- A safety factor ($0<S<1$) is used when a new step is calculated to further ensure that a small enough step is taken.
- When a step fails, i.e. the error bound equation is not satisfied,
  - *dtfailmin*: The new step must change by some minimum factor.
  - *dtfailmax*: The step cannot change by more than some maximum factor
  - *maxattempts*: A limit is placed on the number of times a step is retried.
  - A check must be made to ensure that the new step is larger than machine roundoff. (Check if $t+dt == t$.)
- When a step passes, i.e. $|\Delta_{est}(i)|\leq\Delta_{des}(i)|$ is satisfied,
  - *dtpassmin*: The step is not changed unless it is by some minimum factor.
  - *dtpassmax*: The step is not changed by more than some maximum factor.

The only remaining question is what to take for the initial step. In theory, any step can be taken and the stepper will adjust the step accordingly. In practice, if the step is too far off, and the error is much larger than the given tolerance, the stepper will have difficulty. So some care must be taken in choosing the initial step.

Some safeguards can also be taken during the integration by defining,

- *dtmin*: A limit placed on the smallest possible stepsize
- *maxsteps*: A limit placed on the total number of steps taken.

The Python code for the the adaptive stepsize control is discussed further in Appendix Organization.

Problem adaptive

The demos in the previous section solved the Daisyworld equations using the embedded Runge-Kutta methods with adaptive timestep control.

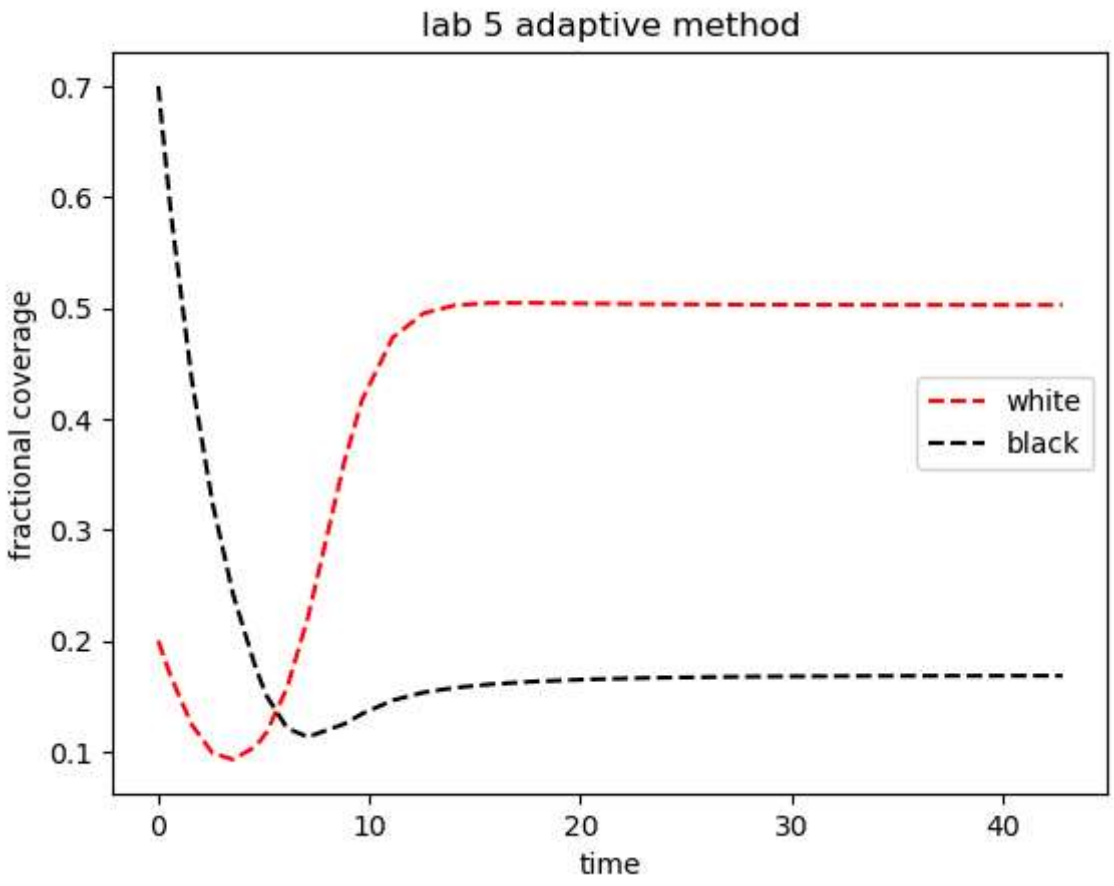
1. Run the code and find solutions of Daisyworld with the default settings found in adapt.yaml using the timeloop5Err adaptive code
2. Find the solutions again but this time with fixed stepsizes (you can copy and paste code for this if you don't want to code your own - be sure to read the earlier parts of the lab before attempting this question if you are stuck on how to do this!) and compare the solutions, the size of the timesteps, and the number of the timesteps between the fixed and adaptive timestep code.

