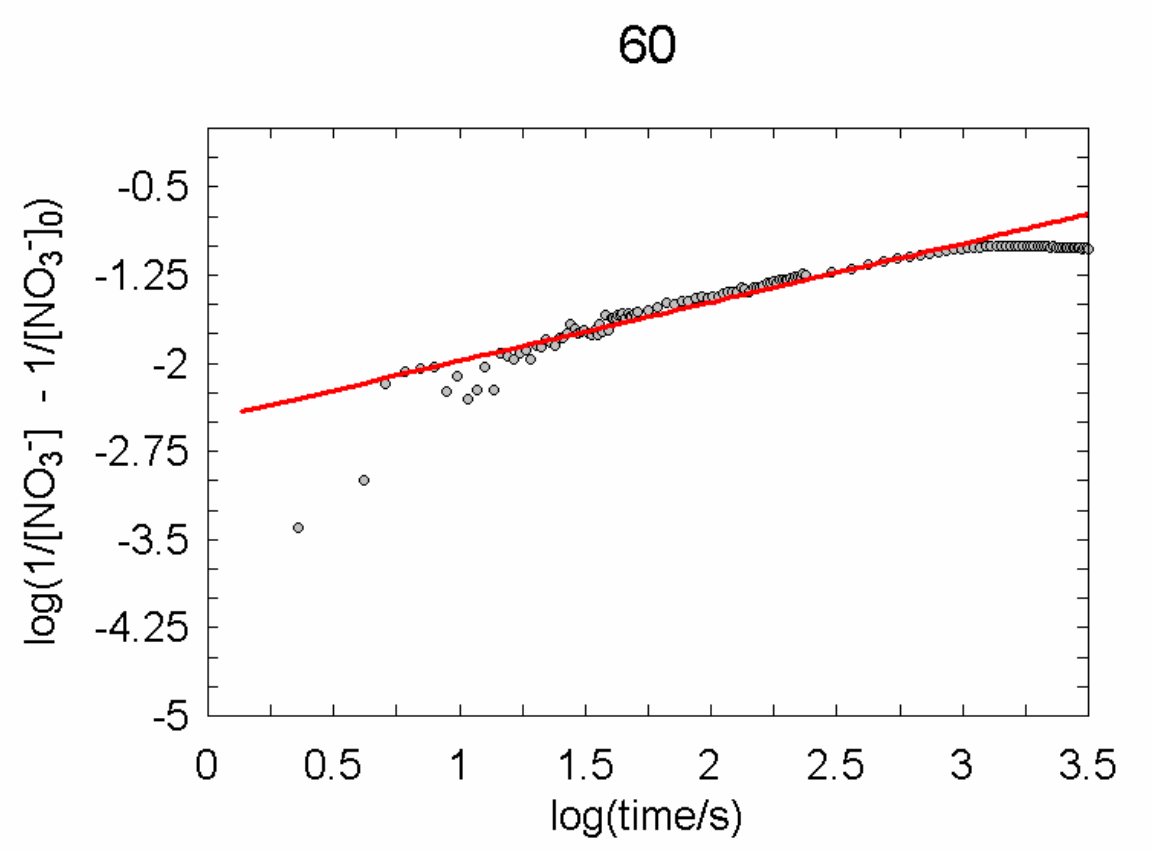

22b
-1.5
-1.75
-2
-2.25
-2.5
-2.75
log(1/[NO3-] - 1/[NO3-]0)
0.75
1.25
1.75
2.25
2.75
log(time/s)

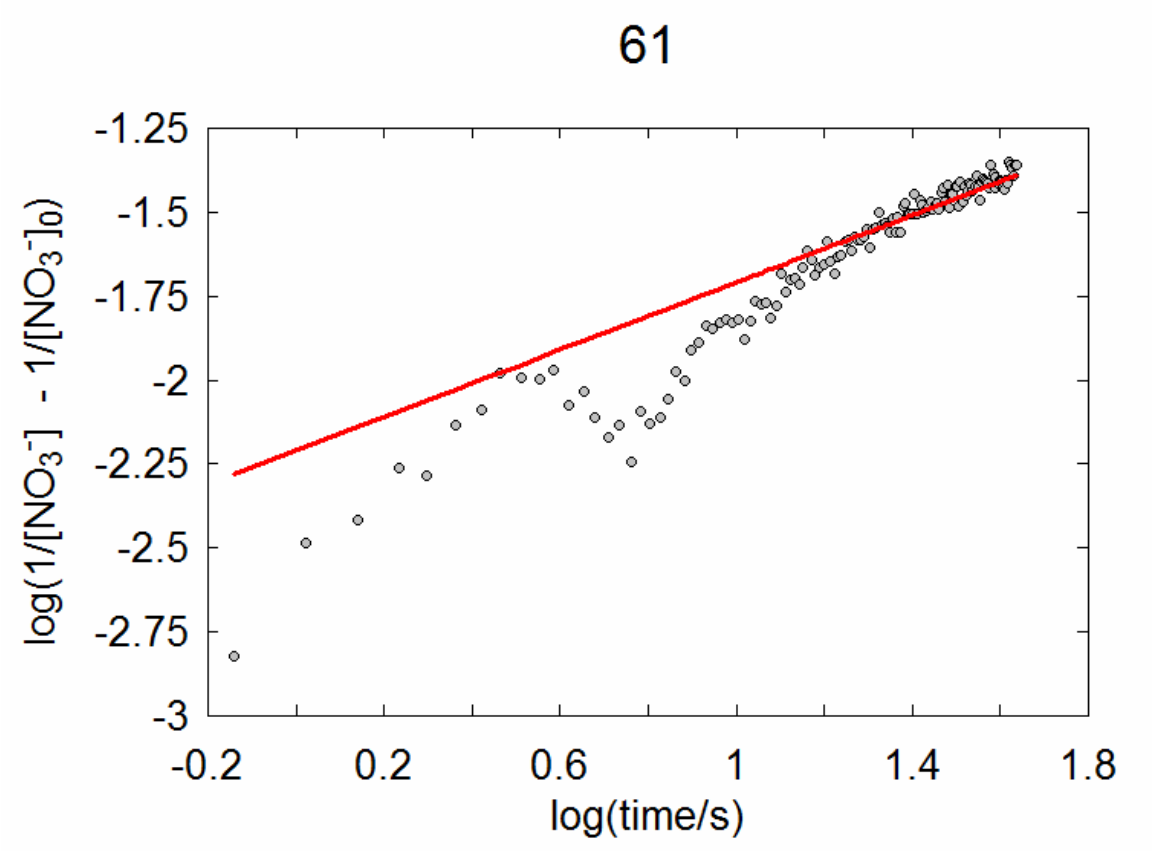

22c
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)
-2
-2.5
-3
-3.5
-4
-4.5
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2

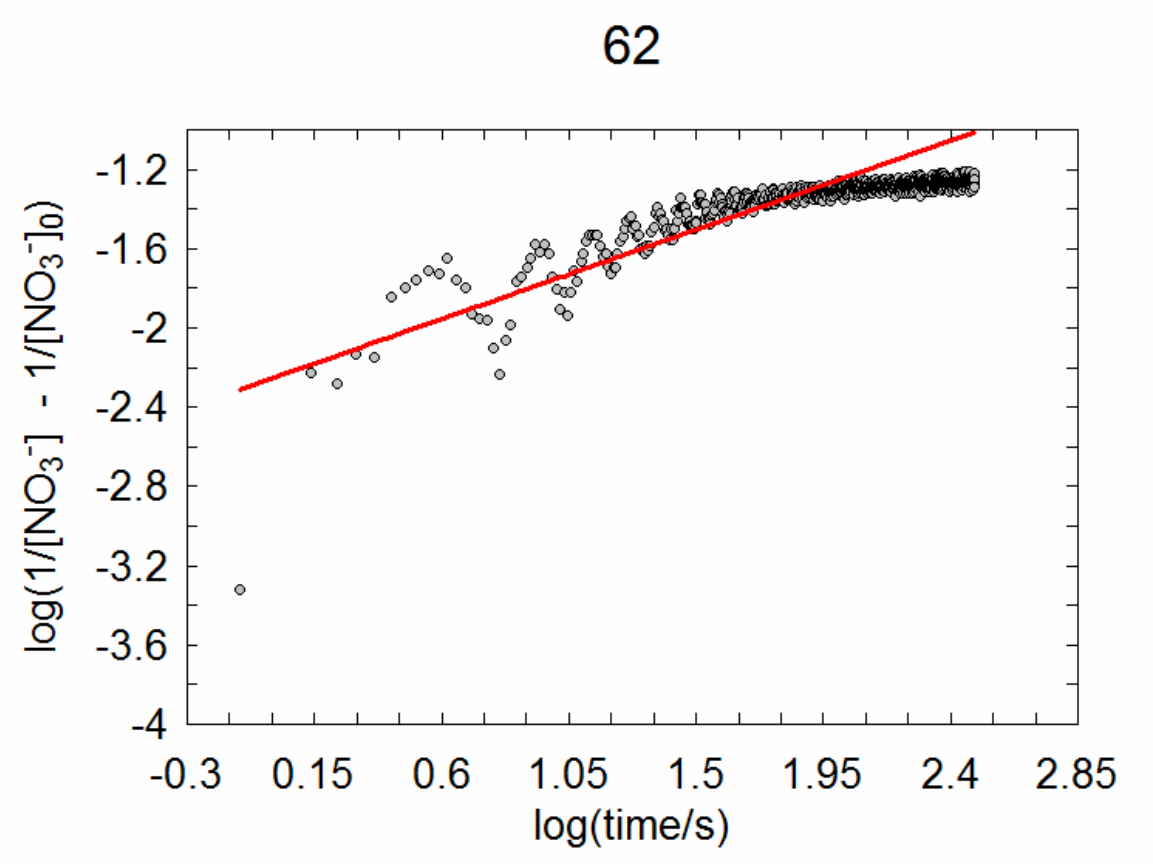

22d
-1.25
-1.5
-1.75
-2
-2.25
-2.5
-2.75
log(1/[NO3-] - 1/[NO3-]0)
0.6
0.9
1.2
1.5
1.8
2.1
2.4
2.7
3
3.3
3.6
log(time/s)

