Benchmarking Time-Series Cross Sectional Methods: Synthetic Data Approach

Introduction

Measuring the causal impact of an intervention has long been a central interest for econometricians, and has more recently become a focus in industry, where large scale experiments are frequently run to inform strategic decisions. In this world, the data often contain a large time series component on the outcome of interest, with few relevant covariates, staggered treatment times, and many more control/donor units than treated. Typical empirical applications rely on panel models such as the Synthetic Control Method - henceforth SCM - (Abadie, Diamond, and Hainmueller, 2010; 2015) which impute the unobserved potential outcome for the treated units and compute the Average Treatment Effect on the Treated (ATT) as the average difference between observed and counterfactual outcomes.

The original formulation of SCM predicted counterfactual outcomes as a convex combination of donor units, with weights determined by goodness-of-fit on lagged outcome and auxiliary covariates. However, a number of alternatives have since been proposed, including the introduction of time weights as well as unit weights in Athey et al. (2019)'s SCDID, or Ben-Michael et al (2019)'s bias corrected Augmented Synthetic Control Method (ASCM). More generally, researchers have suggested numerous ways to optimally impute counterfactual outcomes, ranging from Matrix Completion (MC) methods common in computer science (Athey et al. 2018), modeling the outcome as composed of latent factors and estimating these time-varying factors and individual specific loadings (Bai, 2009; Gobillon and Magnac, 2016; Xu, 2017), or using Bayesian Structural Time Series models, as in CausalImpact (Scott and Varian, 2014; Brodersen et al, 2015).

While this expansive and expanding toolkit provides practitioners with an abundance of choice, it can be hard for practicioners to determine which method is most suitable for their specific purpose. In this paper, our goal is to provide a systematic empirical comparison of these methods as well as an R package intended to guide users to methodologies most suited to their data. Our Monte Carlo comparison relies on a common synthetic DGP framework nesting selection into treatment (random assignment, selection on observables, selection on unobservables), overlap in treated and control time series, heterogeneity in treatment impact and decay, and heterogeneity in the auto-correlation of outcomes. Because the number of potential models is quite large, we focus on a subset primarily based on existing package infrastructure in R – namely CausalImpact, Gsynth, Matrix Completion, and SCDID.

Several papers provide a starting point for our analysis, most notably Gobillon and Magnac (2016) who conduct Monte Carlo experiments in a similar envirionment, comparing several variations of Bai (2009)'s IFE model[1], as well as SCM and standard Difference-in-differences. They conclude that each method is unbiased in their baseline case with random assignment to treatment, but upon introducing correlations between loadings and treatment assignment, all methods become severely biased except for standard IFE and IFE with a treatment dummy[2].

In a similar vein, Xu (2017) reports results comparing IFE and SCM to their proposed GSC (Gsynth) estimator, which first estimates latent factors using donor units, then estimates factor loadings for each treated observation, and ultimately combines the two to compute counterfactual. The Monte Carlo experiments, using a modified version of the DGP in Bai (2009) and Gobillon and Magnac (2016), suggest that the Gsynth method is less biased than traditional difference-in-differences when there exist unobservable, decomposable

[1]These include standard IFE to compute counterfactual, IFE with a treatment dummy, IFE with counterfactuals based on matches from estimated factor loadings, IFE with counterfactual constrained to SCM weights based on factor loadings

[2]Difference-in-differences is also unbiased here, but the authors demonstrate that this is an artifact of their main DGP.

time-varying confounders, and is less biased than IFE under heterogeneous treatment effects. Lastly, they show that Gsynth tends to be more efficient than SCM, and demonstrate the robustness of these results by recomputing the metrics using cross-validation to estimate the number of factors instead of assuming the correct model specification.

Finally, Gardeazabal and Vega-Bayo (2017) sample from existing data as well as their synthetic DGP to compare SCM to a panel data approach (IFE) by Hsiao et al (2012) under different circumstances. Interestingly, they explore how robust the methodologies are to changes in the donor pool – selecting the subsample of donors from those that have positive weights according to SCM or those that are statististically significant under the IFE approach – and conclude that SCM is more robust to these alterations. However, in simulation studies focusing on cases with bad pre-treatment matches (defined by Mean Absolute Error, MAE), the results suggest that SCM is much more biased, while IFE models are unbiased but have large confidence intervals.

Our benchmarking exercise adds to this existing literature in several ways. First, we provide a more expansive set of comparisons by introducing generalized models on a common DGP; SCDID generalizates both Difference-in-differences and SCM; the Gsynth package (Xu, 2017) provides implementations for one or two-way fixed effects, IFE using EM (Gobillon and Magnac, 2016), Matrix Completion (Athey et al 2018), as well as the Gsynth algorithm; CausalImpact covers a large number of time series models under the umbrella of BSTS (including ARIMA models). Second, the flexibility of our DGP allows us to examine a number of interesting cases with the change of input parameters. Third, inspired by an expansive literature on forecasting time-series, we provide a simple characterization of the data using time-series features such as ACF and entropy, and analyze whether differences in these features are predictive of differences in methodology performance (Kang et al 2017). Fourth, borrowing from machine learning, we study whether ensembles can provide performance boosts in this context (Hastie et al, 2013; Athey et al, 2019).

The rest of the paper proceeds as follows. Section @ref(sec:methods) provides a brief overview of each of the methods we analyze herein, with appropriate references for further information. Section @ref(sec:dgp) provides an annotated walkthrough our DGP, which serves as the basis for all of our MCMC simulations.

Methods Overview

Notation

Throughout the paper, we refer to individuals as $i \in (1, 2, \dots, N)$ and time periods as $t \in (1, 2, \dots, T)$. A subset of the N units are treated at varying times, and once treated, remain so for the duration. Thus, each unit can be decomposed into treated (N_{tr}) and control (N_c) such that $N = N_{tr} + N_c$. The total number of time periods, T, can be similarly decomposed into the pre-treatment periods, $t \in (1, 2, \dots, T_0)$, and post-treatment periods ($t \in (T_0 + 1, T_0 + 2, \dots, T)$) given the time $T_0 + 1$ that unit i is first treated. Let D_{it} represent whether unit i received treatment in time t, so $D_{it} = 1$ if $i \in N_{tr}, t \geq T_{0_i}$, and 0 otherwise. Potential outcomes for unit i at time t under control and treatment respectively are $Y_{it}(0), Y_{it}(1)$ while observed outcomes are

$$Y_{it} = \begin{cases} Y_{it}(0) & D_{it} = 0 \\ Y_{it}(1) & D_{it} = 1, \end{cases}$$

which we can rewrite in the Rubin causal framework as $Y_{it} = (1 - D_{it})Y_{it}(0) + D_{it}Y_{it}(1)$ (Rubin, 1974).

