Environmental Radioactivity

Physics Laboratory 3

Visva Loganathan vloganathan@student.ethz.ch

Teaching Assistant

Dr. Miriam Mindová mmindova@phys.ethz.ch

Abstract Environmental radioactivity was investigated with a focus on ^{222}Rn and its short-lived progeny using three Ranger Radiation Alert detectors. After background measurements, the detectors were calibrated with a 1.9 g KCl sample, yielding consistent calibration factors of about 11–12 Bq/cps.

Radon-related activity was then measured in water and air samples. The water sample showed only a weak excess above background, corresponding to $C_0 = 0.29 \pm 0.13\,\mathrm{Bq\,L^{-1}}$, so that no reliable half-life could be determined. The air sample gave a clearer signal with $C_0 = 1.14 \pm 0.09\,\mathrm{Bq\,m^{-3}}$ and an estimated annual dose of $8.60 \pm 0.69\,\mu\mathrm{Sv\,yr^{-1}}$, indicating very low radon exposure in the measured basement environment. Its decay behavior was consistent with the expected order of magnitude of short-lived radon daughters.

An ingrowth measurement on activated carbon showed the buildup of daughter activity toward secular equilibrium, with an effective half-life of 30.74 ± 0.88 min. Overall, the experiment successfully demonstrated radioactive decay, ingrowth, and secular equilibrium in environmental radioactivity measurements, with the air sample providing the most reliable environmental result.

ETH Zürich, April 24, 2026

Visva Loganathan

1 Introduction

Environmental radioactivity originates mainly from naturally occurring radionuclides present in the Earth's crust, atmosphere, and hydrosphere. Among the most important contributors are ^{40}K and the radionuclides belonging to the primordial decay series of ^{238}U, ^{235}U, and ^{232}Th. These nuclides have existed since the formation of the Earth and continue to decay through long chains of radioactive daughters until stable lead isotopes are reached. As a result, natural radioactivity is a permanent component of the human radiation environment [3].

A particularly important radionuclide in environmental physics and radiochemistry is radon. Radon is a noble gas produced in the decay chains of uranium and thorium, which allows it to escape from soil, rocks, building materials, and water. Because of its gaseous nature, it can migrate through pore spaces and accumulate in enclosed or poorly ventilated areas such as basements, caves, and underground rooms. For this reason, radon is responsible for a large fraction of the average public exposure to ionizing radiation. Among the naturally occurring radon isotopes, ^{222}Rn is the most relevant for environmental measurements because it originates from the ^{238}U decay series and has a comparatively long half-life of 3.8 days.

Although ^{222}Rn itself decays by α emission and could in principle be measured directly with a suitable alpha-detection method, the present experimental setup mainly detects the β-emitting short-lived daughter nuclides, especially ^{214}Pb and ^{214}Bi. These daughter products are therefore well suited for indirect radon measurements with low-level radiation detectors. In addition, the time evolution of parent and daughter nuclides illustrates important concepts of nuclear physics such as radioactive decay, ingrowth, and secular equilibrium. In the case of ^{222}Rn and its progeny, the daughter activity builds up over time and approaches equilibrium with the parent activity after several daughter half-lives.

In this experiment, environmental radioactivity was investigated using a Ranger Radiation Alert detector equipped with a Geiger–Mueller tube. Background radiation was first measured in order to determine the detector baseline. The detector was then calibrated using a potassium chloride sample of known ^{40}K activity. Afterward, radon-related activity was measured in different environmental samples, including air, water, and rock-related samples. Special attention was paid to the ingrowth and decay behavior of radon progeny in order to identify radionuclides from their characteristic half-lives and to study the establishment of secular equilibrium.

The aim of this experiment was to measure environmental radioactivity with a focus on ^{222}Rn and its daughter nuclides, to apply the concepts of radioactive decay and ingrowth to real measurements, and to estimate radon activity in different environmental samples from experimentally observed count rates.

2 Experiment

2.1 Detector and data acquisition

All measurements were performed with a *Ranger Radiation Alert* detector equipped with a Geiger–Mueller tube for low-level radiation measurements. The instrument is capable of detecting α, β, γ, and X-ray radiation, however, under the present experimental conditions it was used mainly as a β detector. In sealed samples, α particles were absorbed by the sample covering, while in air and water measurements the collected activity was dominated by the β-emitting radon progeny, in particular ^{214}Pb and ^{214}Bi. The detector output was recorded in counts per second (cps) using the *Radiation Alert Observer USB* software.

For each measurement, the detector was placed on the sample holder such that the sample faced the detector window without touching it. The Ranger was connected to a computer via USB, and the count rate was logged continuously. The data were saved as plain text files. Only the first two columns, corresponding to time stamp and counts recorded in one second, were used for further analysis.

2.2 Background measurements

Background radiation was measured first in order to determine the detector baseline. For this purpose, the Ranger was placed on an empty sample holder and several measurements of different duration were recorded. Following the recommendation of the manual, measurement times between approximately 10 min and 5 h were used. The purpose of these runs was to compare the statistical fluctuations observed in short and long measurements and to determine a representative background count rate.

The measured background count rate was subtracted from all subsequent measurements, including the detector calibration. Depending on the duration of the measurement, the raw data were rebinned into time intervals of 1 min and 1 h in order to reduce the effect of statistical fluctuations.

2.3 Efficiency calibration

The detector was calibrated using a potassium chloride sample, since natural potassium contains the radioactive isotope ^{40}K. For the calibration, 1.9 g of KCl were ground, weighed, and distributed as homogeneously as possible over an area corresponding approximately to the detector window. The sample was then sealed on both sides with adhesive film in order to avoid contamination of the detector.

The KCl sample was counted for 60 min. A binning interval of 5 min was used for the evaluation. From the known activity A of the calibration sample and the measured net count rate $\mathrm{cps}_{\mathrm{net}}$, the calibration factor k was obtained as

$$k = \frac{A}{\mathrm{cps}_{\mathrm{net}}}, \tag{1}$$

where $\mathrm{cps}_{\mathrm{net}}$ is the background-corrected count rate. The detector efficiency was then defined as

$$\varepsilon = \frac{1}{k}. \tag{2}$$

This calibration factor was later used to convert measured count rates of environmental samples into activity values.

2.4 Measurement of radon in environmental samples

2.4.1 Water sample

Radon activity in water was determined by filtering approximately 2.0 L of water through a fiberglass membrane mounted on a porcelain funnel. After filtration, the membrane was dried on a hotplate for about 5 min and then placed in the sample holder below the detector. Since the expected activity was relatively low and a rapid decay of short-lived radon progeny was anticipated, the sample was measured for less than 5 h and analyzed using binning intervals of 5 min.

2.4.2 Air sample

For the air measurement, approximately 25 m^3 of air should be pumped through a glass fiber filter using a dust-collecting pump. The initial and final pumped volumes were recorded to determine the total sampled air volume. However, the total sampled air volume was determined to be only $3.96\,\mathrm{m}^3$, the meter scale had initially been misread. To maximize radon collection, air was preferentially sampled in a poorly ventilated indoor environment, the basement of the HPP building on the ETH Hönggerberg campus. After sampling, the filter was placed in the sample holder and measured for about 10 h. Depending on the count rate evolution, binning times of 10 min were used.

The measured count rates from both water and air samples were converted into activities using the calibration factor determined from the KCl measurement.

2.5 Ingrowth and decay of radon progeny

The ingrowth and decay behavior of radon daughters was studied using activated carbon exposed to radon emanating from radioactive source materials. The purpose of this part of the experiment was to observe the buildup and decay of the short-lived progeny of $^{222}\mathrm{Rn}$ and to identify characteristic half-lives from the measured activity curves.

For the ingrowth measurement, activated carbon was placed in a glass beaker and attached to adhesive film. The activated carbon was then rapidly exposed to the sample container such that the carbon faced the source and adsorbed emanated radon for 5 min. After exposure, the carbon was sealed with a second adhesive film and placed in the sample holder. The activity was measured for approximately 5 h, and the data were rebinned into intervals of 5 or 10 min. The resulting ingrowth curve was later fitted to extract the characteristic buildup of the daughter activity.

For the decay measurement, the same procedure was used, but the activated carbon was exposed to the radon source for a longer period of 2 h in order to collect a larger amount of ^{222}Rn. The sealed sample was then measured over about three days, using a binning interval of 1 h. Because the software is known to show a gradual increase in the time interval between recorded rows during very long measurements, the multi-day decay data were expected to be more reliable for a qualitative than for a fully quantitative half-life determination.

2.6 Data treatment

The raw detector output consisted of one-second count data stored in text format. Because radioactive decay is inherently stochastic, the count rate fluctuated strongly on short time scales. To reduce statistical scatter, the one-second data were grouped into fixed time bins. For each bin, the total number of counts was first calculated,

$$N_{\mathrm{bin}} = \sum_i n_i,$$

with Poisson uncertainty

$$\sigma_{N_{\mathrm{bin}}} = \sqrt{N_{\mathrm{bin}}}.$$

The binned count rate was then obtained as

$$\mathrm{cps}_{\mathrm{bin}} = \frac{N_{\mathrm{bin}}}{\Delta t}.$$

Thus, each bin was treated as one independent measurement point of the decay or ingrowth curve.

Different bin widths were used depending on the measurement type and the expected rate of change of the activity. Shorter bin widths were chosen for rapidly varying samples, such as water and air filters, whereas longer bin widths were applied to slowly varying measurements such as the multi-day decay runs. For all samples, the background count rate was subtracted before further analysis. Activity values were then obtained by multiplying the net count rate by the calibration factor k.

3 Results & Discussion

3.1 Background measurements

Before evaluating the calibration and environmental sample measurements, the background count rate of each detector was analyzed. According to the manual, the relevant information is contained in the first two columns of the raw text files, namely the time stamp and the number of detected counts in one second. Since the raw one-second count rate fluctuates strongly because of the random nature of radioactive decay, the data were additionally binned over longer time intervals in order to obtain a more stable representation of the background behavior.

For detector 1, two background measurements of different duration were available: a short measurement of 10 minutes and a longer measurement of 5 hours. For detectors 2 and 3, background measurements of 2 hours and 3 hours, respectively, were analyzed. The 10-minute run of detector 1 was binned into 1-minute intervals, whereas the longer runs were binned into 1-hour intervals. For each measurement, the total counting time, the mean count rate, its Poisson uncertainty, the standard deviation of the raw one-second counts, and the standard deviation of the binned count rates were determined.

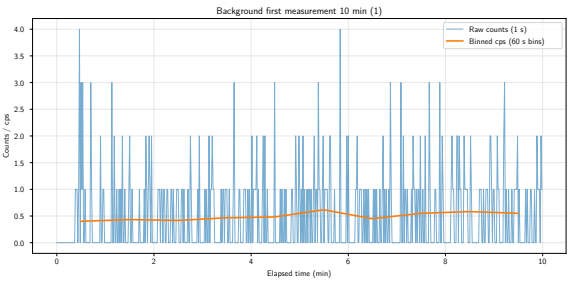
Table 1: Summary of the analyzed background measurements. The quoted uncertainty of the mean count rate was obtained from Poisson counting statistics.

Measurement	Time (min)	Bin size (s)	Mean cps	σ_{mean}	σ_{binned}
Detector 1	10	60	0.495	0.029	0.075
Detector 1	300	3600	0.503	0.005	0.015
Detector 2	120	3600	0.540	0.009	0.011
Detector 3	180	3600	0.539	0.007	0.020

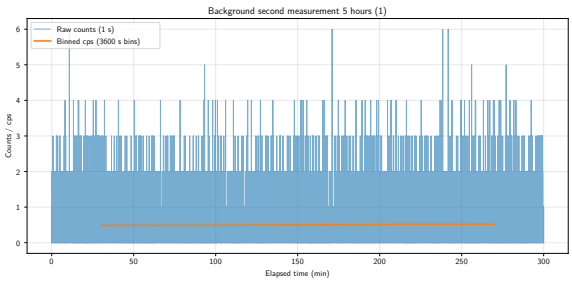
The corresponding background curves are shown in Fig. 1. For all three detectors, the background behavior is stable over the full measurement interval, and the mean count rates are mutually consistent. Detector 2 and detector 3 yield nearly identical mean background values, both close to 0.54 cps, while detector 1 gives 0.495 ± 0.029 cps for the 10-minute run and 0.503 ± 0.005 cps for the 5-hour run. The agreement between the two measurements of detector 1 shows that the detector background is reproducible and of the same order of magnitude as for the other two detectors.

As expected, the short 10-minute background run of detector 1 is more strongly affected by statistical fluctuations than the longer measurements. This is reflected in its larger uncertainty of the mean count rate, σ_{mean}, as well as in the larger spread of the binned values, σ_{binned}. Such behavior is consistent with Poisson counting statistics, since a shorter acquisition time provides fewer total counts and therefore leads to larger relative fluctuations. In contrast, the longer background measurements yield smoother binned curves and smaller statistical uncertainties. Overall, the background analysis confirms that all

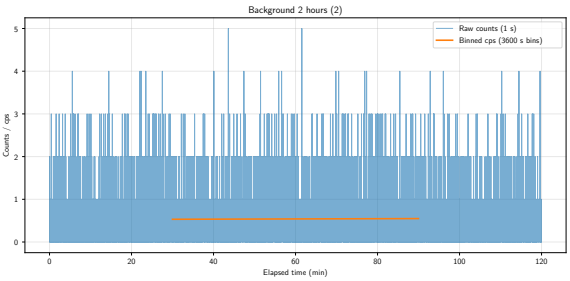
detectors provide stable and consistent baseline count rates, which were subsequently used for the background correction of the KCl calibration measurements.



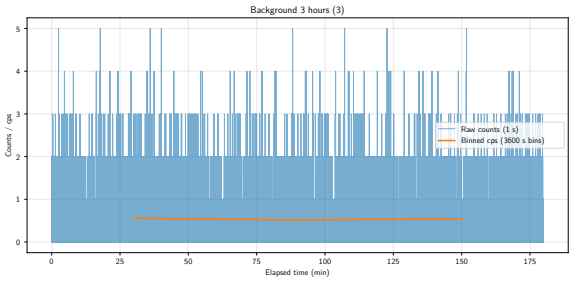
(a) Detector 1, first background measurement (10 min, 60 s bins).



(b) Detector 1, second background measurement (5 h, 3600 s bins).



(c) Detector 2 background measurement (2 h, 3600 s bins).



(d) Detector 3 background measurement (3 h, 3600 s bins).

Figure 1: Background measurements for the three Ranger detectors. The blue curves show the raw one-second counts, while the orange curves represent the binned count rates over longer time intervals. All three detectors exhibit stable and consistent background levels. As expected, the short 10-minute run of detector 1 is more affected by statistical fluctuations than the longer measurements.

Regarding the reduction and control of the measured background, several practical improvements can be suggested. To improve the determination of the detector background, longer and repeated background measurements should be preferred, since they reduce the relative statistical uncertainty and make transient fluctuations easier to identify. Moreover, the detector should be kept in a stable environment and as far as possible from radioactive samples, contaminated surfaces, or temporarily elevated radon concentrations. A constant detector position and unchanged measurement geometry are also important in order to avoid artificial variations between runs. If available, additional shielding or measurements in a location with lower environmental radiation could further reduce the background level and improve the sensitivity for weak samples.

3.2 Calibration efficiency

The activity of the KCl calibration sample was calculated from the abundance of the radioactive isotope ^{40}K in natural potassium. Since each KCl molecule contains one potassium atom, the number of potassium atoms in the sample is obtained from the sample mass and the molar mass of KCl. Multiplying this by the natural isotopic abundance of

^{40}K yields the number of radioactive atoms,

$$N_{^{40}\mathrm{K}} = \frac{m_{\mathrm{KCl}}}{M_{\mathrm{KCl}}} N_A f_{^{40}\mathrm{K}}. \tag{3}$$

The activity then follows from

$$A_{^{40}\mathrm{K}} = \lambda N_{^{40}\mathrm{K}}, \qquad \lambda = \frac{\ln 2}{T_{1/2}}, \tag{4}$$

where $T_{1/2}$ is the half-life of ^{40}K, not of KCl itself. Using $m_{\mathrm{KCl}} = 1.9\,\mathrm{g}$, $M_{\mathrm{KCl}} = 74.5513\,\mathrm{g/mol}$ [2], $f_{^{40}\mathrm{K}} = 0.000117$, and $T_{1/2}(^{40}\mathrm{K}) = (1.248 \pm 0.003) \times 10^9\,\mathrm{y}$ [1], the theoretical activity of ^{40}K in the calibration sample is found to be

$$A_{^{40}\mathrm{K}} = 31.60\,\mathrm{Bq}. \tag{5}$$

For each detector, the KCl calibration measurement was performed over 1 h, and the recorded count-rate data were additionally binned into 5 min intervals. The net count rate was then obtained by subtracting the detector-specific background count rate,

$$\mathrm{cps}_{\mathrm{net}} = \mathrm{cps}_{\mathrm{raw}} - \mathrm{cps}_{\mathrm{bg}}. \tag{6}$$

From the net count rate, the calibration factor was calculated as

$$k = \frac{A_{^{40}\mathrm{K}}}{\mathrm{cps}_{\mathrm{net}}}, \tag{7}$$

and the detector efficiency was determined from

$$\varepsilon = \frac{1}{k}. \tag{8}$$

As a consistency check, detector 1 was calibrated using two different background assignments: first with the short 10 min background measurement, and second with the long 5 h background measurement. The resulting calibration plots are shown in Fig. 2, and the numerical results are summarized in Table 2.

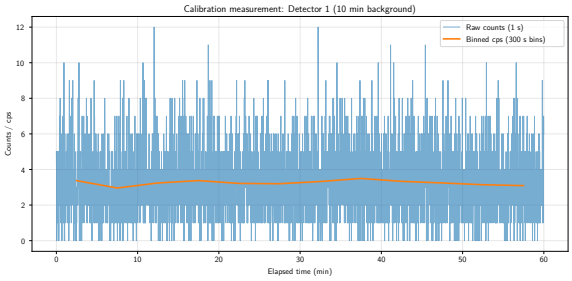
Table 2: Calibration results for the three Ranger detectors. For detector 1, two background assignments were evaluated using the 10 min and 5 h background measurements, respectively. Both cases yield consistent calibration results within uncertainty.