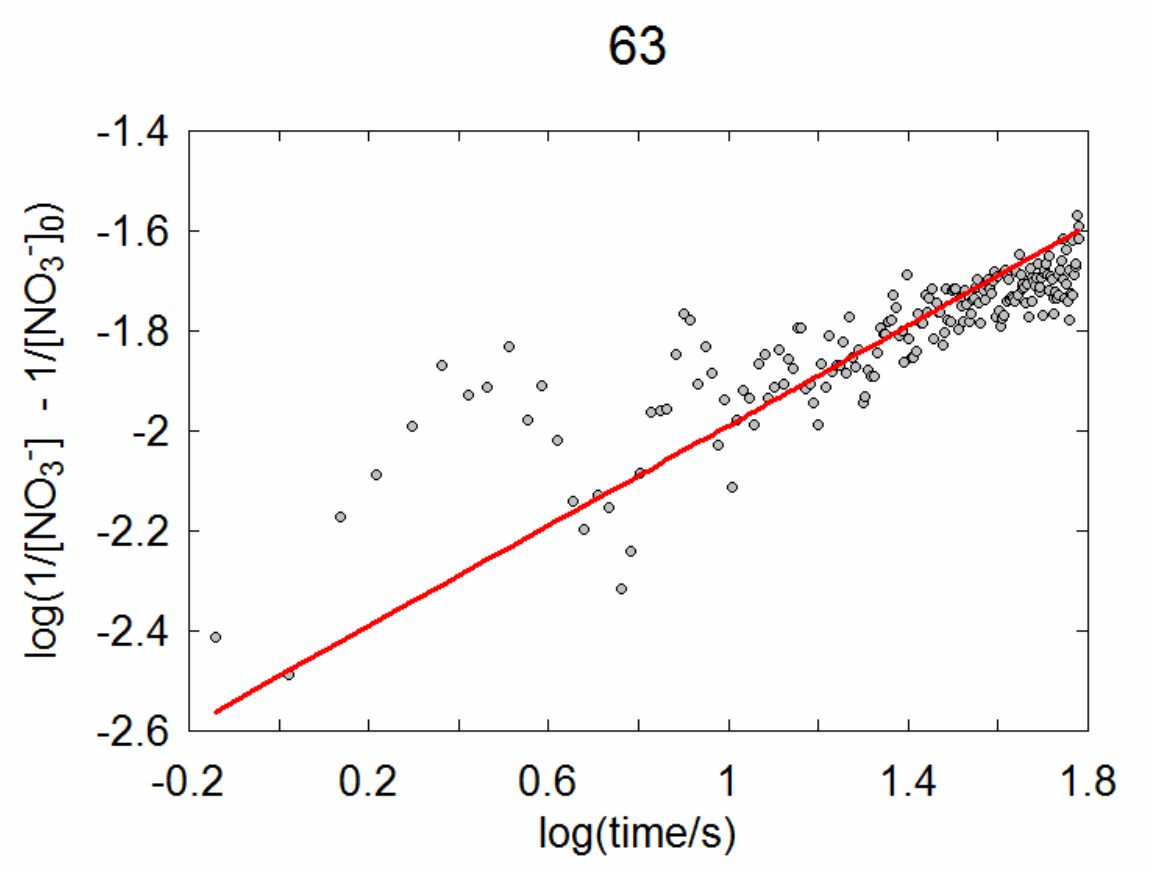

22e
log(1/[NO3-] - 1/[NO3-]0)
-1.2
-1.6
-2
-2.4
-2.8
-3.2
-3.6
-4
0.6
0.9
1.2
1.5
1.8
2.1
2.4
2.7
log(time/s)

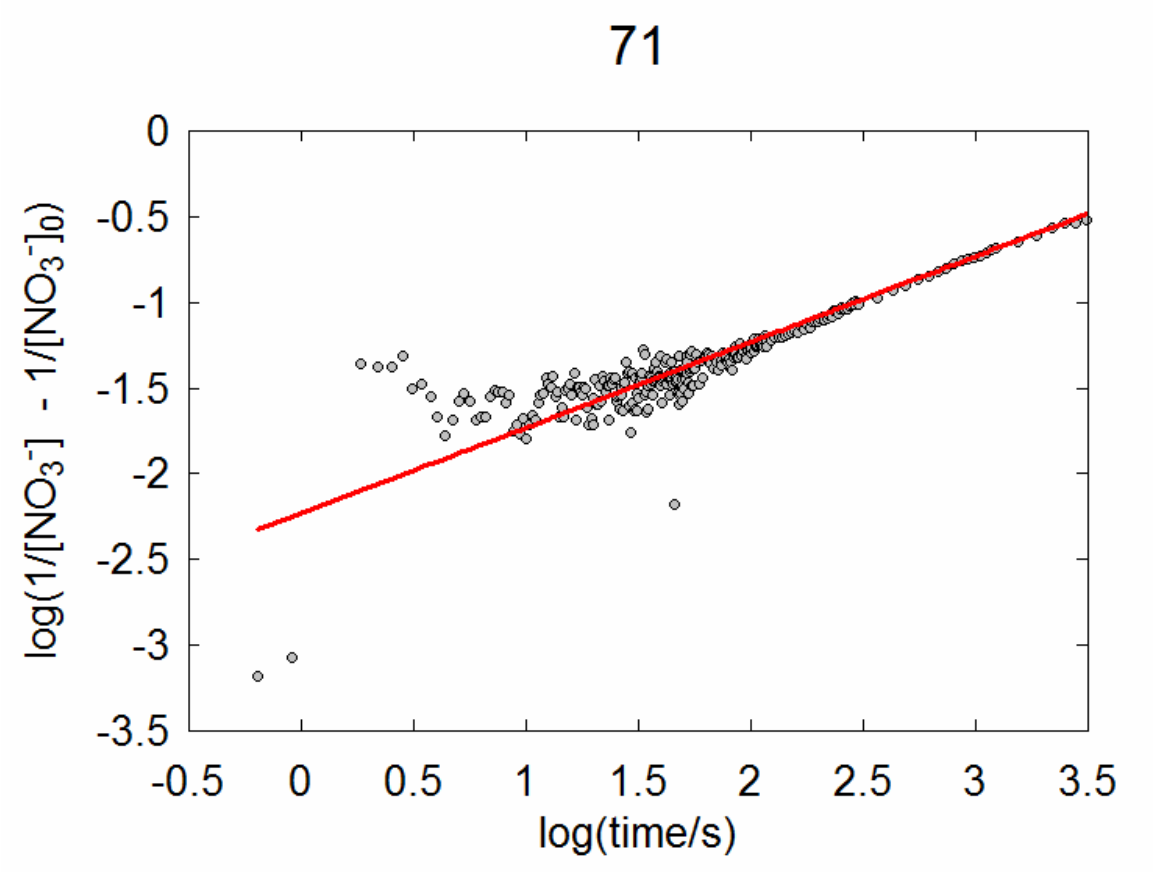

22f
log(1/[NO3-] - 1/[NO3-]0)
-1.2
-1.6
-2
-2.4
-2.8
-3.2
-3.6
-4
0.4
0.8
1.2
1.6
2
2.4
2.8
3.2
3.6
4
log(time/s)

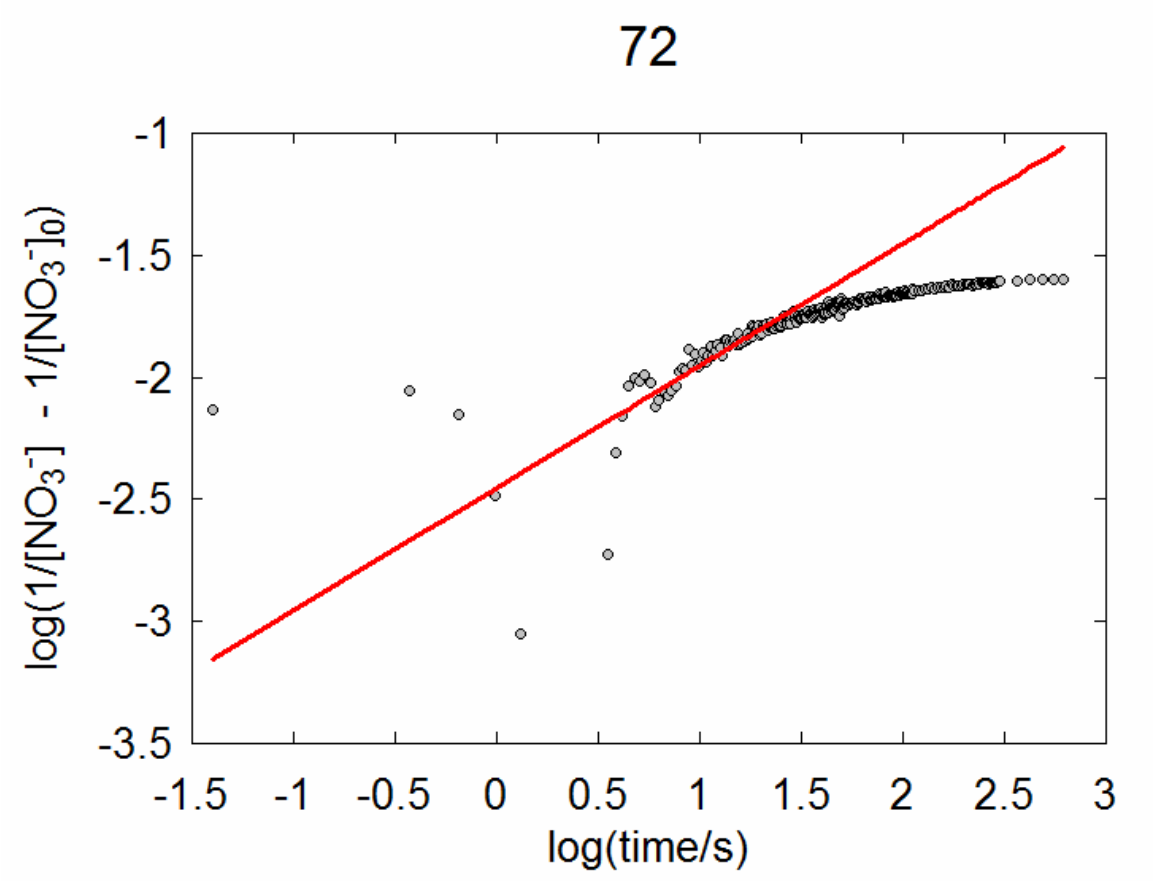

60
log(1/[NO3-] - 1/[NO3-]0)
-0.5
-1.25
-2
-2.75
-3.5
-4.25
-5
0
0.5
1
1.5
2
2.5
3
3.5
log(time/s)