DGP

In this section, we outline the DGP used as the basis of our benchmarking exercises. We begin by giving a highlighting the key features of our DGP in a high-level overview, and then go into code and details in a separate subsection for interested readers.

Key Features

Our data is generating from the following factor augmented autoregressive process of order one:

$$
\begin{aligned}
lnY_{it}(0) &= \alpha_i + \rho_i lnY_{i,t-1}(0) + F_t\lambda_i' + \epsilon_{it} \\
lnY_{it}(1) &= lnY_{it}(0) + D_{it}\tau_{it}.
\end{aligned}
$$

Each unit has an individual specific intercept (α_i), auto-correlation coefficient (ρ_i), as well as factor loadings, (λ_i, dimension $1 \times r$, where r is the number of unobserved factors) which scale unobserved, time-varying factors (F_t, dimension $1 \times r$ for a given t). The noise term, ϵ_{it} is iid $\mathcal{N} \sim (0, 0.5^2)$. Finally, treatment impact is modeled as a shock to counterfactual outcomes, with our binary indicator D_{it} encoding treatment status and τ_{it} representing the impact of treatment for unit i at time t.

We allow for a number of treatment assignment mechanisms, such as random assignment ($E(\alpha_i|D_{it} = 1) = E(\alpha_i|D_{it} = 0)$ as well as for ρ_i, λ_i), selection on observables (D_{it} is correlated with at least one of $\alpha_i, \rho_i, \lambda_i$), and selection on unobservables [3]. Extreme observations can also be made more frequent by tweaking an overlap parameter, which shifts the distribution of covariates[4] for a fraction of observations, and then randomly assigns treatment.[5]

Furthermore, we allow for varying degrees of heterogeneity in both treatment impact and decay. Our treatment effect $\tau_{it} = \omega_i * \delta_{it}$ flexibly covers many cases, including impact decay by setting (for example) $\delta_{it} = \delta_i^{t-T_{0i}}$ – where δ_i itself can display heterogeneity – as well as impact heterogeneity via ω_i.

Implementation Details

We walk through the details of our DGP code for ease of use. The DGP has a core function that is called with all of the arguments, and then sends much of the work to various helper functions to help keep the flow clear. The following chunk of code details the inputs to the DGP; after verifying the function inputs are feasible, we determine the length of the total time period (which depends on input dates and time frequency) as well as the time when treatment is allowed to begin. We then send the relevant subset of inputs to a helper function which creates the time-invariant (unit level) variables, another relevant subset to a helper creating the time-varying variables (factors), bring the two together to form our unit by time grid, generate the counterfactual outcome process, and add the treatment to the counterfactual.

```r
#TODO(alexdkellogg): allow dgp to be covariates only or factors only
#TODO(alexdkellogg): allow num_periods directly rather than dates
#TODO(alexdkellogg): introduce coeff vector and have xBeta go into Y
factor_synthetic_dgp<-function(num_entries=2000,
                               date_start="2017-01-01",
                               first_treat="2018-07-01",
                               date_end="2020-01-01",
                               freq=c("daily", "weekly", "monthly"),
                               prop_treated=0.4,
                               treat_impact_mean=0.1,
                               treat_impact_sd=0.1,
                               treat_decay_mean=0.7,
                               treat_decay_sd=1,
                               selection=c("random", "observables",
```

[3]For now, the outcome is actually not effected by additional observables or unobservables. However, these variables have been created, selection on them is modeled, and the outcome can be made to depend on them by modeling $\alpha_i = a_i + \beta_{obs}X_{1i} + \beta_{unobs}X_{2i}$.

[4]For now, we just shift observable and unobservable X, which do not interact with Y, though we should easily be able to shift ρ, λ.

