Empirical Demand Estimation in Insurance Markets (replication exercise)

READABILITY NOTE

The answers to the questions posed in the problem set instructions are contained in boxes like this one. This is to allow a brief reading for the answers to the actual questions. The rest of this document is just exposition (documentation for myself, mostly) on the process of optimizing the log-likelihood function and notes for optimizing in Julia in the future.

1 Summary Statistics

Table 1: Year 1 population mean of all draws of out-of-pocket expenditures, by plan

Plan	Mean
250	1697.07
500	2296.09
1200	2540.26

Table 2: Year 1 population mean of standard deviation of all draws of out-of-pocket expenditures, by plan

Plan	Std Dev
250	961.73
500	1355.43
1200	1412.39

Table 3: Year 1 population mean of out-of-pocket expenditures, by plan, by family status

Plan	Family Status	Mean	Std Dev	Count
250	Single	1697.93	958.65	988
	With Spouse & Children	1648.93	941.17	757
	With Children	1800.03	995.02	146
	With Spouse	1775.26	1006.72	378
	Unknown (11)	1335.70	846.02	19
	Unknown (12)	625.40	432.83	2
500	Single	2297.85	1353.94	988
	With Spouse & Children	2241.22	1335.19	757
	With Children	2439.46	1392.22	146
	With Spouse	2376.91	1397.17	378
	Unknown (11)	1832.90	1205.60	19
	Unknown (12)	853.64	601.96	2
1200	Single	2539.60	1412.77	988
	With Spouse & Children	2487.90	1393.95	757
	With Children	2683.56	1428.95	146
	With Spouse	2623.68	1451.76	378
	Unknown (11)	2056.62	1292.34	19
	Unknown (12)	1050.43	698.78	2

Table 4: Family Status Tabulation

Family Status	Count
Single	988
With Spouse & Children	757
With Children	146
With Spouse	378
Unknown (11)	19
Unknown (12)	2

Table 5: Family and Individual Tabulations

Variable	Mean	Std Dev	Median	Min	Max	Count
Chronic condition in family	0.46	0.50	0	0	1	2290
Enrollment in Flexible Spending Account	0.34	0.47	0	0	1	2290
Employee Manager Status	0.30	0.46	0	0	1	2290
Income	2.57	1.21	2	1	5	2290
Family Size	1.94	1.01	2	1	4	2290
Age (by individual)	38.45	20.68	41.09	0.00	74.99	4450
Gender (by individual)	0.50	0.50	1	0	1	4450

2 The Log-Likelihood Function

Code in attached files. One note: When I evaluated the function at the suggested starting parameter values, I got about 5525. This is on the order of magnitude of 2000-3000, but hopefully not incorrect.

3 vNM-utility-based Choice Predictions

Code in attached files. I used a mix of for loops and element-wise operations to complete this. The benefit of using Julia is that loops are not a bottleneck like they are for python, R, or matlab. This is because Julia is a compile-at-runtime language and both loops and matrix operations are written in Julia. So after compiling, the loops run about as fast as matrix operations.

4 Estimating Parameters (Maximizing the Likelihood)

Below, I compare an open-source Julia-based optimization algorithm with the Knitro package from Artelys.

4.1 Julia Open-source Optimization Packages

There are many optimizing algorithms in the main Julia optimizing package Optim.jl. Most of the algorithms require analytical gradients or use a julia method of auto-differentiation to obtain an exact gradient (and Hessian in some cases) from the definition of the function without having to symbolically solve for the derivative. Because of the way I wrote the likelihood function, it is not auto-differentiable. There may be a way to write the function so that it is auto-differentiable, or perhaps even a way to derive the gradient. However, I just focused on the algorithms that can use finite differences to estimate the gradient at each step. Unfortunately these require many more function evaluations and are slower than using gradient methods.