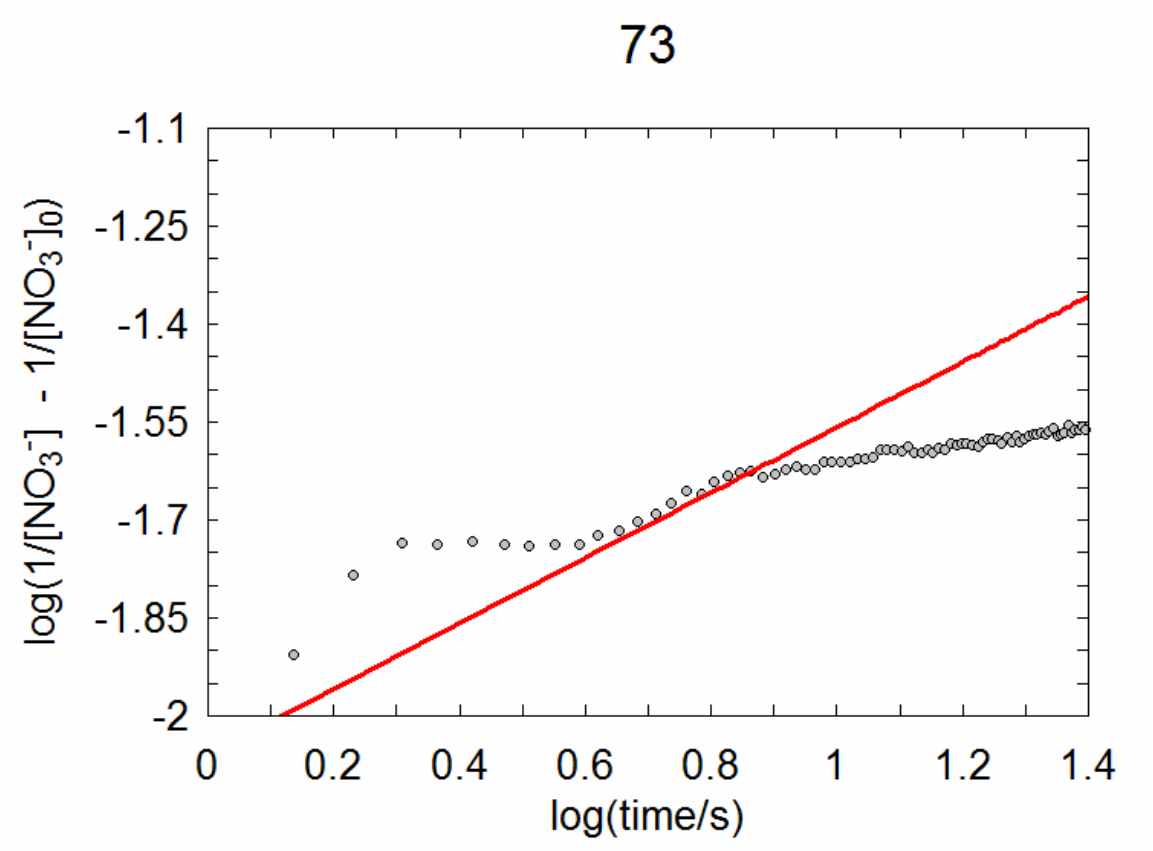

61
-1.25
-1.5
-1.75
-2
-2.25
-2.5
-2.75
-3
log(1/[NO3-] - 1/[NO3-]0)
-0.2
0.2
0.6
1
1.4
1.8
log(time/s)

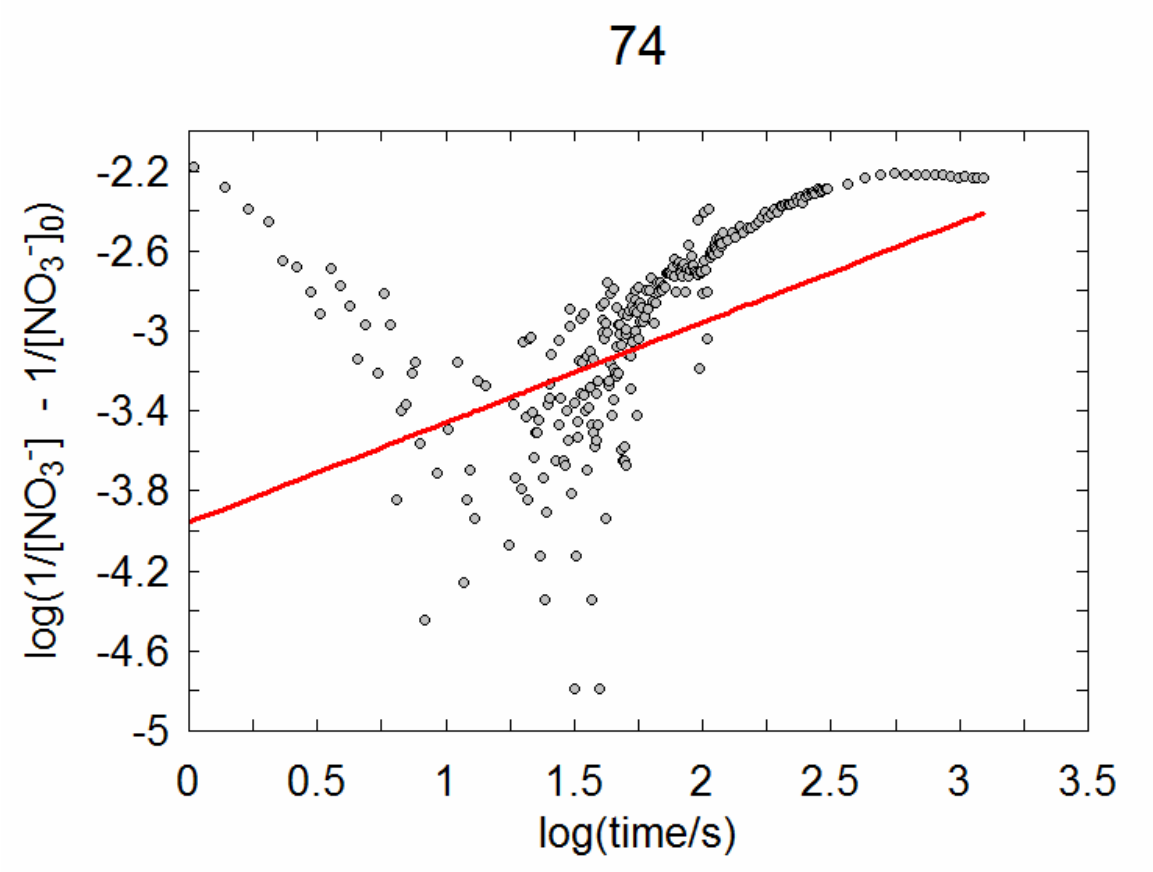

62
-1.2
-1.6
-2
-2.4
-2.8
-3.2
-3.6
-4
log(1/[NO3-] - 1/[NO3-]0)
-0.3
0.15
0.6
1.05
1.5
1.95
2.4
2.85
log(time/s)

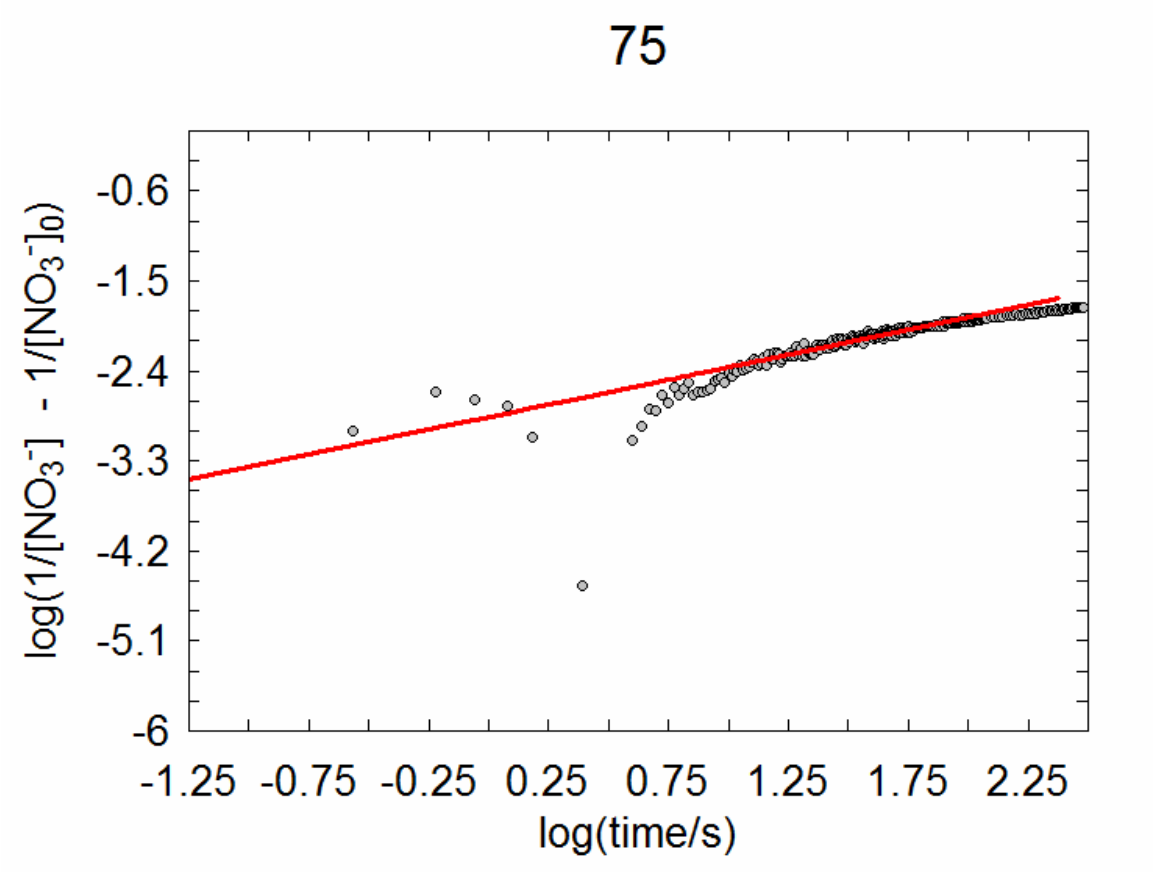

63
-1.4
-1.6
-1.8
-2
-2.2
-2.4
-2.6
log(1/[NO3-] - 1/[NO3-]0)
-0.2
0.2
0.6
1
1.4
1.8
log(time/s)

