Fixed Point Acceleration in R

by Stuart Baumann, Margaryta Klymak

Abstract A fixed point problem is one where we seek a vector, X, for a function, f, such that f(X) = X. The solution of many such problems can be accelerated by using a fixed point acceleration algorithm. With the release of the **FixedPoint** package there is now a number of algorithms available in **R** that can be used for accelerating the finding of a fixed point of a function. These algorithms include Newton acceleration, Aitken acceleration and Anderson acceleration as well as epsilon extrapolation methods and minimal polynomial methods. This paper demonstrates the use of fixed point accelerators in solving numerical mathematics problems using the algorithms of the **FixedPoint** package as well as the squarem method of the **SQUAREM** package.

Introduction

R has had a number of packages providing optimisation algorithms for many years. These include traditional optimisers through the optim function, genetic algorithms through the **rgenoud** package (Mebane, Jr. and Sekhon, 2011) and response surface global optimisers through packages like **DiceKriging** (Roustant et al., 2012). It also has several rootfinders like the uniroot method and the methods of the **BB** package (Varadhan and Gilbert, 2009).

Fixed point accelerators are conceptually similar to both optimisation and root finding algorithms but thus far implementations of fixed point finders have been rare in **R**. Prior to **FixedPoint**'s (Baumann and Klymak, 2018) release the squarem method of the **SQUAREM** package[1] (Varadhan, 2010) was the only effective fixed point acceleration algorithm available in **R**.[2] In some part this is likely because there is often an obvious method to find a fixed point by merely feeding a guessed fixed point into a function, taking the result and feeding it back into the function. By doing this repeatedly a fixed point is often found. This method (that we will call the "Simple" method) is often convergent but it is also often slow which can be prohibitive when the function itself is expensive.

This paper shows how the finding of a fixed point of a function can be accelerated using fixed point accelerators in **R**. The next section starts by with a brief explanation of fixed points before a number of fixed point acceleration algorithms are discussed. The algorithms examined include the Newton, Aitken and Scalar Epsilon Algorithm (SEA) methods that are designed for accelerating the convergence of scalar sequences. Five algorithms for accelerating vector sequences are also discussed including the Vector Epsilon Algorithm (VEA), Anderson acceleration and three minimal polynomial algorithms (MPE, RRE and the squarem method provided in the **SQUAREM** package). The **FixedPoint** package is then introduced with applications of how it can be used to find fixed points. In total five problems are described which show how fixed point accelerators can be used in solving problems in asset pricing, machine learning and economics. Here the intent is not only to showcase the capabilities of **FixedPoint** and **SQUAREM** but also to demonstrate how various problems may be able to be recast in an iterate way in order to be able to exploit fixed point accelerators. Finally this paper uses the presented numerical problems to perform a speed of convergence test on all of the algorithms presented in this paper.

Fixed point acceleration

Fixed point problems

A fixed point problem is one where we look for a vector, $X \in \Re^N$, so that for a given real valued function $f : \Re^N \rightarrow \Re^N$ we have:

$$f(X) = X \tag{1}$$

If $f : \Re^1 \rightarrow \Re^1$ and thus any solution X will be a scalar then one way to solve this problem would be to use a rootfinder on the function $g(x) = f(x) - x$ or to use an optimiser to minimise a function like $h(x) = (f(x) - x)^2$. These techniques will not generally work however if $f : \Re^N \rightarrow \Re^N$ where N is large. Consider for instance using a multidimensional Newtonian optimiser to minimise $h(x) = \sum_{i=1}^{N} (f_i(x) - x_i)^2$ where $f_i(x)$ is the i'th element output by $f(x)$. The estimation of gradients for each individual dimension may take an unfeasibly long time. In addition this method may not make use all of the available information. Consider for instance that we know that the solution for

[1] The squarem method has also been implemented in the **turboEM** package (Bobb and Varadhan, 2014).
[2] The Anderson method has since been implemented in the **daarem** package (Henderson and Varadhan, 2018).

x will be an increasing vector (so $x_i > x_j$ for any entries of x with $i > j$) with many elements. This information can be preserved and used in the fixed point acceleration algorithms that we present but would be more difficult to exploit in a standard optimisation algorithm.

Much of the intuition behind the use of optimisers and rootfinders carries over to the use of fixed point acceleration algorithms. Like a function may have multiple roots and multiple local optima, a function may have multiple fixed points. The extreme case of this is the identity mapping $f(x) = x$ for which every x is a fixed point. Some functions have no roots or optima and likewise some functions do not possess fixed points. This is the case for the function $f(x) = \frac{-1}{x}$. From a practical standpoint, it is often useful to have access to multiple optimisers and rootfinders as different algorithms are better suited to different types of functions. This is also the case for finding fixed points and the **FixedPoint** package is useful in this regard, offering eight fixed point algorithms.

The first algorithm implemented in **FixedPoint** is the "simple" method which merely takes the output of a function and feeds it back into the function. For instance starting with a guess of x_0, the next guess will be $x_1 = f(x_0)$. The guess after that will be $x_2 = f(x_1)$ and so on. Under some conditions f will be a contraction mapping and so the simple method will be guaranteed to converge to a unique fixed point (Stokey et al., 1989). Even when this is the case however the simple method may only converge slowly which can be inconvenient. The other seven methods implemented in **FixedPoint** and the squarem method of **SQUAREM** are designed to be faster than the simple method but may not be convergent for every problem.

Fixed point acceleration algorithms

Newton acceleration

Here we will define $g(x) = f(x) - x$. The general approach is to solve $g(x)$ with a rootfinder. The x that provides this root will be a fixed point. Thus after two iterates we can approximate the fixed point with:

$$\text{Next guess} = x_i - \frac{g(x_i)}{g'(x_i)} \tag{2}$$

FixedPoint approximates the derivative $g'(x_i)$ such that we use to get an estimated fixed point of:

$$\text{Next guess} = x_i - \frac{g(x_i)}{\frac{g(x_i)-g(x_{i-1})}{x_i - x_{i-1}}} \tag{3}$$

The implementation of the Newton method in **FixedPoint** uses this formula to predict the fixed point given two previous function iterates. This method is designed for use with scalar functions. If it is used with higher dimensional functions that take and return vectors then it will be used elementwise.

Aitken acceleration

Consider that a sequence of scalars $\{x_i\}_{i=0}^{\infty}$ that converges linearly to its fixed point of $\hat{x}$. This implies that for a some i:

$$\frac{\hat{x} - x_{i+1}}{\hat{x} - x_i} \approx \frac{\hat{x} - x_{i+2}}{\hat{x} - x_{i+1}} \tag{4}$$

For a concrete example consider that every iteration halves the distance between the current value of x_i and the fixed point. In this case the left hand side will be one half which will equal the right hand side which will also be one half. Equation 4 can be simply rearranged to give a formula predicting the fixed point that is used as the subsequent iterate. This is:

$$\text{Next guess} = x_i - \frac{(x_{i+1} - x_i)^2}{x_{i+2} - 2x_{i+1} + x_i} \tag{5}$$

The implementation of the Aitken method in **FixedPoint** uses this formula to predict the fixed point given two previous iterates. This method is designed for use with scalar functions. If it is used with higher dimensional functions that take and return vectors then it will be used elementwise.