3. Given the difference in the number of timesteps, how much faster would the fixed timeloop need to be to give the same performance as the adaptive timeloop for this case?

1. Run the yaml file

In [15]:
```
theSolver = Integ54('adapt.yaml')
timevals, yvals, errorlist = theSolver.timeloop5Err()
daisies = pd.DataFrame(yvals, columns=['white', 'black'])

thefig, theAx = plt.subplots(1, 1)
line1, = theAx.plot(timevals, daisies['white'])
line2, = theAx.plot(timevals, daisies['black'])
line1.set(linestyle='--', color='r', label='white')
line2.set(linestyle='--', color='k', label='black')
theAx.set_title('lab 5 adaptive method')
theAx.set_xlabel('time')
theAx.set_ylabel('fractional coverage')
out = theAx.legend(loc='center right')
```

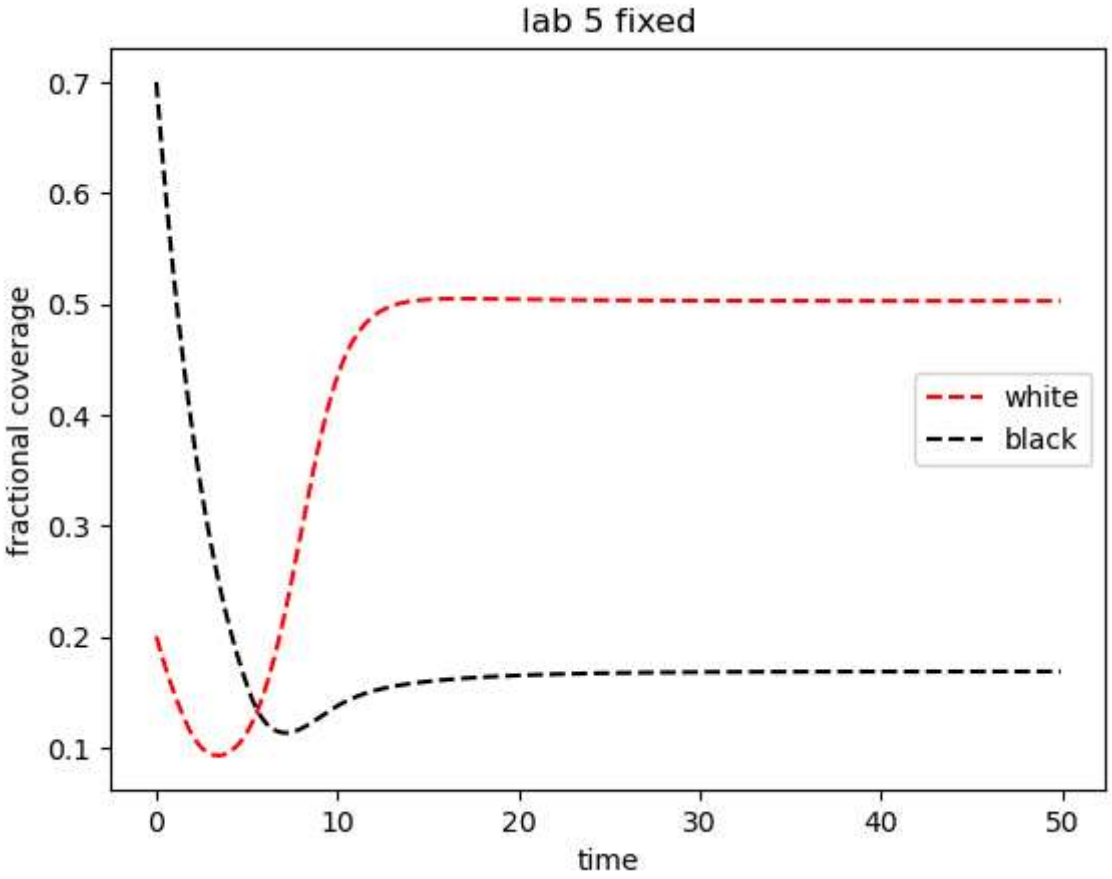


2. Run the fixed code and plot

In [16]:
```
theSolver = Integ54('adapt.yaml')
timevals_fixed, yvals, errorlist = theSolver.timeloop5fixed()
```

```
daisies = pd.DataFrame(yvals, columns=['white', 'black'])

thefig, theAx = plt.subplots(1, 1)
line1, = theAx.plot(timevals_fixed, daisies['white'])
line2, = theAx.plot(timevals_fixed, daisies['black'])
line1.set(linestyle='--', color='r', label='white')
line2.set(linestyle='--', color='k', label='black')
theAx.set_title('lab 5 fixed')
theAx.set_xlabel('time')
theAx.set_ylabel('fractional coverage')
out = theAx.legend(loc='center right')
```