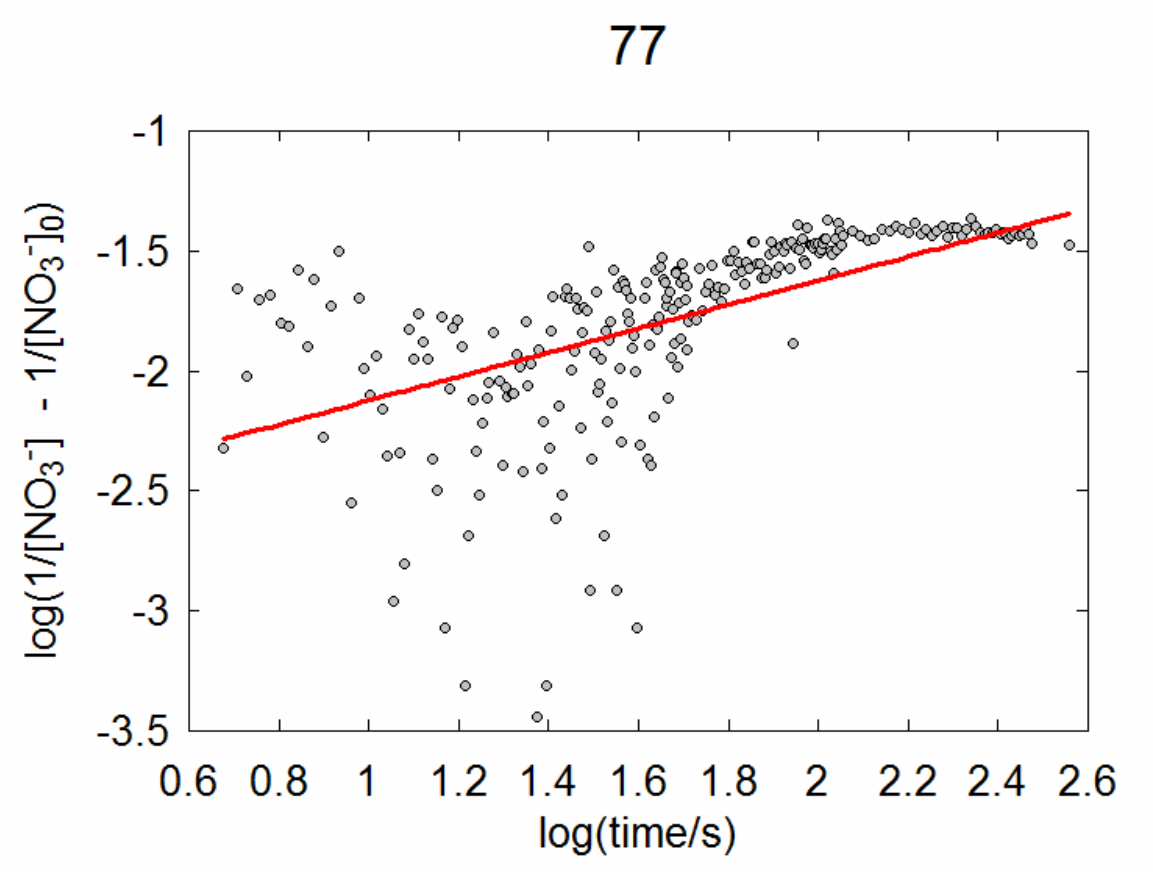

71
0
-0.5
-1
-1.5
-2
-2.5
-3
-3.5
log(1/[NO3-] - 1/[NO3-]0)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
log(time/s)

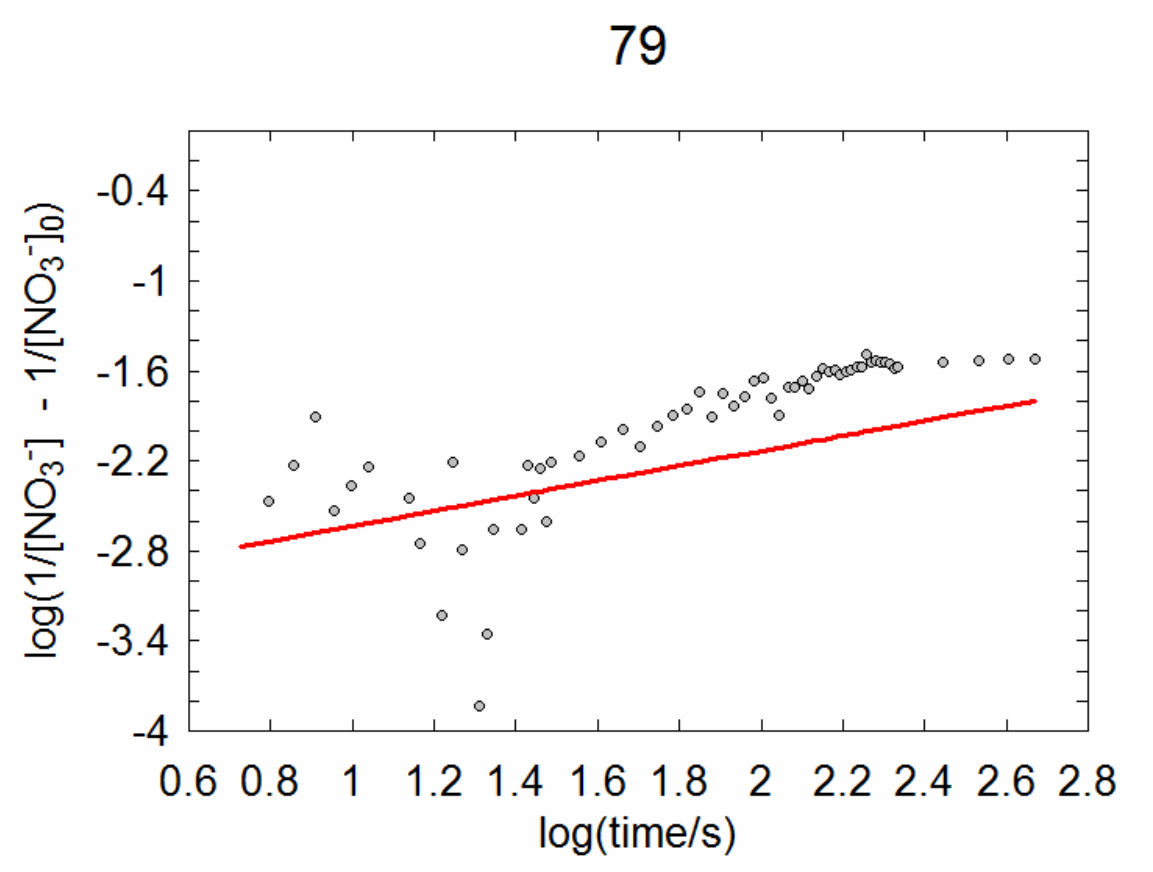

72
-1
-1.5
-2
-2.5
-3
-3.5
log(1/[NO3-] - 1/[NO3-]0)
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
3
log(time/s)

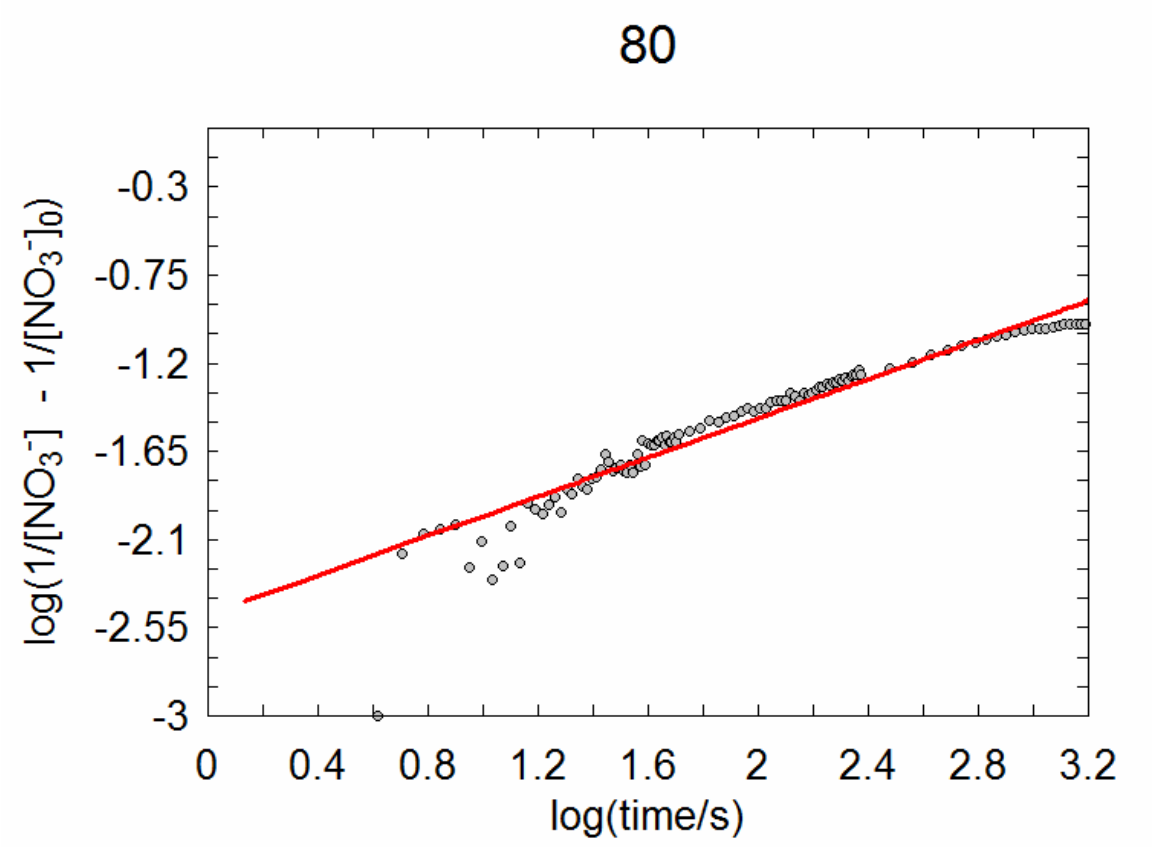

73
-1.1
-1.25
-1.4
-1.55
-1.7
-1.85
-2
log(1/[NO3-] - 1/[NO3-]0)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
log(time/s)

