A random virus detector: computer practical

Peter C. de Greef, Laurens H.J. Krah & Rob J. de Boer, Theoretical Biology UU.

In this practical we work with an 'artificial immune system' composed of a large set of random detectors (as a model for lymphocyte receptors) that are defined as random strings of letters that can match 'input' strings ('kmers' of letters). These inputs are introduced (expressed) by intruders and by the 'self'-environment of our system. Random immune systems like this have been proposed as 'learning' virus scanners for computers [2]. Another elegant example for systems like this is the detection of a language by randomly sampling words from a text [4]. In our system a detector is defined to match an input kmer whenever the kmer is present within the detector. For example, the detector `abcdef` matches the 3-mers `abc` and `cde`, but not the input `bca`. The lengths of the alphabet, the detector and the kmer define the probability that a detector matches a particular input.

Consider an alphabet of A letters, input kmers of K letters, and detectors of L letters. The probability that a single kmer is matching the detector at a particular position of the detector is $p_K = (1/A)^K$, because all K letters should be identical. The number of unique kmers that can be matched by a single detector is maximally $m = 1 + L - K$, and hence the probability that a single kmer is matching the detector at any position becomes $p = 1 - (1 - p_K)^m$, i.e., one minus the probability that the kmer matches nowhere. This parameter p defines the specificity of the random immune system. If $p \to 1$, almost all kmers will be recognized by each detector, when $p \to 0$ we need a lot of detectors, i.e., a diverse repertoire, to have at least one detector matching a randomly chosen kmer. Defining R as the number of functional detectors, i.e., the diversity of the functional repertoire, a kmer is expected to be matched by one detector when $R = 1/p$.

On the webpage tbb.bio.uu.nl/rdb/practicals/scanner we provide a Python script implementing this model. The main question that you can study with this script is how one should parametrize this random virus detector to allow it to detect most viruses. The challenge is that the system is also composed of 'self'-strings that should not be attacked by the immune system (e.g., self proteins in the immune system, or native code on a computer). Let there be S self-kmers that should be tolerated. The larger S the more challenging the problem becomes, because each detector should match none of the self kmers. One can also quantify the fraction of the total space covered by the self-kmers, which is $c = S/A^K$ because there are A^K different kmers in the system. For your information, in the human body there are about 10^4 different self proteins, and if each is sampled 10 times into short peptides (called epitopes), one would estimate that $S = 10^5$ kmers. In the Python code we scale this down to $S = 10^4$ to speed up the calculations.

The script starts simple with computing and printing the probability p and the coverage c of sequence space by the S self-kmers. from the basic parameters A, K and L. Next, it defines functions returning one or n random string(s) of a particular length. To allow for a computationally efficient system, even for a very diverse repertoire of detectors (millions), we define a data structure `kmer_dict` containing all A^K kmers. This 'dictionary' is indexed by the string defining the kmer, and contains (1) a boolean variable indicating whether or not this kmer is a self-kmer, and (2) a list of all detectors in the repertoire matching this kmer (where detectors are identified by a unique integer). This would look like

```
aaa: T [13, 23, 17]
aab: F [101, 213]
...: . []
zzz: F []
```

where the first column identifies the kmer, the T or F define whether or not this kmer is contained in self, and the integers refer to the indices of the detectors. The data structure defining all detectors is a list of integers representing the state of each receptor, ordered from the first one made, to the last one. We allow for 3 states, naive '0', tolerant '1', and memory '2'. Receptors are born in a naive state when they match none of the self-kmers, they become tolerant whenever they match at least one of the self-kmers, and they turn from the naive to the memory state after matching a foreign kmer. After seeing foreign challenges the list of states, i.e., the repertoire, may look like

```
repertoire = [0, 0, 1, 0, 2, ...., 0],
```

where detectors 0 and 1 are naive, 2 is tolerant, 3 is naive, 4 is memory, and the last detector is naive.