		Columns					
		A	B	C	D	E	F
		Zeros	Iterates Performed	Epsilon Triangle Columns			
Row Numbers	1	0	1.000	-2.175	0.728	179.993	0.742
	2	0	0.540	3.152	0.734	303.615	
	3	0	0.858	-4.920	0.737		
	4	0	0.654	7.184			
	5	0	0.793				
		0					

Figure 1: The Epsilon Algorithm applied to the cos(x) function

Epsilon algorithms

The epsilon algorithms introduced by Wynn (1962) provides an alternate method to extrapolate to a fixed point. This paper will present a brief numerical example and refer readers to Wynn (1962) or Smith et al. (1987) for a mathematical explanation of why it works. The basic epsilon algorithm starts with a column of simple function iterates. If i iterates have been performed then this column will have a length of $i+1$ (the initial starting guess and the results of the i iterations). Then a series of columns are generated by means of the below equation:

$$\epsilon^{r}_{c+1} = \epsilon^{r+1}_{c-1} + (\epsilon^{r+1}_{c} - \epsilon^{r}_{c})^{-1} \tag{6}$$

Where c is a column index and r is a row index. The algorithm starts with the ϵ_0 column being all zeros and ϵ_1 being the column of the sequence iterates. The value in the furthest right column ends up being the extrapolated value.

This can be seen in the figure 1 which uses an epsilon method to find the fixed point of $\cos(x)$ with an initial guess of a fixed point of 1. In this figure B1 is the initial guess of the fixed point. Then we have the iterates $B2 = \cos(B1)$, $B3 = \cos(B2)$ and so on. Moving to the next column we have $C1 = A2 + 1/(B2 - B1)$ and $C2 = A3 + 1/(B3 - B2)$ and so on before finally we get $F1 = D2 + 1/(E2 - E1)$. As this is the last entry in the triangle it is also the extrapolated value.

Note that the values in columns C and E are poor extrapolations. Only the even columns D,F provide reasonable extrapolation values. For this reason an even number of iterates (an odd number of values including the starting guess) should be used for extrapolation. **FixedPoint** will enforce this by throwing away the first iterate provided if necessary to get an even number of iterates.

In the vector case this algorithm can be visualised by considering each entry in the above table to contain a vector going into the page. In this case the complication emerges from the inverse term in equation 6: there is no clear interpretation of $(\epsilon^{r+1}_{c} - \epsilon^{r}_{c})^{-1}$ when $(\epsilon^{r+1}_{c} - \epsilon^{r}_{c})$ represents a vector. The Scalar Epsilon Algorithm (SEA) uses elementwise inverses to solve this problem which ignores the vectorised nature of the function. The Vector Epsilon Algorithm (VEA) uses the Samuelson inverse of each vector $(\epsilon^{r+1}_{c} - \epsilon^{r}_{c})$ as described in Smith et al. (1987).

Minimal polynomial algorithms

FixedPoint implements two minimal polynomial algorithms, Minimal Polynomial Extrapolation (MPE) and Reduced Rank Extrapolation (RRE). The key intuition for these methods is that a linear combination of previous iterates is taken to generate a new guess vector. The coefficients of the previous iterates are taken so that this new guess vector is expected to not be changed much by the function.[3]

To first define notation, each vector (the initial guess and subsequent iterates) is defined by $x_0, x_1, ...$. The first differences are denoted $u_j = x_{j+1} - x_j$ and the second differences are denoted $v_j = u_{j+1} - u_j$. If we have already completed $k-1$ iterations (and so we have k terms) then we will use matrices of first and second differences with $U = [u_0, u_1, ..., u_{k-1}]$ and $V = [v_0, v_1, ..., v_{k-1}]$.

[3]For more details an interested reader is directed to Cabay and Jackson (1976) or Smith et al. (1987) for a detailed explanation.

For the MPE method the extrapolated vector, is found by:

$$\text{Next guess} = \frac{\sum_{j=0}^{k} c_j x_j}{\sum_{j=0}^{k} c_j} \tag{7}$$

Where the coefficient vector is found by $c = -U^{+}u_k$ where U^{+} is the Moore-Penrose generalised inverse of the U matrix. In the case of the RRE method the extrapolated vector, is found by:

$$\text{Next guess} = x_0 - UV^{+}u_0 \tag{8}$$

The only effective fixed point accelerator that was available in R prior to the release of the **FixedPoint** package was the squarem method provided in the **SQUAREM** packages. This method modifies the minimal polynomial algorithms with higher order terms and can thus be considered as a variant of the MPE algorithms. The squarem method is primarily intended to accelerate convergence in the solution of expectation maximisation problems but can be used more generally with any function that is a contraction mapping (Varadhan and Roland, 2008).

Anderson acceleration

Anderson (1965) acceleration is an acceleration algorithm that is well suited to functions of vectors. Similarly to the minimal polynomial algorithms it takes a weighted average of previous iterates. It is different however to all previous algorithms in that the previous iterates used to generate a guess vector need not be sequential but any previous iterates can be used. Thus it is well suited to parallelising the finding of a fixed point.[4]

Consider that we have previously run an N-dimensional function M times. We can define a matrix $G_i = [g_{i-M}, g_{i-M+1}, ..., g_i]$ where $g(x_j) = f(x_j) - x_j$. Each column of this matrix can be interpreted as giving the amount of "movement" that occurred in a run of the function.

In Anderson acceleration we assign a weight to apply to each column of the matrix. This weight vector is M-dimensional and can be denoted $\alpha = \{\alpha_0, \alpha_1, ..., \alpha_M\}$. These weights are determined by means of the following optimisation:

$$\min_{\alpha} ||G_i \alpha||_2 \tag{9}$$
$$s.t. \sum_{j=0}^{M} \alpha_j = 1$$

Thus we choose the weights that will be predicted to create the lowest "movement" in an iteration.

With these weights we can then create the expression for the next iterate as:

$$\text{Next guess} = \sum_{j=0}^{M} \alpha_j f(x_{i-M+j}) \tag{10}$$

Robustness of fixed point algorithms

Functions with restricted input spaces

Some functions have a restricted input space. Acceleration schemes can perform badly in these settings by proposing vectors that sit outside of the required input space. As an example consider the following $\Re^2 \rightarrow \Re^2$ function, that we try to find the fixed point for with the Anderson method:

$$\text{Output} = \left(\frac{\sqrt{\text{Input}[1] + \text{Input}[2]}}{2} \quad , \quad \left| \frac{3\text{Input}[1]}{2} + \frac{\text{Input}[2]}{2} \right| \right) \tag{11}$$

```
library(FixedPoint)
SimpleVectorFunction = function(x){c(0.5*sqrt(x[1] + x[2]), abs(1.5*x[1] + 0.5*x[2]))}
FPSolution = FixedPoint(Function = SimpleVectorFunction, Inputs = c(0.3,900),
                        Method = "Anderson")
```

[4] An example of this is shown in the appendix for the consumption smoothing problem described later in this paper.