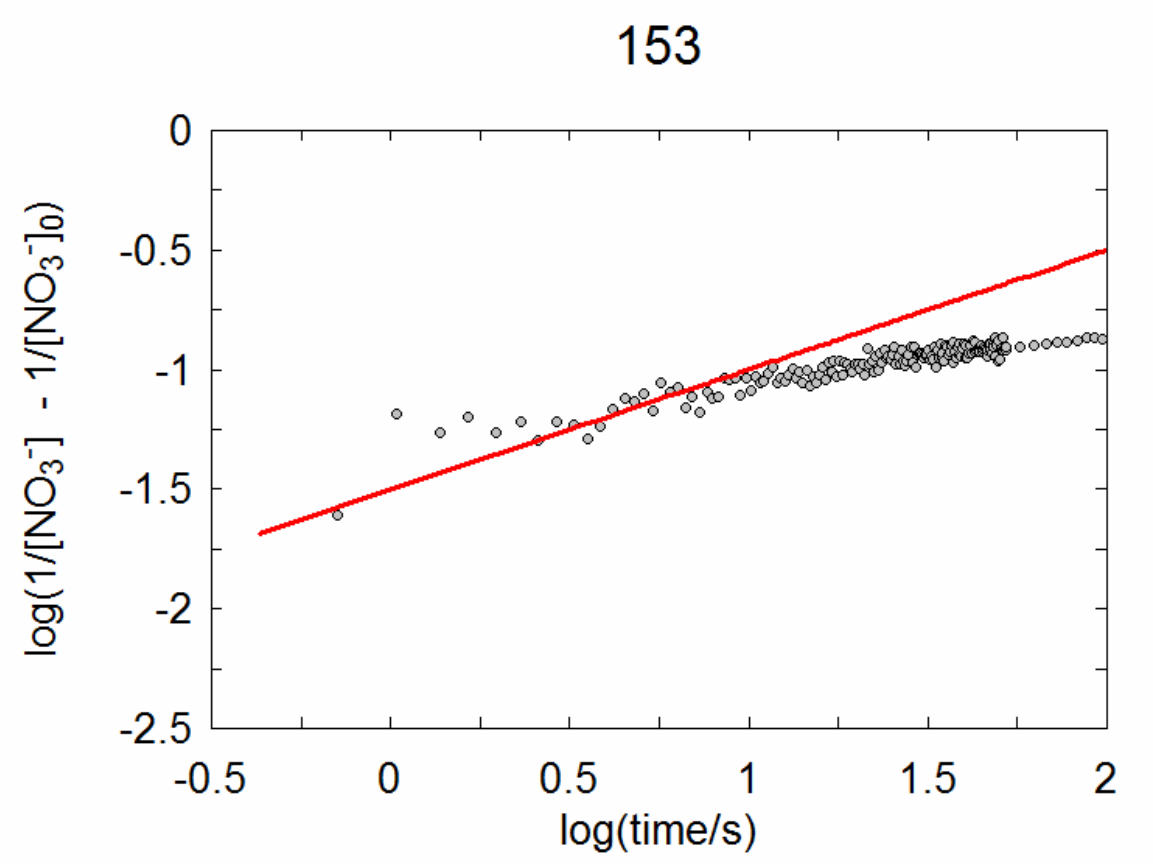

74
-2.2
-2.6
-3
-3.4
-3.8
-4.2
-4.6
-5
log(1/[NO3-] - 1/[NO3-]0)
0
0.5
1
1.5
2
2.5
3
3.5
log(time/s)

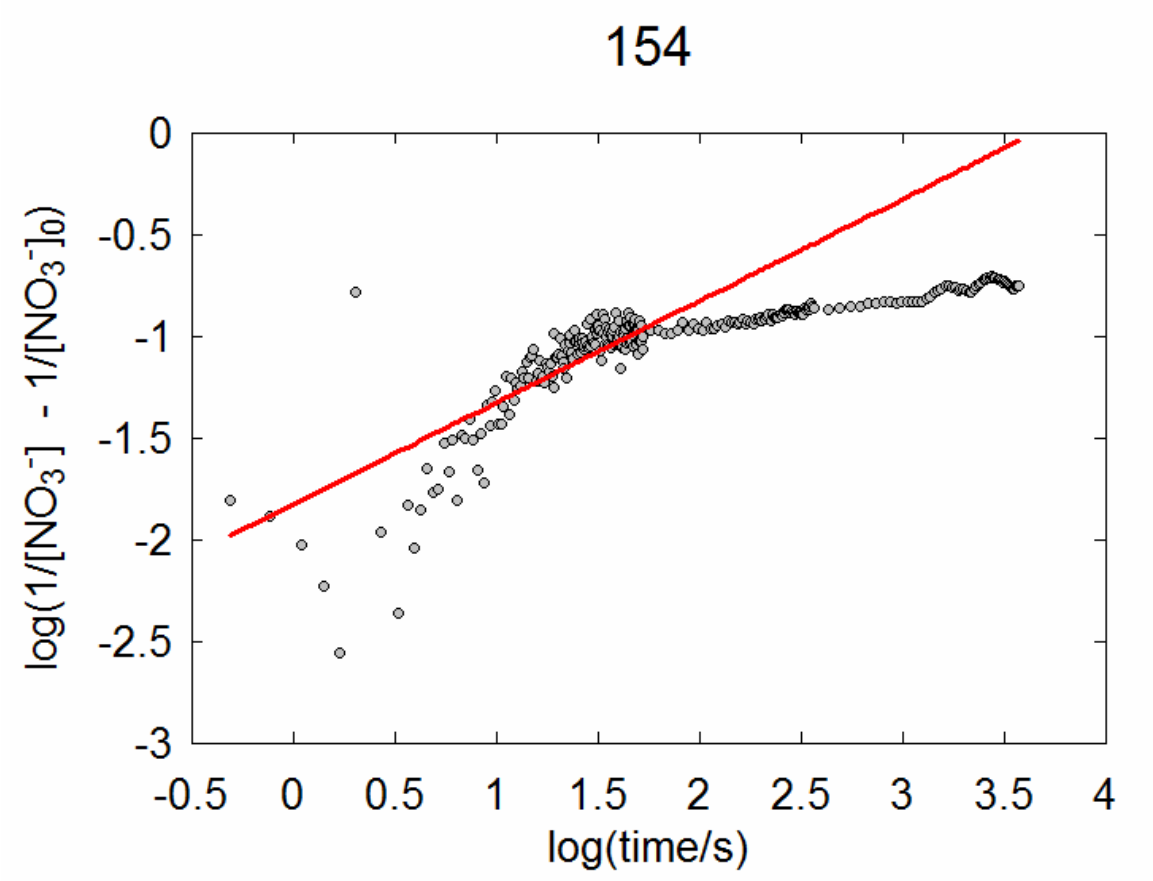

75
-0.6
-1.5
-2.4
-3.3
-4.2
-5.1
-6
log(1/[NO3-] - 1/[NO3-]0)
-1.25
-0.75
-0.25
0.25
0.75
1.25
1.75
2.25
log(time/s)

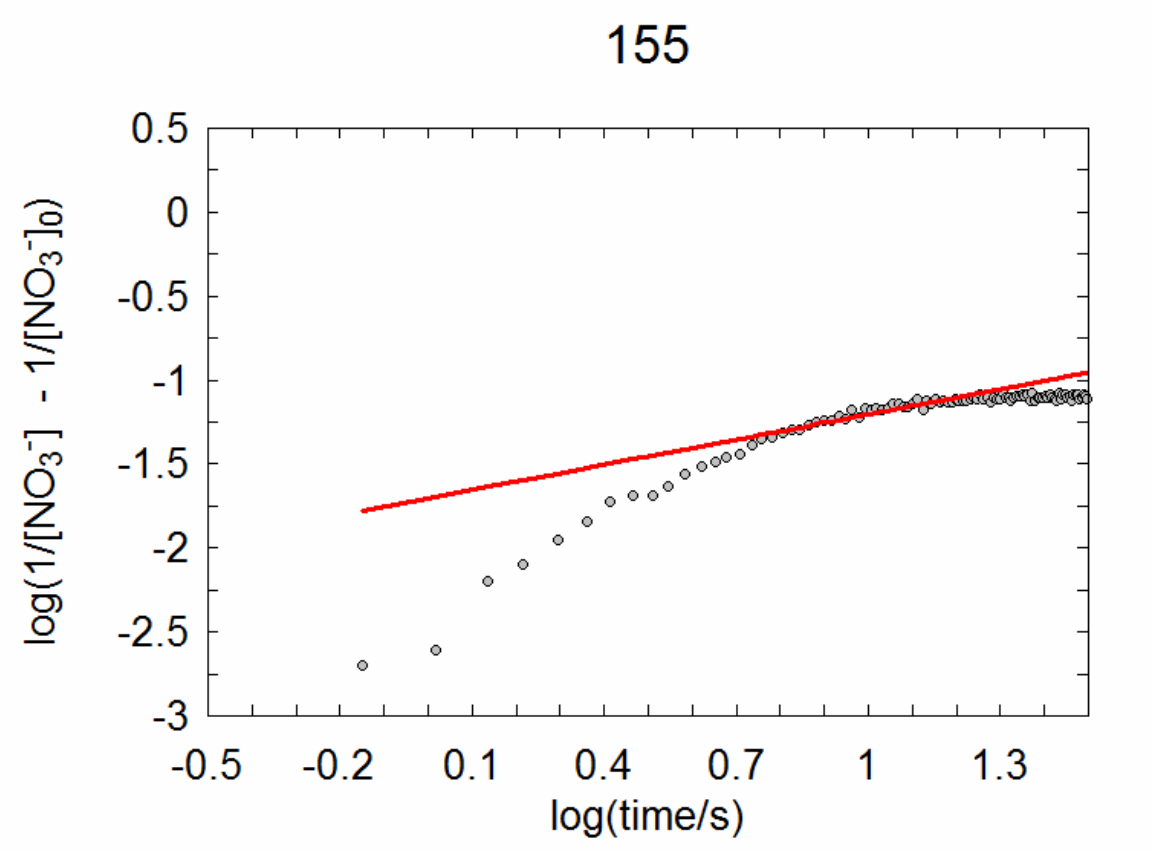

77
log(1/[NO3-] - 1/[NO3-]0)
-1
-1.5
-2
-2.5
-3
-3.5
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
log(time/s)

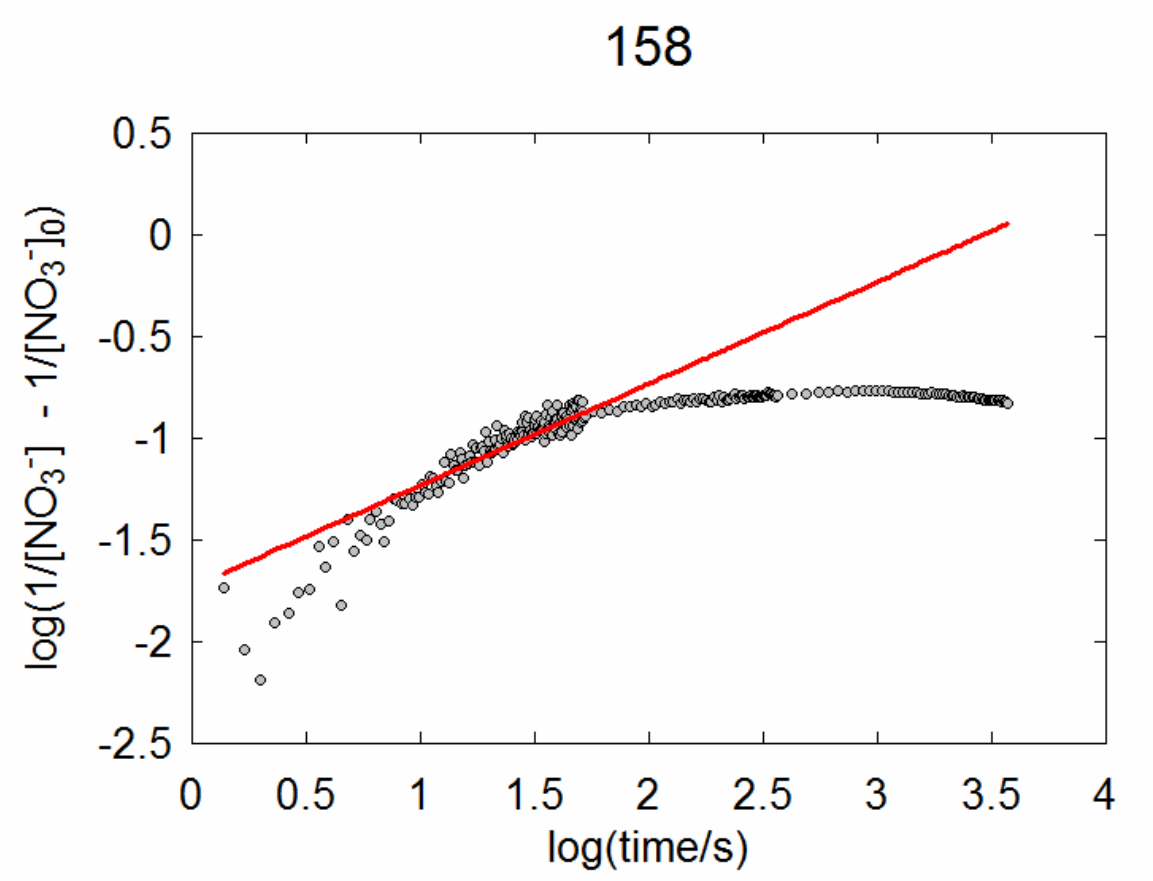

79
log(1/[NO3-] - 1/[NO3-]0)
-0.4
-1
-1.6
-2.2
-2.8
-3.4
-4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
log(time/s)