The simulator first fills the data structure defining all kmers:

```
a_matrix = [[i for i in letters] for j in range(K)]
all_kmers = ["".join(f) for f in list(itertools.product(*a_matrix))]
kmer_dict = {kmer:[False, []] for kmer in all_kmers}
```

where `a_matrix` contains K columns of A different letters, which are combined by an iterative loop into all possible strings of K letters in `all_kmers`. These are collected into a 'dictionary' called `kmer_dict` that can be indexed by the kmer, and has the attributes 'False' and an empty list of detectors. Initially, the dictionary will look like (for $A = 3$ and $K = 2$):

```
{'aa':[False,[]], 'ab':[False,[]], ... , 'cb':[False,[]], 'cc':[False,[]]}
```

Next we define S random kmers to be 'self', e.g., words sampled from trusted documents on a computer or peptides sampled from self proteins in an organism, and set their self/non-self attribute in the kmer-dictionary to 'true':

```
self_kmers = nrandom_strings(K, S)
for kmer in self_kmers:
    kmer_dict[kmer][0] = True
```

Random repertoire. Subsequently we build a repertoire of R naive random detectors, while setting their state to 'tolerant' (1) whenever they match a least one of the self-kmers. This is done in the form of a while-loop because one cannot predict how many random detectors need to be made to acquire R naive detectors.

```
repertoire = []
detec_id = 0
n_naives = 0
while n_naives < int(R):
    Lmer = arandom_string(L) # make one new detector
    repertoire.append(0)
    for i in range(nKmers):
        kmer = Lmer[i:i+K]
        # add detector (if not already present for this k−mer):
        if not detec_id in kmer_dict[kmer][1]:
            kmer_dict[kmer][1] += [detec_id]
```

```
        # set state to tolerant if k−mer in self repertoire:
        if kmer_dict[kmer][0]:
            repertoire[detec_id] = 1
    # add to number of naive detectors if not tolerant:
    if repertoire[detec_id] == 0:
        n_naives += 1
    detec_id += 1
print("R, T and R0:",np.bincount(repertoire),detec_id)
```

where after some initialization, we keep by calling `arandom_string` until R naive detectors have been made. The state of each detector is appended to the `repertoire`, and the receptor is split up into m kmers. The index of the new detector is added to each of these kmers (if not already present), and if the kmer turns out to be a self-kmer, its state is set to tolerant. After all kmers of the new detector have been recorded and checked for selfness, the number of naive detectors is increased by one if the detector has not been tolerized by any of its kmers. At the end of this block there will R naive detectors in the repertoire, and an unknown number, T, of tolerized receptors. These numbers, with total number of receptors tried, $R_0 = R + T$, is printed by the last statement. The system is wasteful whenever $R_0 \gg R$.

Challenging intruders. Finally, we start challenging the system with foreign 'intruders' each composed of 10 random kmers until the system fails to respond and 'dies'. After some initializations, we enter a while-loop during which a novel pathogen with 10 kmers is created by calling `nrandom_strings(K,10)`. The detectors matching one of these kmers are aggregated into the list `responders`. Subsequently, the states of the matching detectors are collected (in `resp_states`), and aggregated into a distribution of the number of naive, tolerant, and memory detectors involved in the response to this intruder:

```
n_intruders = 0
alive = True
while alive:
    n_intruders += 1
    intruder = nrandom_strings(K, 10)
    responders = []
    for kmer in intruder:
        responders += kmer_dict[kmer][1]
    resp_states = [repertoire[i] for i in responders]
    distribution = np.bincount(resp_states, minlength = 3)
```

Next a decision is made whether or not to respond to the intruder. In the code below the system responds to the pathogen if, and only if, there is at least one naive or memory detector responding (the tolerant receptors for now do not play any role):

```
if distribution[0] + distribution[2] > 0: # respond
        response = True
        for i in responders: # update states
            if repertoire[i] == 0:
                repertoire[i] = 2
    else: # no response
        alive = False
        print("No response to intruder %i"%n_intruders)
```

Thus, if a decision to respond is made, the naive detectors that were involved acquire a memory state, and if the system fails to respond, it dies and exists the while-loop.

Practical. The simulator can be downloaded as a Jupyter notebook (scanner.ipynb).During the practical first make yourself familiar with this code (e.g., by setting the A, K and L parameters to small values and printing some of the data structures. After that study the following questions.