Detector	$\mathrm{cps}_{\mathrm{bg}}$ (s^{-1})	$\mathrm{cps}_{\mathrm{raw}}$ (s^{-1})	$\mathrm{cps}_{\mathrm{net}}$ (s^{-1})	k (Bq/cps)	ε (cps/Bq)
1a	0.495 ± 0.029	3.250 ± 0.030	2.755 ± 0.042	11.47 ± 0.17	0.0872 ± 0.0013
1b	0.503 ± 0.005	3.250 ± 0.030	2.747 ± 0.031	11.50 ± 0.13	0.0869 ± 0.0010
2	0.540 ± 0.009	3.360 ± 0.031	2.820 ± 0.032	11.21 ± 0.13	0.0892 ± 0.0010
3	0.539 ± 0.007	3.182 ± 0.030	2.643 ± 0.031	11.96 ± 0.14	0.0836 ± 0.0010

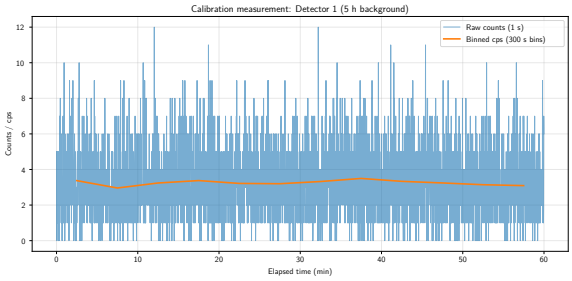
The calibration results of detectors 2 and 3 are mutually consistent. Both detectors yield similar background-subtracted count rates and therefore similar calibration factors and efficiencies. Detector 1 now shows the same overall behavior. Using either the 10 min or the 5 h background measurement, the resulting net count rates remain clearly positive and lead to calibration factors and efficiencies that are in very good agreement with those of detectors 2 and 3. The two background assignments for detector 1 give $k = 11.47 \pm 0.17$ Bq/cps and $k = 11.50 \pm 0.13$ Bq/cps, respectively, showing that the calibration result is stable with respect to the choice of background measurement.

As expected, the use of the longer 5 h background measurement for detector 1 leads to a smaller uncertainty than the 10 min background case, because the background count rate is determined more precisely from the larger number of recorded counts. Nevertheless, both detector 1 calibrations are fully consistent within uncertainty and also agree well with the corresponding values obtained for detectors 2 and 3. Overall, all three Ranger detectors exhibit comparable efficiencies, with calibration factors in the range of about 11–12 Bq/cps and efficiencies close to 0.09 cps/Bq.

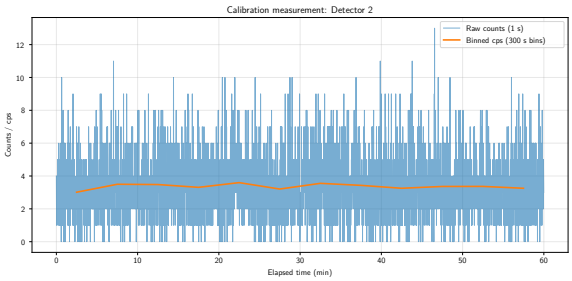
For the following evaluation, the calibration based on the 5 h background measurement is adopted for detector 1, since it provides the smaller statistical uncertainty. The 10 min background case is retained as a consistency check and confirms that the detector response is reproducible within the expected counting statistics.



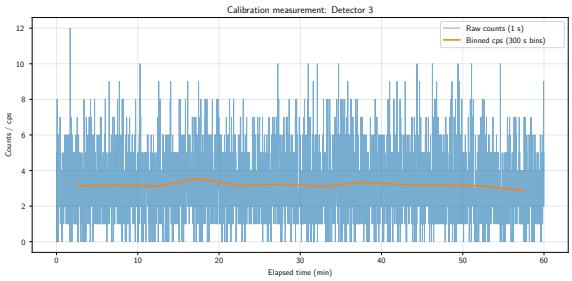
(a) Detector 1 using the 10 min background.



(b) Detector 1 using the 5 h background.



(c) Detector 2 calibration measurement.



(d) Detector 3 calibration measurement.

Figure 2: Calibration measurements of the three Ranger detectors using the 1.9 g KCl reference sample. The blue curves show the raw one-second counts, while the orange curves represent the count rate binned in 300 s intervals. Two background assignments were evaluated for detector 1, using the 10 min and 5 h background measurements, respectively. All four cases, including both background assignments for detector 1, yield consistent background-corrected count rates and calibration results within uncertainty.

In the present experiment, no separate calibration factor for ^{222}Rn was determined independently. Instead, the manual instructs using the detector calibration factor obtained from the KCl efficiency calibration to convert the measured count rates into radon-related activity. Thus, for the subsequent ^{222}Rn analysis, the effective k factor is taken to be the same detector-specific calibration factor determined from the KCl reference measurement. This gives $k = 11.50 \pm 0.13\,\mathrm{Bq/cps}$ for detector 1, $k = 11.21 \pm 0.13\,\mathrm{Bq/cps}$ for detector 2, and $k = 11.96 \pm 0.14\,\mathrm{Bq/cps}$ for detector 3. For detector 1, the calibration obtained with the 10 min background measurement is fully consistent within uncertainty and therefore confirms the robustness of the adopted value.

3.3 Measurement of Rn in Environmental Samples

3.3.1 Water sample

The background-corrected activity curve of the water sample, obtained from approximately 2.0 L of filtered water, is shown in Fig. 3. After conversion with the detector-specific calibration factor of detector 2, only a weak net signal is obtained. The fitted initial activity is

$$A_0 = 0.58 \pm 0.27\,\mathrm{Bq},$$

which corresponds to an initial radon concentration of

$$C_0 = 0.29 \pm 0.13\,\mathrm{Bq\,L^{-1}}.$$

Thus, the water sample exhibits only a small excess above background, and the statistical uncertainties remain comparatively large over the full measurement interval.
A single-exponential fit to the decay curve yields an apparent half-life of

$$T_{1/2} = 190 \pm 390\,\mathrm{min}.$$

However, this result is not meaningful from a physical point of view, since the uncertainty is larger than the central value itself. In addition, the fitted half-life is not compatible with the literature values of the short-lived radon progeny responsible for the detected β activity, namely ^{214}Pb with $T_{1/2} = 26.9\,\mathrm{min}$ and ^{214}Bi with $T_{1/2} = 19.7\,\mathrm{min}$ [3]. The water measurement is therefore best interpreted as a low-activity sample for which a rough concentration estimate can still be obtained, whereas a reliable half-life determination is not possible.

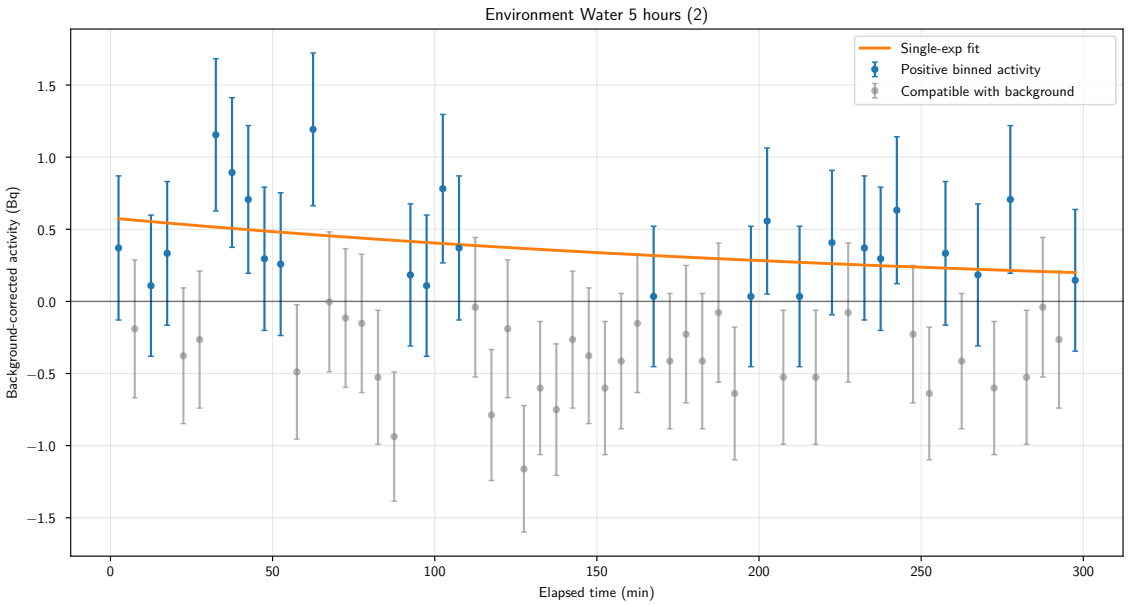


Figure 3: Background-corrected activity of the water sample measured with detector 2 as a function of elapsed time. The data were binned in 300 s intervals and converted into activity using the calibration factor determined from the ^{40}K activity in the KCl calibration sample. Blue points indicate positive background-corrected activity values, whereas gray points are bins compatible with background after subtraction and were not used for the exponential fit. Because the net signal is weak and strongly affected by statistical fluctuations, the extracted half-life should be interpreted with caution.

3.3.2 Air sample

The activity curve obtained from the air filter sample is shown in Fig. 4. In contrast to the water measurement, the air sample exhibits a clear decay immediately after the start of the measurement. From the gas meter readings before and after sampling, the total sampled air volume was determined to be only 3.96 m^3, the meter scale had initially been misread. Using the calibration factor of detector 3, the fitted initial activity is

$$A_0 = 4.52 \pm 0.37\,\mathrm{Bq},$$

which corresponds to an initial radon concentration of

$$C_0 = 1.14 \pm 0.09\,\mathrm{Bq\,m^{-3}}.$$

To estimate the corresponding annual dose, the relation given in the laboratory manual was used,

$$D = C_0\,(\varepsilon_r + \varepsilon_d F)\,O, \tag{9}$$

where C_0 is the radon activity concentration in Bq m^{-3}, $\varepsilon_r = 0.17\,\mathrm{nSv\,h^{-1}\,(Bq\,m^{-3})^{-1}}$ is the dose conversion factor for radon itself, $\varepsilon_d = 9\,\mathrm{nSv\,h^{-1}\,(Bq\,m^{-3})^{-1}}$ is the dose conversion factor for the short-lived progeny, F is the equilibrium factor between radon and its daughters, and O is the indoor occupancy time. Following the manual, $F = 0.4$ was

used. For the occupancy factor, a conventional workplace value of $O = 2000\,\mathrm{h\,yr^{-1}}$ was assumed, consistent with UNSCEAR practice for indoor workplaces [4]. This yields an annual dose estimate of

$$D = (8.60 \pm 0.69)\,\mu\mathrm{Sv\,yr^{-1}}.$$

This dose is very small. Likewise, the measured radon concentration is low compared with typical indoor radon concentrations reported in the literature. According to the WHO, outdoor radon concentrations are usually of the order of 5 to $15\,\mathrm{Bq\,m^{-3}}$, whereas indoor concentrations commonly start at about $10\,\mathrm{Bq\,m^{-3}}$ and may reach well above $100\,\mathrm{Bq\,m^{-3}}$, depending on ventilation and local geology [7]. The value measured in the ETH basement is therefore lower than typical indoor concentrations and even below the usual outdoor range.

This suggests that the HPP basement is not strongly radon contaminated. However, the result should be interpreted with caution. The sampled air volume was smaller than originally intended, and the filter method detects mainly the short-lived radon progeny rather than gaseous ^{222}Rn directly. The obtained concentration should therefore be regarded as an approximate lower estimate rather than a precise absolute radon concentration for the basement.

The decay behavior of the air-filter activity was analyzed using two different models. First, a single-exponential fit was applied in order to describe the data by one effective decay component only. The corresponding model is

$$A(t) = A_0 e^{-\lambda t},$$

where A_0 is the initial activity and λ is the decay constant. The associated half-life is then given by

$$T_{1/2} = \frac{\ln 2}{\lambda}.$$

This model provides an effective half-life for the overall decay of the collected radon progeny.
Applying this fit to the measured air-sample activity yields

$$T_{1/2} = (43.6 \pm 4.6)\,\mathrm{min}.$$

This value is somewhat larger than the literature half-lives of the dominant short-lived radon daughters, namely ^{214}Pb with $T_{1/2} = 26.9\,\mathrm{min}$ and ^{214}Bi with $T_{1/2} = 19.7\,\mathrm{min}$, but it is still of the same order of magnitude. Since the measured count rate originates from the combined contribution of radon progeny collected on the filter, such an effective half-life is physically plausible.

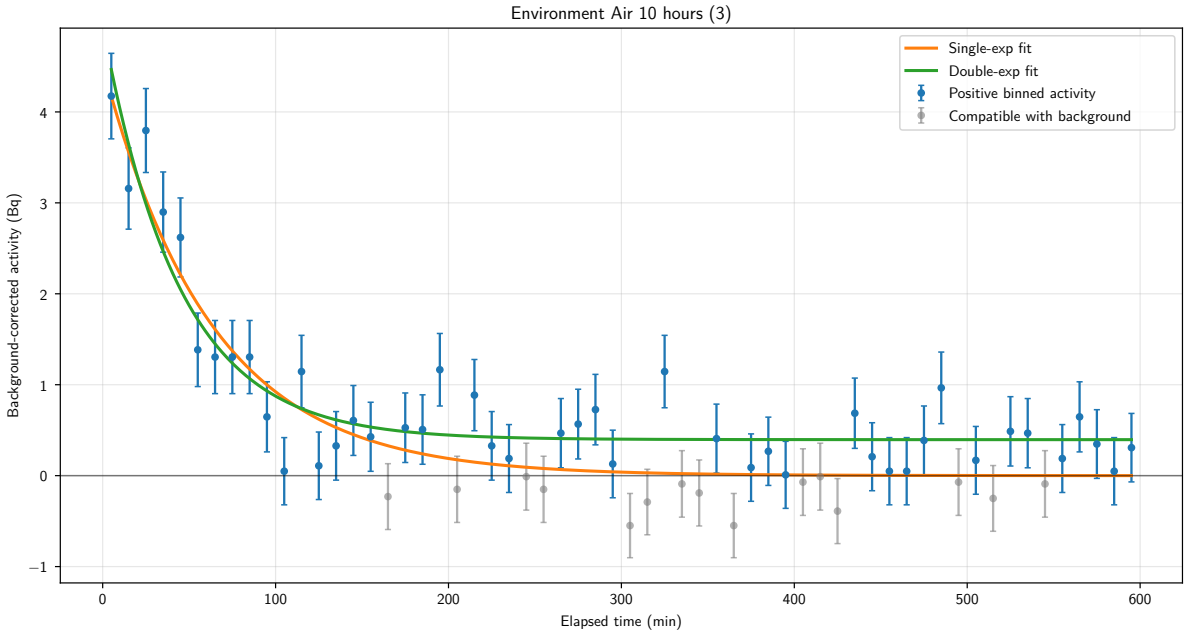


Figure 4: Background-corrected activity of the air filter sample measured with detector 3 as a function of elapsed time. The data were binned in 600 s intervals and converted into activity using the ^{40}K calibration factor. Blue points indicate positive background-corrected activity values, whereas gray points are bins compatible with background after subtraction and were not used for the exponential fits. Both a single-exponential and a double-exponential fit are shown. The rapid initial decrease is clearly visible, whereas the late-time data approach the background level.

To test whether more than one decay component contributes to the measured activity, the data were also fitted with a double-exponential model of the form

$$A(t) = A_1 e^{-\lambda_1 t} + A_2 e^{-\lambda_2 t},$$

where A_1 and A_2 are the amplitudes of the two components and λ_1, λ_2 are their respective decay constants. The corresponding half-lives are

$$T_{1/2,1} = \frac{\ln 2}{\lambda_1}, \qquad T_{1/2,2} = \frac{\ln 2}{\lambda_2}.$$

This more flexible model was used to check whether the decay curve contains more than one characteristic slope, as expected if different short-lived daughter isotopes contribute with different relative weights.

For this purpose, the air data were additionally displayed on a logarithmic scale and fitted with the double-exponential model, as shown in Fig. 5. The fast component yields

$$T_{1/2,\mathrm{fast}} = (30.8 \pm 9.8)\,\mathrm{min},$$

which is reasonably consistent with the expected decay of the short-lived ^{222}Rn daughters, mainly ^{214}Pb and ^{214}Bi, within uncertainty. The slow component yields a much longer half-life. In principle, such a component could indicate contributions from other radon

decay chains, for example from ^{220}Rn in the thorium series through its daughter ^{212}Pb, or from later daughters in the ^{222}Rn decay chain. However, the slow component is not well constrained by the present data and appears mainly at late times, where the activity is close to the background level. Therefore, it should not be interpreted as a reliable identification of an additional isotope.