Unfortunately an error will occur here. After four iterates the Anderson method decides to try the vector $(-1.085113, -3.255338)$. This results in the square root of a negative number and hence the output is undefined.

In cases like this there are a few things a user can try. The first is to change the function to another function that retains the same fixed points. In the above case we could change the function to take the absolute value of the sum of the two inputs before taking the square root. Then after finding a fixedpoint we can verify if the sum of the two entries is positive and hence it is also a solution to the original function. Another measure that could be tried is to change the initial guess. Finally we could change the acceleration method. The simple method will be robust in this case as the function will never return an Output vector that sums to a negative number. It is still likely to be slow however.[5] A special feature of the FixedPoint package is that it allows methods to be changed while retaining previous iterates. So in this case we can run the preceding code until an error causes the acceleration to stop, switch to the simple method for a few iterates and then switch back to the anderson method. No error will result as we are close enough to the fixedpoint that each new guess sums to be positive:

```
FPSolution = FixedPoint(Function = SimpleVectorFunction, Inputs = FPSolution$Inputs,
                        Outputs = FPSolution$Outputs, Method = "Simple", MaxIter = 5)
# Now we switch to the Anderson Method again. No error results because we are
# close to fixed point.
FPSolution = FixedPoint(Function = SimpleVectorFunction, Inputs = FPSolution$Inputs,
                        Outputs = FPSolution$Outputs, Method = "Anderson")
```

Another example of a restricted input space is shown in the consumption smoothing example presented later in this paper. In this example the input vector must reproduce a monotonic and concave function. All of the vectorised methods presented in this paper take a combination of previous iterates all of which take and return vectors representing monotonic and concave functions. As a result these methods will only propose vectors representing monotonic and concave functions. By contrast the Newton, SEA and Aitken methods do not take into account the entire vector when proposing the fixedpoint value for each element of the vector and as a result some of the input vectors proposed by these methods may not be valid. Ensuring that a vectorised method is chosen is thus sufficient in this case to ensure that each vector tried is within the input space of the function for which a fixedpoint is sought.

Convergence by constant increments

Most fixed point acceleration algorithms will fail in finding the fixed point of a function that converges by a fixed increment. For instance we may have a function that takes x and returns x shifted 1 unit (in Euclidean norm) in a straight line towards its fixed point. A realistic example of this type of convergence is the training of a perceptron classifier which is explored later in this paper.

This type of convergence is problematic for all algorithms presented except for the simple method. The basic problem can be illustrated simply by looking at the Newton and Aitken methods. For the Newton method consider the derivative in equation 3 which is approximated by $\frac{g(x_i)-g(x_{i-1})}{x_i-x_{i-1}}$. When there is convergence by constant increments then $g(x_i) = g(x_{i-1})$ and the derivative is zero which means calculating the Newton method's recommended new guess of the fixed point involves division by zero. Now considering the Aitken method of equation 5 the new guess is given by $x_i - \frac{(x_{i+1}-x_i)^2}{x_{i+2}-2x_{i+1}+x_i}$. When there is convergence by constant increments then $x_i - x_{i+1} = x_{i+1} - x_{i+2}$ and so we have $x_{i+2} - 2x_{i+1} + x_i = (x_i - x_{i+1}) - (x_{i+1} - x_{i+2}) = 0$. It is not possible to calculate the new guess.

More generally, when there is convergence by constant increments then then the fixed point method receives information about what direction to go in but no information about how far to go. This is a complication that is common to all fixed point acceleration methods. In these cases it may be possible to change the function to make it converge by varying increments while retaining the same set of fixed points. An example of this is shown in the perceptron example presented later in this paper. In other cases where it is not possible to modify the function, it is advisable to use the simple method.

[5] As the simple method is so often monotonic and convergent the **FixedPoint** package has a "dampening" parameter which allows users to create guesses by linearly combining the guesses of their desired acceleration method with the simple iterates. This allows users to combine the robustness advantages of the simple method with the speed advantages of another method.

Applications of fixed point acceleration with the FixedPoint package

Simple examples with analytical functions

For the simplest possible example we will use the **FixedPoint** package to accelerate the solution for a square root. Consider we want to estimate a square root using the Babylonian method. To find the square root of a number x, given an initial guess t_0, the following sequence converges to the square root:

$$t_{n+1} = \frac{1}{2}\left[t_n + \frac{x}{t_n}\right] \tag{12}$$

This is a fast converging and inexpensive sequence which probably makes an acceleration algorithm overkill but for sake of exposition we can implement this in **FixedPoint**. In the next code block we find the square root of 100 with the SEA method and an initial guess of six:

```
library(FixedPoint)
SequenceFunction = function(tn){0.5*(tn + 100/tn)}
FP = FixedPoint(Function = SequenceFunction, Inputs = 6, Method = "SEA")
```

The benefit of fixed point accelerators is more apparent when applied to vectorised functions. For a simple example consider the below function where each element of the returned vector depends on both elements of the input vector:

```
Vec_Function = function(x){c(0.5*sqrt(abs(x[1] + x[2])), 1.5*x[1] + 0.5*x[2])}
FP_Simple   = FixedPoint(Function = Vec_Function, Inputs = c(0.3,900),
                         Method = "Simple")
FP_Anderson = FixedPoint(Function = Vec_Function, Inputs = c(0.3,900),
                         Method = "Anderson")
```

Here it takes 105 iterates to find a fixed point with the simple method but only 14 with the Anderson acceleration method.

Gas diffusion

For a more complex example consider we want to model the diffusion of gas in a two dimensional space. We set up a two dimensional grid split into ϕ divisions along the side so there are ϕ^2 grid squares in total. Pure nitrogen is being released at location $(1,1)$ and pure oxygen is being released at location (ϕ,ϕ). We are interested in determining the steady state gas concentrations in each square of the grid. We will model equilibrium as occurring when each square has a gas concentration equal to the average of itself with its contiguous squares.

```
phi = 10
Numbering = matrix(seq(1,phi^2,1), phi)           # Numbering scheme for squares

NeighbourSquares = function(n,phi){
  SurroundingIndexes = c(n)
  if (n %% phi != 1){SurroundingIndexes = c(SurroundingIndexes, n-1)}    # above
  if (n %% phi != 0){SurroundingIndexes = c(SurroundingIndexes, n+1)}    # below
  if (n > phi){SurroundingIndexes = c(SurroundingIndexes, n-phi)}        # right
  if (n <= phi^2-phi){SurroundingIndexes = c(SurroundingIndexes, n+phi)} # left
  return(SurroundingIndexes)
}

TwoDimensionalDiffusionIteration = function(x, phi){
  xnew = x
  for (i in 1:(phi^2)){
    Subset = NeighbourSquares(i, phi)
    xnew[i] = mean(x[Subset])
  }
  xnew[1] = 0
  xnew[phi^2] = 1
  return(xnew)
}
```