1. Specificity
The analytical model presented in the lecture proved that a random immune system operates most efficiently when $p \simeq 1/S$. Otherwise the probability of mounting an immune response becomes too low, or the total number of receptors that has to subjected to the tolerance process ($R_0 = R + T$) becomes prohibitively high (see the Appendix). In the simulator the specificity is solely determined by the number of letters in the alphabet (A), and the lengths of the kmer (K) and detector (L) strings. Run the first block of code to study how the specificity, p, and the coverage of the sequence space by the self kmers, c, depend on the basic parameters A, K, and L. Can you confirm the results of the analytical model by changing the diversity and the specificity of the system, and running the full code to study how many intruders the system survives? Hint: by changing A, K, and L, and running the first block of code, one can just search for a desired specificity by trial and error. Alternatively, try this Mathematica notebook (scanner.nb), that computes L for a given p, A and K).

2. Coverage
A new property of this model is the coverage, c, of the sequence space by the self-kmers, which is defined by S, K, and A. Can you find parameter settings that are similar in specificity but different in this coverage? Does the survival of intruders depend on the coverage? Can you explain why? Remember that the coverage is defined as $c = S/A^K$. Can you find an expression for the number of intruders the system is expected to survive? What are the relative roles of specificity and coverage in the number of challenges this random immune system can survive?

3. Immune system
The T cell immune system works with kmers of $K = 5$ amino acids (i.e., 20 letters), and contains about $S = 10^5$ self-kmers. One of the smallest vertebrates is the fish species *Paedocypris*, which is known to have no more than $R = 37000$ T cells (lymphocytes with a random receptor, like a random detector) [3]. Can this indeed be a functional immune system, or do the fish have an enormous waste of useless tolerant receptors to make this work?

Appendix. A simple mathematical model for the fraction of detectors escaping from tolerization is

$$R = R_0(1-p)^S \ , \tag{1}$$

where one writes that the diversity of the functional repertoire, R, is the size of the total number of detectors that is tested, R_0, times the probability that a detector does not match any of the S self-kmers [1]. Remember that p is the probability of matching a kmer. One could argue that this functional repertoire becomes protective when every foreign kmer has a fair chance to be recognized by the repertoire as a whole. With a recognition probability of p per detector, this is achieved when $R \simeq 1/p$, as at that diversity each kmer is expected to be recognized by exactly one clone. To address the question how diverse R_0 would have

to be, we substitute $R = 1/p$ in Eq. (1) and solve for R_0,

$$\frac{1}{p} = R_0(1-p)^S \simeq R_0 \mathrm{e}^{-pS} \quad \text{or} \quad R_0 \simeq \frac{\mathrm{e}^{pS}}{p} \ . \tag{2}$$

Plotting R_0 as a function of p reveals that this function has a minimum at $p = 1/S$, and that at this minimum $R_0 = S\mathrm{e}^1$, i.e., one needs to make $\mathrm{e} = 2.73$-fold more receptors than there are self-kmers in the system to have a functional repertoire. For p values larger than $1/S$, this R_0 rapidly becomes prohibitively large. Finally, note the probability of not mounting an immune response to a single foreign kmer would be $(1-p)^R$, and hence that the probability of not responding to an intruder composed of 10 foreign kmers is $(1-p)^{10pR} \simeq \mathrm{e}^{-10pR}$.

References

[1] **De Boer, R. J. and Perelson, A. S.**, 1993. How diverse should the immune system be? Proc. R. Soc. Lond., B, Biol. Sci. **252**:171–175.

[2] **Forrest, S., Hofmeyr, S. A., and Somayaji, A.**, 1997. Computer immunology. Commun. ACM **40**:88–96.

[3] **Giorgetti, O. B., Shingate, P., O'Meara, C. P., Ravi, V., Pillai, N. E., Tay, B. H., Prasad, A., Iwanami, N., Tan, H. H., Schorpp, M., Venkatesh, B., and Boehm, T.**, 2021. Antigen receptor repertoires of one of the smallest known vertebrates. Sci. Adv. **7**.

[4] **Wortel, I. M. N., Kesmir, C., De Boer, R. J., Mandl, J. N., and Textor, J.**, 2020. Is T Cell Negative Selection a Learning Algorithm? Cells **9**.

December 14, 2021