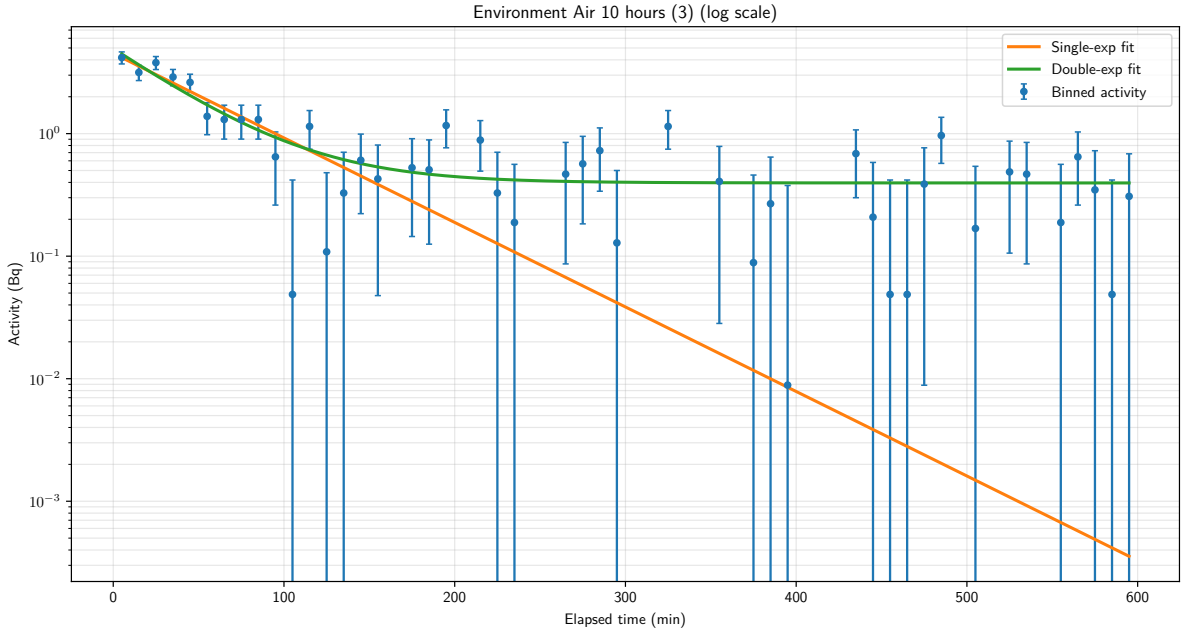


Figure 5: Same air-sample activity data as in Fig. 4, but shown on a logarithmic scale. The logarithmic representation highlights the approximately exponential initial decay and shows that the second, very slow component of the double-exponential fit is not physically meaningful.

Overall, the air sample provides the more reliable environmental radon result. Compared with the water sample, it shows a much clearer excess above background and allows at least an approximate identification of the relevant short-lived radon progeny through the observed decay behavior.

For additional perspective, the estimated annual dose from the air sample,

$$D = 8.60 \pm 0.69\,\mu\mathrm{Sv\,yr^{-1}},$$

is extremely small compared with other common sources of radiation exposure. According to the US EPA, consuming one banana corresponds to a dose of about 0.1 μSv due to naturally occurring ^{40}K [5]. The dose estimated for the HPP basement air sample is therefore comparable to the dose from about 86 bananas per year. At the same time, it is negligible compared with the average annual radiation dose to the public, which the US NRC gives as about 6.2 mSv yr^{-1} in the United States [6]. This comparison further supports the conclusion that the measured radon-related dose in the investigated HPP basement is very low.

3.4 Ingrowth and Decay of ^{222}Rn and its Progeny

This part of the experiment was designed to study both the buildup of the short-lived radon daughters after a short exposure to emanated radon and the subsequent long-time decay behavior after a longer adsorption period on activated carbon. The ingrowth measurement was recorded over about 4 h and evaluated with a binning interval of 300 s, while the long decay measurement extended over about 72 h and was evaluated with 3600 s bins.

3.4.1 Ingrowth

The ingrowth curve is shown in Fig. 6. As expected, the measured activity rises rapidly at early times and then gradually approaches a constant saturation level, indicating that secular equilibrium between ^{222}Rn and its short-lived progeny is being established.
To describe this behavior, the Bateman equations for a parent–daughter decay chain were used in their simplified form. Since the half-life of ^{222}Rn is much longer than the duration of the ingrowth measurement, the parent activity can be treated as approximately constant over the observed time interval. In this limit, the daughter activity follows

$$A(t) = A_\infty \left(1 - e^{-\lambda t}\right),$$

where A_∞ is the equilibrium activity and λ is the effective decay constant of the daughter activity. The corresponding half-life is given by

$$T_{1/2} = \frac{\ln 2}{\lambda}.$$

The fit yields an equilibrium activity of

$$A_\infty = (37.76 \pm 0.32)\,\mathrm{Bq}$$

and an effective half-life of

$$T_{1/2} = (30.74 \pm 0.88)\,\mathrm{min}.$$

This value is of the same order of magnitude as the literature half-lives of the dominant short-lived radon daughters, in particular ^{214}Pb with $T_{1/2} = 26.9\,\mathrm{min}$ and ^{214}Bi with $T_{1/2} = 19.7\,\mathrm{min}$. Since the detected β-activity originates from the combined contribution of these daughter nuclides, such an effective half-life is physically plausible.
Using the fitted half-life, seven daughter half-lives correspond to

$$7\,T_{1/2} = (3.59 \pm 0.10)\,\mathrm{h}.$$

Equivalently, the fitted ingrowth model gives a time of

$$t_{99\%} = (3.40 \pm 0.10)\,\mathrm{h}$$

to reach 99% of the equilibrium activity. In addition, after seven half-lives the fractional approach to equilibrium is

$$1 - 2^{-7} = 0.9922,$$

that is, about 99.2%. This demonstrates quantitatively that secular equilibrium between ^{222}Rn and ^{214}Pb is effectively reached after about seven ^{214}Pb half-lives, in agreement with the expectation from the Bateman equations.

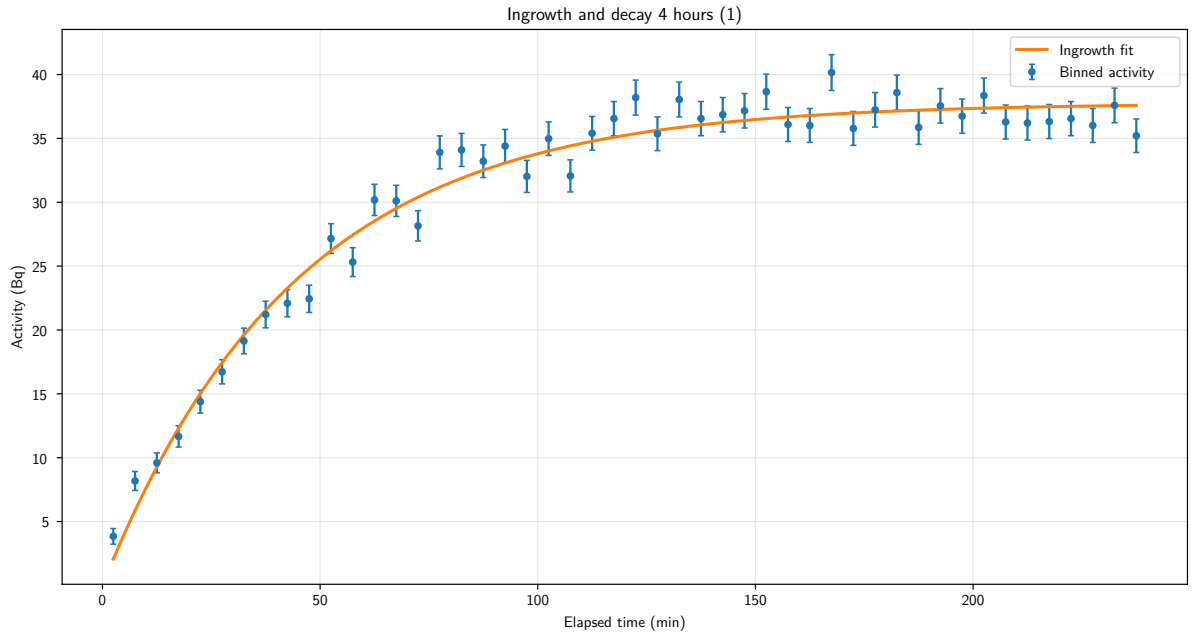


Figure 6: Background-corrected activity of the ingrowth measurement as a function of elapsed time. The data were binned in 300 s intervals and fitted with an ingrowth model approaching a saturation activity A_∞. The curve shows the gradual establishment of secular equilibrium between ^{222}Rn and its short-lived progeny.

3.4.2 Long decay measurement

The long decay measurement is shown in Fig. 7. Immediately after the start of the measurement, the activity decreases rapidly, whereas at later times the data approach a nearly constant residual level. A single-exponential model,

$$A(t) = A_0 e^{-\lambda t},$$

was fitted to the decay data, giving

$$A_0 = 136.89 \pm 1.51\,\mathrm{Bq}$$

and

$$T_{1/2} = 1.339 \pm 0.010\,\mathrm{h}.$$

This fitted half-life is far smaller than the literature half-life of ^{222}Rn,

$$T_{1/2}(^{222}\mathrm{Rn}) = 91.76\,\mathrm{h},$$

and therefore cannot be interpreted as the true decay constant of ^{222}Rn. Instead, the measured decay is dominated by the activity of short-lived daughter nuclides at early times. In the ^{222}Rn decay chain, ^{218}Po has a very short half-life of about 3.1 min and therefore decays rapidly to ^{214}Pb without significant accumulation on the time scale of the measurement. The observed β activity is thus mainly governed by ^{214}Pb and ^{214}Bi, while later daughters such as ^{210}Pb or ^{210}Bi could only contribute to a slow residual component because of their much longer half-lives. Contributions from other radon decay chains, for example, ^{220}Rn from the thorium series, cannot be fully excluded either. The fitted value should therefore be regarded as an effective half-life of a mixed daughter activity rather than a direct measurement of the ^{222}Rn half-life.
The logarithmic representation in Fig. 8 further shows that the late-time data deviate strongly from a pure single-exponential decay and remain close to a residual background-like level. This confirms that the long measurement is not well described by a single decay component over the full 72 h interval.

Consequently, the long decay run should be interpreted mainly qualitatively. It clearly demonstrates the rapid loss of daughter activity after the sample has been removed from the radon source, but it does not provide a reliable experimental determination of the true ^{222}Rn half-life. This is consistent with the known limitation of long measurements in the manual, namely that the effective time spacing between recorded rows may drift over long acquisition times.
Under the simplifying assumptions of secular equilibrium between ^{226}Ra and its daughters before sampling and 100% adsorption efficiency of ^{222}Rn on the activated carbon, the measured initial activity can be used to estimate the activity of ^{226}Ra in the uranium rock. Using the 2 h adsorption time, this gives

$$A(^{226}\mathrm{Ra}) = (9.13 \pm 0.10) \times 10^3\,\mathrm{Bq}.$$

This value should be regarded only as a rough order-of-magnitude estimate, since it depends directly on the strong assumptions of ideal secular equilibrium and complete ^{222}Rn adsorption on the activated carbon. In reality, the adsorption efficiency may be smaller than 100%, and part of the radon may not be trapped or may desorb again during the measurement. In addition, small leaks in the adhesive-film sealing could allow some radon to escape from the sample. Such effects would reduce the measured activity and could also

make the observed decay appear faster than expected. Therefore, the calculated ^{226}Ra activity should not be interpreted as a precise quantitative value.

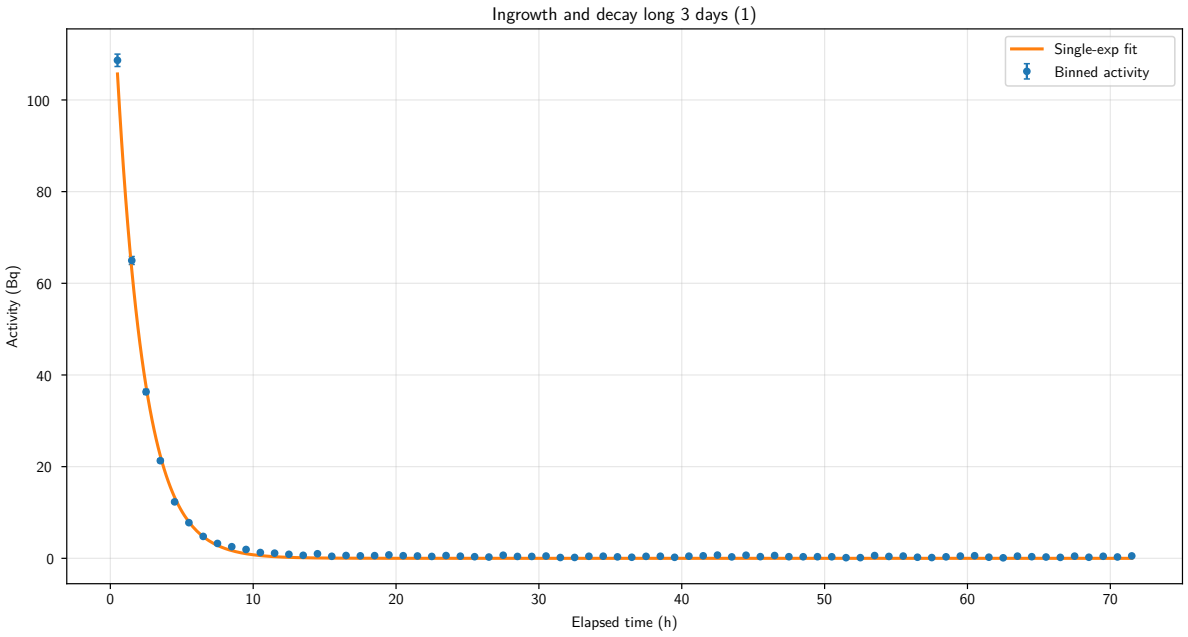


Figure 7: Background-corrected activity of the long decay measurement as a function of elapsed time. The data were binned in 3600 s intervals and fitted with a single-exponential decay model. The rapid initial decrease is visible on the linear scale, but the late-time behavior is not well described by a pure exponential.

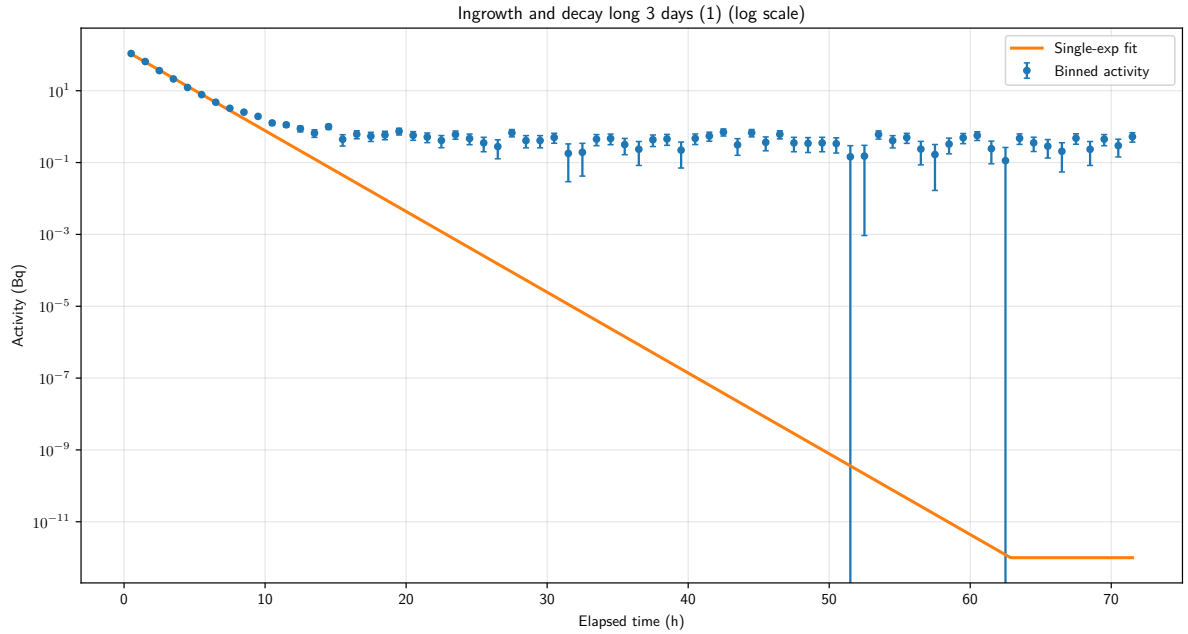


Figure 8: Same long decay data as in Fig. 7, but shown on a logarithmic scale. The logarithmic representation highlights the clear deviation from a pure single-exponential decay at late times, where the activity approaches a residual background-like level.

4 Conclusion

In this experiment, environmental radioactivity was investigated with a focus on ^{222}Rn and its short-lived progeny. The background measurements showed that all three Ranger detectors provided stable and mutually consistent baseline count rates. As expected from Poisson statistics, the shorter background run exhibited larger relative fluctuations, while the longer measurements yielded smaller uncertainties. The subsequent efficiency calibration with the KCl reference sample also gave consistent results for all detectors, with calibration factors in the range of about 11–12 Bq/cps and efficiencies close to 0.09 cps/Bq. This confirms that the detector response was reproducible and that the KCl calibration was suitable for the conversion of measured count rates into radon-related activity.

For the environmental samples, the water measurement showed only a weak excess above background. The fitted initial activity was $A_0 = 0.58 \pm 0.27\,\mathrm{Bq}$, corresponding to a radon concentration of $C_0 = 0.29 \pm 0.13\,\mathrm{Bq\,L^{-1}}$. Because of the low signal and the large statistical fluctuations, the fitted half-life was not physically meaningful, so the water sample mainly allows a rough concentration estimate. In contrast, the air filter sample yielded a much clearer signal. The initial activity was found to be $A_0 = 4.52 \pm 0.37\,\mathrm{Bq}$, corresponding to $C_0 = 1.14 \pm 0.09\,\mathrm{Bq\,m^{-3}}$, and the associated annual dose estimate was $D = 8.60 \pm 0.69\,\mu\mathrm{Sv\,yr^{-1}}$. This value is very small, indicating that the investigated HPP basement was not strongly radon contaminated. The decay analysis of the air sample gave an effective half-life of $T_{1/2} = 43.6 \pm 4.6\,\mathrm{min}$ for the single-exponential model, while the fast component of the double-exponential fit yielded $T_{1/2,\mathrm{fast}} = 30.8 \pm 9.8\,\mathrm{min}$. Both values are of the expected order of magnitude for the short-lived radon daughters and therefore support their identification as the dominant contributors to the measured β activity.