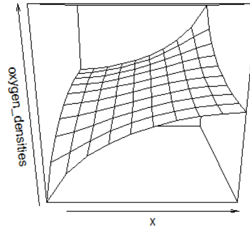

Figure 2: Equilibrium concentrations of Oxygen found by the fixedpoint function

```
FP = FixedPoint(Function = function(x) TwoDimensionalDiffusionIteration(x,phi),
                Inputs =  c(rep(0,50), rep(1,50)), Method = "RRE")
```

The fixed point found here can then be used to plot the density of oxygen over the space. The code for this is below while the plot can be found in figure 2.

```
x = 1:phi
y = 1:phi
oxygen_densities = matrix(FP$FixedPoint, phi)
persp(x, y, oxygen_densities)
```

Finding equilibrium prices in a pure exchange economy

Consider now we are modeling a pure exchange economy and want to determine the equilibrium prices given household preferences and endowments. We have N households. Every household has preferences over G types of good. Household $n \in N$ has a utility function of

$$U_n = \sum_{i=1}^{G} \gamma_{n,i} \log(c_{n,i}) \tag{13}$$

Where $\gamma_{n,i}$ is a parameter describing household n's taste for good i, $c_{n,i}$ is household n's consumption of good i. Each household is endowed with an amount of each good. They can then trade goods before consumption. We have data on each household's endowment and preferences for each good and want to determine the equilibrium prices for this pure exchange economy.

We will choose good 1 as the numeraire, so we will have $P_1 = 1$. First we will find an expression for demand given a price vector. Setting up the lagrangian for household n:

$$L_n = \sum_{i=1}^{G} \gamma_{n,i} \log(c_{n,i}) + \lambda_n [\sum_{i=1}^{G} P_i (e_{n,i} - c_{n,i})] \tag{14}$$

Where λ_n is household n's shadow price, $e_{n,i}$ is this household's endowment of good i and P_i is the price of good i. Taking the first order condition with respect to c_i of this lagrangian yields:

$$c_{n,i} = \frac{\gamma_{n,i}}{P_i \lambda_n} \tag{15}$$

and taking the first order condition with respect to λ_n yields the budget constraint. Subbing the above equation into the budget constraint and rearranging yields:

$$\lambda_n = \frac{\sum_{i=1}^{G} \gamma_{n,i}}{\sum_{i=1}^{G} P_i e_{n,i}} \tag{16}$$

We can also sum over households to find total demand for each good as:

$$D_i = \frac{1}{P_i} \sum_{n=1}^{G} \frac{\gamma_{n,i}}{\lambda_n} \tag{17}$$

We will find the equilibrium price vector by using an approximate price vector to estimate the λs using equation 16. We can then find an estimate of the equilibrium price P_i which solves clears the market, $D_i = \sum_{n=1}^{G} e_{n,i}$:

$$P_i = \frac{\sum_{n=1}^{G} \frac{\gamma_{n,i}}{\lambda_n}}{\sum_{n=1}^{G} e_{n,i}} \tag{18}$$

We use this approach in the code below for the case of 10 goods with 8 households. For exposition sake we generate some data below before proceeding to find the equilibrium price vector.

```
# Generating data
set.seed(3112)
N = 8
G = 10
Endowments = matrix(rlnorm(N*G), nrow = G)
Gamma      = matrix(runif(N*G), nrow = G)
# Every column here represents a household and every row is a good.
# So Endowments[1,2] is the second household's endowment of good 1.

# We now start solving for equilibrium prices:
TotalEndowmentsPerGood  = apply(Endowments, 1, sum)
TotalGammasPerHousehold = apply(Gamma, 2, sum)
LambdasGivenPriceVector = function(Price){
  ValueOfEndowmentsPerHousehold = Price * Endowments
  TotalValueOfEndowmentsPerHousehold = apply(ValueOfEndowmentsPerHousehold, 2, sum)
  return(TotalGammasPerHousehold /TotalValueOfEndowmentsPerHousehold)
}

IterateOnce = function(Price){
  Lambdas = LambdasGivenPriceVector(Price)                      # eqn 16
  GammaOverLambdas = t(apply(Gamma, 1, function(x) x / Lambdas))
  SumGammaOverLambdas = apply(GammaOverLambdas,1,sum)
  NewPrices = SumGammaOverLambdas/ TotalEndowmentsPerGood       # eqn 18
  NewPrices = NewPrices/NewPrices[1]       # normalising with numeraire
  return(NewPrices)
}

InitialGuess = rep(1,10)
FP = FixedPoint(Function = IterateOnce, Inputs =  InitialGuess, Method = "VEA")
```

The fixed point contained in the FP object is the vector of equilibrium prices.

The training of a perceptron classifier

The perceptron is one of the oldest and simplest machine learning algorithms (Rosenblatt, 1958). In its simplest form, for each observation it is applied it uses an N-dimensional vector of features x together with N+1 weights **w** to classify the observation as being of type one or type zero. It classifies observation j as a type one if $w_0 + \sum_{i=1}^{N} w_i x_{i,j} > 0$ and as a type zero otherwise.

The innovation of the perceptron was its method for training its weights, **w**. This is done by looping over a set of observations that can be used for training (the "training set") and for which the true category information is available. The perceptron classifies each observation. When it classifies an observation correctly no action is taken. On the other hand when the perceptron makes an error then it updates its weights with the following expressions.

$$w_0' = w_0 + (d_j - y_j) \tag{19}$$

$$w_i' = w_i + (d_j - y_j)x_{j,i} \qquad \text{for } i > 0 \tag{20}$$

Where w_i is the old weight for the i'th feature and w_i' is the updated weight. $x_{j,i}$ is the feature

value for observation j's feature i, d_j is the category label for observation j and y_j is the perceptron's prediction for this observation's category.

This training algorithm can be rewritten as fixed point problem. We can write a function that takes perceptron weights, loops over the data updating these weights and then returns the updated weight vector. If the perceptron classifies every observation correctly then the weights will not update and we are at a fixed point.[6]

Most acceleration algorithms perform poorly in accelerating the convergence of this perceptron training algorithm. This is due to the perceptron often converging by a fixed increment. This occurs because multiple iterates can result in the same observations being misclassified and hence the same change in the weights. As a result we will use the simple method which is guaranteed to be convergent for this problem (Novikoff, 1963).

```
# Generating linearly separable data
set.seed(10)
data = data.frame(x1 = rnorm(100,4,2), x2 = rnorm(100,8,2), y = -1)
data = rbind(data,data.frame(x1 = rnorm(100,-4,2), x2 = rnorm(100,12), y = 1))

# Iterating training of Perceptron
IteratePerceptronWeights = function(w, LearningRate = 1){
  intSeq = 1:length(data[,"y"])
  for (i in intSeq){
    target = data[i,c("y")]
    score  =  w[1] + (w[2]*data[i, "x1"]) + (w[3]*data[i, "x2"])
    ypred  = 2*(as.numeric( score > 0 )-0.5)
    update = LearningRate * 0.5*(target-ypred)
    w[1]   = w[1] + update
    w[2]   = w[2] + update*data[i, "x1"]
    w[3]   = w[3] + update*data[i, "x2"]
  }
  return(w)
}

InitialGuess = c(1,1,1)
FP = FixedPoint(Function = IteratePerceptronWeights, Inputs =  InitialGuess,
                Method = "Simple", MaxIter = 1200)
```

The result of this algorithm can be seen in figure 3. It can be seen that the classification line perfectly separates the two groups of observations.