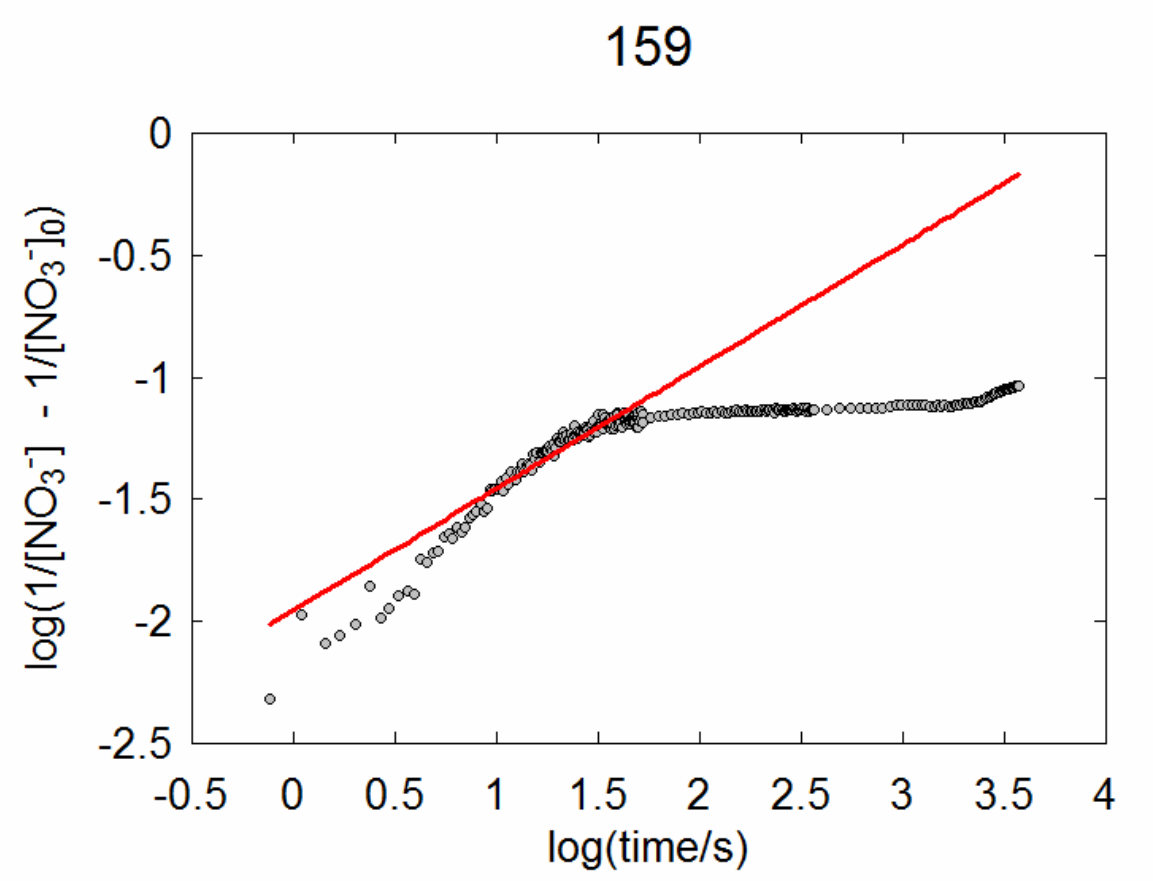

80
log(1/[NO3-] - 1/[NO3-]0)
-0.3
-0.75
-1.2
-1.65
-2.1
-2.55
-3
0
0.4
0.8
1.2
1.6
2
2.4
2.8
3.2
log(time/s)

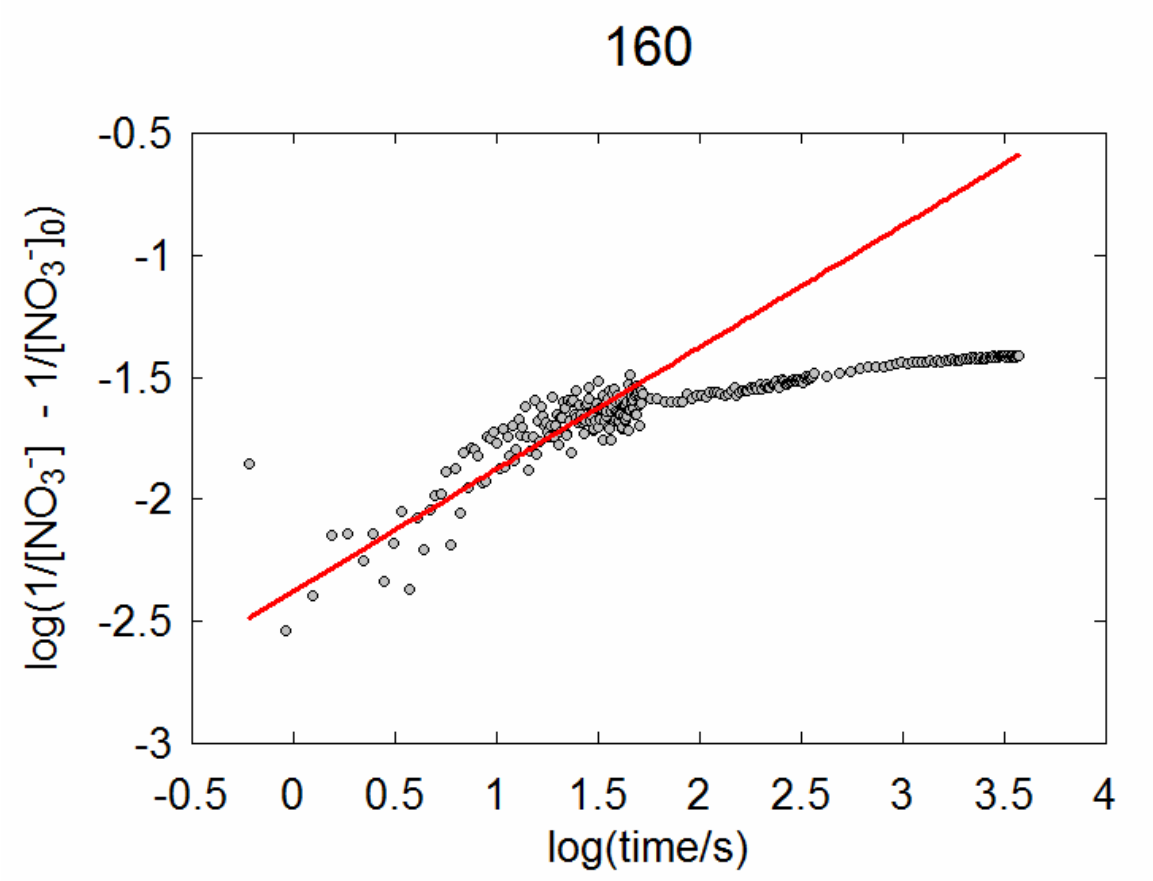

153
0
-0.5
-1
-1.5
-2
-2.5
-0.5
0
0.5
1
1.5
2
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)

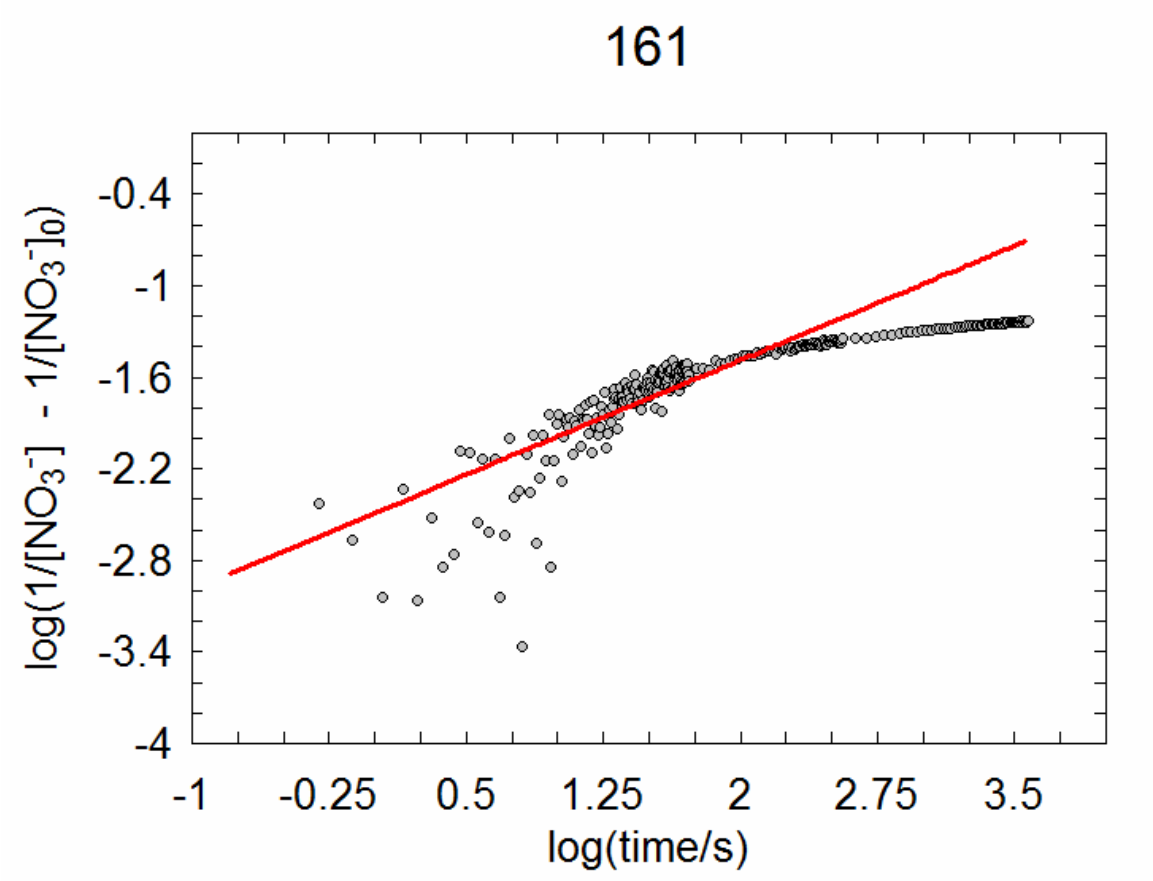

154
0
-0.5
-1
-1.5
-2
-2.5
-3
log(1/[NO3-] - 1/[NO3-]0)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(time/s)