[5]I can't tell if this is helpful above and beyond traditional selection.

```
                                    "unobservables"),
                         rho=0.9,
                         rho_scale=0.2,
                         rho_shift=0,
                         cov_overlap_scale=0,
                         num_factors=3,
                         loading_scale=0,
                         intercept_scale=0,
                         seed=19){
#rescale_y_max=5e8
#conditional_impact_het=0,

#Generates tibble of long form panel data using a factor-augmented AR 1
#Each row is time period x unit unique combination.

#Args
#num_entries: number of units to be generated
#date_start: string "yyyy-mm-dd" input for the first simulated date
#first_treat: string "yyyy-mm-dd" input first treatment period.
#date_end:: string "yyy-mm-dd" input for the last simulated date
#freq: string indicating time unit, either "daily", "weekly", "monthly"
#prop_treated: proportion of entries that should receive treatment
#treat_impact_mean: initial period treatment impact mean, drawn from truncated
#   normal distribution centered here. The end points of the dist are [0,0.25]
#   by default, but shift to [a,b] where b=mean+0.25 if the mean is larger
#   than 0.25 (max of 1)  or a=mean-0.25 if mean is below 0.
#treat_impact_sd: standard deviation of the truncated normal mean impact.
#treat_decay_mean: initial period treatment decay mean, drawn from truncated
#   normal distribution centered here. The end points of the dist are [0,0.9]
#   by default, but shift to [0,1+eps] if mean>0.9. This allows for units to
#   have no decay (value of 1). Propagates as mean**(post_treat_period).
#treat_impact_sd: standard deviation of the truncated normal decay factor
#selection= string in "random", "observables", "unobservables" dictating
#   treatment assignment mechanism.
#rho: mean of truncated normal distribution of the autocorrelation of outcome,
#   with bounds [0, 0.995].
#rho_scale: standard deviation of truncated normal for the autocorrelation.
#rho_shift: multiplier on the mean rho for control units,
#   overall mean stay below 1 (rho*rho_shift<1).
#cov_overlap_scale: (-1,1) shifts distribution of covariates for a fraction
#   (prop_treated) of x variables. A value of 1 shifts the distribution for
#   treatment up on all x's, whereas -1 shifts up the distribution for donors.
#num_factors: number (3+) of time-varying, unobserved factors to simulate
#loading_scale: (-1,1) shift in factor distribution, -1 shifts loadings up
#   for control units, positive values shift loadings distribution up for
#   treated units.
#intercept_scale: (-1,1) shifts the mean of a truncated normal distribution
#   for control unit intercepts: >0 shifts the mean down (treatment is larger)
#   and <0 shift control mean above the treatment mean.
#seed: random number seed
```

```
#Output
#Long form tibble with columns for observed, potential treated, and
#  potential untreated outcomes, treatment time, and
#  the relevant x variables (loadings, intercept, observables).




#Match the arguments to valid entries
set.seed(seed)
if (missing(selection)) {
  selection <- "random"
} else{
  selection <- match.arg(selection)
}

if (missing(freq)) {
  freq <- "monthly"
} else{
  freq <- match.arg(freq)
}
#require 10% min in both treat and control
stopifnot(prop_treated >= 0.1 & prop_treated <= 0.9)
#require cov_overlap_scale to be between -1 (shift treat down) and 1
stopifnot(cov_overlap_scale <= 1 & cov_overlap_scale >= -1)
#require 3 factors of more
stopifnot(num_factors>2)
#require less than 100% TE
stopifnot(treat_impact_mean<1)

#given the dates and frequency, identify total number of periods
num_periods=time_interval_calculator(start_t=date_start, end_t=date_end,
                                     freq_t=freq)
#Store how long after start date the treatment begins
treat_start_int=time_interval_calculator(start_t=date_start,
                                          end_t=first_treat, freq_t=freq)
#Stop if there are too few pre treat periods
stopifnot(treat_start_int < 0.8*num_periods)

#
#assign covariates and treatment given selection and overlap
synth_data_unit=unit_level_simulation(n_inp=num_entries,
                             type_inp=selection,
                             cov_overlap_inp=cov_overlap_scale,
                             loading_scale_inp=loading_scale,
                             num_factors_inp=num_factors,
                             int_scale_inp=intercept_scale,
                             rho_inp=rho,
                             rho_scale_inp=rho_scale,
                             rho_shift_inp=rho_shift,
                             prop_treated_inp=prop_treated,
                             treat_start=treat_start_int,
                             num_periods_inp=num_periods,
```

```r
                              impact_mean_inp=treat_impact_mean,
                              impact_sd_inp=treat_impact_sd,
                              decay_mean_inp=treat_decay_mean,
                              decay_sd_inp=treat_decay_sd)

  #next, work on time varying characteristics
  synth_data_factors=generate_factors(num_factors_inp=num_factors,
                                      num_periods_inp=num_periods,
                                      num_entry_inp=num_entries,
                                      date_start_inp=date_start,
                                      date_end_inp=date_end,
                                      freq_inp=freq)

  #Next, we generate both potential outcomes
  #Option for factor only (y=factor*loadings), Random Effect only
  # (y=xB), or both.
  #Then, generate the treatment impact
  unit_time_grid <-tidyr::expand_grid(entry = seq_len(num_entries),
                                      time = seq_len(num_periods))

  synth_data_full=unit_time_grid %>%
    dplyr::left_join(synth_data_unit,by=c("entry")) %>%
    dplyr::left_join(synth_data_factors,by=c("time"))

  #generate the counterfactual outcomes
  synth_data_full=generate_counterfactual(synth_data_full,
                                          num_periods_inp=num_periods)



  #generate the per period impact (taking acocunt of decay)
  #TODO(alexdkellogg): add conditional_boost as option, larger TE for big cf
  synth_data_full=generate_treat_impact(data_inp=synth_data_full,
                                cond_impact_inp=conditional_impact_het)  %>%
    dplyr::mutate(y=exp(y),
                  y0=exp(y0),
                  y1=exp(y1))

  return(synth_data_full)
}
```

Individual Level Work

Similar to the overall process, the individual level helper function itself splits off tasks to more specific functions. In this case, there are several processes that need handling: treatment period assignment, treatment status assignment, covariate creation, and treatment impact assignment. The flow for these processes can be seen in the code below. These helper function aggregate the covariates, treatment assignment, and treatment start date into an N row tibble. These in hand, the process concludes by drawing individual specific treatment impact and decay rate: ω_i, δ_i. $\omega_i \sim TN[a,b](\mu_\omega, s_\omega^2)$, where μ_ω, s_ω come from user input, allowing for impact to be concentrated or dispersed. One note, mentioned before, is that the bounds of the truncated normal distribution default to $[a,b] = [0, 0.25]$, but can shift to $[0, \min(\mu_\omega + 0.25, 1)]$ if $\mu_\omega > 0.25$, or $[\mu_\omega - 0.25, 0.25]$ if $\mu_\omega < 0$. Decay rate is also drawn from a truncated normal, $\delta_i \sim TN[0,b](\mu_\delta, s_\delta^2)$, where $b = 0.9$ by default, but moves to $b = 1 + \epsilon$ to allow for no decay if the user wishes.

```
unit_level_simulation <- function(n_inp,
                                  type_inp,
                                  cov_overlap_inp,
                                  loading_scale_inp,
                                  num_factors_inp,
                                  int_scale_inp,
                                  rho_inp,
                                  rho_scale_inp,
                                  rho_shift_inp,
                                  prop_treated_inp,
                                  treat_start,
                                  num_periods_inp,
                                  impact_mean_inp,
                                  impact_sd_inp,
                                  decay_mean_inp,
                                  decay_sd_inp){
  #Gather data that is time invariant
  #First, call to assign treatment, which generates covariates
  unit_level_tib=assign_treat(n_inp=n_inp,
               type_inp=type_inp,
               cov_overlap_inp=cov_overlap_inp,
               loading_scale_inp=loading_scale_inp,
               num_factors_inp=num_factors_inp,
               rho_inp=rho_inp,
               rho_scale_inp=rho_scale_inp,
               rho_shift_inp=rho_shift_inp,
               int_scale_inp= int_scale_inp,
               prop_treated_inp=prop_treated_inp)

  #Merge in the treatment period assignment
  unit_level_tib=assign_treat_time(unit_level_tib, treat_start, num_periods_inp)
  #allow wiggle room for the TE without getting NA
  impact_ub=ifelse(impact_mean_inp<0.25,0.25,
                   min(1, impact_mean_inp+(0.25)) )
  impact_lb=ifelse(impact_mean_inp<=0,impact_mean_inp-0.25, 0)
  decay_ub=ifelse(decay_mean_inp>0.9,1.000000001, 0.9)
  #assign treatment effect impact and decay parameters per unit
  unit_level_tib=unit_level_tib %>%
    dplyr::mutate(
      treat_impact=truncnorm::rtruncnorm(n=dplyr::n(), a=impact_lb,b=impact_ub,
                                         mean=impact_mean_inp, sd=impact_sd_inp),
      treat_decay=truncnorm::rtruncnorm(n=dplyr::n(), a=0,b=decay_ub,
                                        mean=decay_mean_inp, sd=decay_sd_inp))

  return(unit_level_tib)
}
```