Only the simple method is convergent here and it is relatively slow taking 1121 iterations. We can still get a benefit from accelerators however if we can modify the training algorithm to give training increments that change depending on distance from the fixed point. This can be done by updating the weights by an amount proportional to a concave function of the norm of $w_0 + \sum_{i=1}^{N} w_i x_{i,j}$. Note that the instances in which the weights are not updated stay the same and hence the modified training function will result in the same set of fixed points as the basic function. This is done in the next piece of code where the MPE method is used. It can be seen that there is a substantial increase in speed with only 54 iterations required by the MPE method.

```
IteratePerceptronWeights = function(w, LearningRate = 1){
  intSeq = 1:length(data[,"y"])
  for (i in intSeq){
    target = data[i,c("y")]
    score  =  w[1] + (w[2]*data[i, "x1"]) + (w[3]*data[i, "x2"])
    ypred  = 2*(as.numeric( score > 0 )-0.5)
    if ((target-ypred) != 0){
      update = LearningRate * -sign(score) * sqrt(abs(score))
      w[1] = w[1] + update
      w[2] = w[2] + update*data[i, "x1"]
      w[3] = w[3] + update*data[i, "x2"]
```

[6] Note that when a perceptron has one fixed point then there are uncountably many such fixed points where the perceptron correctly classifies the entire training set and will not further update. This is because a scalar multiple of any set of weights will generate the same classification line and the new set of weights will also be a fixed point. There may also be multiple linearly independent hyperplanes that correctly classify every observation. On the other hand it is possible that the data is not linearly separable in which case there may be no fixed point and the weights will continue to update forever.

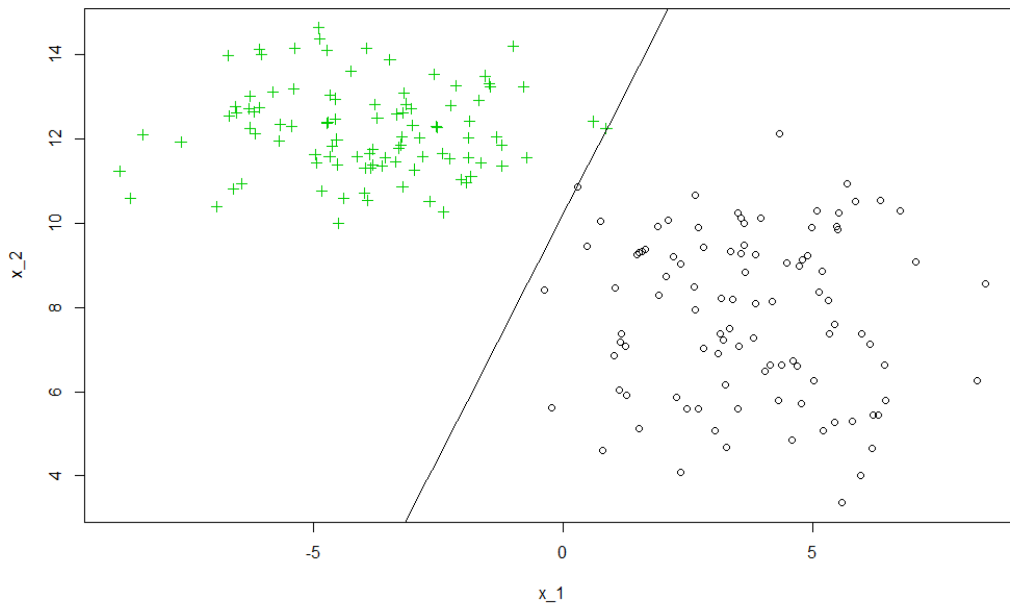

Figure 3: The perceptron linear classifier

```
    }
  }
  return(w)
}
FP = FixedPoint(Function = IteratePerceptronWeights, Inputs =  InitialGuess,
                Method = "MPE")
```

Valuation of a perpetual American put option

For an application in finance consider the pricing of a perpetual American put option on a stock. It never expires unless it is exercised. Its value goes to zero however if the spot price rises to become α times as much as the strike price, denoted S.[7] We will denote x to be the current spot price, σ is the market volatility, d is the risk free rate. In each period the underlying price either increases by a multiple of e^{σ} (which happens with probability p) or decreases by a multiple of $e^{-\sigma}$ (which happens with probability $1-p$) in each unit of time. We have $-\sigma < d < \sigma$.

Given the risk neutral pricing principle the returns from holding the stock must equal the risk-free rate. Hence we must have $pe^{\sigma} + (1-p)e^{-\sigma} = e^{d}$. This implies that:

$$p = \frac{e^{d} - e^{-\sigma}}{e^{\sigma} - e^{-\sigma}} \tag{21}$$

The price of this option at any given spot price of the stock can be solved by means of a fixed point algorithm as shown below:[8]

```
d = 0.05
sigma = 0.1
alpha = 2
S = 10
chi = 0
p = (exp(d) - exp(-sigma) ) / (exp(sigma) - exp(-sigma))
```

[7]This is a common approximation when pricing American options with a finite difference method. While no option's price will ever become exactly zero, at a sufficiently high spot price the option will be low enough value for this to be a good approximation.

[8]In this case the **SQUAREM** package is used with the squarem method. To use the MPE method through the **SQUAREM** package we could add list($K = 2$, method="mpe", square=FALSE) as the control argument to the squarem function call. RRE can be implemented analogously.