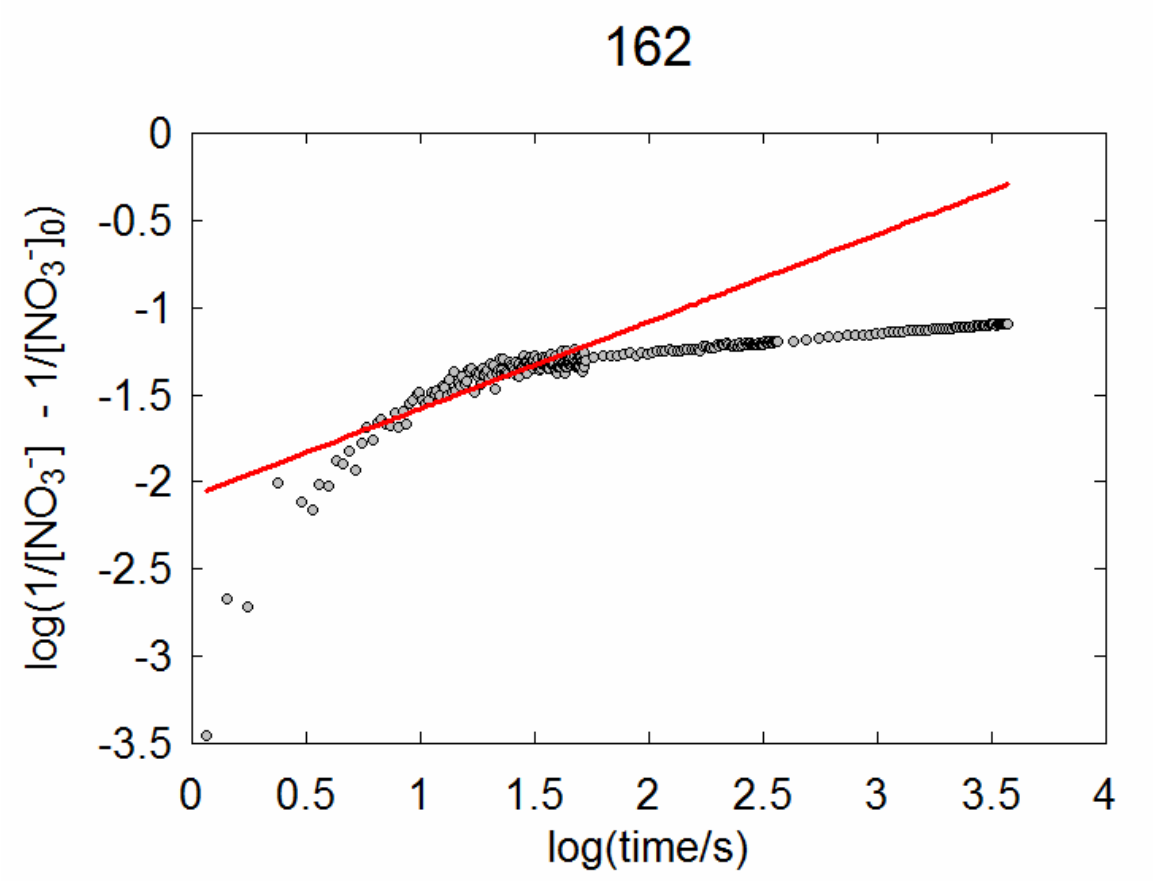

155
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
-0.5
-0.2
0.1
0.4
0.7
1
1.3
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)

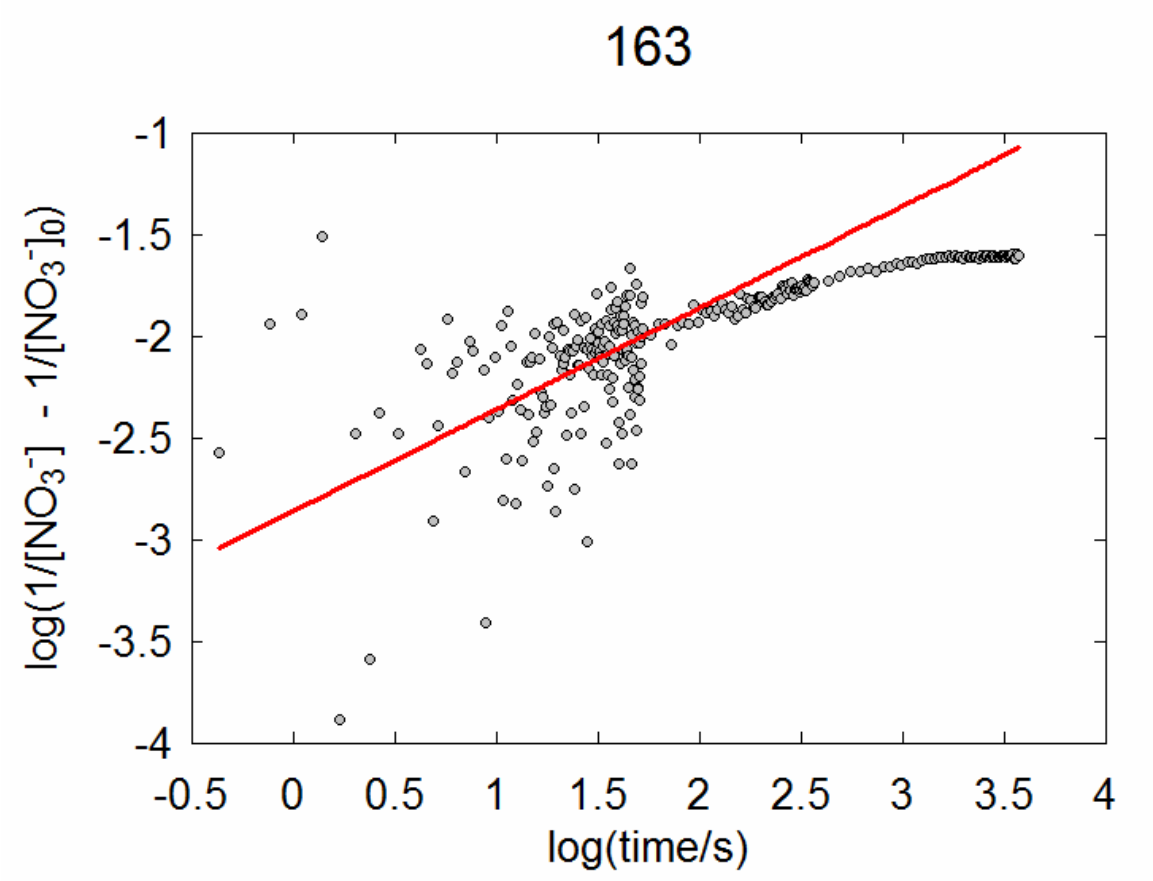

158
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)

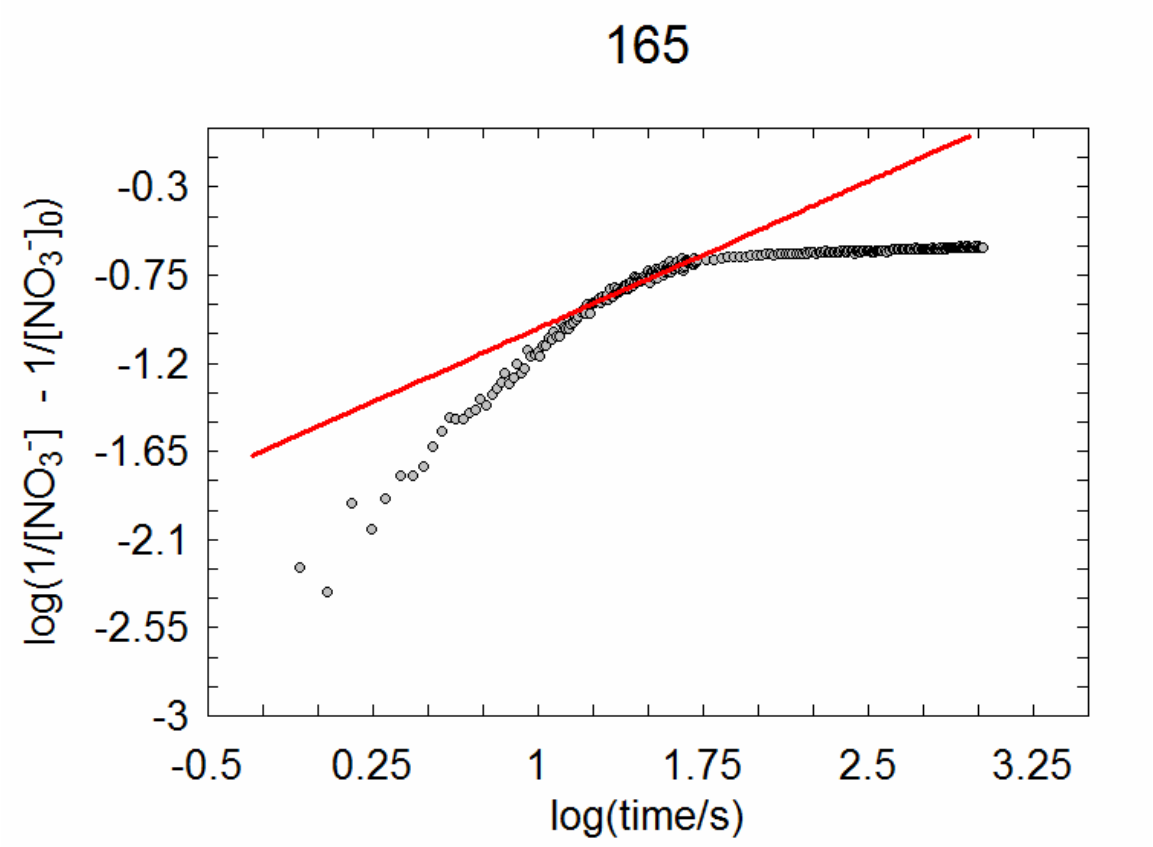

159
0
-0.5
-1
-1.5
-2
-2.5
log(1/[NO3-] - 1/[NO3-]0)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(time/s)

