Supporting Information

Replacing Uridine with 2-Thio-Uridine Enhances the Rate and Fidelity of Nonenzymatic RNA Primer Extension

Benjamin D. Heuberger,[†§] Ayan Pal,[§] Francesca Del Frate, Ved V. Topkar, and Jack W. Szostak*

Howard Hughes Medical Institute, Center for Computational and Integrative Biology, and Department of Molecular Biology, Simches Research Center, Massachusetts General Hospital, Boston, MA 02114

[†]Present address: Lathrop & Gage, LLP, 28 State St., Boston, MA 02109

*szostak@molbio.mgh.harvard.edu

Table of Contents

1. Sequences of oligonucleotides used in this study (RNA, *DNA*)

Primers:
P1: Cy5-$^{5'}$GCG UAG ACU GAC UGG$^{3'}$
P2: BiotinTEG-$^{5'}$*AAT GAT ACG GCG ACC ACC GAG ATC TAC ACG TTC AGA GTT CTA CAG TCC GAC GAT C*GC GUA GAC UGA CUG G$^{3'}$

Templates:
T1: $^{5'}$AAA AAA CCA GUC AGU CUA CGC$^{3'}$
T2: $^{5'}$UUU UUU CCA GUC AGU CUA CGC$^{3'}$
T3: $^{5'}$s^{2}Us^{2}Us^{2}Us ^{2}Us^{2}Us^{2}U CCA GUC AGU CUA CGC$^{3'}$
T4: $^{5'}$s^{2}Ts^{2}Ts^{2}T s^{2}Ts^{2}Ts^{2}T CCA GUC AGU CUA CGC$^{3'}$
T5: $^{5'}$CCC CCU CCA GUC AGU CUA CGC$^{3'}$
T6: $^{5'}$CCC CCs^{2}U CCA GUC AGU CUA CGC$^{3'}$
T7: $^{5'}$CCC CCs^{2}T CCA GUC AGU CUA CGC$^{3'}$
T8: $^{5'}$CCC CCA CCA GUC AGU CUA CGC$^{3'}$
T9: $^{5'}$GAG AGA CCA GUC AGU CUA CGC$^{3'}$

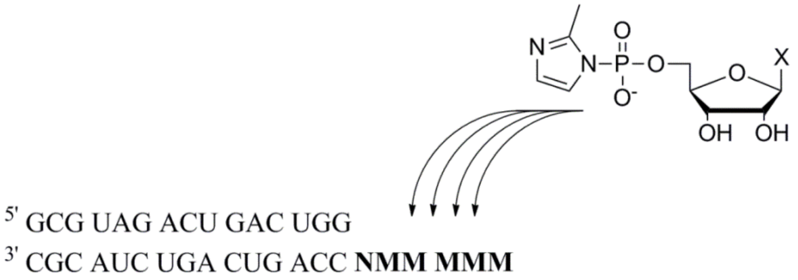
Adaptors and Primers for Sequence Analysis:

Adaptor ligated to the 3'-end of the extended primers for NGS: $^{5'}$AGA TCG GAA GAG CAC ACG TCT$^{3'}$-$^{3'}$T$^{5'}$
RT primer for making the library: $^{5'}$AGA CGT GTG CTC TTC CGA TCT$^{3'}$
PCR primers used to amplify the library: $^{5'}$AAT GAT ACG GCG ACC ACC GAG ATC TAC ACG TTC AGA GTT CTA CAG TCC G-s-A$^{3'}$ (where -s- indicates phosphorothioate bond)