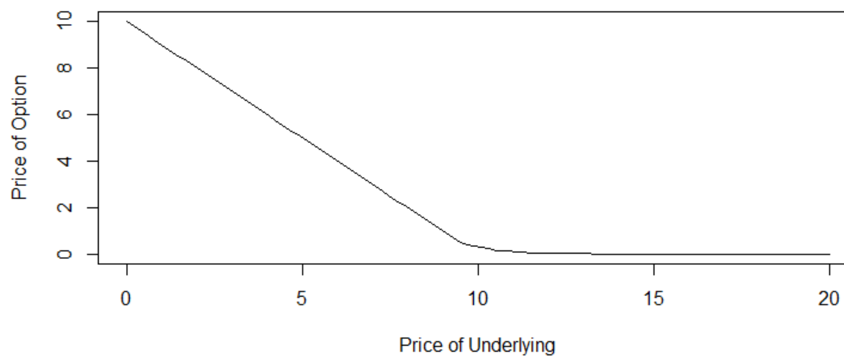

Figure 4: Price of Perpetual American put for each level of the spot price

```
# Initially we guess that the option value decreases linearly from S
# (when the spot price is 0) to 0 (when the spot price is \alpha S).
UnderlyingPrices = seq(0,alpha*S, length.out = 100)
OptionPrice = seq(S,chi, length.out = 100)

ValueOfExercise = function(spot){S-spot}
ValueOfHolding = function(spot, EstimatedValueOfOption){
  if (spot > alpha*S-1e-10){return(chi)}
  IncreasePrice = exp(sigma)*spot
  DecreasePrice = exp(-sigma)*spot
return((p*EstimatedValueOfOption(IncreasePrice) +
          (1-p)*EstimatedValueOfOption(DecreasePrice)))
}
ValueOfOption = function(spot, EstimatedValueOfOption){
  Holding = ValueOfHolding(spot, EstimatedValueOfOption)*exp(-d)
  Exercise = ValueOfExercise(spot)
  return(max(Holding, Exercise))
}
IterateOnce = function(OptionPrice){
  EstimatedValueOfOption = approxfun(UnderlyingPrices, OptionPrice, rule = 2)
  for (i in 1:length(OptionPrice)){
    OptionPrice[i] = ValueOfOption(UnderlyingPrices[i], EstimatedValueOfOption)
  }
  return(OptionPrice)
}

library(SQUAREM)
FP = squarem(par=OptionPrice, IterateOnce)

plot(UnderlyingPrices,FP$par, type = "l",
     xlab = "Price of Underlying", ylab = "Price of Option")
```

Here the fixed point gives the price of the option at any given level of the underlying asset's spot price. This can be visualized as seen in figure 4.

```
plot(UnderlyingPrices,FP$FixedPoint, type = "l",
     xlab = "Price of Underlying", ylab = "Price of Option")
```

A consumption smoothing problem

A common feature of macroeconomic models is the simulation of consumer spending patterns over time. These computations are not trivial, in order for a consumer to make a rational spending decision they need to know their future wellbeing as a function of their future wealth. Often models exhibit infinitely lived consumers without persistent shocks and in this setting the relationship between wealth and wellbeing can be found with a fixed point algorithm. Consider an infinitely lived consumer

that has a budget of B_t at time t and a periodic income of 1. She has a periodic utility function given by $\epsilon_t x_t^\delta$, where x_t is spending in period t and ϵ_t is the shock in period t drawn from some stationary nonnegative shock process with pdf $f(\epsilon)$ defined on the interval $[y,z]$. The problem for the consumer in period t is to maximise her value function:

$$V(B_t|\epsilon_t) = \max_{0<x_t<B_t} \quad \epsilon_t x_t^\delta + \beta \int_y^z V(B_{t+1}|\epsilon)f(\epsilon)d\epsilon \tag{22}$$

Where β is a discounting factor and $B_{t+1} = 1 + B_t - x_t$.

Our goal is to find a function that gives the optimal spending amount, $\hat{x}(B_t,\epsilon_t)$, in period t which is a function of the shock magnitude ϵ_t and the available budget B_t in this period. If we knew the function $\int_y^z V(B_{t+1}|\epsilon)f(\epsilon)d\epsilon$ then we could do this by remembering $B_{t+1} = 1 + B_t - x_t$ and using the optimisation:

$$\hat{x}(B_t,\epsilon_t) = \text{argmax}_{0<x_t<B_t} \quad \epsilon_t x_t^\delta + \beta \int_y^z V(B_{t+1}|\epsilon)f(\epsilon)d\epsilon \tag{23}$$

So now we need to find the function $\int_y^z V(B_{t+1}|\epsilon)f(\epsilon)d\epsilon$. Note as the shock process is stationary, the consumer lives forever and income is always 1, this function will not vary with t. As a result we will rewrite it as simply $f(b)$, where b is the next period's budget.

Now we will construct a vector containing a grid of budget values, $\bar{b}$, for instance $\bar{b} = [0, 0.01, 0.02, ..., 5]$ (we will use bars to describe approximations gained from this grid). If we could then approximate a vector of the corresponding function values, $\bar{f}$, so we had for instance $\bar{f} = [f(0), f(0.01), f(0.02), ..., f(5)]$ then we could approximate the function by constructing a spline $\bar{f}(b)$ between these points. Then we can get the function:

$$\bar{x}(B_t,\epsilon_t) = \text{argmax}_{0<x<B_t} \quad \epsilon_t x_t^\delta + \bar{f}(B_t - x) \tag{24}$$

So this problem reduces to finding the vector of function values at a discrete number of points, $\bar{f}$. This can be done as a fixed point problem. We can first note that this problem is a contraction mapping problem. In this particular example this means that if we define a sequence $\bar{f}_0 = f_0$ where f_0 is some initial guess and $f_{i+1} = g(f_i)$ where g is given by the IterateOnce function below then this sequence will be convergent.[9] Convergence would be slow however so below we will actually use the Anderson method:

```
library(FixedPoint)
library(schumaker)
library(cubature)
delta = 0.2
beta = 0.99
BudgetStateSpace = c(seq(0,1, 0.015), seq(1.05,3,0.05))
InitialGuess = sqrt(BudgetStateSpace)

ValueGivenShock = function(Budget, epsilon, NextValueFunction){
  optimize(f = function(x) epsilon*(x^delta) + beta*NextValueFunction(Budget - x + 1),
           lower = 0, upper = Budget, maximum = TRUE)
}

ExpectedUtility = function(Budget, NextValueFunction){
  if (Budget > 0.001){
    adaptIntegrate(f = function(epsilon) ValueGivenShock(Budget,
        epsilon,NextValueFunction)$objective * dlnorm(epsilon),
        lowerLimit = qlnorm(0.0001), upperLimit = qlnorm(0.9999))$integral
  } else {
    beta*NextValueFunction(1)
  }
```

[9]We use two additional packages in solving this problem. The first is the **cubature** package (Narasimhan and Johnson, 2017) which is used for the integral in equation 23. The second is the **schumaker** package (Baumann and Klymak, 2017) which generates a spline representing $\bar{f}(B_t - x)$ in equation 24. It is necessary for this spline to be shape preserving to ensure there is a unique local maxima to be found for the optimiser used in evaluating this expression.

```
}

IterateOnce = function(BudgetValues){
  NextValueFunction = schumaker::Schumaker(BudgetStateSpace, BudgetValues,
                                           Extrapolation = "Linear")$Spline
  for (i in 1:length(BudgetStateSpace)){   # This is often a good loop to parallelise
    BudgetValues[i] = ExpectedUtility(BudgetStateSpace[i], NextValueFunction)
  }
  return(BudgetValues)
}
FP = FixedPoint(Function = IterateOnce, Inputs = InitialGuess,
                        Method = "Anderson")
```

This takes 71 iterates which is drastically better than the 2316 iterates it takes with the simple method. Now the optimal spending amount can be found for any given budget and any income shock. For instance with the following code we can work out what a consumer with a budget of 1.5 and a shock of 1.2 would spend:

```
NextValueFunction = Schumaker(BudgetStateSpace, FP$FixedPoint)$Spline
ValueGivenShock(1.5, 1.2, NextValueFunction)$maximum
```

Using parallelisation with the Anderson method

It takes 71 iterates for the Anderson method to find the fixed point, however we might want to get it going even faster though parallelisation. The easiest way to do this for this particular problem is to parallelise the for loop through the budgetspace. For exposition however we show how to do this by doing multiple iterates at the same time. We will do this by using six cores and using the parallel capabilities of the **foreach** and **doParallel** packages (Revolution Analytics and Weston, 2015; Microsoft Corporation and Weston, 2017). Each node will produce an different guess vector through the Anderson method. This will be done by giving each node a different subset of the previous iterates that have been completed. The first node will have all previous iterate information. For $i > 1$, the ith node will have all previous iterates except for the ith most recent iterate. The code for this approach is presented in the appendix.