Four "derivative-free" (finite differences) methods stand out from the usual julia ecosystem that

allow for bounds on the parameter space:

1. NelderMead (the default)
2. SAMIN (Simulated Annealing with bounds)
3. Particle Swarm
4. BlackBoxOptim

After some testing, the Simulated Annealing algorithm seemed to preform better – faster iterations and lower objective function values for the same number of iterations. I did not test Particle Swam since it is relatively new to the Optim.jl package and has not been vetted as thoroughly by programmers. This does not mean that SAMIN is more likely than the NelderMead (or Particle Swarm) algorithm to converge on the global minimum, but for the purposes of testing different starting parameter values with a relatively small number of iterations (compared to what I will be using with the Knitro package), Simulated Annealing can give me a better quick idea of how starting parameter values affect the estimated minimizing parameter values. BlackBoxOptim did not work because much of the parameter space returns NaN values (see the Precision Errors section).

Below I show the results of minimizing the objective function using the SAMIN algorithm with five different starting parameters, limiting the runtime to 8 hours total (about minutes for each set of starting parameters). I also compare these results to the Knitro package, run for 7.1 hours, from the first suggested starting values. Just due to lack of time, I end the estimation there, with the understanding that for real research, these may need to run for longer on a better computer with higher precision data types (see below section).

The five starting parameter vectors are presented in Table 6. Note that several of the staring parameter vectors, as well as the lower and upper bounds, have `NaN` function values. This is seemingly due to a precision issue, discussed in a later section.

4.2 KNITRO

The Knitro package has changed a little bit since the 2013 paper but I was able to get it working in Julia. Sadly, the Julia wrapper has some limitations – it cannot parallelize (there is an open GitHub issue about this, and needs to be resolved by the folks at Artelys / Knitro). This means that I could not run the multi-start function efficiently, which would have allowed me to explore effects of starting parameters much better. So I chose to just use one run of Knitro, and compare it to other runs of the Julia Optim-SMAIN alogrithm with the other starting parameter values. Knitro takes longer, and in initial tests, the SAMIN minimizer seems to outperform Knitro in terms of speed of minimization. However, without more extensive simulation testing, I can't say for sure algorithm performs better on this likelihood function. It would likely depend on the function being evaluated as well.

Table 7 shows the parameter estimates from minimizing the negative log likelihood with the KNITRO and SAMIN algorithms, from 5 different starting parameter values (alpha0 - alpha4). The last column (alpha4) is cut off, but is not important – the runs starting with the alpha4 parameter vector did not converge and only returned NaN values. The initial function evaluation at alpha4 returns NaN as well, and it seems that the SAMIN alogrithm was unable to search in the neighbood of alpha4 to find a non-NaN value. However, the initial function evaluation of alpha2 also returned NaN, but the SAMIN solver was able to bounce to a non-NaN path. Both the KNITRO and SAMIN algorithm implementations ignore NaN values, treating them as parameter vectors to skip. This seems problematic and is discussed more in the Precision Errors section.

4.b: Interpretation of the function value at alpha0

The function evaluates to 5525. This is on the order of magnitude of the 2000-3000 suggested in the instructions, and could be due either to sampling variation (since part of the setup requires sampling from random distributions). [a] This means the log-likelihood value is -5525 and the likelihood is $e^{-5525} \approx 2.97 \cdot 10^{-2400}$. This is the joint probability of observing the data we have given the parameters in alpha0. For comparison, we can assume for a moment that all the families are independent, then we can take the geometric mean to find the probability that one of the families came from the distribution described by alpha0. The geometric mean – taking the $\texttt{nIs}^{th}$ root – is about 0.0896, or 8.96% chance on average that a family in our observations came from this distribution.

[a] There could also be a mistake in my code, but after combing through it, I will assume for now that a coding error is not the issue. If I were doing this for research, I would corroborate my results by re-programming the log-likelihood from scratch, going off the equations in the paper, and being much less efficient by putting the random sampling directly into the LL function.