2. Supporting Figure

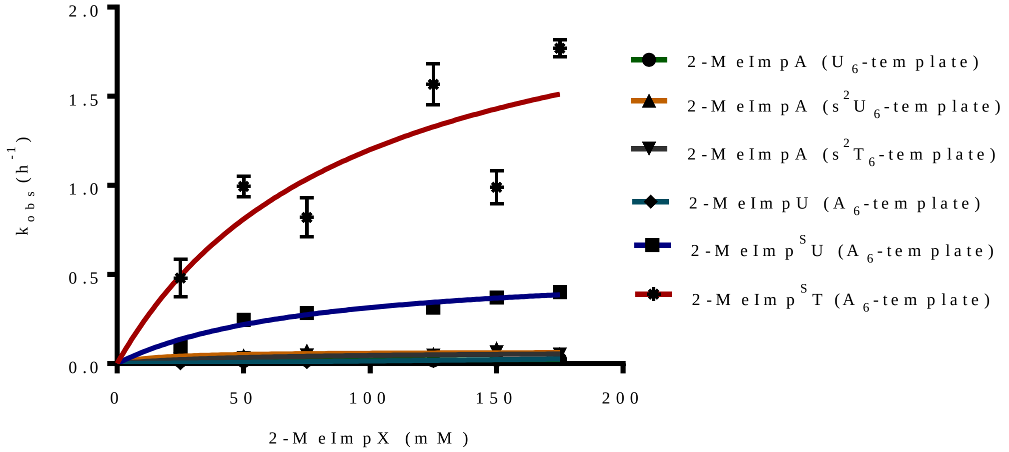
Supporting Figure 1. Kinetics of Nonenzymatic Primer Extension Reactions

a. General scheme for nonenzymatic primer extension.

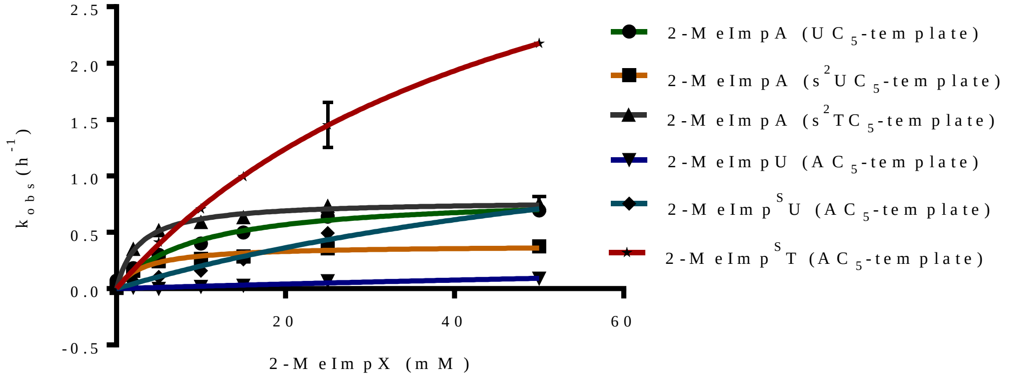


b. Observed rates vs. monomer concentration.

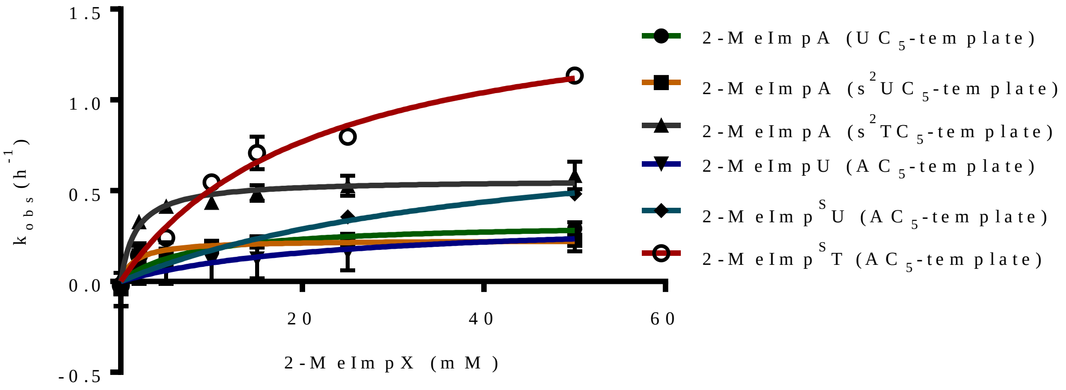
(1) M=N



(2) M=C, high salt buffer 1.