This parallel method takes 102 iterates when using six cores which takes approximately the same time as running $6 + \frac{96}{6} = 22$ iterates sequentially. This is a significant speedup and is possible with the Anderson method as previous iterates do not need to be sequential. The simple parallel algorithm here may also be able to be modified for better performance, for instance different methods could be used in each core or the dampening parameter could be modified.

Speed of convergence comparison

All of the algorithms of the **FixedPoint** package as well as the squarem algorithm of the **SQUAREM** package were run for a variety of problems. In addition to all of the above problems fixed points were found for some basic analytical functions such as $cos(x)$, $x^{\frac{1}{3}}$ and the linear case of $95(18 - x)$.[10] The results are shown in table 1.

It can be seen that the Anderson algorithm performed well in almost all cases. The minimal polynomial methods tended to outperform the epsilon extrapolation methods. This is largely in agreement with previous benchmarking performed in Jbilou and Sadok (2000). The MPE tended to generally outperform the RRE and the VEA outperformed the SEA in all cases. The squarem method tended to be outperformed by the standard minimal polynomial methods. While it was generally amongst the slowest methods, the simple method was the most generally applicable, converging in all but one of the test cases studied.

Conclusion

R has had available a multitude of algorithms for rootfinding and multidimensional optimisation for a long time. Until recently however the range of fixed point accelerators available in **R** has been limited.

[10]The starting guesses, convergence criteria, etc can also be found in the test files for **FixedPoint** which are included with the package's source files. The squarem method provided in the **SQUAREM** package checks for convergence in a different way to the **FixedPoint** package. To overcome this the convergence target was adjusted for this package so that in general the squarem achieves slightly less convergence than the **FixedPoint** methods in the convergence tests in this table which results in any bias being slightly in favor of the squarem method.

Case	Dimensions	Function
1	1	Babylonian Square Root
2	1	cos(x)
3	6	$x^{1/3}$
4	6	$95(18 - x)$
5	2	Simple Vector Function
6	100	Gas Diffusion
7	3	Perceptron
8	3	Modified Perceptron
9	10	Equilibrium Prices
10	100	Perpetual American Put
11	107	Consumption Smoothing

Case	Simple	Anderson	Aitken	Newton	VEA	SEA	MPE	RRE	squarem
1	6	7	7	7	6	6	6	6	6
2	58	7	11	9	13	13	19	25	55
3	22	12	9	9	13	13	9	10	12
4	*	5	3	3	25	*	19	7	*
5	105	14	67	239	20	25	31	31	44
6	221	26	323	*	150	221	44	50	159
7	1121	*	*	*	*	*	*	*	*
8	1156	*	*	20	75	158	54	129	638
9	11	9	14	24	11	11	11	12	11
10	103	37	203	*	108	103	43	52	103
11	2316	71	*	*	*	*	217	159	285

Table 1: The performance of each algorithm for test cases. An asterix indicates the algorithm did not converge.

Before the release of **FixedPoint**, only the squarem method of the **SQUAREM** package was available as a general use fixed point accelerator.

This paper examines the use of fixed point accelerators in **R**. The algorithms of the **FixedPoint** and **SQUAREM** packages are used to demonstrate the use of fixed point acceleration algorithms in the solution of numerical mathematics problems. A number of applications were shown. First the package was used to accelerate the finding of an equilibrium distribution of gas in a diffusion setting. The package was then used to accelerate the training of a perceptron classifier. The acceleration of this training was complicated by the training function converging in fixed increments however it was possible to speed up the solution using a fixed point accelerator by changing the training algorithm while retaining the same set of fixed points. A number of problems in economics were then examined. First the equilibrium price vector was found for a pure exchange economy. Next a vector was found that gives the price of a perpetual American put option at various values of the underlying asset's spot price. Finally the future value function was found for an infinitely lived consumer facing a consumption smoothing problem.

In all of these example applications it can be noted that the solving for a fixed point was accelerated significantly by the use of a fixed point acceleration algorithm. In many cases an accelerator was available that was more than an order of magnitude faster than the simple method. The results indicate that large speedups are available to **R** programmers that are able to apply fixed point acceleration algorithms to their numerical problem of interest.

Bibliography

D. G. Anderson. Iterative procedures for nonlinear integral equations. *Journal of the ACM*, 12(4): 547–560, 1965. URL https://doi.org/10.1145/321296.321305. [p4]

S. Baumann and M. Klymak. *Schumaker: Schumaker Shape-Preserving Spline*, 2017. URL https://CRAN.R-project.org/package=schumaker. R package version 1.0. [p12]

S. Baumann and M. Klymak. *FixedPoint: Algorithms for Finding Fixed Point Vectors of Functions*, 2018. URL https://cran.r-project.org/package=FixedPoint. [p1]

J. F. Bobb and R. Varadhan. *turboEM: A Suite of Convergence Acceleration Schemes for EM, MM and Other*

Fixed-Point Algorithms, 2014. URL https://CRAN.R-project.org/package=turboEM. R package version 2014.8-1. [p1]

S. Cabay and L. W. Jackson. A polynomial extrapolation method for finding limits and antilimits of vector sequences. *Siam Journal of Numerical Analysis*, 13(5):734–752, 1976. URL https://doi.org/10.1137/0713060. [p3]

N. Henderson and R. Varadhan. *Daarem: Damped Anderson Acceleration with Epsilon Monotonicity for Accelerating EM-Like Monotone Algorithms*, 2018. URL https://cran.r-project.org/package=daarem. [p1]

K. Jbilou and H. Sadok. Vector extrapolation methods. applications and numerical comparison. *Journal of Computational and Applied Mathematics*, 122(1-2):149–165, 2000. URL https://doi.org/10.1016/S0377-0427(00)00357-5. [p13]

W. R. Mebane, Jr. and J. S. Sekhon. Genetic optimization using derivatives: The rgenoud package for R. *Journal of Statistical Software*, 42(11):1–26, 2011. URL https://doi.org/10.18637/jss.v042.i11. [p1]