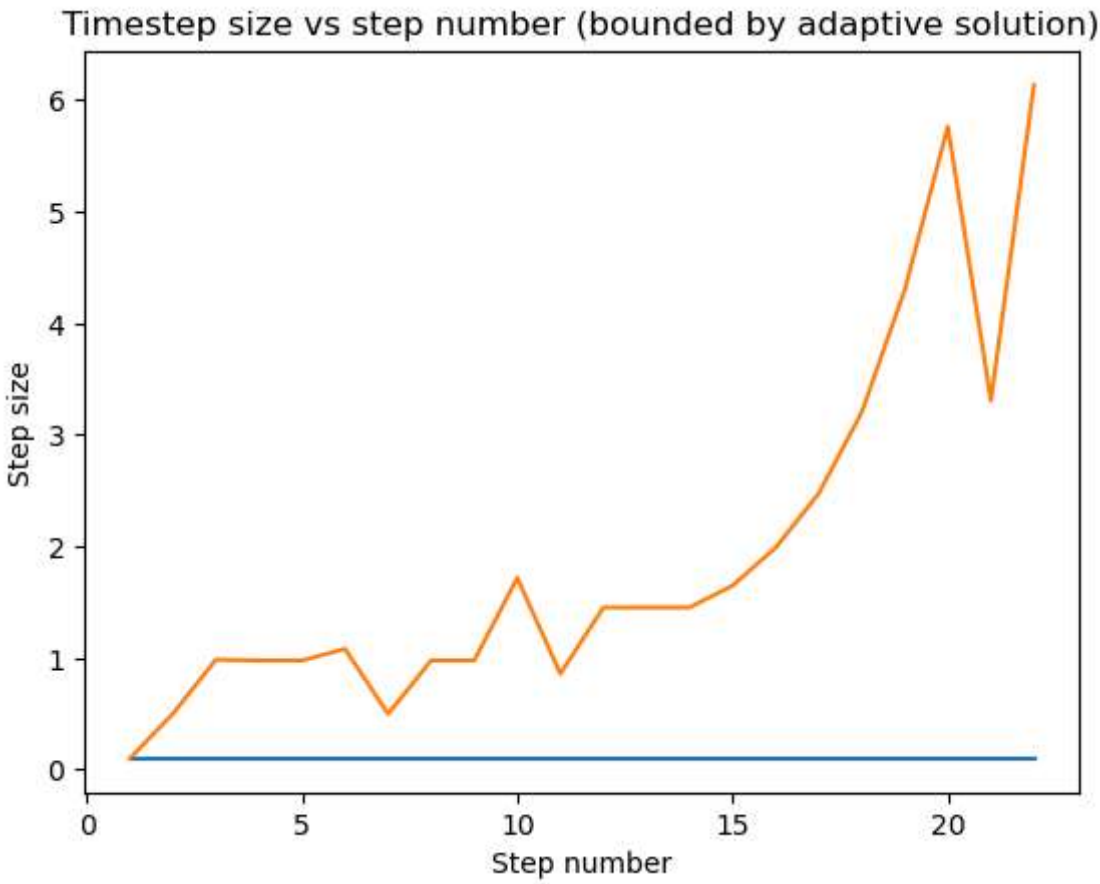
Below we compare the adaptive timesteps and number of timesteps to that of the fixed solution. We see that the adaptive solution takes MANY fewer steps to solve compared to the fixed solution, and the timesteps are on the order of 100 times larger than that of the fixed solution.

In [17]:
```
dt = []
dt_fixed = []
step_num = []
for i in range(len(timevals)-1):
    dt_fixed.append(timevals[i+1] - timevals[i])
    dt.append(0.1)
    step_num.append(i+1)

plt.plot(step_num, dt)
plt.plot(step_num, dt_fixed)
```

```
plt.title("Timestep size vs step number (bounded by adaptive solution)")
plt.xlabel("Step number")
plt.ylabel("Step size")
print(f'The total number of steps taken for fixed is {len(timevals_fixed)} and the
```

```
The total number of steps taken for fixed is 500 and the number of steps taken for
adaptive is 23
```



3. Find the computational time difference by finding the difference in number of steps for both methods if we assume that each step takes the same amount of time for both schemes.

In [18]:
```
print(abs(len(timevals)-len(timevals_fixed)))
```

```
477
```

The fixed solution would have to reduce the number of steps by 477 to be on par with the adaptive solution, or about 95% faster than it currently is.

Daisyworld Steady States

We can now use the Runge-Kutta code with adaptive timestep control to find some steady states of Daisyworld by varying the luminosity LS_0 in the uservars section of adapt.yaml and recording the daisy fractions at the end of the integration. The code was used in the earlier sections to find some adhoc steady state solutions and the effect of altering some