(3) M=C, low salt buffer 2.



Supporting Figure 1. a) Schematic representation of nonenzymatic primer extension reaction. b) k_{obs} vs concentration of monomer curves. Reaction Condition: 200 mM HEPES pH 7.0, 0.5 μM P1, 1.5 μM template, on ice and (1) 1.0 M NaCl, 200 mM $MgCl_2$; (2) 1.0 M NaCl, 200 mM $MgCl_2$, 40 mM 2-MeImpG; (3) 100 mM $MgCl_2$, 40 mM 2-MeImpG. The curves are fitted to the following equation:

$k_{obs} = k_{max} * [\text{2-MeImpX}] / (K_d + [\text{2-MeImpX}])$

3. Supporting Tables

Supporting Table 1: Distribution of Products shown in Figure 4

1a.

Product	U	s^{2}U	s^{2}T
P2	3.2	0.05	0.08
P2-Y	5.97	0.21	0.24
P2-U*Y	69.84	7.9	8.01
P2-U*CY	14.18	20.29	10.45
P2-U*CU*Y	6.33	31.9	16.93
P2-U*CU*CY	0.38	19.39	21.6
P2-U*CU*CU*Y	0.1	20.26	42.68

1b.

Product	2-MeImpU	2-MeImps^{2}U	2-MeImps^{2}T
P2	0.05	0.62	0.23
P2-R	0.07	0.1	0.13
P2-AR	28.98	24.94	16.01
P2-AGR	40.09	42.19	45.96
P2-AGGR	5.46	5.72	4.86
P2-AGGGR	19.36	20.14	24.58
P2-AGGGGR	5.99	6.29	8.24

Supporting Table 1: Sequencing analysis data of experiment depicted in Fig. 4. Values represent percentage of products obtained in the primer extension reaction.

Supporting Table 2: Distribution of Products shown in Figure 5

3'-AGAGAG template		2-MeImpU + 2-MeImpC		2-MeImps^{2}U + 2-MeImpC		2-MeImps^{2}T + 2-MeImpC	
		Percentage	Sequence Count	Percentage	Sequence Count	Percentage	Sequence Count
Positions 1,2	U,C	94.95	53134	97.53	756413	97.21	617154
	U,U	4.28	2394	1.59	12313	2.03	12856
	C,C	0.38	210	0.51	3919	0.63	4011
	C,U	0.40	222	0.38	2928	0.13	836
Positions 1,2,3	U,C,U	80.67	10566	95.50	675669	97.00	562410
	U,C,C	19.17	2511	4.27	30239	2.85	16553
	U,U,U	0.10	13	0.17	1180	0.12	706
	U,U,C	0.06	8	0.06	411	0.03	836

Supporting Table 2: Sequencing analysis data of experiment depicted in Fig. 5.