Microsoft Corporation and S. Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*, 2017. URL https://CRAN.R-project.org/package=doParallel. R package version 1.0.11. [p13]

B. Narasimhan and S. G. Johnson. *Cubature: Adaptive Multivariate Integration over Hypercubes*, 2017. URL https://CRAN.R-project.org/package=cubature. R package version 1.3-11. [p12]

A. Novikoff. On convergence proofs for perceptrons. *Stanford Research Institute: Technical Report*, 298258, 1963. [p9]

Revolution Analytics and S. Weston. *Foreach: Provides Foreach Looping Construct for R*, 2015. URL https://CRAN.R-project.org/package=foreach. R package version 1.4.3. [p13]

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. URL https://doi.org/10.1037/h0042519. [p8]

O. Roustant, D. Ginsbourger, and Y. Deville. DiceKriging, DiceOptim: Two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization. *Journal of Statistical Software*, 51(1):1–55, 2012. URL http://doi.org/10.18637/jss.v051.i01. [p1]

D. Smith, W. Ford, and A. Sidi. Extrapolation methods for vector sequences. *SIAM Review*, 29(2):199–233, 1987. URL https://doi.org/10.1137/1029042. [p3]

N. Stokey, R. E. Lucas, and E. Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989. ISBN 9780674750968. [p2]

R. Varadhan. *SQUAREM: Squared Extrapolation Methods for Accelerating EM-Like Monotone Algorithms*, 2010. URL https://CRAN.R-project.org/package=SQUAREM. R package version 2017.10-1. [p1]

R. Varadhan and P. Gilbert. BB: An R package for solving a large system of nonlinear equations and for optimizing a high-dimensional nonlinear objective function. *Journal of Statistical Software*, 32(4):1–26, 2009. URL https://doi.org/10.18637/jss.v032.i04. [p1]

R. Varadhan and C. Roland. Simple and globally convergent methods for accelerating the convergence of any em algorithm. *Scandinavian Journal of Statistics*, 35(2):335–353, 2008. URL https://doi.org/10.1111/j.1467-9469.2007.00585.x. [p4]

P. Wynn. Acceleration techniques for iterated vector and matrix problems. *Mathematics of Computation*, 16(79):301–322, 1962. URL https://doi.org/10.2307/2004051. [p3]

Appendix: An algorithm for finding a fixed point while using parallelisation

```
library(foreach)
library(doParallel)
cores = 6

NodeTaskAssigner = function(Inputs, Outputs, i, Function){
  library(FixedPoint)
  library(schumaker)
  library(cubature)
```

```
  Iterates = dim(Inputs)[2]
  if (i > 1.5) {IterateToDrop = Iterates-i+1} else {IterateToDrop = 0}
  IteratesToUse = (1:Iterates)[ 1:Iterates != IterateToDrop]
  Inputs = matrix(Inputs[,IteratesToUse], ncol = length(IteratesToUse), byrow = FALSE)
  Outputs = matrix(Outputs[,IteratesToUse], ncol = length(IteratesToUse), byrow = FALSE)
  Guess = FixedPointNewInput(Inputs = Inputs, Outputs = Outputs, Method = "Anderson")
  Outputs = matrix(Function(Guess), ncol = 1, byrow = FALSE)
  Inputs = matrix(Guess, ncol = 1, byrow = FALSE)

  return(list(Inputs = Inputs, Outputs = Outputs))
}

# This combines the results returned by each node
CombineLists = function(List1, List2){
  width = dim(List1$Inputs)[2] + dim(List2$Inputs)[2]
  C = list()
  C$Inputs  = matrix(c(List1$Inputs , List2$Inputs ), ncol = width, byrow = FALSE)
  C$Outputs = matrix(c(List1$Outputs, List2$Outputs), ncol = width, byrow = FALSE)
  return(C)
}

# ReSortIterations
# This function takes the previous inputs and outputs from the function, removes
# duplicates and then sorts them in order of increasing convergence.
ReSortIterations = function(PreviousIterates,
                            ConvergenceMetric = function(Resids){max(abs(Resids))})
{
  # Removing any duplicates
  NotDuplicated   = (!(duplicated.matrix(PreviousIterates$Inputs, MARGIN = 2)))
  PreviousIterates$Inputs  = PreviousIterates$Inputs[,NotDuplicated]
  PreviousIterates$Outputs = PreviousIterates$Outputs[,NotDuplicated]
  # Resorting
  Resid       = PreviousIterates$Outputs - PreviousIterates$Inputs
  Convergence = ConvergenceVector = sapply(1:(dim(Resid)[2]), function(x)
    ConvergenceMetric(Resid[,x]) )
  Reordering  = order(Convergence, decreasing = TRUE)
  PreviousIterates$Inputs  = PreviousIterates$Inputs[,Reordering]
  PreviousIterates$Outputs = PreviousIterates$Outputs[,Reordering]
  return(PreviousIterates)
}


ConvergenceMetric = function(Resid){max(abs(Resid))}

# Preparing for clustering and getting a few runs to input to later functions:
PreviousRuns = FixedPoint(Function = IterateOnce, Inputs = InitialGuess,
                          Method = "Anderson", MaxIter = cores)
PreviousRuns$Residuals = PreviousRuns$Outputs - PreviousRuns$Inputs
PreviousRuns$Convergence = apply(PreviousRuns$Residuals, 2, ConvergenceMetric)
ConvergenceVal = min(PreviousRuns$Convergence)

registerDoParallel(cores=cores)

iter = cores
while (iter < 100 & ConvergenceVal > 1e-10){
  NewRuns = foreach(i = 1:cores, .combine=CombineLists) %dopar% {
    NodeTaskAssigner(PreviousRuns$Inputs, PreviousRuns$Outputs, i, IterateOnce)
  }
  # Appending to previous runs
  PreviousRuns$Inputs = matrix(c(PreviousRuns$Inputs, NewRuns$Inputs),
                   ncol = dim(PreviousRuns$Inputs)[2] + cores, byrow = FALSE)
  PreviousRuns$Outputs = matrix(c(PreviousRuns$Outputs, NewRuns$Outputs),
                   ncol = dim(PreviousRuns$Outputs)[2] + cores, byrow = FALSE)
  PreviousRuns = ReSortIterations(PreviousRuns)
```

```
  PreviousRuns$Residuals = PreviousRuns$Outputs - PreviousRuns$Inputs
  PreviousRuns$Convergence = apply(PreviousRuns$Residuals, 2, ConvergenceMetric)
  # Finding Convergence
  ConvergenceVal = min(PreviousRuns$Convergence)
  iter = iter + cores
}

stopImplicitCluster()
# And the fixed point comes out to be:
PreviousRuns$Outputs[, dim(PreviousRuns$Outputs)[2]]
```

Stuart Baumann
ORCiD: 0000-0002-9657-0969
stuart@stuartbaumann.com

Margaryta Klymak
University of Oxford
ORCiD: 0000-0003-4376-883X
margaryta.klymak@qeh.ox.ac.uk