The ingrowth measurement further demonstrated the establishment of secular equilibrium between ^{222}Rn and its short-lived progeny. The fitted equilibrium activity was $A_\infty = 37.76 \pm 0.32\,\mathrm{Bq}$ and the effective half-life was $T_{1/2} = 30.74 \pm 0.88\,\mathrm{min}$, which is consistent with the expected behavior of the combined daughter activity. The analysis showed that about 99% of equilibrium is reached after approximately $3.40 \pm 0.10\,\mathrm{h}$, in agreement with the Bateman-based expectation of about seven daughter half-lives. By contrast, the long decay measurement could only be interpreted qualitatively. Although it clearly showed the rapid loss of daughter activity after removal from the radon source, the fitted half-life of $1.339 \pm 0.010\,\mathrm{h}$ is far below the literature half-life of ^{222}Rn and therefore does not represent the true radon decay. The estimated ^{226}Ra activity of the uranium rock, $A(^{226}\mathrm{Ra}) = (9.13 \pm 0.10) \times 10^3\,\mathrm{Bq}$, should therefore only be regarded as an order-of-magnitude estimate because it depends on strong simplifying assumptions, in particular ideal secular equilibrium, complete radon adsorption on the activated carbon, and the absence of leakage or desorption.

Overall, the experiment successfully illustrated the main concepts of environmental radioactivity, radioactive decay, ingrowth, and secular equilibrium using real measurements. The most reliable environmental result was obtained from the air sample, whereas the water measurement was limited by low activity and poor counting statistics. The main limitations of the experiment were the weak signal strength of some samples, the smaller-than-intended sampled air volume, and the limited reliability of very long measurements. Nevertheless, the results are physically consistent and show that the Ranger detector, together with a simple KCl calibration, is well suited for the qualitative and semi-quantitative investigation of environmental radon and its progeny.

Appendix

Acknowledgments

I would like to thank my lab partner, Denis Seibel, for the good collaboration throughout this experiment. We carried out the measurements together and jointly collected the data used in this report.

I would like to thank my teaching assistant, Dr. Miriam Mindová, for her supervision and support during the experiment. Her guidance and explanations were very helpful throughout the laboratory work and the interpretation of the results.

AI Usage Declaration

This report was prepared with assistance from ChatGPT (OpenAI) for language polishing, formatting suggestions, and for helping to develop, debug, and correct the syntax of the Python code. The AI was not used to generate raw data, to select results, or to determine the final conclusions. All scientific content (methods, calculations, results, and interpretation) was critically reviewed and validated by the authors. The authors take full responsibility for the accuracy of the analysis and for any remaining errors. No confidential or personal data were provided to the AI system during the preparation of this report.

Safety Measures

Several safety precautions were followed throughout the experiment in order to minimize unnecessary radiation exposure and to avoid contamination of the detector and the working area. Since the main radionuclides investigated in this experiment were ^{222}Rn and its short-lived progeny, particular care was taken when handling samples that had been exposed to radon emanation.

The detector window of the Ranger instrument was treated as a sensitive component and was never touched either by the sample or by the operator. During all measurements, the samples were placed in the holder in such a way that no direct contact with the detector window occurred. In addition, the activated carbon and KCl samples were sealed with adhesive film before being placed under the detector in order to prevent contamination.

To reduce inhalation exposure, measurements involving airborne radon progeny were carried out in a controlled way and the sampled filters were handled only for the time required to transfer them into the sample holder. The activated carbon exposed to radon was sealed immediately after sampling. More generally, unnecessary proximity to radioactive samples was avoided, and the samples were returned to their designated storage location after the measurements.

Finally, standard laboratory safety rules were respected throughout the experiment. In particular, the workplace was kept organized, the measurement setup was handled carefully, and all experimental procedures were performed according to the instructions given in the laboratory manual and by the teaching assistant.

References

[1] F. G. Kondev et al. "The NUBASE2020 evaluation of nuclear physics properties". In: *Chinese Physics C* 45.3 (2021), p. 030001. DOI: 10.1088/1674-1137/abddae.

[2] National Bureau of Standards. *Certificate of Standard Reference Material 1655: Potassium Chloride, KCl(cr), for Solution Calorimetry*. Standard Reference Material Certificate 1655. National Bureau of Standards, Mar. 1981.

[3] Habacuc Pérez Tribouillier. *Environmental Radioactivity*. ETH Zürich. Zürich, Switzerland, Oct. 2023.

[4] United Nations Scientific Committee on the Effects of Atomic Radiation. *Evaluation of Occupational Exposure to Ionizing Radiation*. Tech. rep. UNSCEAR 2020/2021 Report, Volume IV, accessed 2026-04-09. United Nations, 2022. URL: https://www.unscear.org/unscear/uploads/documents/unscear-reports/UNSCEAR_2020_21_Report_Vol.IV.pdf.

[5] United States Environmental Protection Agency. *Natural Radioactivity in Food*. Accessed 2026-04-09. July 2025. URL: https://www.epa.gov/radtown/natural-radioactivity-food.

[6] United States Nuclear Regulatory Commission. *Doses in Our Daily Lives*. Accessed 2026-04-09. n.d. URL: https://www.nrc.gov/about-nrc/radiation/around-us/doses-daily-lives.

[7] World Health Organization. *Radon*. WHO fact sheet, accessed 2026-04-09. Jan. 2023. URL: https://www.who.int/news-room/fact-sheets/detail/radon-and-health.

Python Code

1. Background Measurement.py

```python
from pathlib import Path
import re
from datetime import datetime
import math

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt



# ============================================================
# PATHS
# ============================================================

# Assumed location:
# Datenauswertung/
#  Python Code/
#     BackgroundAnalysis.py
#  Messdaten/

BASE_DIR = Path(__file__).resolve().parent.parent
DATA_DIR = BASE_DIR / "Messdaten"
OUTPUT_DIR = BASE_DIR / "background_analysis_output"
OUTPUT_DIR.mkdir(exist_ok=True)


# ============================================================
# USER SETTINGS
# ============================================================

# Here you define which background folders you want to analyze
# and which bin size (in seconds) should be used for each one.
#
# Suggested examples:
# 60 s   = 1 min
# 300 s  = 5 min
# 600 s  = 10 min
```

```
# 1800 s = 30 min
# 3600 s = 1 hour

BACKGROUND_CONFIG = {
    "Background first measurement 10 min (1)": 60,
    "Background second measurement 5 hours (1)": 3600,
    "Background 2 hours (2)": 3600,
    "Background 3 hours (3)": 3600,
}

# If you want to also save the binned data as csv files:
SAVE_BINNED_DATA = True

# If you want plots to pop up interactively:
SHOW_PLOTS = True


# ============================================================
# HELPERS
# ============================================================

def extract_device_number(folder_name: str):
    match = re.search(r"\((\d+)\)\s*$", folder_name)
    return int(match.group(1)) if match else None


def extract_datetime_from_filename(filename: str):
    """
    Example:
    Chart 2026_03_30 13.04.44.txt
    """
    match = re.search(r"(\d{4}_\d{2}_\d{2})\s+(\d{2}\.\d{2}\.\d{2})", filename)
    if not match:
        return None
    return datetime.strptime(
        f"{match.group(1)} {match.group(2)}",
        "%Y_%m_%d %H.%M.%S"
    )


def list_txt_files_sorted(folder: Path):
```

```
    files = list(folder.glob("*.txt"))
    return sorted(files, key=lambda f: extract_datetime_from_filename(f.name)
    or datetime.min)


def normalize_name(name: str):
    return name.strip().lower()


def find_folder_by_exact_name(folder_name: str):
    for folder in DATA_DIR.iterdir():
        if folder.is_dir() and normalize_name(folder.name) ==
    normalize_name(folder_name):
            return folder
    raise FileNotFoundError(f"Folder not found: {folder_name}")


# ============================================================
# TXT READING
# ============================================================

def read_measurement_file(file_path: Path):
    """
    Read only the first two columns:
    1) Date/Time
    2) counts (1 sec)

    Ignore all columns from column 3 onward, exactly as stated in the manual.
    """
    encodings = ["utf-8", "latin1", "cp1252"]
    last_error = None

    for enc in encodings:
        try:
            df = pd.read_csv(
                file_path,
                sep="\t",
                encoding=enc,
                usecols=[0, 1]
            )
            break
```

```
        except Exception as e:
            last_error = e
    else:
        raise RuntimeError(f"Could not read {file_path.name}: {last_error}")

    df.columns = ["Date/Time", "counts"]

    out = pd.DataFrame()
    out["datetime"] = pd.to_datetime(
        df["Date/Time"],
        format="%Y.%m.%d %H:%M:%S",
        errors="coerce"
    )
    out["counts"] = pd.to_numeric(df["counts"], errors="coerce")
    out["source_file"] = file_path.name
    out["file_start_datetime"] = extract_datetime_from_filename(file_path.name)

    out = out.dropna(subset=["counts"]).copy()
    out["counts"] = out["counts"].astype(float)

    return out


def read_folder_measurement(folder: Path):
    """
    Read all txt files in chronological order and concatenate them.
    Each row corresponds to one second of measurement.
    """
    txt_files = list_txt_files_sorted(folder)
    if not txt_files:
        raise FileNotFoundError(f"No txt files found in folder: {folder.name}")

    dfs = []
    for f in txt_files:
        dfs.append(read_measurement_file(f))

    merged = pd.concat(dfs, ignore_index=True)
    merged["elapsed_s"] = np.arange(len(merged), dtype=float)
    merged["elapsed_min"] = merged["elapsed_s"] / 60.0
    merged["elapsed_h"] = merged["elapsed_s"] / 3600.0

```

```
    return merged


# ============================================================
# BINNING
# ============================================================

def bin_counts(df: pd.DataFrame, bin_size_s: int):
    """
    Bin the 1-second count data into bins of length bin_size_s.

    For each bin:
      cps_bin = (sum of counts in bin) / (number of rows in bin)

    This matches the manual:
    "sum of counts in a time interval divided by the number of integrated rows"
    """
    if bin_size_s <= 0:
        raise ValueError("bin_size_s must be positive.")

    data = df.copy()

    # Bin index based on row number, since each row = 1 second
    data["bin_index"] = (np.arange(len(data)) // bin_size_s).astype(int)

    grouped = data.groupby("bin_index", as_index=False).agg(
        elapsed_s_start=("elapsed_s", "min"),
        elapsed_s_end=("elapsed_s", "max"),
        n_rows=("counts", "size"),
        counts_sum=("counts", "sum"),
        counts_mean=("counts", "mean"),
    )

    grouped["cps"] = grouped["counts_sum"] / grouped["n_rows"]
    grouped["elapsed_s_center"] = 0.5 * (grouped["elapsed_s_start"] +
    grouped["elapsed_s_end"])
    grouped["elapsed_min_center"] = grouped["elapsed_s_center"] / 60.0
    grouped["elapsed_h_center"] = grouped["elapsed_s_center"] / 3600.0

    # Poisson uncertainty of mean cps in each bin:
    # sigma = sqrt(total counts) / n_rows
```

```
    grouped["sigma_cps"] = np.sqrt(grouped["counts_sum"]) / grouped["n_rows"]

    return grouped


# ============================================================
# SUMMARY STATISTICS
# ============================================================

def measurement_summary(df: pd.DataFrame, binned_df: pd.DataFrame,
    folder_name: str, bin_size_s: int):
    counts = df["counts"].to_numpy(dtype=float)
    n_seconds = len(counts)
    total_counts = counts.sum()

    mean_cps = total_counts / n_seconds
    sigma_mean_cps = math.sqrt(total_counts) / n_seconds

    std_1s = np.std(counts, ddof=1) if len(counts) > 1 else np.nan
    std_binned = np.std(binned_df["cps"], ddof=1) if len(binned_df) > 1 else
    np.nan

    return {
        "folder": folder_name,
        "device": extract_device_number(folder_name),
        "n_seconds": n_seconds,
        "measurement_time_min": n_seconds / 60.0,
        "measurement_time_h": n_seconds / 3600.0,
        "bin_size_s": bin_size_s,
        "n_bins": len(binned_df),
        "total_counts": total_counts,
        "mean_cps": mean_cps,
        "sigma_mean_cps": sigma_mean_cps,
        "std_1s_counts": std_1s,
        "std_binned_cps": std_binned,
    }


# ============================================================
# PLOTTING
# ============================================================
```

```
239
240 def plot_background(df: pd.DataFrame, binned_df: pd.DataFrame, folder_name:
    str, bin_size_s: int):
241     """
242     Plot raw 1-second counts and binned cps.
243     """
244     fig, ax = plt.subplots(figsize=(10, 5))
245
246     # raw 1-second counts
247     ax.plot(df["elapsed_min"], df["counts"], linewidth=0.7, alpha=0.6,
    label="Raw counts (1 s)")
248
249     # binned cps
250     ax.plot(
251         binned_df["elapsed_min_center"],
252         binned_df["cps"],
253         linewidth=2.0,
254         label=f"Binned cps ({bin_size_s} s bins)"
255     )
256
257     ax.set_title(folder_name)
258     ax.set_xlabel("Elapsed time (min)")
259     ax.set_ylabel("Counts / cps")
260     ax.legend()
261     ax.grid(True, alpha=0.3)
262
263     plt.tight_layout()
264
265     safe_name = re.sub(r'[^\w\-_. ]', '_', folder_name).replace(" ", "_")
266     fig_path = OUTPUT_DIR / f"{safe_name}_background_plot.png"
267     plt.savefig(fig_path, dpi=200)
268
269     if SHOW_PLOTS:
270         plt.show()
271     else:
272         plt.close(fig)
273
274     return fig_path
275
276
277 # ============================================================
```

```python
# MAIN ANALYSIS
# ============================================================

def analyze_background_folder(folder_name: str, bin_size_s: int):
    folder = find_folder_by_exact_name(folder_name)

    df = read_folder_measurement(folder)
    binned_df = bin_counts(df, bin_size_s=bin_size_s)

    summary = measurement_summary(
        df=df,
        binned_df=binned_df,
        folder_name=folder_name,
        bin_size_s=bin_size_s
    )

    fig_path = plot_background(df, binned_df, folder_name, bin_size_s)

    if SAVE_BINNED_DATA:
        safe_name = re.sub(r'[^\w\-_. ]', '_', folder_name).replace(" ", "_")
        binned_csv = OUTPUT_DIR / f"{safe_name}_binned.csv"
        binned_df.to_csv(binned_csv, index=False)
    else:
        binned_csv = None

    return summary, fig_path, binned_csv


if __name__ == "__main__":
    print("Base directory :", BASE_DIR)
    print("Data directory :", DATA_DIR)
    print("Output directory:", OUTPUT_DIR)

    all_summaries = []

    for folder_name, bin_size_s in BACKGROUND_CONFIG.items():
        print("\n" + "=" * 70)
        print(f"Analyzing background folder: {folder_name}")
        print(f"Chosen bin size: {bin_size_s} s")
        print("=" * 70)

```

```
        try:
            summary, fig_path, binned_csv =
    analyze_background_folder(folder_name, bin_size_s)

            all_summaries.append(summary)

            print(f"Device                : {summary['device']}")
            print(f"Measurement time [min]:
    {summary['measurement_time_min']:.2f}")
            print(f"Measurement time [h]  :
    {summary['measurement_time_h']:.3f}")
            print(f"Number of seconds     : {summary['n_seconds']}")
            print(f"Number of bins        : {summary['n_bins']}")
            print(f"Mean cps              : {summary['mean_cps']:.6f} pm
    {summary['sigma_mean_cps']:.6f}")
            print(f"Std of 1 s counts     : {summary['std_1s_counts']:.6f}")
            print(f"Std of binned cps     : {summary['std_binned_cps']:.6f}")
            print(f"Plot saved to         : {fig_path}")
            if binned_csv is not None:
                print(f"Binned data saved to  : {binned_csv}")

        except Exception as e:
            print(f"ERROR for folder '{folder_name}': {e}")

    if all_summaries:
        summary_df = pd.DataFrame(all_summaries)
        summary_csv = OUTPUT_DIR / "background_summary.csv"
        summary_df.to_csv(summary_csv, index=False)

        print("\n" + "=" * 70)
        print("FINAL BACKGROUND SUMMARY")
        print("=" * 70)
        print(summary_df.to_string(index=False))
        print(f"\nSummary saved to: {summary_csv}")
    else:
        print("\nNo background results were produced.")



# ============================================================
# PATHS
```

```
# ============================================================

# Assumed folder structure:
# Datenauswertung/
#  Python Code/
#     EnvironmentalRadioactivity.py
#  Messdaten/

BASE_DIR = Path(__file__).resolve().parent.parent
DATA_DIR = BASE_DIR / "Messdaten"
OUTPUT_DIR = BASE_DIR / "calibration_analysis_output"
OUTPUT_DIR.mkdir(exist_ok=True)


# ============================================================
# USER SETTINGS
# ============================================================

# Folder names exactly as they appear in Messdaten
CALIBRATION_CONFIG = {
    1: {
        "background_folder": "Background first measurement 10 min (1)",
        "calibration_folder": "calibration 1 hour (1)",
        "bin_size_s": 300,   # 5 min bins, as recommended in the manual
    },
    2: {
        "background_folder": "Background 2 hours (2)",
        "calibration_folder": "calibration 1 hour (2)",
        "bin_size_s": 300,
    },
    3: {
        "background_folder": "Background 2 hours (3)",
        "calibration_folder": "calibration 1 hour (3)",
        "bin_size_s": 300,
    },
}

SHOW_PLOTS = True
SAVE_BINNED_DATA = True