We start with the simplest process, which designates treatment time. Treatment time is dictated by a geometric distribution as well as user input for when treatment may start; the geometric distribution models the count of independent Bernoulli trials up to the first success when the probability of each trial is p. Depending on the number of post treatment periods available to us, we choose $p \in (0.25, 0.2, 0.15, 0.1)$ to help ensure that we have a reasonable amount of post treatment available for the majority of our observations (See Figure @ref(fig:treat-start-time) for an example).

```
assign_treat_time<-function(treat_tib_inp, treat_start, num_periods_inp){
  #Geometric distribution parameter selected based on time
  #Goal is to have most treatment assigned before end date
  geom_prob=dplyr::case_when(num_periods_inp-treat_start<=20~0.25,
                             num_periods_inp-treat_start<=30~0.2,
                             num_periods_inp-treat_start<=40~0.15,
                             TRUE~0.1 )
  #draw treatment period as first date plus geometric random variable
  return(treat_tib_inp %>%
           dplyr::mutate(
             treatment_period=treat_start+
               stats::rgeom(dplyr::n(), geom_prob)) %>%
           dplyr::group_by(entry) %>%
           dplyr::mutate(treatment_period=
                           min(treatment_period,num_periods_inp)) %>%
           dplyr::ungroup())

}
```

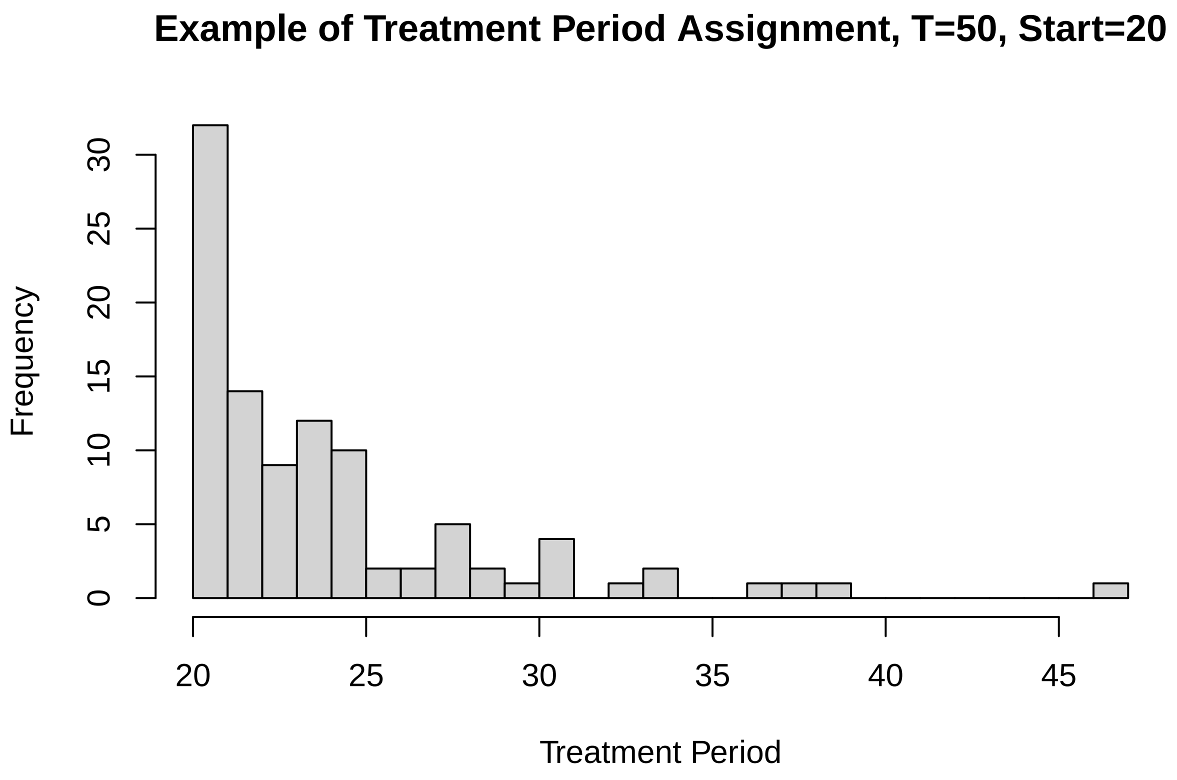

Figure 1: Assignment Example

We next discuss treatment assignment; because we want to handle selection and covariate overlap, our treatment assignment function actually handles much of the work at the individual level. Specifically, it first generates covariates with a fraction (the proportion treated) of the covariates coming from a distribution shifted according to a covariate overlap input (can be shifted up or down).[6] Next, in the case of random assignment, a random uniform variable is generated independent on covariates, and the top proportion

[6]Currently, these covariates only affect selection into treatment and are fixed in number and type. We could allow for generalization here by allowing the user to input the desired number of X's, and then randomly selecting a subset (and storing the name) for use in the selection process?

treated fraction are assigned treatment, and are passed on to another function that assigns individual specific intercepts, loadings, and autocorrelation parameter.

```r
#TODO(alexdkellogg): Generalize to allow for user inputted number of xs
gen_covariates<-function(n_inp, cov_overlap_inp, frac_shifted){
  #Generate several covariates, both observed and unobserved.
  #Shift means according to overlap_inp and frac_shifted

  #first, create a set of correlated variables, mean zero, var/covar Sigma
  Sigma <- matrix(c(16 ,  4,  -4.8,
                    4  , 25,    9,
                    -4.8,  9,    9),3,3)
  corr_xs= MASS::mvrnorm(n = n_inp, rep(0, 3), Sigma)
  colnames(corr_xs)=c("obs_mvnorm1", "unobs_mvnorm", "obs_mvnorm2")

  return(
    x_tib=tibble::tibble(
      entry = seq_len(n_inp),
      to_shift = as.numeric(dplyr::percent_rank(
        runif(n = n_inp,min = 0,max = 1)
                          )>1-frac_shifted)) %>%
    cbind(corr_xs) %>%
    dplyr::group_by(to_shift) %>%
    dplyr::mutate(
      obs_beta=stats::rbeta(n = dplyr::n(),
                          shape1 = 6,
                          shape2 = 5+3*(to_shift*-(cov_overlap_inp))),
      obs_binom=stats::rbinom(n = dplyr::n(),
                          size = 15,
                          prob = 0.5+0.25*(to_shift*(cov_overlap_inp))),
      obs_norm=stats::rnorm(n = dplyr::n(),
                           mean = 0+5*(to_shift*(cov_overlap_inp)),
                           sd = 10),
      unobs_beta=stats::rbeta(n = dplyr::n(),
                          shape1 = 6,
                          shape2 = 5+3*(to_shift*-(cov_overlap_inp))),
      ubobs_binom=stats::rbinom(n = dplyr::n(),
                           size = 15,
                           prob = 0.5+0.25*(to_shift*(cov_overlap_inp))),
      ubobs_norm=stats::rnorm(n = dplyr::n(),
                          mean = 0+5*(to_shift*(cov_overlap_inp)),
                          sd = 10) ) %>%
      dplyr::ungroup() %>%
      dplyr::select(-to_shift)
  )

}
```