4.c: Optimization Keyword Arguments

The options given in the instructions provide stopping conditions to the program. `MaxFunEvals` (`f_calls_limit` in Julia) sets the maximum number of function evaluations before exiting. `MaxIter` (`iterations` in Julia) sets the maximum number of iterations through the algorithm – this is different than `MaxFunEvals` because for each iteration through the algorithm, the program needs to compute the finite-differences gradient estimate. So for each iteration of the algorithm, there are probably around $k + 1$ function evaluations: an initial function evaluation at the new parameter values, plus another function evaluation for each of the parameters, evaluated at the new parameter values plus a tiny "epsilon" change in one of the parameters (to estimate the gradient). Some algorithms use $k + 2$ or $k + 3$ to evaluate the function one more time at the new parameter values plus a positive and/or negative epsilon change in all the parameters.

Finally, `TolFun` (`f_tol` in Julia) is hopefully the binding constraint of the stopping conditions. This is the Function Tolerance condition which is the difference between function evaluations in successive iterations. So if `TolFun` $= 10^{-4}$, then the algorithm would stop as soon as two successive iterations result in function evaluations that are less than 10^{-4} apart, if the other stopping conditions have not been met yet.

The other main options that would effect the optimization are arguments controlling the size of steps to take in the parameter space and random seeds (to make sure a solve run is reproducible). There's also options that would implicitly change the alogorithm used. For example, there is a low memory setting that would change to using a sparse approximation of the gradient and hessian, at risk of being less accurate.

There are many other options. For functions that have given or computable exact gradients or hessians, there are similar arguments for the tolerances and maximum number of calls to those functions. There are also options regarding the saving of results or what to print out during the search, which can be very helpful when comparing the searches of different algorithms. Of course, there is also an argument for maximum time to spend on the optimization (something I used when running the SAMIN optimizations).

Due to time constraints and lack of multithreading in the currently implemented Julia version of

KNITRO, I was unable to use the Multi-start functionality of KNITRO. So I cannot say that I have high confidence that the reported estimates are the global minimum. A bigger issue seems to be the prevalence of precision-error-based NaN values (see next section). Because of this, much of the domain is not actually searched for the global minimum, and I could be missing an important region of the function to fit the distributions to the data.

Table 6: Five starting values for the search to minimize the negative log-likelihood function.

		Starting Values					
Parameter	**Lower Bound**	**alpha0**	**alpha1**	**alpha2**	**alpha3**	**alpha4**	**Upper Bound**
Inertia–single	0	0.06	0.1	0.3	0.5	0.9	2
Inertia–family	-Inf	-2500	900	600	1500	1000	Inf
Inertia–FSA enroll	1	300	800	500	400	150	20000
Inertia–Income	-Inf	-500	-2500	-1500	-1000	-2000	Inf
Inertia–quantitative	-5000	0	20	200	0	-50	5000
Inertia–manager	-5000	0	-100	-100	100	-50	5000
Inertia–chronic condition	-0.5	0.003	0.04	0.1	0.02	0.05	1
Inertia–salient change	0.1	700	0.1	0.1	0.1	0.1	Inf
Risk aversion mean–intercept	1	800	300	200	150	150	20000
Risk aversion mean—income	0	1250	700	700	500	800	20000
Risk aversion mean—age	-5000	0	100	0	50	-50	5000
Risk aversion standard deviation	-0.5	0	0	-0.0003	-0.0003	0.003	1
CDHP—single—RC mean	-Inf	-2200	1500	2500	1700	1500	Inf
CDHP—single—RC SD	1	300	800	500	400	150	20000
CDHP—family—RC mean	0	1750	1500	1200	1500	1050	20000
CDHP—family—RC SD	-5000	0	100	0	200	100	5000
High total cost—PPO1200	0	0.04	0.03	0.08	0.03	0.06	1
ϵ_{500}, σ–single	0.1	800	500	300	1	100	Inf
ϵ_{1200}, σ–single	1	800	300	200	150	150	20000
ϵ_{500}, σ–family	-5000	-500	-200	-800	-500	-700	5000
ϵ_{1200}, σ–family	-5000	0	-100	-200	-100	100	5000
-LogLikelihood	NaN	5525	12952	NaN	55335	NaN	NaN

Table 7: Parameter estimates from minimizing the log likelihood with the KNITRO and SAMIN algorithms, from 5 different starting parameter values (alpha0 - alpha4).