```

```
# ============================================================
# PHYSICAL CONSTANTS FOR KCl CALIBRATION
# ============================================================

KCL_MASS_G = 1.9
K40_ABUNDANCE = 0.000117        # 0.0117 %
K40_HALF_LIFE_Y = 1.248e9       # years
AVOGADRO = 6.02214076e23
MOLAR_MASS_KCL = 74.5513        # g/mol

# Keep total 40K activity for now
K40_BRANCHING_FACTOR = 1.0


# ============================================================
# HELPERS
# ============================================================

def extract_datetime_from_filename(filename: str):
    match = re.search(r"(\d{4}_\d{2}_\d{2})\s+(\d{2}\.\d{2}\.\d{2})", filename)
    if not match:
        return None
    return datetime.strptime(
        f"{match.group(1)} {match.group(2)}",
        "%Y_%m_%d %H.%M.%S"
    )


def normalize_name(name: str):
    return name.strip().lower()


def find_folder_by_exact_name(folder_name: str):
    for folder in DATA_DIR.iterdir():
        if folder.is_dir() and normalize_name(folder.name) ==
    normalize_name(folder_name):
            return folder
    raise FileNotFoundError(f"Folder not found: {folder_name}")


def list_txt_files_sorted(folder: Path):
```

```
    files = list(folder.glob("*.txt"))
    return sorted(files, key=lambda f: extract_datetime_from_filename(f.name)
    or datetime.min)


# ============================================================
# TXT READING
# ============================================================

def read_measurement_file(file_path: Path):
    """
    Reads only the first two columns:
    1) Date/Time
    2) counts (1 sec)

    All columns from column 3 onward are ignored.
    """
    encodings = ["utf-8", "latin1", "cp1252"]
    last_error = None

    for enc in encodings:
        try:
            df = pd.read_csv(
                file_path,
                sep="\t",
                encoding=enc,
                usecols=[0, 1]
            )
            break
        except Exception as e:
            last_error = e
    else:
        raise RuntimeError(f"Could not read {file_path.name}: {last_error}")

    df.columns = ["Date/Time", "counts"]

    out = pd.DataFrame()
    out["datetime"] = pd.to_datetime(
        df["Date/Time"],
        format="%Y.%m.%d %H:%M:%S",
        errors="coerce"
```

```
    )
    out["counts"] = pd.to_numeric(df["counts"], errors="coerce")
    out["source_file"] = file_path.name
    out["file_start_datetime"] = extract_datetime_from_filename(file_path.name)

    out = out.dropna(subset=["counts"]).copy()
    out["counts"] = out["counts"].astype(float)

    return out


def read_folder_measurement(folder: Path):
    txt_files = list_txt_files_sorted(folder)
    if not txt_files:
        raise FileNotFoundError(f"No txt files found in folder: {folder.name}")

    dfs = []
    for f in txt_files:
        dfs.append(read_measurement_file(f))

    merged = pd.concat(dfs, ignore_index=True)
    merged["elapsed_s"] = np.arange(len(merged), dtype=float)
    merged["elapsed_min"] = merged["elapsed_s"] / 60.0
    merged["elapsed_h"] = merged["elapsed_s"] / 3600.0

    return merged


# ============================================================
# BINNING
# ============================================================

def bin_counts(df: pd.DataFrame, bin_size_s: int):
    if bin_size_s <= 0:
        raise ValueError("bin_size_s must be positive.")

    data = df.copy()
    data["bin_index"] = (np.arange(len(data)) // bin_size_s).astype(int)

    grouped = data.groupby("bin_index", as_index=False).agg(
        elapsed_s_start=("elapsed_s", "min"),
```

```
        elapsed_s_end=("elapsed_s", "max"),
        n_rows=("counts", "size"),
        counts_sum=("counts", "sum"),
    )

    grouped["cps"] = grouped["counts_sum"] / grouped["n_rows"]
    grouped["elapsed_s_center"] = 0.5 * (grouped["elapsed_s_start"] +
    grouped["elapsed_s_end"])
    grouped["elapsed_min_center"] = grouped["elapsed_s_center"] / 60.0
    grouped["elapsed_h_center"] = grouped["elapsed_s_center"] / 3600.0
    grouped["sigma_cps"] = np.sqrt(grouped["counts_sum"]) / grouped["n_rows"]

    return grouped


# ============================================================
# STATISTICS / PHYSICS
# ============================================================

def count_rate_stats(df: pd.DataFrame):
    counts = df["counts"].to_numpy(dtype=float)
    n_seconds = len(counts)

    if n_seconds == 0:
        raise ValueError("No data rows found.")

    total_counts = counts.sum()
    mean_cps = total_counts / n_seconds
    sigma_mean_cps = math.sqrt(total_counts) / n_seconds

    return {
        "n_seconds": n_seconds,
        "total_counts": total_counts,
        "mean_cps": mean_cps,
        "sigma_mean_cps": sigma_mean_cps,
    }


def k40_activity_from_kcl_mass(mass_g: float):
    """
    Compute 40K activity in the KCl sample:
```

```
    A = lambda * N
    """
    n_kcl_mol = mass_g / MOLAR_MASS_KCL
    n_k_atoms = n_kcl_mol * AVOGADRO
    n_k40_atoms = n_k_atoms * K40_ABUNDANCE

    t_half_s = K40_HALF_LIFE_Y * 365.25 * 24 * 3600
    decay_constant = math.log(2) / t_half_s

    activity_bq = decay_constant * n_k40_atoms
    activity_bq *= K40_BRANCHING_FACTOR

    return activity_bq


def calibration_from_rates(activity_bq: float, raw_stats: dict, bg_stats:
    dict):
    raw_cps = raw_stats["mean_cps"]
    sigma_raw = raw_stats["sigma_mean_cps"]

    bg_cps = bg_stats["mean_cps"]
    sigma_bg = bg_stats["sigma_mean_cps"]

    net_cps = raw_cps - bg_cps
    sigma_net = math.sqrt(sigma_raw**2 + sigma_bg**2)

    if net_cps <= 0:
        raise ValueError(f"Non-positive net count rate: {net_cps:.6f} cps")

    k = activity_bq / net_cps
    sigma_k = activity_bq * sigma_net / (net_cps**2)

    efficiency = 1.0 / k
    sigma_efficiency = sigma_k / (k**2)

    return {
        "raw_cps": raw_cps,
        "sigma_raw": sigma_raw,
        "bg_cps": bg_cps,
        "sigma_bg": sigma_bg,
        "net_cps": net_cps,
```

```
        "sigma_net": sigma_net,
        "k": k,
        "sigma_k": sigma_k,
        "efficiency": efficiency,
        "sigma_efficiency": sigma_efficiency,
    }


# ============================================================
# PLOTTING
# ============================================================

def plot_calibration(device: int, calib_df: pd.DataFrame, calib_binned:
    pd.DataFrame):
    fig, ax = plt.subplots(figsize=(10, 5))

    ax.plot(
        calib_df["elapsed_min"],
        calib_df["counts"],
        linewidth=0.7,
        alpha=0.6,
        label="Raw counts (1 s)"
    )

    bin_size_s = int(round(calib_binned["n_rows"].iloc[0]))
    ax.plot(
        calib_binned["elapsed_min_center"],
        calib_binned["cps"],
        linewidth=2.0,
        label=f"Binned cps ({bin_size_s} s bins)"
    )

    ax.set_title(f"Calibration measurement detector {device}")
    ax.set_xlabel("Elapsed time (min)")
    ax.set_ylabel("Counts / cps")
    ax.grid(True, alpha=0.3)
    ax.legend()

    plt.tight_layout()

    fig_path = OUTPUT_DIR / f"calibration_detector{device}.png"
```

```
    plt.savefig(fig_path, dpi=200)

    if SHOW_PLOTS:
        plt.show()
    else:
        plt.close(fig)

    return fig_path
```

2. Calibration Efficiency.py

```python
from pathlib import Path
import re
from datetime import datetime
import math
import traceback

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# ============================================================
# PATHS
# ============================================================

# Assumed folder structure:
# Datenauswertung/
#  Python Code/
#     2. Calibration Efficiency.py
#  Messdaten/

BASE_DIR = Path(__file__).resolve().parent.parent
DATA_DIR = BASE_DIR / "Messdaten"
OUTPUT_DIR = BASE_DIR / "calibration_analysis_output"
OUTPUT_DIR.mkdir(exist_ok=True)


# ============================================================
# USER SETTINGS
# ============================================================

# We include 4 calibration cases:
# - detector 1 with 10 min background (main choice)
# - detector 1 with 5 h background (consistency check)
# - detector 2
# - detector 3
CALIBRATION_CONFIG = {
    "detector1_10min_bg": {
        "device": 1,
        "background_folder": "Background first measurement 10 min (1)",
```

```
        "calibration_folder": "Calibration 1 hour (1)",
        "bin_size_s": 300,   # 5 min bins
        "label": "Detector 1 (10 min background)",
    },
    "detector1_5h_bg": {
        "device": 1,
        "background_folder": "Background second measurement 5 hours (1)",
        "calibration_folder": "Calibration 1 hour (1)",
        "bin_size_s": 300,
        "label": "Detector 1 (5 h background)",
    },
    "detector2": {
        "device": 2,
        "background_folder": "Background 2 hours (2)",
        "calibration_folder": "Calibration 1 hour (2)",
        "bin_size_s": 300,
        "label": "Detector 2",
    },
    "detector3": {
        "device": 3,
        "background_folder": "Background 3 hours (3)",
        "calibration_folder": "Calibration 1 hour (3)",
        "bin_size_s": 300,
        "label": "Detector 3",
    },
}

# Set to False to avoid the script blocking on plt.show()
SHOW_PLOTS = True
SAVE_BINNED_DATA = True


# ============================================================
# PHYSICAL CONSTANTS FOR KCl CALIBRATION
# ============================================================

KCL_MASS_G = 1.9
K40_ABUNDANCE = 0.000117        # 0.0117 %
K40_HALF_LIFE_Y = 1.248e9       # years
AVOGADRO = 6.02214076e23
MOLAR_MASS_KCL = 74.5513        # g/mol
```

```
82
83  # Keep total 40K activity
84  K40_BRANCHING_FACTOR = 1.0
85
86
87  # ============================================================
88  # HELPERS
89  # ============================================================
90
91  def extract_datetime_from_filename(filename: str):
92      match = re.search(r"(\d{4}_\d{2}_\d{2})\s+(\d{2}\.\d{2}\.\d{2})", filename)
93      if not match:
94          return None
95      return datetime.strptime(
96          f"{match.group(1)} {match.group(2)}",
97          "%Y_%m_%d %H.%M.%S"
98      )
99
100
101 def normalize_name(name: str):
102     return name.strip().lower()
103
104
105 def find_folder_by_exact_name(folder_name: str):
106     for folder in DATA_DIR.iterdir():
107         if folder.is_dir() and normalize_name(folder.name) ==
    normalize_name(folder_name):
108             return folder
109     raise FileNotFoundError(f"Folder not found: {folder_name}")
110
111
112 def list_txt_files_sorted(folder: Path):
113     files = list(folder.glob("*.txt"))
114     return sorted(files, key=lambda f: extract_datetime_from_filename(f.name)
    or datetime.min)
115
116
117 # ============================================================
118 # TXT READING
119 # ============================================================
120
```

```python
def read_measurement_file(file_path: Path):
    """
    Reads only the first two columns:
    1) Date/Time
    2) counts (1 sec)

    All columns from column 3 onward are ignored.
    """
    encodings = ["utf-8", "latin1", "cp1252"]
    last_error = None

    for enc in encodings:
        try:
            df = pd.read_csv(
                file_path,
                sep="\t",
                encoding=enc,
                usecols=[0, 1]
            )
            break
        except Exception as e:
            last_error = e
    else:
        raise RuntimeError(f"Could not read {file_path.name}: {last_error}")

    df.columns = ["Date/Time", "counts"]

    out = pd.DataFrame()
    out["datetime"] = pd.to_datetime(
        df["Date/Time"],
        format="%Y.%m.%d %H:%M:%S",
        errors="coerce"
    )
    out["counts"] = pd.to_numeric(df["counts"], errors="coerce")
    out["source_file"] = file_path.name
    out["file_start_datetime"] = extract_datetime_from_filename(file_path.name)

    out = out.dropna(subset=["counts"]).copy()
    out["counts"] = out["counts"].astype(float)

    return out
```

```
162
163
164 def read_folder_measurement(folder: Path):
165     txt_files = list_txt_files_sorted(folder)
166     if not txt_files:
167         raise FileNotFoundError(f"No txt files found in folder: {folder.name}")
168
169     dfs = []
170     for f in txt_files:
171         dfs.append(read_measurement_file(f))
172
173     merged = pd.concat(dfs, ignore_index=True)
174     merged["elapsed_s"] = np.arange(len(merged), dtype=float)
175     merged["elapsed_min"] = merged["elapsed_s"] / 60.0
176     merged["elapsed_h"] = merged["elapsed_s"] / 3600.0
177
178     return merged
179
180
181 # ============================================================
182 # BINNING
183 # ============================================================
184
185 def bin_counts(df: pd.DataFrame, bin_size_s: int):
186     if bin_size_s <= 0:
187         raise ValueError("bin_size_s must be positive.")
188
189     data = df.copy()
190     data["bin_index"] = (np.arange(len(data)) // bin_size_s).astype(int)
191
192     grouped = data.groupby("bin_index", as_index=False).agg(
193         elapsed_s_start=("elapsed_s", "min"),
194         elapsed_s_end=("elapsed_s", "max"),
195         n_rows=("counts", "size"),
196         counts_sum=("counts", "sum"),
197     )
198
199     grouped["cps"] = grouped["counts_sum"] / grouped["n_rows"]
200     grouped["elapsed_s_center"] = 0.5 * (grouped["elapsed_s_start"] +
    grouped["elapsed_s_end"])
201     grouped["elapsed_min_center"] = grouped["elapsed_s_center"] / 60.0
```

```
    grouped["elapsed_h_center"] = grouped["elapsed_s_center"] / 3600.0
    grouped["sigma_cps"] = np.sqrt(grouped["counts_sum"]) / grouped["n_rows"]

    return grouped


# ============================================================
# STATISTICS / PHYSICS
# ============================================================

def count_rate_stats(df: pd.DataFrame):
    counts = df["counts"].to_numpy(dtype=float)
    n_seconds = len(counts)

    if n_seconds == 0:
        raise ValueError("No data rows found.")

    total_counts = counts.sum()
    mean_cps = total_counts / n_seconds
    sigma_mean_cps = math.sqrt(total_counts) / n_seconds

    return {
        "n_seconds": n_seconds,
        "total_counts": total_counts,
        "mean_cps": mean_cps,
        "sigma_mean_cps": sigma_mean_cps,
    }


def k40_activity_from_kcl_mass(mass_g: float):
    """
    Compute 40K activity in the KCl sample:
    A = lambda * N
    """
    n_kcl_mol = mass_g / MOLAR_MASS_KCL
    n_k_atoms = n_kcl_mol * AVOGADRO
    n_k40_atoms = n_k_atoms * K40_ABUNDANCE

    t_half_s = K40_HALF_LIFE_Y * 365.25 * 24 * 3600
    decay_constant = math.log(2) / t_half_s

```

```
    activity_bq = decay_constant * n_k40_atoms
    activity_bq *= K40_BRANCHING_FACTOR

    return activity_bq


def calibration_from_rates(activity_bq: float, raw_stats: dict, bg_stats:
    dict):
    raw_cps = raw_stats["mean_cps"]
    sigma_raw = raw_stats["sigma_mean_cps"]

    bg_cps = bg_stats["mean_cps"]
    sigma_bg = bg_stats["sigma_mean_cps"]

    net_cps = raw_cps - bg_cps
    sigma_net = math.sqrt(sigma_raw**2 + sigma_bg**2)

    if net_cps <= 0:
        return {
            "raw_cps": raw_cps,
            "sigma_raw": sigma_raw,
            "bg_cps": bg_cps,
            "sigma_bg": sigma_bg,
            "net_cps": net_cps,
            "sigma_net": sigma_net,
            "k": np.nan,
            "sigma_k": np.nan,
            "efficiency": np.nan,
            "sigma_efficiency": np.nan,
        }

    k = activity_bq / net_cps
    sigma_k = activity_bq * sigma_net / (net_cps**2)

    efficiency = 1.0 / k
    sigma_efficiency = sigma_k / (k**2)

    return {
        "raw_cps": raw_cps,
        "sigma_raw": sigma_raw,
        "bg_cps": bg_cps,
```

```
        "sigma_bg": sigma_bg,
        "net_cps": net_cps,
        "sigma_net": sigma_net,
        "k": k,
        "sigma_k": sigma_k,
        "efficiency": efficiency,
        "sigma_efficiency": sigma_efficiency,
    }



# ============================================================
# PLOTTING
# ============================================================

def plot_calibration(case_name: str, label: str, calib_df: pd.DataFrame,
    calib_binned: pd.DataFrame):
    fig, ax = plt.subplots(figsize=(10, 5))

    ax.plot(
        calib_df["elapsed_min"],
        calib_df["counts"],
        linewidth=0.7,
        alpha=0.6,
        label="Raw counts (1 s)"
    )

    bin_size_s = int(round(calib_binned["n_rows"].iloc[0]))
    ax.plot(
        calib_binned["elapsed_min_center"],
        calib_binned["cps"],
        linewidth=2.0,
        label=f"Binned cps ({bin_size_s} s bins)"
    )

    ax.set_title(f"Calibration measurement: {label}")
    ax.set_xlabel("Elapsed time (min)")
    ax.set_ylabel("Counts / cps")
    ax.grid(True, alpha=0.3)
    ax.legend()