4. Sequence analysis

The following Python script processes the FASTQ output files from the Illumina sequencing and generates an output file containing all primer extension sequences and their corresponding read counts.

```
from __future__ import division
from Bio.Seq import Seq
import Bio.SeqIO as sio
import numpy as np
import collections
import itertools
import sys

ADAPTER = 'GCGTAGACTGACTGG'
PRIMER_EDIT_DISTANCE_THRESHOLD = 1
MAX_LENGTH = 6
TRIM = 100
EXCLUDE_BASES = ('C', 'T') if sys.argv[3] == 'AG' else ('A', 'G')
FORWARD_SLACK = True

#Check if infile is specified
try:
    ForwardReadFileName = sys.argv[1]
    ReverseReadFileName = sys.argv[2]
except:
    print 'Usage: ' + str(sys.argv[0]) + ' ForwardReadFileName ReverseReadFileName'
    sys.exit(1)

# A dynamic programming implementation of Levenshtein edit distance calculation
def levenshtein_distance(source, target):
    if source == target:
        return 0
    if len(source) < len(target):
        return levenshtein_distance(target, source)

    if len(target) == 0:
        return len(source)

    # Tuple() forces strings to be used as sequences
    source = np.array(tuple(source))
    target = np.array(tuple(target))

    # This is a fancy dynamic programming algorithm
    # Note that this is optimized so that we only really need the last 2 rows of the matrix

    previous_row = np.arange(target.size + 1)
    for s in source:

        # Insertion (target grows longer than source):
        current_row = previous_row + 1

        # Substitution or matching:
        # Target and source items are aligned, and either
        # are different (cost of 1), or are the same (cost of 0).
        current_row[1:] = np.minimum(current_row[1:], np.add(previous_row[:-1], target != s))

        # Deletion (target grows shorter than source):
        current_row[1:] = np.minimum(current_row[1:], current_row[0:-1] + 1)

        previous_row = current_row

    return previous_row[-1]

# Define output outfile prefix as the infile minus the .fastq extension
```

```
outPrefix = sys.argv[1].split('.fastq')[0]

# Initialize a counter to be used to count the matching sequences
sequence_counter = collections.Counter()
no_adapter_count = 0
high_edit_distance = 0
excluded_base_count = 0
bad_adapter_count = 0

# Iterate over both
with open(ForwardReadFileName) as ForwardRead, open(ReverseReadFileName) as
ReverseRead:

    # use the Biopython fastq parser and iterate through both files simultaneously
using izip
    for i, forward_record, reverse_record in itertools.izip(itertools.count(),
sio.parse(ForwardRead, 'fastq'), sio.parse(ReverseRead, 'fastq')):

        # for convenience, cast both reads into BioPython Seq type
        # NOTE: the reverse read is reverse-complemented here
        for_seq = forward_record.seq
        rev_seq = reverse_record.seq.reverse_complement()

        # if either sequence doesn't contain the adapter, skip this iteration
        if rev_seq.find(ADAPTER) == -1:
            no_adapter_count += 1
            continue

        if not FORWARD_SLACK:
            if for_seq.find(ADAPTER) == -1:
                no_adapter_count += 1
                continue

        # isolate the actual sequence in the reverse read by finding the sequence
after the adapter
        reverse_read_seq = rev_seq.split(ADAPTER)[-1]
        # if it is longer than 6 bases, clip it to 6 bases
        if len(reverse_read_seq) > MAX_LENGTH:
            reverse_read_seq = reverse_read_seq[:MAX_LENGTH]

        # check if the sequence has a C or T, skip this iteration if it does
        if any(base in reverse_read_seq for base in EXCLUDE_BASES):
            excluded_base_count += 1
            continue


        distance = levenshtein_distance(ADAPTER, for_seq[:16])

        if  distance > PRIMER_EDIT_DISTANCE_THRESHOLD:
            bad_adapter_count += 1
            continue

        # isolate the actual sequence in the forward read by finding the sequence
after the adapter of the same length as the reverse_read_seq
        if not FORWARD_SLACK:
            forward_read_seq = for_seq.split(ADAPTER)[-1][:len(reverse_read_seq)]
        else:
            forward_read_seq = for_seq[15:len(reverse_read_seq)+15]

        # Compare the forward and reverse read sequences and increment the counter if
        # they are within the Edit Distance Threshold of one another
        if str(forward_read_seq) != str(reverse_read_seq):
            high_edit_distance += 1
            continue
        else:
            sequence_counter[reverse_read_seq.tostring()] += 1

        if i % 1000 == 0:
            print i
```

```
    # Compute basic statistics
    num_thrown_out = i - sum(sequence_counter.values())
    sys.stdout.flush()


with open(outPrefix + '_counts.txt', 'w') as outFile:
    for record, count in sequence_counter.most_common():
        outFile.write("%d %s\n" % (count, record))


    outFile.write("\n%d of %d (%f %%) sequences thrown out" % (num_thrown_out, i, 
100*(num_thrown_out/i)))
    outFile.write("\n%d lacked exact adapters, %d lacked fuzzily matched adapters,%d 
had mismatched forward and reverse reads, %d had an incorrect base" % 
(no_adapter_count, bad_adapter_count,  high_edit_distance, excluded_base_count))
```