Parameter	**KNITRO alpha0**	**SAMIN alpha0**	**SAMIN alpha1**	**SAMIN alpha2**	**SAMIN alpha3**	**SAMIN alp**
Inertia–single	20000.00	19796.69	17954.56	17954.56	17347.85	1050
Inertia–family	12642.21	9464.11	16059.15	16059.15	15961.01	800
Inertia–FSA enroll	4639.08	2744.43	800.65	800.65	2008.02	-700
Inertia–Income	5000.00	4566.84	4149.55	4149.55	3260.32	-50
Inertia–quantitative	-2422.14	2618.89	-740.63	-740.63	-1108.49	-50
Inertia–manager	5000.00	4945.55	3230.35	3230.35	3793.49	100
Inertia–chronic condition	5000.00	4599.03	730.98	730.98	1915.93	100
Inertia–salient change	1295.03	4252.69	3658.94	3658.94	2102.07	-50
Risk aversion mean–intercept	0.00580	0.09690	0.33000	0.33000	0.07080	0.9
Risk aversion mean—income	-0.00048	0.00320	-0.02040	-0.02040	0.00740	0.05
Risk aversion mean—age	-0.00001	-0.00010	-0.45060	-0.45060	0.00020	0.003
Risk aversion standard deviation	0.00003	0.02310	0.07670	0.07670	0.04610	0.06
CDHP—single—RC mean	-2500	-2500	600	600	1500	1000
CDHP—single—RC SD	701.98	700	0.1	0.1	0.1	0.1
CDHP—family—RC mean	-2200.00	-2200	2500	2500	1700	1500
CDHP—family—RC SD	801.69	800	300	300	1	100
High total cost—PPO1200	-500.03	-500	-1500	-1500	-1000	-2000
ϵ_{500}, σ–single	187.59	1263.03	10088.42	10088.42	803.48	150
ϵ_{1200}, σ–single	1.00	32.60	5327.08	5327.08	143.56	150
ϵ_{500}, σ–family	7657.18	11822.50	5641.20	5641.20	9511.09	150
ϵ_{1200}, σ–family	1.00	5843.94	1470.16	1470.16	4080.02	150
-LogLikelihood	2576.65	2636.08	7006.38	7006.38	2749.41	NaN
Function Evals	11643	3683	3804	3804	3794	3709
Hours run	7.13	1.6	1.6	1.6	1.6	1.6
Converged?	N	N	N	N	N	N

4.d: Population distribution of inertial costs

Fig. 1 plots a kernel density of the inertial costs for each year, after accounting for given demographic heterogeniety by evaluating at the LL maximizing parameters given from the Knitro run in Table 7. This strikes me as a lot of money – that for an average family, they would give up $35,000 to stay on the same plan as last year? My parameter estimates are very far from the published paper however, so these may just be very overblown by poor estimation.