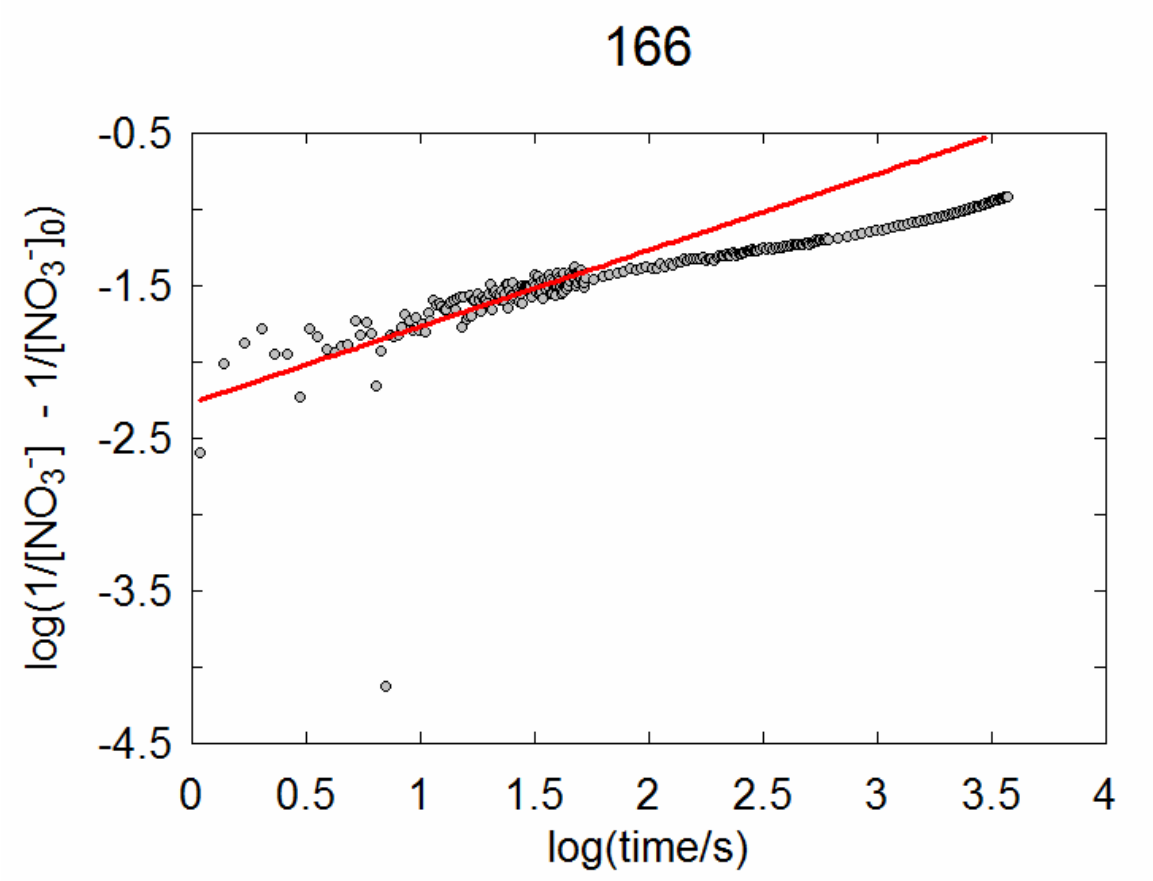

160
-0.5
-1
-1.5
-2
-2.5
-3
log(1/[NO3-] - 1/[NO3-]0)
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(time/s)

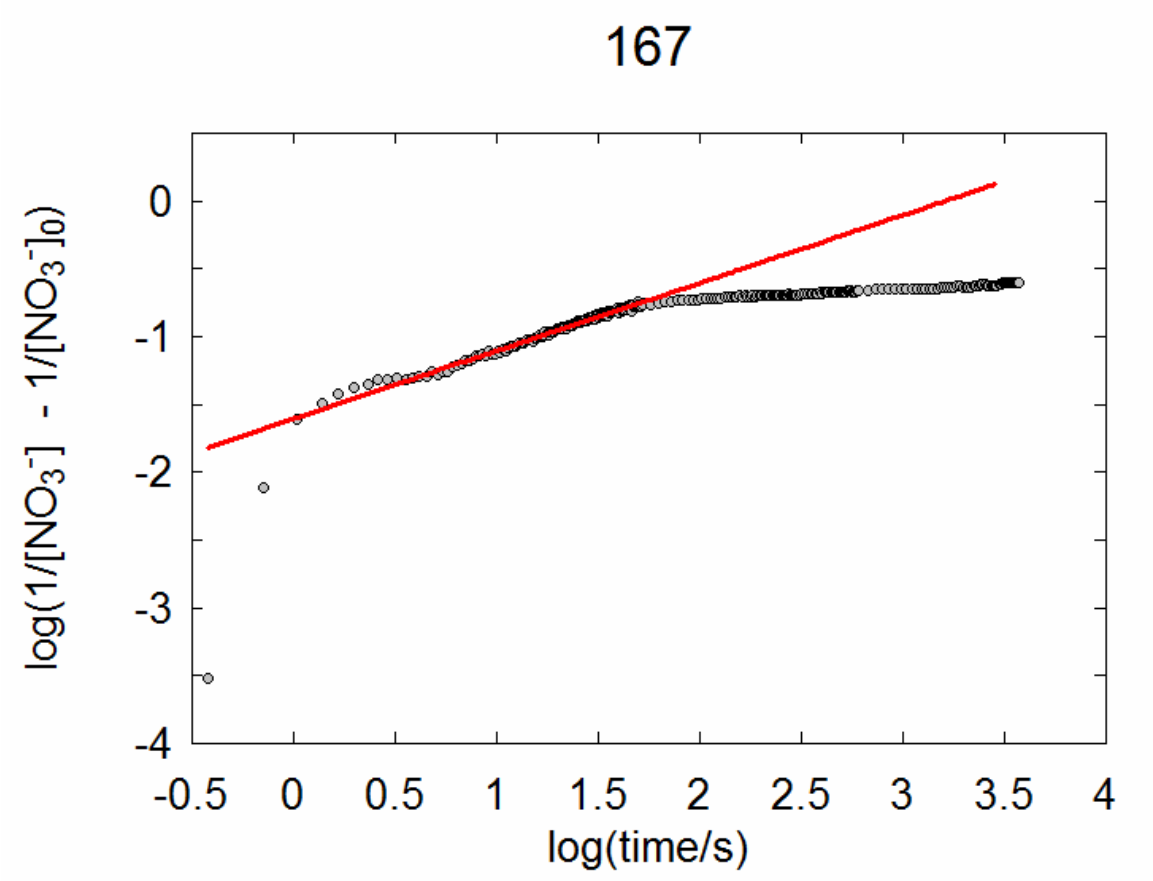

161
log(1/[NO3-] - 1/[NO3-]0)
-0.4
-1
-1.6
-2.2
-2.8
-3.4
-4
-1
-0.25
0.5
1.25
2
2.75
3.5
log(time/s)

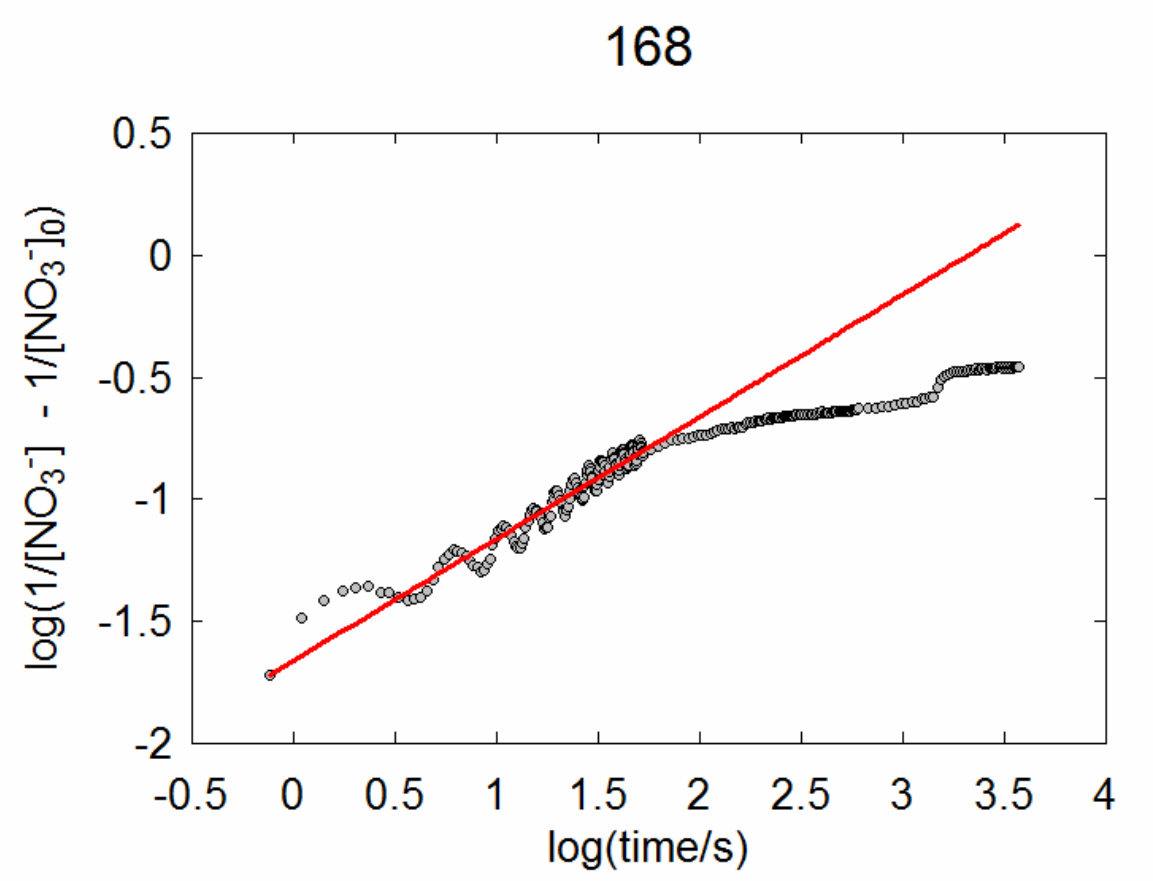

162
0
-0.5
-1
-1.5
-2
-2.5
-3
-3.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)


163
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)
-1
-1.5
-2
-2.5
-3
-3.5
-4
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4


165
-0.3
-0.75
-1.2
-1.65
-2.1
-2.55
-3
log(1/[NO3-] - 1/[NO3-]0)
-0.5
0.25
1
1.75
2.5
3.25
log(time/s)


166
-0.5
-1.5
-2.5
-3.5
-4.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)


167
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)
0
-1
-2
-3
-4
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4


168
0.5
0
-0.5
-1
-1.5
-2
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
log(1/[NO3-] - 1/[NO3-]0)
log(time/s)