```

```
    plt.tight_layout()

    fig_path = OUTPUT_DIR / f"{case_name}.png"
    plt.savefig(fig_path, dpi=200)

    if SHOW_PLOTS:
        plt.show()
    else:
        plt.close(fig)

    return fig_path


# ============================================================
# MAIN CALIBRATION ANALYSIS
# ============================================================

def analyze_calibration_case(case_name: str, config: dict):
    device = config["device"]
    label = config["label"]

    bg_folder = find_folder_by_exact_name(config["background_folder"])
    cal_folder = find_folder_by_exact_name(config["calibration_folder"])
    bin_size_s = config["bin_size_s"]

    bg_df = read_folder_measurement(bg_folder)
    cal_df = read_folder_measurement(cal_folder)

    cal_binned = bin_counts(cal_df, bin_size_s=bin_size_s)

    bg_stats = count_rate_stats(bg_df)
    cal_stats = count_rate_stats(cal_df)

    activity_bq = k40_activity_from_kcl_mass(KCL_MASS_G)
    cal_results = calibration_from_rates(activity_bq, cal_stats, bg_stats)

    fig_path = plot_calibration(case_name, label, cal_df, cal_binned)

    if SAVE_BINNED_DATA:
        binned_csv = OUTPUT_DIR / f"{case_name}_binned.csv"
        cal_binned.to_csv(binned_csv, index=False)
```

```
    else:
        binned_csv = None

    summary = {
        "case_name": case_name,
        "label": label,
        "device": device,
        "background_folder": config["background_folder"],
        "calibration_folder": config["calibration_folder"],
        "background_n_seconds": bg_stats["n_seconds"],
        "background_mean_cps": cal_results["bg_cps"],
        "background_sigma_cps": cal_results["sigma_bg"],
        "calibration_n_seconds": cal_stats["n_seconds"],
        "calibration_raw_mean_cps": cal_results["raw_cps"],
        "calibration_raw_sigma_cps": cal_results["sigma_raw"],
        "calibration_net_cps": cal_results["net_cps"],
        "calibration_net_sigma_cps": cal_results["sigma_net"],
        "kcl_activity_bq": activity_bq,
        "k_factor": cal_results["k"],
        "sigma_k_factor": cal_results["sigma_k"],
        "efficiency": cal_results["efficiency"],
        "sigma_efficiency": cal_results["sigma_efficiency"],
        "bin_size_s": bin_size_s,
    }

    return summary, fig_path, binned_csv


if __name__ == "__main__":
    print("Base directory :", BASE_DIR)
    print("Data directory :", DATA_DIR)
    print("Output directory:", OUTPUT_DIR)

    all_results = []

    for case_name, config in CALIBRATION_CONFIG.items():
        print("\n" + "=" * 70)
        print(f"Analyzing calibration case: {case_name}")
        print("=" * 70)

        try:
```

```
            summary, fig_path, binned_csv =
    analyze_calibration_case(case_name, config)
            all_results.append(summary)

            print(f"Label                  : {summary['label']}")
            print(f"Background folder      : {summary['background_folder']}")
            print(f"Calibration folder     : {summary['calibration_folder']}")
            print(f"Background mean cps    :
    {summary['background_mean_cps']:.6f} pm
    {summary['background_sigma_cps']:.6f}")
            print(f"Calibration raw cps    :
    {summary['calibration_raw_mean_cps']:.6f} pm
    {summary['calibration_raw_sigma_cps']:.6f}")
            print(f"Calibration net cps    :
    {summary['calibration_net_cps']:.6f} pm
    {summary['calibration_net_sigma_cps']:.6f}")
            print(f"KCl activity [Bq]      : {summary['kcl_activity_bq']:.6f}")
            print(f"k factor [Bq/cps]      : {summary['k_factor']:.6f} pm
    {summary['sigma_k_factor']:.6f}")
            print(f"Efficiency [cps/Bq]    : {summary['efficiency']:.6f} pm
    {summary['sigma_efficiency']:.6f}")
            print(f"Plot saved to          : {fig_path}")
            if binned_csv is not None:
                print(f"Binned data saved to   : {binned_csv}")

        except Exception as e:
            print(f"ERROR for case {case_name}: {e}")
            traceback.print_exc()

    if all_results:
        results_df = pd.DataFrame(all_results)
        results_csv = OUTPUT_DIR / "calibration_summary_all_cases.csv"
        results_df.to_csv(results_csv, index=False)

        print("\n" + "=" * 70)
        print("FINAL CALIBRATION SUMMARY")
        print("=" * 70)
        print(results_df.to_string(index=False))
        print(f"\nSummary saved to: {results_csv}")
    else:
        print("\nNo calibration results were produced.")
```

3. Rn Environmental Samples Air and Water.py

```python
from __future__ import annotations

import math
from pathlib import Path
from typing import Optional, Dict, Any, List

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit


# ============================================================
# USER CONFIGURATION
# ============================================================

BASE_DIR = Path(__file__).resolve().parents[1]
DATA_DIR = BASE_DIR / "Messdaten"
OUTPUT_DIR = BASE_DIR / "environmental_samples_analysis_output"

# Enter your real measured sample volumes here ###############
WATER_VOLUME_L = 2.00   # manual: 2-3 L
AIR_VOLUME_M3 = 3.96    # manual: 25.0 m^3

# Plotting/fit convention:
# Negative background-subtracted activities are interpreted as compatible with background.
# The fit functions below use only activity_bq > 0.

# Dose model from manual
DOSE_EPS_R = 0.17       # nSv h^-1 per (Bq m^-3)
DOSE_EPS_D = 9.0        # nSv h^-1 per (Bq m^-3)
DOSE_F = 0.4
DOSE_OCCUPANCY_H_PER_YEAR = 2000.0

# Corrected calibration values from your latest calibration output
DETECTORS: Dict[int, Dict[str, float]] = {
    2: {
        "bg_cps": 0.540278,
        "sigma_bg_cps": 0.008662,
```

```
        "k_bq_per_cps": 11.207092,
        "sigma_k_bq_per_cps": 0.126203,
    },
    3: {
        "bg_cps": 0.539259,
        "sigma_bg_cps": 0.007066,
        "k_bq_per_cps": 11.957791,
        "sigma_k_bq_per_cps": 0.138263,
    },
}

ANALYSES: List[Dict[str, Any]] = [
    {
        "name": "water",
        "label": "Environment Water 5 hours (2)",
        "folder": "Environment Water 5 hours (2)",
        "detector": 2,
        "bin_s": 300,               # 5 min
        "fit_tmin_min": 0.0,
        "fit_tmax_min": 180.0,      # adjust if needed
        "volume_l": WATER_VOLUME_L,
    },
    {
        "name": "air",
        "label": "Environment Air 10 hours (3)",
        "folder": "Environment Air 10 hours (3)",
        "detector": 3,
        "bin_s": 600,               # 10 min
        "fit_tmin_min": 0.0,
        "fit_tmax_min": 300.0,      # adjust if needed
        "volume_m3": AIR_VOLUME_M3,
    },
]

# Literature half-lives
T12_PB214_MIN = 26.9
T12_BI214_MIN = 19.7


# ============================================================
# HELPER FUNCTIONS
```

```
# ============================================================

def pm_str(value: float, sigma: Optional[float] = None, digits: int = 3) ->
    str:
    if sigma is None or not np.isfinite(sigma):
        return f"{value:.{digits}f}"
    return f"{value:.{digits}f} pm {sigma:.{digits}f}"


def safe_mkdir(path: Path) -> None:
    path.mkdir(parents=True, exist_ok=True)


def read_txt_file(path: Path) -> pd.DataFrame:
    """
    Read only the first two columns:
      1) Date/Time
      2) counts (1 sec)
    Ignore all columns from the 3rd onward.

    The Ranger text files are often saved in Windows ANSI encoding
    (typically cp1252), not UTF-8.
    """
    last_error = None

    for enc in ["utf-8", "cp1252", "latin-1"]:
        try:
            df = pd.read_csv(
                path,
                sep=r"\t+",
                engine="python",
                usecols=[0, 1],
                header=0,
                encoding=enc,
            )
            break
        except UnicodeDecodeError as exc:
            last_error = exc
    else:
        raise last_error

```

```
    df.columns = ["DateTime", "counts"]

    df["DateTime"] = pd.to_datetime(
        df["DateTime"],
        format="%Y.%m.%d %H:%M:%S",
        errors="coerce"
    )
    df["counts"] = pd.to_numeric(df["counts"], errors="coerce")

    df = df.dropna(subset=["DateTime", "counts"]).copy()
    df["counts"] = df["counts"].astype(float)
    return df


def read_measurement_folder(folder: Path) -> pd.DataFrame:
    txt_files = sorted(folder.glob("*.txt"))
    if not txt_files:
        raise FileNotFoundError(f"No .txt files found in: {folder}")

    frames = []
    for txt_file in txt_files:
        df = read_txt_file(txt_file)
        df["source_file"] = txt_file.name
        frames.append(df)

    data = pd.concat(frames, ignore_index=True)
    data = data.sort_values("DateTime").drop_duplicates(subset=["DateTime"]).reset_index(drop=True)

    t0 = data["DateTime"].iloc[0]
    data["elapsed_s"] = (data["DateTime"] - t0).dt.total_seconds()
    data["elapsed_min"] = data["elapsed_s"] / 60.0
    data["elapsed_h"] = data["elapsed_s"] / 3600.0
    return data


def bin_time_series(data: pd.DataFrame, bin_s: int) -> pd.DataFrame:
    out = data.copy()
    out["bin_index"] = (out["elapsed_s"] // bin_s).astype(int)

    grouped = out.groupby("bin_index", as_index=False)
```

```
161
162     binned = grouped.agg(
163         t_start_s=("elapsed_s", "min"),
164         t_end_s=("elapsed_s", "max"),
165         n_rows=("counts", "size"),
166         counts_sum=("counts", "sum"),
167     )
168
169     binned["t_mid_s"] = 0.5 * (binned["t_start_s"] + binned["t_end_s"])
170     binned["t_mid_min"] = binned["t_mid_s"] / 60.0
171     binned["t_mid_h"] = binned["t_mid_s"] / 3600.0
172
173     # mean cps in each bin
174     binned["raw_cps"] = binned["counts_sum"] / binned["n_rows"]
175
176     # Poisson uncertainty on mean cps
177     binned["sigma_raw_cps"] = np.sqrt(np.clip(binned["counts_sum"], 0.0,
    None)) / binned["n_rows"]
178
179     return binned
180
181
182 def apply_background_and_calibration(
183     binned: pd.DataFrame,
184     bg_cps: float,
185     sigma_bg_cps: float,
186     k_bq_per_cps: float,
187     sigma_k_bq_per_cps: float,
188 ) -> pd.DataFrame:
189     out = binned.copy()
190
191     out["bg_cps"] = bg_cps
192     out["sigma_bg_cps"] = sigma_bg_cps
193     out["net_cps"] = out["raw_cps"] - bg_cps
194     out["sigma_net_cps"] = np.sqrt(out["sigma_raw_cps"] ** 2 + sigma_bg_cps **
    2)
195
196     out["activity_bq"] = out["net_cps"] * k_bq_per_cps
197     out["sigma_activity_bq"] = np.sqrt(
198         (k_bq_per_cps * out["sigma_net_cps"]) ** 2 +
199         (out["net_cps"] * sigma_k_bq_per_cps) ** 2
```

```
    )

    return out


def select_fit_window(
    df: pd.DataFrame,
    tmin_min: float = 0.0,
    tmax_min: Optional[float] = None,
) -> pd.DataFrame:
    mask = df["t_mid_min"] >= tmin_min
    if tmax_min is not None:
        mask &= df["t_mid_min"] <= tmax_min
    return df.loc[mask].copy()


# ============================================================
# FIT MODELS
# ============================================================

def model_decay_single(t_s: np.ndarray, a0: float, lamb: float) -> np.ndarray:
    return a0 * np.exp(-lamb * t_s)


def model_decay_double(t_s: np.ndarray, a1: float, lamb1: float, a2: float,
    lamb2: float) -> np.ndarray:
    return a1 * np.exp(-lamb1 * t_s) + a2 * np.exp(-lamb2 * t_s)


def fit_single_decay(df: pd.DataFrame) -> Optional[Dict[str, float]]:
    fit_df = df[(df["activity_bq"] > 0) & (df["sigma_activity_bq"] > 0)].copy()
    if len(fit_df) < 4:
        return None

    t = fit_df["t_mid_s"].to_numpy(dtype=float)
    y = fit_df["activity_bq"].to_numpy(dtype=float)
    s = fit_df["sigma_activity_bq"].to_numpy(dtype=float)

    p0 = [max(y[0], y.max()), np.log(2.0) / (30.0 * 60.0)]
    bounds = ([0.0, 0.0], [np.inf, np.inf])