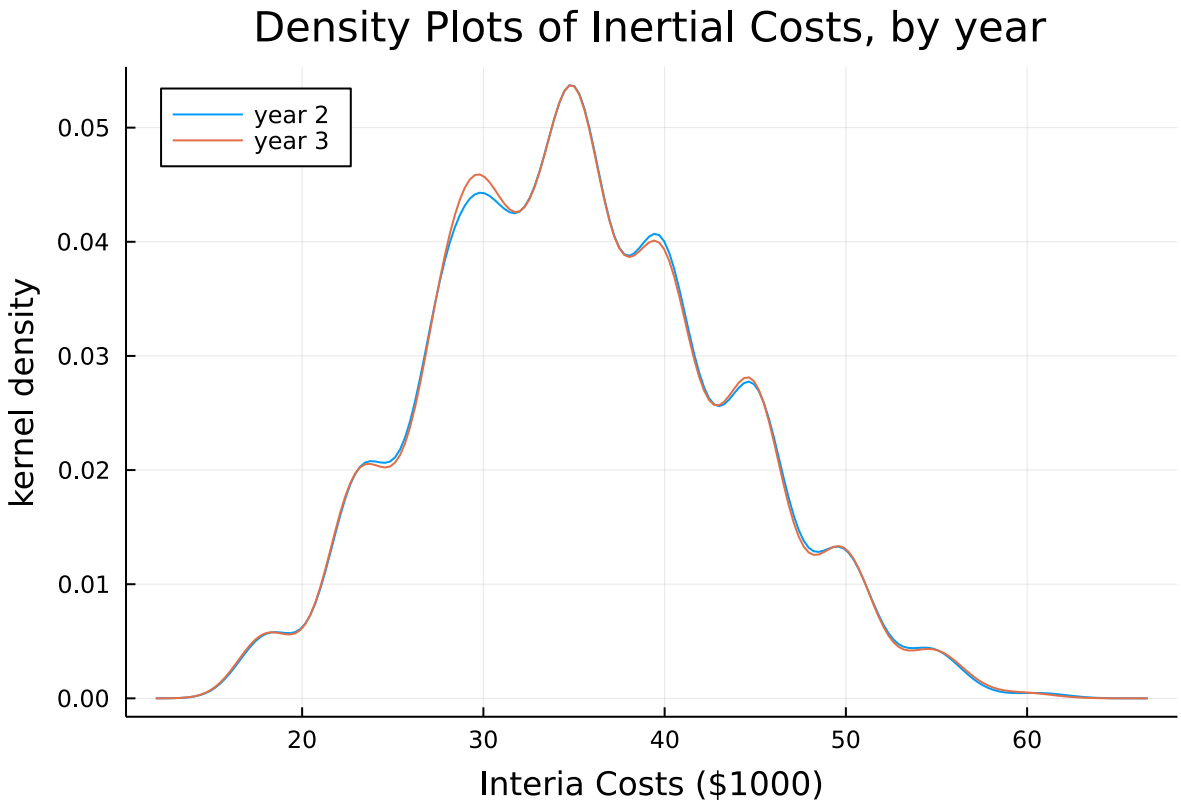

Figure 1: Estimated kernel densities of the inertial costs, evaluated at the minimizing parameter values returned by the Knitro optimizer.

4.e: CARA risk and indifference for a gamble

Consider the mean of the CARA risk preference distribution of random coefficients. Translate this coefficient into the value of X that makes a family indifferent between no gamble and a gamble where they win $100 with 50% probability and lose $X with 50% probability.

Taking the mean of the risk preference random coefficient distribution, evaluated using the random coefficients generated for this problem and the distribution of income and age, I get a mean of about $3.996 \cdot 10^{-5}$. Consider this coefficient fixed, and the potential X fixed for which a family is made indifferent. Because this is a CARA utility, the initial level of wealth does not matter. So we can solve a simple indifference case of the utility generated from zero additional consumption ($u(0)$) compared to the expected utility generated by the gamble ($\mathbb{E}$[gamble]). Here, the only random variable is winning or losing the gamble, so to calculate the level of lost consumption X that would make the family indifferent, we solve the following indifference condition:

$$u(0) = \mathbb{E}[\text{gamble}]$$

$$u(0) = \frac{1}{2}u(100) + \frac{1}{2}u(-X)$$

Noting that the utility function is

$$u(x) = -\frac{1}{\gamma}e^{-\gamma x}$$

then the inverse utility function is

$$u^{-1}(z) = -\frac{1}{\gamma}\log(-\gamma y)$$

Solving for X in the indifference conditions yields:

$$X = -u^{-1}\big(2u(0) - u(100)\big)$$

And plugging in my estimate of the mean family coefficient gives $X \approx 99.6$. This means that, on average, a family would take the gamble if they were only at risk of losing less than $99.6.