Intercepts are distributed according to $\alpha_i \sim TN_{[5,10]}(\mu_\alpha, 0.7^2)$, where $\mu_\alpha = g(\rho) - (1 - D_i) * het_{int}$ is itself a function of the autocorrelation input and treatment status. In particular, the mean lies between 5.5 and 9 (decreases with ρ input), and is shifted up or down for control units according to the sign (negative or positive, respectively) of user input "intercept_scale"$=het_{int}$.[7] Individual autocorrelation is drawn from a

[7]This is done because we have a particular outcome range we wish to target, with the 25th percentile around 1000 and the 80th percentile around 10,000,000. If autocorrelation is drawn close to 1, the outcome process can explode at the top end, which we aim to avoid.

truncated normal as well, $\rho_i \sim TN_{[0,0.995]}(\mu_\rho * (1 - D_i) * het_\rho, s_\rho^2)$, with $\mu_\rho, het_\rho, s_\rho^2$ all coming from user input. This allows for homogeneity in ρ, heterogeneity alike for treated and untreated units, or differential means by treatment status. Finally, r loadings – according to the number of factors the user requested – are drawn from $\lambda_i \sim \beta(2 - (1 - D_i)het_{load}, 2 - D_i het_{load})$, so that for $het_{load} > 0$, loadings are drawn from a distribution shifted towards one for treated and shifted toward zero for control units (opposite if below 0).

```
#TODO(alexdkellogg): Do we want to adjust outcomes by autocorr (case_when)?
generate_loadings <- function(treat_tib_inp, loading_scale_inp,num_factors_inp,
                              int_scale_inp,rho_inp, rho_scale_inp, rho_shift_inp){

  loadings_mat=matrix(0, nrow=nrow(treat_tib_inp), ncol=num_factors_inp)
  colnames(loadings_mat)=glue::glue("loading{1:num_factors_inp}")



  mean_int=case_when(rho_inp<0.5~9,
                     rho_inp<0.8~7.5,
                     rho_inp<0.9~6.5,
                     TRUE~5.5)

  stopifnot(mean_int-int_scale_inp>5,
            mean_int-int_scale_inp<10)

  tib_pre_loadings=treat_tib_inp %>%
    dplyr::group_by(treated) %>%
    dplyr::mutate(
      #intercept=stats::rexp(dplyr::n(), 1+(1-treated)*int_scale_inp),
      intercept=truncnorm::rtruncnorm(n=dplyr::n(), a=5,b=10,
                                      mean=mean_int-(1-treated)*int_scale_inp,
                                      sd=0.7),
      #intercept=stats::runif(dplyr::n(), min=5,max=8-(1-treated)*int_scale_inp),
      # autocorr=stats::runif(dplyr::n(),ifelse(treated,rho_inp,
      #                                         rho_inp*rho_scale_inp),
      #                       max=0.95),
      autocorr=truncnorm::rtruncnorm(n=dplyr::n(), a=0,b=0.995,
                                     mean=ifelse(treated,rho_inp,
                                                 rho_inp*rho_shift_inp),
                                     sd=rho_scale_inp)) %>%
    dplyr::bind_cols(tibble::as_tibble(loadings_mat)) %>%
    dplyr::ungroup()
  return(
    tib_pre_loadings %>%
      dplyr::group_by(treated) %>%
      dplyr::mutate(
        dplyr::across(tidyselect::starts_with("loading"),
        .fns=list(load=~stats::rbeta(dplyr::n(), 2 -(1-treated)*(loading_scale_inp) ,
                             2 - (treated)*(loading_scale_inp)) ),
        .names ="{col}")
      ) %>% dplyr::ungroup()

  )


}
```

If instead a selection mechanism was desired, the relevant covariates (observed or unobserved) are selected, multiplied by a random normal coefficient vector, and mapped into the probability space by the logistic function. To add additional noise to the selection, a random uniform is added to the probability after this transformation (which results in scores in [0,2]), and the top proportion_treated are assigned treatment. Given this treatment assignment, we send the data off to a helper function which creates the intercept, loadings, and autocorrelation as describe above.[8] An example of shifting the loadings up fro the treated group is demonstrated in Figure @ref(fig:selection-ex-fig).

```r
assign_treat<-function( n_inp, type_inp, cov_overlap_inp,
                        loading_scale_inp,rho_inp=rho_inp,
                        rho_scale_inp,rho_shift_inp, num_factors_inp,
                        int_scale_inp, prop_treated_inp){
  #Creates a tibble with treatment assignment and time constant covariates
  #Covariate distribution can differ by treatment depending on cov_overlap_inp,
  #and selection into treatment is modelled.

  #generate covariates with varying levels of overlap
  covariate_tib=gen_covariates(n_inp=n_inp, cov_overlap_inp=cov_overlap_inp,
                               frac_shifted=prop_treated_inp)

  if (type_inp == "random") {
    #Randomly assign treatment to prop_treated_inp
    treat_covar_tib=tibble::tibble(
      entry = seq_len(n_inp),
      treated = as.numeric(dplyr::percent_rank(
        runif(n = n_inp,min = 0,max = 1)
      )>1-prop_treated_inp)) %>%
      inner_join(covariate_tib, by="entry")

    unit_tib=generate_loadings(treat_tib_inp=treat_covar_tib,
                                     loading_scale_inp=loading_scale_inp,
                               num_factors_inp=num_factors_inp,
                                     int_scale_inp=int_scale_inp,
                               rho_inp=rho_inp,
                                     rho_scale_inp=rho_scale_inp,
                               rho_shift_inp = rho_shift_inp)
    return(unit_tib)
  }
  else{
    #Assign treatment based on either observable or unobservable xs
    relevant_xs=ifelse(type_inp=="observables", "obs", "unobs")
    #First, create covariates with
    z <- covariate_tib %>%
      dplyr::select(tidyr::starts_with(relevant_xs)) %>%
      as.matrix()

    # combine the variables for each observation to form a predicted score
    # rescale with logistic function to map into the probability space (0,1)
    prob_treat= tibble::tibble(score=
                                 z %*% stats::rnorm(n = ncol(z),mean = 0,sd = 1)) %>%
      dplyr::mutate(prop_score = 1 / (1 + exp(score)))
```