```

```
    try:
        popt, pcov = curve_fit(
            model_decay_single,
            t,
            y,
            p0=p0,
            sigma=s,
            absolute_sigma=True,
            bounds=bounds,
            maxfev=20000,
        )
    except Exception:
        return None

    perr = np.sqrt(np.diag(pcov))
    a0, lamb = popt
    sigma_a0, sigma_lamb = perr

    t12_s = np.log(2.0) / lamb
    sigma_t12_s = np.log(2.0) * sigma_lamb / (lamb ** 2)

    y_fit = model_decay_single(t, *popt)
    rss = float(np.sum((y - y_fit) ** 2))

    return {
        "a0_bq": a0,
        "sigma_a0_bq": sigma_a0,
        "lambda_per_s": lamb,
        "sigma_lambda_per_s": sigma_lamb,
        "t12_s": t12_s,
        "sigma_t12_s": sigma_t12_s,
        "rss": rss,
    }


def fit_double_decay(df: pd.DataFrame) -> Optional[Dict[str, float]]:
    fit_df = df[(df["activity_bq"] > 0) & (df["sigma_activity_bq"] > 0)].copy()
    if len(fit_df) < 6:
        return None

    t = fit_df["t_mid_s"].to_numpy(dtype=float)
```

```
    y = fit_df["activity_bq"].to_numpy(dtype=float)
    s = fit_df["sigma_activity_bq"].to_numpy(dtype=float)

    ymax = max(y.max(), 1e-9)
    p0 = [
        0.6 * ymax,
        np.log(2.0) / (20.0 * 60.0),    # faster component
        0.4 * ymax,
        np.log(2.0) / (90.0 * 60.0),    # slower component
    ]
    bounds = ([0.0, 0.0, 0.0, 0.0], [np.inf, np.inf, np.inf, np.inf])

    try:
        popt, pcov = curve_fit(
            model_decay_double,
            t,
            y,
            p0=p0,
            sigma=s,
            absolute_sigma=True,
            bounds=bounds,
            maxfev=50000,
        )
    except Exception:
        return None

    perr = np.sqrt(np.diag(pcov))
    a1, lamb1, a2, lamb2 = popt
    sa1, sl1, sa2, sl2 = perr

    # sort so component 1 is always the faster one
    if lamb2 > lamb1:
        a1, a2 = a2, a1
        lamb1, lamb2 = lamb2, lamb1
        sa1, sa2 = sa2, sa1
        sl1, sl2 = sl2, sl1

    t12_1_s = np.log(2.0) / lamb1
    t12_2_s = np.log(2.0) / lamb2
    sigma_t12_1_s = np.log(2.0) * sl1 / (lamb1 ** 2)
    sigma_t12_2_s = np.log(2.0) * sl2 / (lamb2 ** 2)
```

```
322
323     y_fit = model_decay_double(t, a1, lamb1, a2, lamb2)
324     rss = float(np.sum((y - y_fit) ** 2))
325
326     return {
327         "a1_bq": a1,
328         "sigma_a1_bq": sa1,
329         "lambda1_per_s": lamb1,
330         "sigma_lambda1_per_s": sl1,
331         "a2_bq": a2,
332         "sigma_a2_bq": sa2,
333         "lambda2_per_s": lamb2,
334         "sigma_lambda2_per_s": sl2,
335         "t12_1_s": t12_1_s,
336         "sigma_t12_1_s": sigma_t12_1_s,
337         "t12_2_s": t12_2_s,
338         "sigma_t12_2_s": sigma_t12_2_s,
339         "rss": rss,
340     }
341
342
343 # ============================================================
344 # PLOTTING
345 # ============================================================
346
347 def plot_raw_and_binned(raw_df: pd.DataFrame, binned_df: pd.DataFrame, label:
    str, outpath: Path) -> None:
348     plt.figure(figsize=(11, 6))
349     plt.plot(raw_df["elapsed_min"], raw_df["counts"], linewidth=0.8,
    label="Raw counts (1 s)")
350     plt.plot(binned_df["t_mid_min"], binned_df["raw_cps"], linewidth=2.0,
    label="Binned cps")
351     plt.xlabel("Elapsed time (min)")
352     plt.ylabel("Counts / cps")
353     plt.title(label)
354     plt.grid(True, alpha=0.3)
355     plt.legend()
356     plt.tight_layout()
357     plt.savefig(outpath, dpi=200)
358     plt.show()
359     plt.close()
```

```python


def plot_activity_linear(
    binned_df: pd.DataFrame,
    label: str,
    outpath: Path,
    single_fit: Optional[Dict[str, float]] = None,
    double_fit: Optional[Dict[str, float]] = None,
) -> None:
    t_min = binned_df["t_mid_min"].to_numpy(dtype=float)
    t_s = binned_df["t_mid_s"].to_numpy(dtype=float)
    y = binned_df["activity_bq"].to_numpy(dtype=float)
    s = binned_df["sigma_activity_bq"].to_numpy(dtype=float)

    positive_mask = y > 0
    negative_mask = y <= 0

    plt.figure(figsize=(11, 6))

    # Positive, physically meaningful background-subtracted activities
    if np.any(positive_mask):
        plt.errorbar(
            t_min[positive_mask],
            y[positive_mask],
            yerr=s[positive_mask],
            fmt="o",
            ms=4,
            capsize=2,
            label="Positive binned activity",
        )

    # Negative background-subtracted bins:
    # not interpreted as negative activity, only as compatible with background
    if np.any(negative_mask):
        plt.errorbar(
            t_min[negative_mask],
            y[negative_mask],
            yerr=s[negative_mask],
            fmt="o",
            ms=4,
            capsize=2,
```

```
            color="gray",
            ecolor="gray",
            alpha=0.6,
            label="Compatible with background",
        )

    if single_fit is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_single(tfit, single_fit["a0_bq"],
    single_fit["lambda_per_s"])
        plt.plot(tfit / 60.0, yfit, linewidth=2.0, label="Single-exp fit")

    if double_fit is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_double(
            tfit,
            double_fit["a1_bq"],
            double_fit["lambda1_per_s"],
            double_fit["a2_bq"],
            double_fit["lambda2_per_s"],
        )
        plt.plot(tfit / 60.0, yfit, linewidth=2.0, label="Double-exp fit")

    plt.axhline(0.0, color="black", linewidth=1.0, alpha=0.5)
    plt.xlabel("Elapsed time (min)")
    plt.ylabel("Background-corrected activity (Bq)")
    plt.title(label)
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath, dpi=200)
    plt.show()
    plt.close()


def plot_activity_log(
    binned_df: pd.DataFrame,
    label: str,
    outpath: Path,
    single_fit: Optional[Dict[str, float]] = None,
    double_fit: Optional[Dict[str, float]] = None,
```

```
) -> None:
    t_min = binned_df["t_mid_min"].to_numpy(dtype=float)
    t_s = binned_df["t_mid_s"].to_numpy(dtype=float)
    y = binned_df["activity_bq"].to_numpy(dtype=float)
    s = binned_df["sigma_activity_bq"].to_numpy(dtype=float)

    mask = y > 0
    if not np.any(mask):
        return

    plt.figure(figsize=(11, 6))
    plt.errorbar(t_min[mask], y[mask], yerr=s[mask], fmt="o", ms=4, capsize=2,
    label="Binned activity")

    if single_fit is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_single(tfit, single_fit["a0_bq"],
    single_fit["lambda_per_s"])
        yfit = np.clip(yfit, 1e-12, None)
        plt.plot(tfit / 60.0, yfit, linewidth=2.0, label="Single-exp fit")

    if double_fit is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_double(
            tfit,
            double_fit["a1_bq"],
            double_fit["lambda1_per_s"],
            double_fit["a2_bq"],
            double_fit["lambda2_per_s"],
        )
        yfit = np.clip(yfit, 1e-12, None)
        plt.plot(tfit / 60.0, yfit, linewidth=2.0, label="Double-exp fit")

    plt.yscale("log")
    plt.xlabel("Elapsed time (min)")
    plt.ylabel("Activity (Bq)")
    plt.title(label + " (log scale)")
    plt.grid(True, which="both", alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath, dpi=200)
```

```
    plt.show()
    plt.close()


# ============================================================
# ANALYSIS
# ============================================================


def analyze_water(config: Dict[str, Any]) -> Dict[str, Any]:
    folder = DATA_DIR / config["folder"]
    detector = DETECTORS[config["detector"]]
    outdir = OUTPUT_DIR / config["name"]
    safe_mkdir(outdir)

    raw_df = read_measurement_folder(folder)
    binned = bin_time_series(raw_df, config["bin_s"])
    binned = apply_background_and_calibration(
        binned,
        bg_cps=detector["bg_cps"],
        sigma_bg_cps=detector["sigma_bg_cps"],
        k_bq_per_cps=detector["k_bq_per_cps"],
        sigma_k_bq_per_cps=detector["sigma_k_bq_per_cps"],
    )

    fit_df = select_fit_window(binned, config["fit_tmin_min"],
    config["fit_tmax_min"])
    single_fit = fit_single_decay(fit_df)

    plot_raw_and_binned(raw_df, binned, config["label"], outdir /
    "water_raw_binned.png")
    plot_activity_linear(binned, config["label"], outdir /
    "water_activity_linear.png", single_fit=single_fit)

    binned.to_csv(outdir / "water_binned.csv", index=False)

    result: Dict[str, Any] = {
        "name": "water",
        "label": config["label"],
        "detector": config["detector"],
        "bin_s": config["bin_s"],
```

```
        "n_raw_rows": len(raw_df),
        "duration_h": raw_df["elapsed_h"].iloc[-1],
        "single_fit_ok": single_fit is not None,
    }

    print("=" * 70)
    print(f"Analyzing: {config['label']}")
    print("=" * 70)
    print(f"Folder                  : {folder}")
    print(f"Detector                : {config['detector']}")
    print(f"Background cps          : {pm_str(detector['bg_cps'],
detector['sigma_bg_cps'], 6)}")
    print(f"k factor [Bq/cps]       : {pm_str(detector['k_bq_per_cps'],
detector['sigma_k_bq_per_cps'], 6)}")
    print(f"Rows read               : {len(raw_df)}")
    print(f"Duration [h]            : {raw_df['elapsed_h'].iloc[-1]:.3f}")
    print(f"Bin size [s]            : {config['bin_s']}")

    if single_fit is not None:
        t12_min = single_fit["t12_s"] / 60.0
        sigma_t12_min = single_fit["sigma_t12_s"] / 60.0

        result["a0_bq"] = single_fit["a0_bq"]
        result["sigma_a0_bq"] = single_fit["sigma_a0_bq"]
        result["t12_min"] = t12_min
        result["sigma_t12_min"] = sigma_t12_min

        print("Single exponential fit : OK")
        print(f"A0 [Bq]                 : {pm_str(single_fit['a0_bq'],
single_fit['sigma_a0_bq'], 3)}")
        print(f"T1/2 [min]              : {pm_str(t12_min, sigma_t12_min, 3)}")

        volume_l = config.get("volume_l", None)
        if volume_l is not None and volume_l > 0:
            c0_bq_per_l = single_fit["a0_bq"] / volume_l
            sigma_c0_bq_per_l = single_fit["sigma_a0_bq"] / volume_l
            result["c0_bq_per_l"] = c0_bq_per_l
            result["sigma_c0_bq_per_l"] = sigma_c0_bq_per_l
            print(f"C0 water [Bq/L]         : {pm_str(c0_bq_per_l,
sigma_c0_bq_per_l, 4)}")
        else:
```

```
            print("C0 water [Bq/L]        : Volume not set")
    else:
        print("Single exponential fit : FAILED")

    print(f"Reference Pb-214 [min] : {T12_PB214_MIN:.1f}")
    print(f"Reference Bi-214 [min] : {T12_BI214_MIN:.1f}")
    print(f"Saved binned CSV       : {outdir / 'water_binned.csv'}")
    print(f"Saved plots            : {outdir}")

    return result


def analyze_air(config: Dict[str, Any]) -> Dict[str, Any]:
    folder = DATA_DIR / config["folder"]
    detector = DETECTORS[config["detector"]]
    outdir = OUTPUT_DIR / config["name"]
    safe_mkdir(outdir)

    raw_df = read_measurement_folder(folder)
    binned = bin_time_series(raw_df, config["bin_s"])
    binned = apply_background_and_calibration(
        binned,
        bg_cps=detector["bg_cps"],
        sigma_bg_cps=detector["sigma_bg_cps"],
        k_bq_per_cps=detector["k_bq_per_cps"],
        sigma_k_bq_per_cps=detector["sigma_k_bq_per_cps"],
    )

    fit_df = select_fit_window(binned, config["fit_tmin_min"],
    config["fit_tmax_min"])
    single_fit = fit_single_decay(fit_df)
    double_fit = fit_double_decay(fit_df)

    plot_raw_and_binned(raw_df, binned, config["label"], outdir /
    "air_raw_binned.png")
    plot_activity_linear(
        binned,
        config["label"],
        outdir / "air_activity_linear.png",
        single_fit=single_fit,
        double_fit=double_fit,
```

```
    )
    plot_activity_log(
        binned,
        config["label"],
        outdir / "air_activity_log.png",
        single_fit=single_fit,
        double_fit=double_fit,
    )

    binned.to_csv(outdir / "air_binned.csv", index=False)

    result: Dict[str, Any] = {
        "name": "air",
        "label": config["label"],
        "detector": config["detector"],
        "bin_s": config["bin_s"],
        "n_raw_rows": len(raw_df),
        "duration_h": raw_df["elapsed_h"].iloc[-1],
        "single_fit_ok": single_fit is not None,
        "double_fit_ok": double_fit is not None,
    }

    print("=" * 70)
    print(f"Analyzing: {config['label']}")
    print("=" * 70)
    print(f"Folder                 : {folder}")
    print(f"Detector               : {config['detector']}")
    print(f"Background cps         : {pm_str(detector['bg_cps'],
    detector['sigma_bg_cps'], 6)}")
    print(f"k factor [Bq/cps]      : {pm_str(detector['k_bq_per_cps'],
    detector['sigma_k_bq_per_cps'], 6)}")
    print(f"Rows read              : {len(raw_df)}")
    print(f"Duration [h]           : {raw_df['elapsed_h'].iloc[-1]:.3f}")
    print(f"Bin size [s]           : {config['bin_s']}")

    if single_fit is not None:
        t12_min = single_fit["t12_s"] / 60.0
        sigma_t12_min = single_fit["sigma_t12_s"] / 60.0

        result["single_a0_bq"] = single_fit["a0_bq"]
        result["single_sigma_a0_bq"] = single_fit["sigma_a0_bq"]
```

```
        result["single_t12_min"] = t12_min
        result["single_sigma_t12_min"] = sigma_t12_min

        print("Single exponential fit : OK")
        print(f"A0 [Bq]                : {pm_str(single_fit['a0_bq'],
    single_fit['sigma_a0_bq'], 3)}")
        print(f"T1/2 [min]             : {pm_str(t12_min, sigma_t12_min, 3)}")
    else:
        print("Single exponential fit : FAILED")

    if double_fit is not None:
        t12_fast_min = double_fit["t12_1_s"] / 60.0
        sigma_t12_fast_min = double_fit["sigma_t12_1_s"] / 60.0
        t12_slow_min = double_fit["t12_2_s"] / 60.0
        sigma_t12_slow_min = double_fit["sigma_t12_2_s"] / 60.0

        result["double_t12_fast_min"] = t12_fast_min
        result["double_sigma_t12_fast_min"] = sigma_t12_fast_min
        result["double_t12_slow_min"] = t12_slow_min
        result["double_sigma_t12_slow_min"] = sigma_t12_slow_min

        print("Double exponential fit : OK")
        print(f"Fast T1/2 [min]        : {pm_str(t12_fast_min,
    sigma_t12_fast_min, 3)}")
        print(f"Slow T1/2 [min]        : {pm_str(t12_slow_min,
    sigma_t12_slow_min, 3)}")
    else:
        print("Double exponential fit : FAILED")

    volume_m3 = config.get("volume_m3", None)
    if volume_m3 is not None and volume_m3 > 0 and single_fit is not None:
        c0_bq_per_m3 = single_fit["a0_bq"] / volume_m3
        sigma_c0_bq_per_m3 = single_fit["sigma_a0_bq"] / volume_m3

        dose_nsv_per_year = c0_bq_per_m3 * (DOSE_EPS_R + DOSE_EPS_D * DOSE_F)
    * DOSE_OCCUPANCY_H_PER_YEAR
        sigma_dose_nsv_per_year = sigma_c0_bq_per_m3 * (DOSE_EPS_R +
    DOSE_EPS_D * DOSE_F) * DOSE_OCCUPANCY_H_PER_YEAR

        result["c0_bq_per_m3"] = c0_bq_per_m3
        result["sigma_c0_bq_per_m3"] = sigma_c0_bq_per_m3
```

```
        result["dose_nsv_per_year"] = dose_nsv_per_year
        result["sigma_dose_nsv_per_year"] = sigma_dose_nsv_per_year
        result["dose_msv_per_year"] = dose_nsv_per_year / 1e6
        result["sigma_dose_msv_per_year"] = sigma_dose_nsv_per_year / 1e6

        print(f"C0 air [Bq/m^3]        : {pm_str(c0_bq_per_m3,
    sigma_c0_bq_per_m3, 4)}")
        print(f"Dose [nSv/year]        : {pm_str(dose_nsv_per_year,
    sigma_dose_nsv_per_year, 2)}")
        print(f"Dose [mSv/year]        : {pm_str(dose_nsv_per_year / 1e6,
    sigma_dose_nsv_per_year / 1e6, 6)}")
    elif single_fit is not None:
        print("C0 air [Bq/m^3]        : Volume not set")

    print(f"Reference Pb-214 [min] : {T12_PB214_MIN:.1f}")
    print(f"Reference Bi-214 [min] : {T12_BI214_MIN:.1f}")
    print(f"Saved binned CSV       : {outdir / 'air_binned.csv'}")
    print(f"Saved plots            : {outdir}")

    return result


# ============================================================
# MAIN
# ============================================================

def main() -> None:
    safe_mkdir(OUTPUT_DIR)

    print(f"Base directory         : {BASE_DIR}")
    print(f"Data directory         : {DATA_DIR}")
    print(f"Output directory       : {OUTPUT_DIR}")
    print("")

    all_results: List[Dict[str, Any]] = []

    for config in ANALYSES:
        if config["name"] == "water":
            result = analyze_water(config)
        elif config["name"] == "air":
            result = analyze_air(config)
```

```
        else:
            raise ValueError(f"Unknown analysis name: {config['name']}")

        all_results.append(result)
        print("")

    summary_df = pd.DataFrame(all_results)
    summary_path = OUTPUT_DIR / "environmental_samples_summary.csv"
    summary_df.to_csv(summary_path, index=False)

    print("=" * 70)
    print("FINAL SUMMARY")
    print("=" * 70)
    print(summary_df.to_string(index=False))
    print("")
    print(f"Summary saved to       : {summary_path}")


if __name__ == "__main__":
    main()
```

4. Ingrowth and Decay of Rn and Progeny.py

```python
from __future__ import annotations

from pathlib import Path
from typing import Optional, Dict, Any, List

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit


# ============================================================
# USER CONFIGURATION
# ============================================================

BASE_DIR = Path(__file__).resolve().parents[1]
DATA_DIR = BASE_DIR / "Messdaten"
OUTPUT_DIR = BASE_DIR / "ingrowth_decay_analysis_output"

# Detector 1: use the final corrected calibration
DETECTOR_1 = {
    "bg_cps": 0.502667,
    "sigma_bg_cps": 0.005284,
    "k_bq_per_cps": 11.503519,
    "sigma_k_bq_per_cps": 0.127739,
}

# Folder names from your current structure
CONFIG = {
    "ingrowth": {
        "label": "Ingrowth and decay 4 hours (1)",
        "folder": "Ingrowth and decay 4 hours (1)",
        "bin_s": 300,              # 5 min
        "fit_tmin_min": 0.0,
        "fit_tmax_min": 240.0,
    },
    "long_decay": {
        "label": "Ingrowth and decay long 3 days (1)",
        "folder": "Ingrowth and decay long 3 days (1)",
        "bin_s": 3600,             # 1 h
```

```
        "fit_tmin_h": 0.0,
        "fit_tmax_h": None,
        "adsorption_time_h": 2.0,  # from manual for decay measurement
    },
}

# Literature values
T12_PB214_MIN = 26.9
T12_BI214_MIN = 19.7
T12_RN222_D = 3.8235
T12_RN222_H = T12_RN222_D * 24.0


# ============================================================
# HELPERS
# ============================================================

def safe_mkdir(path: Path) -> None:
    path.mkdir(parents=True, exist_ok=True)


def pm_str(value: float, sigma: Optional[float] = None, digits: int = 3) ->
    str:
    if sigma is None or not np.isfinite(sigma):
        return f"{value:.{digits}f}"
    return f"{value:.{digits}f} pm {sigma:.{digits}f}"


def read_txt_file(path: Path) -> pd.DataFrame:
    """
    Read only the first two columns:
      1) Date/Time
      2) counts (1 sec)
    Ignore all columns from the 3rd onward.
    """
    last_error = None

    for enc in ["utf-8", "cp1252", "latin-1"]:
        try:
            df = pd.read_csv(
                path,
```

```
                sep=r"\t+",
                engine="python",
                usecols=[0, 1],
                header=0,
                encoding=enc,
            )
            break
        except UnicodeDecodeError as exc:
            last_error = exc
    else:
        raise last_error

    df.columns = ["DateTime", "counts"]

    df["DateTime"] = pd.to_datetime(
        df["DateTime"],
        format="%Y.%m.%d %H:%M:%S",
        errors="coerce"
    )
    df["counts"] = pd.to_numeric(df["counts"], errors="coerce")

    df = df.dropna(subset=["DateTime", "counts"]).copy()
    df["counts"] = df["counts"].astype(float)
    return df


def read_measurement_folder(folder: Path) -> pd.DataFrame:
    txt_files = sorted(folder.glob("*.txt"))
    if not txt_files:
        raise FileNotFoundError(f"No .txt files found in: {folder}")

    frames = []
    for txt_file in txt_files:
        df = read_txt_file(txt_file)
        df["source_file"] = txt_file.name
        frames.append(df)

    data = pd.concat(frames, ignore_index=True)
    data = data.sort_values("DateTime").drop_duplicates(subset=["DateTime"]).reset_index(drop=True)