4.f: Starting Parameter Values & Reported Estimates

Table 7 shows the results from using the starting parameter vectors in Table 6. The parameter estimates change quite a bit. The obvious criteria for the best of these estimates is the one with the lowest function value (negative log likelihood). This means it is closest to what could be considered a global maximum in the likelihood – maximizing the probability that the model fits the data. So the Knitro results would be my main choice for reporting estimates, however, that is a bit artificial as we can see from Table 6 that I let it run significantly longer than the other runs, and it is the only run using Knitro.

If I were doing this again, I would probably use the Optim.jl package's SAMIN algorithm instead of Knitro, and implement my own version of Knitro's multi-start. I believe the default method is to choose parameter values uniformly over the given restricted parameter space. I could fairly ea sly parallelize this with the Distributed package, and perhaps let it run longer on our department server or on an AWS elastic compute service.

4.3 Precision Errors

The loglikelihood function returns `NaN` at some parameter values within the restricted parameter domain (between the lower and upper bounds suggested). This happens because some of the X values are large in the utility function, so when calculating $e^{-\gamma * x}$, this evaluates to zero when using the Float64 data type. Then, when inverting the elements of the matrix in the normalization process, it is dividing by zero and returning `NaN`. Assuming I have correctly programmed the function and transformed the data, then this is merely a round-off issue. The obvious solution would be to use a more precise data type, and, in Julia, BigFloat is the next largest (without creating my own custom datatype) and has arbitrary precision. But when the numbers are very small (on the order of 10^{-2000}), this can take up a large amount of memory. As a result, each evaluation of the log-likelihood function takes a very long time to run when using BigFloats. It is possible that I could compute this matrix using BigFloats first, then immediately convert it back to Float64 – this issue really occurs at the division. But even computing the one element-wise division using BigFloats seems to take a while.

There may be another way to deal with this issue – for example, if the goal in the utility step is to create relative utilities compared to the 1200 plan, then I could create a rule for how to deal with these. If the element-wise division would be something like $Y/0$ ($Y \in \mathbb{R}$), then just assign `Inf` to it. The issue comes when you want to evaluate Y/Z for very small Y and Z. I can't think of another way to deal with this issue. Using higher precision might be the best solution to fully explore the parameter space, but sadly my computer doesn't seem to have enough RAM to run this with BigFloats, and it would seem to take an extremely long time. This could be parallelized with the Distributed library. I am working on a maximum likelihood project in Julia over the summer and will be testing parallization. I am keeping these methods in mind for more complex likelihood functions that don't have easy gradients to derive (or they don't exist ex-ante). If I needed to use BigFloats for a research project like this, I would probably spin-up an AWS elastic compute instance to run something like this in parallel, where the RAM and CPU can scale easily.

4.4 Alternative Algorithms

For optimizers in Julia, Optim.jl is the main package that developers have devoted their time. There are three derivative-free methods one can employ that only require the function in question and utilize finite differences to estimate the gradient: `NelderMead`, `SAMIN`, and `Particle Swarm`. Outside of Optim.jl, I also found the BlackBoxOptim package. This is designed to work well as a derivative-free method, so it's possible it could outperform the main Optim.jl methods in some cases. However, it does not work well when much of the domain of the of the function returns NaN. So due to the precision issue mentioned above, I was not able to test this alogrithm here.

There are also a wealth of other optimizers for situations when you have more information about the function (Gradient, Hessian, etc). See this link for a quick list of different solvers available in Optim.jl: julianlsolvers.github.io/Optim.jl/stable/#user/config