[8]Note that, as of now, the selection is not technically governed the loadings/intercept/autocorrelation, but the distribution of these variables can optionally be shifted up or down by treatment status so that treatment assignment is correlated by these variables.

```
    #Ad random noise to the score and take the top fraction as treated
    treat_covar_tib= prob_treat %>%
      dplyr::mutate( u = runif(n = n_inp, min = 0, max = 1),
                     p_new = prop_score+u,
                     treated=as.numeric(dplyr::percent_rank(p_new)>
                                          1-prop_treated_inp),
                     entry = seq_len(n_inp)) %>%
      dplyr::select(entry, treated) %>%
      inner_join(covariate_tib, by="entry")

    unit_tib=generate_loadings(treat_tib_inp=treat_covar_tib,
                               loading_scale_inp=loading_scale_inp,
                               num_factors_inp=num_factors_inp,
                               int_scale_inp=int_scale_inp,
                               rho_inp=rho_inp,
                               rho_scale_inp=rho_scale_inp,
                               rho_shift_inp = rho_shift_inp)

    return(unit_tib)
  }
}
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

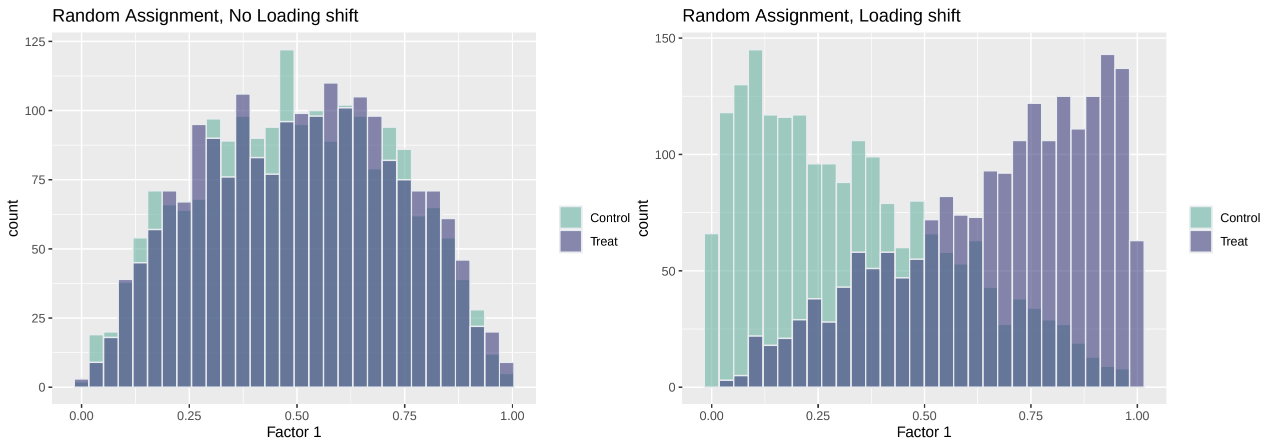

Figure 2: Overlap Example

Time-Varying Factors

The next component of our DGP is the creation of our factors, which are common across all individuals but vary over time. These are loosely modeled on existing data, from which we estimated 4 factors resembling the following processes:

$$
\begin{aligned}
F_{1t} &= t/T + \nu_{1t}\\
F_{2t} &= \rho_{F_2} F_{2,t-1} + \alpha_q + \nu_{2t}\\
F_{3t} &= \rho_{F_3} F_{3,t-1} + \alpha_m + \nu_{3t}\\
F_{4t} &= \rho_{F_4} F_{4,t-1} + \alpha_r + \nu_{4t}.
\end{aligned}
$$

At a conceptual level, the first factor represents a time trend, the second captures a seasonal component every quarter, the third a seasonal component every month, and the fourth (and beyond) represent random fixed effects at varying intervals. Mathematically, $\rho_{Fj} = 0.2\ \forall j$ so that the autoregressive component of each factor is fixed at 0.2. The shocks, $\alpha_j \sim U[-1,1]$, are iid and drawn each quarter ($j = q$), month ($j = m$), or stochastically – with the number of shocks drawn from a discrete $U[1,13]$ and their location drawn from discrete $U[1,51]$.[9] Idiosyncratic noise for each factor is drawn from $\nu_j \overset{\text{iid}}{\sim} \mathcal{N}(0, 0.1^2)$. In building the AR(1) process for each factor, we allow a burn in period of 500.

```
generate_factors<-function(num_factors_inp,
                           num_periods_inp, num_entry_inp,
                           date_start_inp, date_end_inp, freq_inp){

  #generate factors from an AR 1 process
  factor_mat <-matrix(0, nrow=num_periods_inp,ncol = num_factors_inp )
  colnames(factor_mat) <- glue::glue("factor{1:num_factors_inp}")
  #ar model description -- AR 1 with auto correlation and sd inputs
  ar_model=list(order=c(1,0,0), ar=0.2)
  #TODO(alexdkellogg): check with AP if shocks trend over time, assumed fixed
  quarter_effects=tibble::tibble(q_shock=stats::runif(4, -1, 1),
                                 quarter_num=seq_len(4))
  month_effects=tibble::tibble(m_shock=stats::runif(12, -1, 1),
                               month_num=seq_len(12))


  #combine the zero matrix of factors with date indicators
  factor_tib=generate_time_grid(date_start_inp=date_start_inp,
                                num_periods_inp=num_periods_inp,
                                freq_inp=freq_inp,
                                num_entry_inp=num_entry_inp) %>%
    dplyr::bind_cols(tibble::as_tibble(factor_mat)) %>%
    dplyr::inner_join(quarter_effects, by="quarter_num") %>%
    dplyr::inner_join(month_effects, by="month_num")


  #add first factor, period/total + noise
  factor_tib=factor_tib %>%
    dplyr::mutate(factor1=time/dplyr::n()+
                    stats::rnorm(dplyr::n(),mean=0,sd=0.1)) %>%
    dplyr::group_by(quarter_num) %>%
    dplyr::mutate(
      factor2=stats::arima.sim(model=ar_model, n=dplyr::n(),
                               innov = q_shock+stats::rnorm(dplyr::n(),
                                                            sd=0.1),
                               n.start = 500)) %>%
```

[9] In using the R-package "lubirdate", we generate a time grid with information on the quarter and month for each row, which we use to assign the shocks at the proper intervals.