```

```
    t0 = data["DateTime"].iloc[0]
    data["elapsed_s"] = (data["DateTime"] - t0).dt.total_seconds()
    data["elapsed_min"] = data["elapsed_s"] / 60.0
    data["elapsed_h"] = data["elapsed_s"] / 3600.0
    return data


def bin_time_series(data: pd.DataFrame, bin_s: int) -> pd.DataFrame:
    out = data.copy()
    out["bin_index"] = (out["elapsed_s"] // bin_s).astype(int)

    grouped = out.groupby("bin_index", as_index=False)

    binned = grouped.agg(
        t_start_s=("elapsed_s", "min"),
        t_end_s=("elapsed_s", "max"),
        n_rows=("counts", "size"),
        counts_sum=("counts", "sum"),
    )

    binned["t_mid_s"] = 0.5 * (binned["t_start_s"] + binned["t_end_s"])
    binned["t_mid_min"] = binned["t_mid_s"] / 60.0
    binned["t_mid_h"] = binned["t_mid_s"] / 3600.0

    binned["raw_cps"] = binned["counts_sum"] / binned["n_rows"]
    binned["sigma_raw_cps"] = np.sqrt(np.clip(binned["counts_sum"], 0.0,
    None)) / binned["n_rows"]

    return binned


def apply_background_and_calibration(
    binned: pd.DataFrame,
    bg_cps: float,
    sigma_bg_cps: float,
    k_bq_per_cps: float,
    sigma_k_bq_per_cps: float,
) -> pd.DataFrame:
    out = binned.copy()

    out["bg_cps"] = bg_cps
```

```
    out["sigma_bg_cps"] = sigma_bg_cps
    out["net_cps"] = out["raw_cps"] - bg_cps
    out["sigma_net_cps"] = np.sqrt(out["sigma_raw_cps"] ** 2 + sigma_bg_cps ** 2)

    out["activity_bq"] = out["net_cps"] * k_bq_per_cps
    out["sigma_activity_bq"] = np.sqrt(
        (k_bq_per_cps * out["sigma_net_cps"]) ** 2 +
        (out["net_cps"] * sigma_k_bq_per_cps) ** 2
    )
    return out


# ============================================================
# MODELS
# ============================================================

def model_ingrowth_bateman(t_s: np.ndarray, a_inf: float, lamb_d: float) -> np.ndarray:
    """
    Effective daughter ingrowth toward equilibrium:
        A_d(t) = A_inf * (1 - exp(-lambda_d t))
    """
    return a_inf * (1.0 - np.exp(-lamb_d * t_s))


def model_decay_single(t_s: np.ndarray, a0: float, lamb: float) -> np.ndarray:
    return a0 * np.exp(-lamb * t_s)


# ============================================================
# FITS
# ============================================================

def fit_ingrowth(df: pd.DataFrame) -> Optional[Dict[str, float]]:
    fit_df = df[(df["activity_bq"] >= 0) & (df["sigma_activity_bq"] > 0)].copy()
    if len(fit_df) < 4:
        return None

    t = fit_df["t_mid_s"].to_numpy(dtype=float)
```

```
    y = fit_df["activity_bq"].to_numpy(dtype=float)
    s = fit_df["sigma_activity_bq"].to_numpy(dtype=float)

    p0 = [max(y.max(), 1e-6), np.log(2.0) / (26.9 * 60.0)]
    bounds = ([0.0, 0.0], [np.inf, np.inf])

    try:
        popt, pcov = curve_fit(
            model_ingrowth_bateman,
            t,
            y,
            p0=p0,
            sigma=s,
            absolute_sigma=True,
            bounds=bounds,
            maxfev=20000,
        )
    except Exception:
        return None

    perr = np.sqrt(np.diag(pcov))
    a_inf, lamb_d = popt
    sigma_a_inf, sigma_lamb_d = perr

    t12_s = np.log(2.0) / lamb_d
    sigma_t12_s = np.log(2.0) * sigma_lamb_d / (lamb_d ** 2)

    # time to 99% equilibrium and 7 half-lives
    t_99_s = -np.log(0.01) / lamb_d
    sigma_t_99_s = np.log(100.0) * sigma_lamb_d / (lamb_d ** 2)
    t_7hl_s = 7.0 * t12_s
    sigma_t_7hl_s = 7.0 * sigma_t12_s

    y_fit = model_ingrowth_bateman(t, *popt)
    rss = float(np.sum((y - y_fit) ** 2))

    return {
        "a_inf_bq": a_inf,
        "sigma_a_inf_bq": sigma_a_inf,
        "lambda_d_per_s": lamb_d,
        "sigma_lambda_d_per_s": sigma_lamb_d,
```

```
        "t12_s": t12_s,
        "sigma_t12_s": sigma_t12_s,
        "t_99_s": t_99_s,
        "sigma_t_99_s": sigma_t_99_s,
        "t_7hl_s": t_7hl_s,
        "sigma_t_7hl_s": sigma_t_7hl_s,
        "rss": rss,
    }


def fit_long_decay(df: pd.DataFrame) -> Optional[Dict[str, float]]:
    fit_df = df[(df["activity_bq"] > 0) & (df["sigma_activity_bq"] > 0)].copy()
    if len(fit_df) < 4:
        return None

    t = fit_df["t_mid_s"].to_numpy(dtype=float)
    y = fit_df["activity_bq"].to_numpy(dtype=float)
    s = fit_df["sigma_activity_bq"].to_numpy(dtype=float)

    p0 = [max(y[0], y.max()), np.log(2.0) / (3.8 * 24.0 * 3600.0)]
    bounds = ([0.0, 0.0], [np.inf, np.inf])

    try:
        popt, pcov = curve_fit(
            model_decay_single,
            t,
            y,
            p0=p0,
            sigma=s,
            absolute_sigma=True,
            bounds=bounds,
            maxfev=20000,
        )
    except Exception:
        return None

    perr = np.sqrt(np.diag(pcov))
    a0, lamb = popt
    sigma_a0, sigma_lamb = perr

    t12_s = np.log(2.0) / lamb
```

```
    sigma_t12_s = np.log(2.0) * sigma_lamb / (lamb ** 2)

    y_fit = model_decay_single(t, *popt)
    rss = float(np.sum((y - y_fit) ** 2))

    return {
        "a0_bq": a0,
        "sigma_a0_bq": sigma_a0,
        "lambda_per_s": lamb,
        "sigma_lambda_per_s": sigma_lamb,
        "t12_s": t12_s,
        "sigma_t12_s": sigma_t12_s,
        "rss": rss,
    }


# ============================================================
# THEORY / DERIVED QUANTITIES
# ============================================================

def secular_equilibrium_ratio_after_n_half_lives(n: float) -> float:
    """
    Daughter ingrowth fraction for A_d/A_inf = 1 - 2^{-n}
    """
    return 1.0 - 2.0 ** (-n)


def estimate_ra226_activity_from_rn_ingrowth(a_rn0_bq: float,
    adsorption_time_h: float) -> float:
    """
    Assume:
    - secular equilibrium between 226Ra and daughters
    - 100% adsorption efficiency of 222Rn on activated carbon
    - activity accumulated during adsorption time t_ads:
        A_Rn,ads = A_Ra * (1 - exp(-lambda_Rn * t_ads))
    Therefore:
        A_Ra = A_Rn,ads / (1 - exp(-lambda_Rn * t_ads))
    """
    lambda_rn_h = np.log(2.0) / T12_RN222_H
    factor = 1.0 - np.exp(-lambda_rn_h * adsorption_time_h)
    if factor <= 0:
```

```
        return np.nan
    return a_rn0_bq / factor


# ============================================================
# PLOTTING
# ============================================================

def plot_raw_and_binned(raw_df: pd.DataFrame, binned_df: pd.DataFrame, label: str, outpath: Path) -> None:
    plt.figure(figsize=(11, 6))
    plt.plot(raw_df["elapsed_min"], raw_df["counts"], linewidth=0.8, label="Raw counts (1 s)")
    plt.plot(binned_df["t_mid_min"], binned_df["raw_cps"], linewidth=2.0, label="Binned cps")
    plt.xlabel("Elapsed time (min)")
    plt.ylabel("Counts / cps")
    plt.title(label)
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath, dpi=200)
    plt.show()
    plt.close()


def plot_ingrowth_activity(
    binned_df: pd.DataFrame,
    fit_result: Optional[Dict[str, float]],
    label: str,
    outpath: Path,
) -> None:
    t_min = binned_df["t_mid_min"].to_numpy(dtype=float)
    t_s = binned_df["t_mid_s"].to_numpy(dtype=float)
    y = binned_df["activity_bq"].to_numpy(dtype=float)
    s = binned_df["sigma_activity_bq"].to_numpy(dtype=float)

    plt.figure(figsize=(11, 6))
    plt.errorbar(t_min, y, yerr=s, fmt="o", ms=4, capsize=2, label="Binned activity")

```

```
    if fit_result is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_ingrowth_bateman(tfit, fit_result["a_inf_bq"],
    fit_result["lambda_d_per_s"])
        plt.plot(tfit / 60.0, yfit, linewidth=2.0, label="Ingrowth fit")

    plt.xlabel("Elapsed time (min)")
    plt.ylabel("Activity (Bq)")
    plt.title(label)
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath, dpi=200)
    plt.show()
    plt.close()


def plot_decay_activity(
    binned_df: pd.DataFrame,
    fit_result: Optional[Dict[str, float]],
    label: str,
    outpath_linear: Path,
    outpath_log: Path,
) -> None:
    t_h = binned_df["t_mid_h"].to_numpy(dtype=float)
    t_s = binned_df["t_mid_s"].to_numpy(dtype=float)
    y = binned_df["activity_bq"].to_numpy(dtype=float)
    s = binned_df["sigma_activity_bq"].to_numpy(dtype=float)

    plt.figure(figsize=(11, 6))
    plt.errorbar(t_h, y, yerr=s, fmt="o", ms=4, capsize=2, label="Binned
    activity")

    if fit_result is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_single(tfit, fit_result["a0_bq"],
    fit_result["lambda_per_s"])
        plt.plot(tfit / 3600.0, yfit, linewidth=2.0, label="Single-exp fit")

    plt.xlabel("Elapsed time (h)")
    plt.ylabel("Activity (Bq)")
```

```
    plt.title(label)
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath_linear, dpi=200)
    plt.show()
    plt.close()

    mask = y > 0
    plt.figure(figsize=(11, 6))
    plt.errorbar(t_h[mask], y[mask], yerr=s[mask], fmt="o", ms=4, capsize=2,
    label="Binned activity")

    if fit_result is not None:
        tfit = np.linspace(t_s.min(), t_s.max(), 500)
        yfit = model_decay_single(tfit, fit_result["a0_bq"],
    fit_result["lambda_per_s"])
        yfit = np.clip(yfit, 1e-12, None)
        plt.plot(tfit / 3600.0, yfit, linewidth=2.0, label="Single-exp fit")

    plt.yscale("log")
    plt.xlabel("Elapsed time (h)")
    plt.ylabel("Activity (Bq)")
    plt.title(label + " (log scale)")
    plt.grid(True, which="both", alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath_log, dpi=200)
    plt.show()
    plt.close()


# ============================================================
# ANALYSIS
# ============================================================

def analyze_ingrowth() -> Dict[str, Any]:
    cfg = CONFIG["ingrowth"]
    folder = DATA_DIR / cfg["folder"]
    outdir = OUTPUT_DIR / "ingrowth"
    safe_mkdir(outdir)
```

```
    raw_df = read_measurement_folder(folder)
    binned = bin_time_series(raw_df, cfg["bin_s"])
    binned = apply_background_and_calibration(
        binned,
        bg_cps=DETECTOR_1["bg_cps"],
        sigma_bg_cps=DETECTOR_1["sigma_bg_cps"],
        k_bq_per_cps=DETECTOR_1["k_bq_per_cps"],
        sigma_k_bq_per_cps=DETECTOR_1["sigma_k_bq_per_cps"],
    )

    fit_df = binned[
        (binned["t_mid_min"] >= cfg["fit_tmin_min"]) &
        (binned["t_mid_min"] <= cfg["fit_tmax_min"])
    ].copy()

    fit_result = fit_ingrowth(fit_df)

    plot_raw_and_binned(raw_df, binned, cfg["label"], outdir / "ingrowth_raw_binned.png")
    plot_ingrowth_activity(binned, fit_result, cfg["label"], outdir / "ingrowth_activity_fit.png")
    binned.to_csv(outdir / "ingrowth_binned.csv", index=False)

    result: Dict[str, Any] = {
        "section": "ingrowth",
        "label": cfg["label"],
        "bin_s": cfg["bin_s"],
        "n_raw_rows": len(raw_df),
        "duration_h": raw_df["elapsed_h"].iloc[-1],
        "fit_ok": fit_result is not None,
    }

    print("=" * 70)
    print(f"Analyzing: {cfg['label']}")
    print("=" * 70)
    print(f"Folder                    : {folder}")
    print(f"Background cps            : {pm_str(DETECTOR_1['bg_cps'], DETECTOR_1['sigma_bg_cps'], 6)}")
    print(f"k factor [Bq/cps]         : {pm_str(DETECTOR_1['k_bq_per_cps'], DETECTOR_1['sigma_k_bq_per_cps'], 6)}")
```

```
    print(f"Rows read                  : {len(raw_df)}")
    print(f"Duration [h]               : {raw_df['elapsed_h'].iloc[-1]:.3f}")
    print(f"Bin size [s]               : {cfg['bin_s']}")

    if fit_result is not None:
        t12_min = fit_result["t12_s"] / 60.0
        sigma_t12_min = fit_result["sigma_t12_s"] / 60.0
        t99_h = fit_result["t_99_s"] / 3600.0
        sigma_t99_h = fit_result["sigma_t_99_s"] / 3600.0
        t7hl_h = fit_result["t_7hl_s"] / 3600.0
        sigma_t7hl_h = fit_result["sigma_t_7hl_s"] / 3600.0

        eq_fraction_7hl = secular_equilibrium_ratio_after_n_half_lives(7.0)

        result.update({
            "a_inf_bq": fit_result["a_inf_bq"],
            "sigma_a_inf_bq": fit_result["sigma_a_inf_bq"],
            "t12_min": t12_min,
            "sigma_t12_min": sigma_t12_min,
            "t99_h": t99_h,
            "sigma_t99_h": sigma_t99_h,
            "t7hl_h": t7hl_h,
            "sigma_t7hl_h": sigma_t7hl_h,
            "eq_fraction_7hl": eq_fraction_7hl,
        })

        print("Ingrowth fit               : OK")
        print(f"A_inf [Bq]                  : {pm_str(fit_result['a_inf_bq'],
    fit_result['sigma_a_inf_bq'], 3)}")
        print(f"Effective T1/2 [min]        : {pm_str(t12_min, sigma_t12_min,
    3)}")
        print(f"Reference Pb-214 [min]      : {T12_PB214_MIN:.1f}")
        print(f"Reference Bi-214 [min]      : {T12_BI214_MIN:.1f}")
        print(f"7 half-lives [h]            : {pm_str(t7hl_h, sigma_t7hl_h, 3)}")
        print(f"99% equilibrium time [h]    : {pm_str(t99_h, sigma_t99_h, 3)}")
        print(f"Eq. fraction after 7 hl     : {eq_fraction_7hl:.6f}")
    else:
        print("Ingrowth fit               : FAILED")

    print(f"Saved binned CSV           : {outdir / 'ingrowth_binned.csv'}")
    print(f"Saved plots                : {outdir}")
```

```
511
512     return result
513
514
515 def analyze_long_decay() -> Dict[str, Any]:
516     cfg = CONFIG["long_decay"]
517     folder = DATA_DIR / cfg["folder"]
518     outdir = OUTPUT_DIR / "long_decay"
519     safe_mkdir(outdir)
520
521     raw_df = read_measurement_folder(folder)
522     binned = bin_time_series(raw_df, cfg["bin_s"])
523     binned = apply_background_and_calibration(
524         binned,
525         bg_cps=DETECTOR_1["bg_cps"],
526         sigma_bg_cps=DETECTOR_1["sigma_bg_cps"],
527         k_bq_per_cps=DETECTOR_1["k_bq_per_cps"],
528         sigma_k_bq_per_cps=DETECTOR_1["sigma_k_bq_per_cps"],
529     )
530
531     fit_df = binned[binned["t_mid_h"] >= cfg["fit_tmin_h"]].copy()
532     if cfg["fit_tmax_h"] is not None:
533         fit_df = fit_df[fit_df["t_mid_h"] <= cfg["fit_tmax_h"]].copy()
534
535     fit_result = fit_long_decay(fit_df)
536
537     plot_raw_and_binned(raw_df, binned, cfg["label"], outdir /
    "long_decay_raw_binned.png")
538     plot_decay_activity(
539         binned,
540         fit_result,
541         cfg["label"],
542         outdir / "long_decay_activity_linear.png",
543         outdir / "long_decay_activity_log.png",
544     )
545     binned.to_csv(outdir / "long_decay_binned.csv", index=False)
546
547     result: Dict[str, Any] = {
548         "section": "long_decay",
549         "label": cfg["label"],
550         "bin_s": cfg["bin_s"],
```

```
        "n_raw_rows": len(raw_df),
        "duration_h": raw_df["elapsed_h"].iloc[-1],
        "fit_ok": fit_result is not None,
    }

    print("=" * 70)
    print(f"Analyzing: {cfg['label']}")
    print("=" * 70)
    print(f"Folder                    : {folder}")
    print(f"Background cps            : {pm_str(DETECTOR_1['bg_cps'],
DETECTOR_1['sigma_bg_cps'], 6)}")
    print(f"k factor [Bq/cps]         : {pm_str(DETECTOR_1['k_bq_per_cps'],
DETECTOR_1['sigma_k_bq_per_cps'], 6)}")
    print(f"Rows read                 : {len(raw_df)}")
    print(f"Duration [h]              : {raw_df['elapsed_h'].iloc[-1]:.3f}")
    print(f"Bin size [s]              : {cfg['bin_s']}")

    if fit_result is not None:
        t12_h = fit_result["t12_s"] / 3600.0
        sigma_t12_h = fit_result["sigma_t12_s"] / 3600.0

        a_ra = estimate_ra226_activity_from_rn_ingrowth(
            fit_result["a0_bq"],
            cfg["adsorption_time_h"],
        )
        sigma_a_ra = estimate_ra226_activity_from_rn_ingrowth(
            fit_result["sigma_a0_bq"],
            cfg["adsorption_time_h"],
        )

        result.update({
            "a0_bq": fit_result["a0_bq"],
            "sigma_a0_bq": fit_result["sigma_a0_bq"],
            "t12_h": t12_h,
            "sigma_t12_h": sigma_t12_h,
            "a_ra226_bq": a_ra,
            "sigma_a_ra226_bq": sigma_a_ra,
        })

        print("Decay fit                 : OK")
```

```
        print(f"A0 [Bq]                    : {pm_str(fit_result['a0_bq'],
    fit_result['sigma_a0_bq'], 3)}")
        print(f"T1/2 [h]                   : {pm_str(t12_h, sigma_t12_h, 3)}")
        print(f"Reference Rn-222 [h]       : {T12_RN222_H:.3f}")
        print(f"Estimated Ra-226 [Bq]      : {pm_str(a_ra, sigma_a_ra, 3)}")
        print(f"Adsorption time [h]        : {cfg['adsorption_time_h']:.2f}")
    else:
        print("Decay fit                 : FAILED")

    print("Note                       : Long measurement may be only partly
    quantitative.")
    print(f"Saved binned CSV           : {outdir / 'long_decay_binned.csv'}")
    print(f"Saved plots                : {outdir}")

    return result


# ============================================================
# MAIN
# ============================================================

def main() -> None:
    safe_mkdir(OUTPUT_DIR)

    print(f"Base directory             : {BASE_DIR}")
    print(f"Data directory             : {DATA_DIR}")
    print(f"Output directory           : {OUTPUT_DIR}")
    print("")

    results: List[Dict[str, Any]] = []

    results.append(analyze_ingrowth())
    print("")
    results.append(analyze_long_decay())
    print("")

    summary_df = pd.DataFrame(results)
    summary_path = OUTPUT_DIR / "ingrowth_decay_summary.csv"
    summary_df.to_csv(summary_path, index=False)

    print("=" * 70)
```

```
    print("FINAL SUMMARY")
    print("=" * 70)
    print(summary_df.to_string(index=False))
    print("")
    print(f"Summary saved to          : {summary_path}")


if __name__ == "__main__":
    main()
```