```
    dplyr::ungroup() %>%
    dplyr::group_by(month_num) %>%
    dplyr::mutate(
      factor3=stats::arima.sim(model=ar_model, n=dplyr::n(),
                               innov = m_shock+stats::rnorm(dplyr::n(),
                                                            sd=0.1),
                               n.start = 500)) %>%
    dplyr::ungroup()
  if(num_factors_inp>3){
    #this process works for one factor. Want to lapply to all columns >4
    extra_factors_names=setdiff(
      names(factor_tib %>% dplyr::select(tidyselect::contains("factor"))),
      c(glue::glue("factor{1:3}"))
      )
    extra_factor_tib=purrr::map(.x=extra_factors_names,
                                       .f=~add_extra_factors(factor_tib=factor_tib,
                                 num_factors_inp=num_factors_inp,
                                 num_periods_inp=num_periods_inp,
                                 ar_model_inp=ar_model, col_in = .x)) %>%
      dplyr::bind_cols()

    factor_tib=factor_tib %>%
      dplyr::select(-tidyselect::all_of(extra_factors_names)) %>%
      dplyr::bind_cols(extra_factor_tib)
  }

  return(factor_tib %>% dplyr::select(-tidyselect::contains("shock")))

}

generate_time_grid <- function(date_start_inp,num_periods_inp,
                               freq_inp, num_entry_inp){
  period_dates=switch(freq_inp,
                      "daily"=format(lubridate::ymd(date_start_inp)+lubridate::days(0:(num_periods_inp-
                      "weekly"=format(lubridate::ymd(date_start_inp)+lubridate::weeks(0:(num_periods_inp
                      "monthly"=format(lubridate::ymd(date_start_inp)+months(0:(num_periods_inp-1)))
  )

  #Identidy the relevant components of the date (day/week/etc)
  date_info_tib=switch(freq_inp,
                        "daily" = tibble::tibble(time=seq_len(num_periods_inp),
                                                 date_t=period_dates,
                                                 day_num=lubridate::day(period_dates),
                                                 week_num=lubridate::week(period_dates),
                                                 month_num=lubridate::month(period_dates),
                                                 quarter_num=lubridate::quarter(period_dates),
                                                 year_num=lubridate::year(period_dates)),
                        "weekly"=tibble::tibble(time=seq_len(num_periods_inp),
                                                date_t=period_dates,
                                                week_num=lubridate::week(period_dates),
                                                month_num=lubridate::month(period_dates),
                                                quarter_num=lubridate::quarter(period_dates),
                                                year_num=lubridate::year(period_dates)),
                        "monthly"=tibble::tibble(time=seq_len(num_periods_inp),
                                                  date_t=period_dates,
                                                  month_num=lubridate::month(period_dates),
                                                  quarter_num=lubridate::quarter(period_dates),
                                                  year_num=lubridate::year(period_dates))
                        )

  return( date_info_tib )
}
```

Our DGP supports a minimum of 3 factors so that the trend and seasonal components do a reasonable job mimicking the existing data we estimated.[10] More than 3 factors can also be supported, where each factor above 3 follows the process defined for F_4 above (all independent from one another). An example with 4 factors is provided in Figure @ref(fig:factors-ex-fig).

```
add_extra_factors<-function(factor_tib,col_in,num_factors_inp,
                            num_periods_inp,
                             ar_model_inp){

  #compute the number shocks and their respective locations
  num_shocks=sample(1:13,1)
  shock_locs=c(0,sort(sample(1:52, size =num_shocks, replace = F )),52)
  extra_shocks=stats::runif(n=num_shocks+1, min=-1, max=1)
  shock_seq=rep(rep(extra_shocks, diff(shock_locs)), length.out=num_periods_inp)




  shockXwalk=tibble::tibble(time=seq_len(num_periods_inp),
                            e_shocks=shock_seq)
  factor_tib=factor_tib %>%
    dplyr::left_join(shockXwalk, by="time") %>%
    dplyr::mutate(
    !!as.name(col_in):=
      stats::arima.sim(model=ar_model_inp, n=dplyr::n(),
                      innov = e_shocks+stats::rnorm(dplyr::n(),
                                                    sd=0.1),
                      n.start = 500))

  return(factor_tib %>% dplyr::select(tidyselect::all_of(col_in)))

}
```

Computing Counterfactuals

The final step in the DGP computes the counterfactual outcome process and adds the treatment impact at each time period to round out our two potential outcomes as well as our observed outcome. As defined above, the outcome follows an AR(1) process with innovations governed by the time-varying factors and their individual specific loadings, as well as iid noise. Thus, we first compute the $T \times 1$ dimensional column $F_t\lambda_i'$ for each individual in parallel. With these computed, we simulate the AR(1) process for the counterfactual, where the process is once again governed by an individual specific parameter – ρ_i – allowing for parallelization

[10]If we would like, we can change this to be a purely random effects model, akin to Ignacio's DGP. That is to say, drop the factors and loadings entirely and use only the observed (and unobserved) covariates to model Y, which can still have a trend.

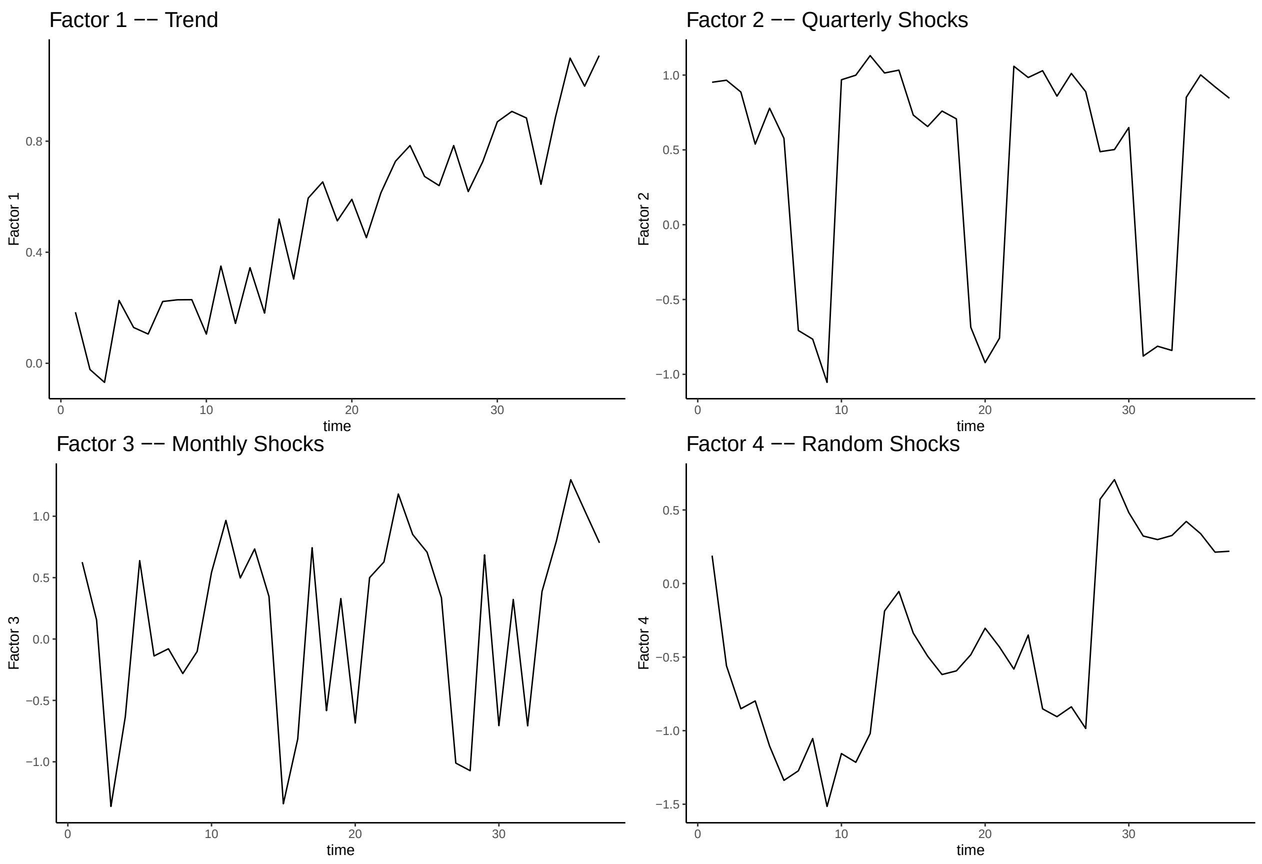

Figure 3: Factors Example

of the code.[11] With this series generated, we add the intercept by entry and a noise term drawn iid from $\mathcal{N}(0, 0.5^2)$ for each entry-time combination.

```
generate_counterfactual<-function(data_inp,num_periods_inp){

  outcome_series=data_inp %>%
    dplyr::select(time, entry, intercept, autocorr,
                  tidyselect::matches("loading|factor"))

  computed_factor_vec=compute_factor_loadings(outcome_series)

  outcome_series=outcome_series %>%
    dplyr::select(-tidyselect::matches("loading|factor")) %>%
    dplyr::mutate(factor_loadings=computed_factor_vec)

  #TODO(alexdkellogg): add in the covariates
  # Should this be in the outcome process or just y0=outcome_ar+xB?
  outcome_ar=compute_outcome_process(outcome_series,num_periods_inp)

  outcome_series=outcome_series %>%
    dplyr::mutate(factor_loadings=computed_factor_vec,
                  y0=outcome_ar+intercept+stats::rnorm(dplyr::n(), sd=0.5)) %>%
    dplyr::select(time, entry, factor_loadings, y0)

  return(data_inp %>%
           dplyr::inner_join(outcome_series, by=c("time", "entry")) %>%
           dplyr::select(time, entry, treated, treatment_period,y0,
                         factor_loadings, dplyr::everything())
         )


}

compute_factor_loadings <- function(data_inp){
  #create a list of factor loadings split by individual
  loadings_vec_list=data_inp %>%
    dplyr::select(entry, tidyselect::contains("loading")) %>%
    dplyr::distinct(entry, .keep_all=T) %>%
    dplyr::group_split(entry, .keep=F) %>%
    lapply(as.matrix)
  #create a list of factor matrices split by individual
  factor_mat_list=data_inp %>%
    dplyr::select(entry, tidyselect::contains("factor")) %>%
    dplyr::group_split(entry, .keep=F) %>%
    lapply(as.matrix)
  #Compute the matrix product of loadings and factors for each individual
  return(furrr::future_map2(.x=loadings_vec_list,.y=factor_mat_list,
                                  .f=~(.y)%*%t(.x) ) %>%
    unlist())
}

#TODO(alexdkellogg): check with AP about 0 noise AR for y
compute_outcome_process<-function(data_inp,num_periods_inp){
```

[11] As before, we allow a burn in period of 500.

```r
  #create a list of AR models, with individual specific noise and autocorr
  ar_param_inp=data_inp %>%
    dplyr::select(entry, autocorr) %>%
    dplyr::distinct(entry, .keep_all=T) %>%
    dplyr::group_split(entry, .keep=F) %>%
    lapply(function(x){ list(order=c(1,0,0), ar=x[[1]])} )


  innov_list=data_inp %>%
    dplyr::select(entry, tidyselect::contains("loading")) %>%
    dplyr::group_split(entry, .keep=F)   %>%
    lapply(as.matrix)


  return(furrr::future_map2(.x=ar_param_inp, .y=innov_list,
                             .f=~stats::arima.sim(model=.x,
                                                   n=num_periods_inp,
                                                   n.start = 500,
                                                   innov = .y)) %>%
    unlist())

}
```

The final step is then to add the treatment impact. Throughout our DGP, we have assigned each unit a hypothetical treatment assignment time, treatment impact, and treatment decay rate; this allows us to generate the treatment impact for all units, regardless of their true treatment status, and store each of the two potential outcomes in our final tibble. For each time period, we compute the individual specific treatment impact as the product of the original impact and the amount of decay by that particular point in time. For time before the treatment period, this is simply 0, yielding our counterfactual. For the post-treatment periods, we can add this quantity to the potential (untreated) outcome to form our potential (treated) outcome for each individual, and finally, assign the observed outcome following the standard framework. Lastly, we take the exponent of each of the three outcome variables and scale it to ensure the distribution resembles its empirical counterpart.

```r
#TODO(alexdkellogg): add conditional impact
generate_treat_impact<-function(data_inp=synth_data_full,
                                cond_impact_inp){
  #Define the treatment propogation for each unit and time combo
  data_inp=data_inp %>%
    dplyr::group_by(entry) %>%
    dplyr::mutate(
      post_treat_t=time-treatment_period,
      decay_t=dplyr::case_when(
        post_treat_t<0~0,
        post_treat_t>=0~treat_decay**post_treat_t),
      impact_t=decay_t*(treat_impact),
      ) %>%
    dplyr::ungroup() %>%
    dplyr::select(-c(treat_decay, treat_decay, decay_t))

  #Add the treatment impact to create y1
  return(data_inp %>%
    dplyr::mutate(y1=impact_t+y0,
                  y=treated*y1+(1-treated)*y0) %>%
      dplyr::select(time, entry, treated,
                    post_treat_t, treatment_period, impact_t,
                    y, y0,y1,
                    dplyr::everything()))
}
```